

Contents

Active Template Library (ATL)

Active Template Library (ATL) concepts

Active Template Library (ATL) concepts

Introduction to COM and ATL

Introduction to COM and ATL

Introduction to COM

Introduction to COM

Interfaces (ATL)

IUnknown

Reference counting

QueryInterface

Marshaling

Aggregation

Introduction to ATL

Introduction to ATL

Using a template library

Scope of ATL

Recommendations for choosing between ATL and MFC

Fundamentals of ATL COM objects

Fundamentals of ATL COM objects

Implementing CComObjectRootEx

Implementing CComObject, CComAggObject, and CComPolyObject

Supporting IDispatch and IErrorInfo

Supporting IDispatchImpl

Changing the default class factory and aggregation model

Creating an aggregated object

Dual interfaces and ATL

Dual interfaces and ATL

Implementing a dual interface

Multiple dual interfaces

nonextensible attribute

Dual interfaces and events

ATL collections and enumerators

ATL collections and enumerators

ATL collection and enumerator classes

Design principles for collection and enumerator interfaces

Implementing a C++ Standard Library-based collection

Implementing a C++ Standard Library-based collection

ATL copy policy classes

ATL composite control fundamentals

ATL composite control fundamentals

Inserting a composite control

Modifying the ATL project

Adding functionality to the composite control

Building and testing the ATL project

ATL control containment FAQ

ATL COM property pages

ATL COM property pages

Specifying property pages

Implementing property pages

Implementing property pages

Example: Implementing a property page

ATL support for DHTML controls

ATL support for DHTML controls

Identifying the elements of the DHTML control project

Calling C++ code from DHTML

Creating an ATL DHTML control

Testing the ATL DHTML control

Modifying the ATL DHTML control

Testing the modified ATL DHTML control

ATL connection points

ATL connection points

ATL connection point classes

Adding connection points to an object

ATL connection point example

Event handling and ATL

Event handling and ATL

Event handling principles

Implementing the event handling interface

Using IDispatchImpl

Using IDispatchSimpleImpl

ATL event handling summary

ATL and the free threaded marshaler

Specifying the threading model for a project (ATL)

ATL module classes

ATL services

ATL services

CAtlServiceModuleT::Start function

CAtlServiceModuleT::ServiceMain function

CAtlServiceModuleT::Run function

CAtlServiceModuleT::Handler function

Registry entries

DCOMCNFG

Debugging tips

Debugging tips

Using Task Manager

Displaying assertions

Running the program as a local server

ATL window classes

ATL window classes

Introduction to ATL window classes

Using a window

Implementing a window

Implementing a window

[Adding an ATL message handler](#)

[Message maps \(ATL\)](#)

[Message handler functions](#)

[Message handler functions](#)

[CommandHandler](#)

[MessageHandler](#)

[NotifyHandler](#)

[Implementing a window with CWindowImpl](#)

[Implementing a dialog box](#)

[Using contained windows](#)

[Understanding window traits](#)

[ATL collection classes](#)

[ATL registry component \(Registrar\)](#)

[ATL registry component \(Registrar\)](#)

[Creating Registrar scripts](#)

[Creating Registrar scripts](#)

[Understanding Backus-Naur form \(BNF\) syntax](#)

[Understanding parse trees](#)

[Registry scripting examples](#)

[Using replaceable parameters \(The Registrar's preprocessor\)](#)

[Invoking scripts](#)

[Setting up a static link to the Registrar code \(C++ only\)](#)

[Programming with ATL and C Run-time code](#)

[Programming with ATL and C Run-time code](#)

[Benefits and tradeoffs of the method used to link to the CRT](#)

[Link to the CRT in your ATL project](#)

[Programming with CComBSTR \(ATL\)](#)

[ATL encoding reference](#)

[ATL utilities reference](#)

[Active Template Library \(ATL\) tutorial](#)

[Active Template Library \(ATL\) tutorial](#)

[Creating the project \(ATL tutorial, part 1\)](#)

[Adding a control \(ATL tutorial, part 2\)](#)

[Adding a property to the control \(ATL tutorial, part 3\)](#)

[Changing the drawing code \(ATL tutorial, part 4\)](#)

[Adding an event \(ATL tutorial, part 5\)](#)

[Adding a property page \(ATL tutorial, part 6\)](#)

[Putting the control on a web page \(ATL tutorial, part 7\)](#)

[ATL class overview by category](#)

[ATL class overview](#)

[Class factories classes](#)

[Class information classes](#)

[Collection classes](#)

[COM modules classes](#)

[Composite controls classes](#)

[Connection points classes](#)

[Control containment classes](#)

[Controls: General support classes](#)

[Data transfer classes](#)

[Data types classes](#)

[Debugging and exceptions classes](#)

[Dual interfaces classes](#)

[Enumerators and collections classes](#)

[Error information classes](#)

[File handling classes](#)

[Interface pointers classes](#)

[IUnknown implementation classes](#)

[Memory management classes](#)

[MMC snap-in classes](#)

[Object safety classes](#)

[Persistence classes](#)

[Properties and property pages classes](#)

[Registry support classes](#)

[Running objects classes](#)

Security classes

Service provider support classes

Site information classes

String and text classes

Tear-off interfaces classes

Thread pooling classes

Threading models and critical sections classes

UI support classes

Utility classes

Windows support classes

ATL library reference

ATL class and struct reference

ATL classes and structs

_ATL_BASE_MODULE70 structure

_ATL_COM_MODULE70 structure

_ATL_FUNC_INFO structure

_ATL_MODULE70 structure

_ATL_WIN_MODULE70 structure

_AtlCreateWndData structure

ATL_DRAWINFO structure

_U_MENUorID class

_U_RECT class

_U_STRINGorID class

CA2AEX class

CA2CAEX class

CA2WEX class

CAccessToken class

CAcl class

CAdapt class

CAtlArray class

CAtlAutoThreadModule class

CAtlAutoThreadModuleT class

CAtlBaseModule class
CAtlComModule class
CAtlDebugInterfacesModule class
CAtlDllModuleT class
CAtlException class
CAtlExeModuleT class
CAtlFile class
CAtlFileMapping class
CAtlFileMappingBase class
CAtlList class
CAtlMap class
CAtlModule class
CAtlModuleT class
CAtlPreviewCtrlImpl class
CAtlServiceModuleT class
CAtlTemporaryFile class
CAtlTransactionManager class
CAtlWinModule class
CAutoPtr class
CAutoPtrArray class
CAutoPtrElementTraits class
CAutoPtrList class
CAutoRevertImpersonation class
CAutoVectorPtr class
CAutoVectorPtrElementTraits class
CAxDialogImpl class
CAxWindow class
CAxWindow2T class
CBindStatusCallback class
CComAggObject class
CComAllocator class
CComApartment class

CComAutoCriticalSection class
CComAutoDeleteCriticalSection class
CComAutoThreadModule class
CComBSTR class
CComCachedTearOffObject class
CComClassFactory class
CComClassFactory2 class
CComClassFactoryAutoThread class
CComClassFactorySingleton class
CComCoClass class
CComCompositeControl class
CComContainedObject class
CComControl class
CComControlBase class
CComCriticalSection class
CComCritSecLock class
CComCurrency class
CComDynamicUnkArray class
CComEnum class
CComEnumImpl class
CComEnumOnSTL class
CComFakeCriticalSection class
CComGITPtr class
CComHeap class
CComHeapPtr class
CComModule class
CComMultiThreadModel class
CComMultiThreadModelNoCS class
CComObject class
CComObjectGlobal class
CComObjectNoLock class
CComObjectRoot class

CComObjectRootEx class
CComObjectStack class
CComPolyObject class
CComPtr class
CComPtrBase class
CComQIPtr class
CComQIPtrElementTraits class
CComSafeArray class
CComSafeArrayBound class
CComSafeDeleteCriticalSection class
CComSimpleThreadAllocator class
CComSingleThreadModel class
CComTearOffObject class
CComUnkArray class
CComVariant class
CContainedWindowT class
CCRTAllocator class
CCRTHeap class
CDacl class
CDebugReportHook class
CDefaultCharTraits class
CDefaultCompareTraits class
CDefaultElementTraits class
CDefaultHashTraits class
CDialogImpl class
CDynamicChain class
CElementTraits class
CElementTraitsBase class
CFirePropNotifyEvent class
CGlobalHeap class
CHandle class
CHeapPtr class

CHeapPtrBase class
CHeapPtrElementTraits class
CHeapPtrList class
CInterfaceArray class
CInterfaceList class
CLocalHeap class
CMessageMap class
CNonStatelessWorker class
CNoWorkerThread class
CPathT class
CPrimitiveElementTraits class
CPrivateObjectSecurityDesc class
CRBMap class
CRBMultiMap class
CRBTree class
CRegKey class
CRTThreadTraits class
CSacl class
CSecurityAttributes class
CSecurityDesc class
CSid class
CSimpleArray class
CSimpleArrayEqualHelper class
CSimpleArrayEqualHelperFalse class
CSimpleDialog class
CSimpleMap class
CSimpleMapEqualHelper class
CSimpleMapEqualHelperFalse class
CSnapInItemImpl class
CSnapInPropertyPageImpl class
CSocketAddr class
CStockPropImpl class

CStringElementTraits class
CStringElementTraitsI class
CStringRefElementTraits class
CThreadPool class
CTokenGroups class
CTokenPrivileges class
CUrl class
CW2AEX class
CW2CWEX class
CW2WEX class
CWin32Heap class
CWindow class
CWindowImpl class
CWinTraits class
CWinTraitsOR class
CWndClassInfo class
CWorkerThread class
IAtlAutoThreadModule class
IAtlMemMgr class
IAxWinAmbientDispatch interface
IAxWinAmbientDispatchEx interface
IAxWinHostWindow interface
IAxWinHostWindowLic interface
ICollectionOnSTLImpl class
IConnectionPointContainerImpl class
IConnectionPointImpl class
IDataObjectImpl class
IDispatchImpl class
IDispEventImpl class
IDispEventSimpleImpl class
IDocHostUIHandlerDispatch interface
IEnumOnSTLImpl class

- IObjectSafetyImpl class
- IObjectWithSiteImpl class
- IOleControllImpl class
- IOleInPlaceActiveObjectImpl class
- IOleInPlaceObjectWindowlessImpl class
- IOleObjectImpl class
- IPerPropertyBrowsingImpl class
- IPersistPropertyBagImpl class
- IPersistStorageImpl class
- IPersistStreamInitImpl class
- IPointerInactiveImpl class
- IPropertyNotifySinkCP class
- IPropertyPage2Impl class
- IPropertyPageImpl class
- IProvideClassInfo2Impl class
- IQuickActivateImpl class
- IRegistrar interface
- IRunnableObjectImpl class
- IServiceProviderImpl class
- ISpecifyPropertyPagesImpl class
- ISupportErrorInfoImpl class
- IThreadPoolConfig interface
- IViewObjectExImpl class
- IWorkerThreadClient interface
- Win32ThreadTraits class
- Worker archetype
- ATL_URL_SCHEME enumeration
- ATL function reference
 - ATL functions
 - ATL HTTP utility functions
 - ATL text encoding functions
 - ATL path functions

- COM map global functions
- Composite control global functions
- Connection point global functions
- Debugging and error reporting global functions
- Device context global functions
- Event handling global functions
- Marshaling global functions
- Pixel-HIMETRIC conversion global functions
- Registry and TypeLib global functions
- Security global functions
- Security identifier global functions
- Server registration global functions
- WinModule global functions

ATL macro reference

- ATL macros
- Aggregation and class factory macros
- Category macros
- COM map macros
- COM interface entry macros
- Compiler options macros
- Composite control macros
- Connection point macros
- Debugging and error reporting macros
- Exception handling macros
- Message map macros (ATL)
- Object map macros
- Object status macros
- Property map macros
- Registry Data Exchange macros
- Registry macros
- Service map macros
- Snap-in object macros

[String conversion macros](#)

[Window class macros](#)

[Windows messages macros](#)

[ATL operator reference](#)

[ATL global variable reference](#)

[ATL typedef reference](#)

[ATL projects in Visual Studio](#)

[ATL wizard and dialog box reference](#)

[ATL wizards and dialog boxes](#)

[ATL Active Server Page Component Wizard reference](#)

[ATL Active Server Page Component Wizard](#)

[ATL Active Server Page Component Wizard, ASP](#)

[ATL Active Server Page Component Wizard, Options](#)

[ATL COM+ 1.0 Component Wizard reference](#)

[ATL COM+ 1.0 Component Wizard](#)

[COM+ 1.0, ATL COM+ 1.0 Component Wizard](#)

[ATL Control Wizard reference](#)

[ATL Control Wizard](#)

[ATL Control Wizard, Appearance](#)

[ATL Control Wizard, Interfaces](#)

[ATL Control Wizard, Options](#)

[ATL Control Wizard, Stock Properties](#)

[ATL Dialog Wizard reference](#)

[ATL OLE DB Consumer Wizard reference](#)

[ATL OLE DB Provider Wizard reference](#)

[ATL Project Wizard reference](#)

[ATL Project Wizard](#)

[Application Settings, ATL Project Wizard](#)

[ATL Property Page Wizard reference](#)

[ATL Property Page Wizard](#)

[ATL Property Page Wizard, Options](#)

[ATL Property Page Wizard, Strings](#)

[ATL Simple Object Wizard reference](#)

[ATL Simple Object Wizard](#)

[ATL Simple Object Wizard, Options](#)

[Add a new interface in an ATL project](#)

[Add an ATL Active Server Page component](#)

[Add an ATL COM+ 1.0 component](#)

[Add an ATL control](#)

[Add an ATL dialog box](#)

[Add an ATL OLE DB consumer](#)

[Add an ATL OLE DB provider](#)

[Add an ATL property page](#)

[Add an ATL simple object](#)

[Add objects and controls to an ATL project](#)

[Create an ATL project](#)

[COM+ 1.0 support in ATL projects](#)

[Default ATL project configurations](#)

[Make an ATL object noncreatable](#)

[MFC support in ATL projects](#)

[Specify compiler optimization for an ATL project](#)

ATL COM Desktop Components

4/1/2019 • 2 minutes to read • [Edit Online](#)

The ATL Reference documents the Active Template Library (ATL), a set of template-based C++ classes that simplify the programming of Component Object Model (COM) objects. COM is a binary specification for creating and consuming software components on Windows. To fully take advantage of ATL, a working familiarity with COM is highly recommended. For more information about COM, see [Component Object Model \(COM\)](#).

In This Section

[ATL Class Overview](#)

Provides links to and brief descriptions of the ATL classes organized by category.

[ATL Classes and structs](#)

Provides reference material on the classes and structs organized alphabetically.

[ATL Functions](#)

Provides reference material on the global functions organized alphabetically. Includes topics organizing the functions into categories.

[ATL Global Variables](#)

Provides reference material on the global variables organized alphabetically.

[ATL Macros](#)

Provides reference material on the macros organized alphabetically. Includes topics organizing the macros into categories.

[ATL Typedefs](#)

Provides reference material on the typedefs organized alphabetically

[Worker Archetype](#)

Provides a links to the ATL Worker archetype.

Related Sections

[ATL](#)

Provides topics on how to program using the Active Template Library (ATL).

[ATL Tutorial](#)

Leads you through the creation of a control and demonstrates some ATL fundamentals in the process.

[ATL Samples](#)

Sample code that shows how to use ATL to write COM objects.

[OLE DB Templates](#)

Provides reference material for the OLE DB consumer and provider templates, a set of template classes that implement many commonly used OLE DB interfaces.

Active Template Library (ATL) Concepts

5/17/2019 • 3 minutes to read • [Edit Online](#)

The Active Template Library (ATL) is a set of template-based C++ classes that let you create small, fast Component Object Model (COM) objects. It has special support for key COM features, including stock implementations, dual interfaces, standard COM enumerator interfaces, connection points, tear-off interfaces, and ActiveX controls.

If you do a lot of ATL programming, you will want to learn more about COM and .NET attributes, which is designed to simplify COM programming. For more information, see [Attributed Programming](#). (COM and .NET attributes are not to be confused with the `[[attribute]]` feature in the C++ standard.)

In This Section

[Introduction to COM and ATL](#)

Introduces the major concepts behind the Component Object Model (COM). This article also briefly explains what ATL is and when you should use it.

[Fundamentals of ATL COM Objects](#)

Discusses the relationship among various ATL classes and how those classes are implemented.

[Dual Interfaces and ATL](#)

Describes dual interfaces from an ATL perspective.

[ATL Collections and Enumerators](#)

Describes the implementation and creation of collections and enumerators in ATL.

[Composite Control Fundamentals](#)

Provides step-by-step instructions for creating a composite control. A composite control is a type of ActiveX control that can contain other ActiveX controls or Windows controls.

[ATL Control Containment FAQ](#)

Covers the fundamental questions related to hosting controls with ATL.

[ATL COM Property Pages](#)

Shows you how to specify and implement COM property pages.

[ATL Support for DHTML Controls](#)

Provides step-by-step instructions for creating a DHTML control.

[ATL Connection Points](#)

Explains what connection points are and how ATL implements them.

[Event Handling and ATL](#)

Describes the steps that you need to take to handle COM events using ATL's `IDispatchImpl` and `IDispatchSimpleImpl` classes.

[ATL and the Free Threaded Marshaler](#)

Provides details on the ATL Simple Object Wizard's option that allows your class to aggregate the free threaded marshaler (FTM).

[Specifying the Project's Threading Model](#)

Describes the macros that are available to control run-time performance related to threading in your project.

[ATL Module Classes](#)

Discusses the module classes new for ATL 7.0. Module classes implement the basic functionality required by ATL.

[ATL Services](#)

Covers the series of events that occur when a service is implemented. Also talks about some of the concepts related to developing a service.

[ATL Window Classes](#)

Describes how to create, superclass, and subclass windows in ATL. The ATL window classes are not COM classes.

[ATL Collection Classes](#)

Describes how to use arrays and maps in ATL.

[The ATL Registry Component \(Registrar\)](#)

Discusses ATL scripting syntax and replaceable parameters. It also explains how to set up a static link to the Registrar.

[Programming with ATL and C Run-Time Code](#)

Discusses the benefits of linking statically or dynamically to the C Run-Time Library (CRT).

[Programming with CComBSTR](#)

Discusses several situations that require caution when programming with `CComBSTR`.

[Encoding Reference](#)

Provides functions and macros that support encoding in a range of common Internet standards such as uuencode, hexadecimal, and UTF8 in `atlenc.h`.

[Utilities Reference](#)

Provides code for manipulating paths and URLs in the form of `CPathT` and `CUrl`. A thread pool, `CThreadPool`, can be used in your own applications. This code can be found in `atlpath.h` and `atlutil.h`.

Related Sections

[ATL Tutorial](#)

Leads you through the creation of a control and demonstrates some ATL fundamentals in the process.

[ATL Samples](#)

Provides descriptions of and links to the ATL sample programs.

[Creating an ATL Project](#)

Contains information on the ATL Project Wizard.

[ATL Control Wizard](#)

Discusses how to add classes.

[Attributed Programming](#)

Provides an overview on using attributes to simplify COM programming plus a list of links to more detailed topics.

[ATL Class Overview](#)

Provides reference information and links to the ATL classes.

Introduction to COM and ATL

3/4/2019 • 2 minutes to read • [Edit Online](#)

This section provides a brief introduction to COM and ATL.

In This Section

[Introduction to COM](#)

Provides an overview of the Component Object Model's (COM) fundamental concepts, including interfaces, `IUnknown`, reference counting, `QueryInterface`, marshaling, and aggregation.

[Introduction to ATL](#)

Discusses, briefly, what the Active Template Library (ATL) was designed for, template libraries, and ATL version numbers. Includes recommendations for choosing between ATL and MFC.

Related Sections

[The Component Object Model](#)

The Windows SDK material on COM.

[ATL](#)

Provides links to conceptual topics on how to program using the Active Template Library.

[ATL Class Overview](#)

Provides reference information and links to the ATL classes.

Introduction to COM

3/4/2019 • 2 minutes to read • [Edit Online](#)

COM is the fundamental "object model" on which ActiveX Controls and OLE are built. COM allows an object to expose its functionality to other components and to host applications. It defines both how the object exposes itself and how this exposure works across processes and across networks. COM also defines the object's life cycle.

Fundamental to COM are these concepts:

- [Interfaces](#) — the mechanism through which an object exposes its functionality.
- [IUnknown](#) — the basic interface on which all others are based. It implements the reference counting and interface querying mechanisms running through COM.
- [Reference counting](#) — the technique by which an object (or, strictly, an interface) decides when it is no longer being used and is therefore free to remove itself.
- [QueryInterface](#) — the method used to query an object for a given interface.
- [Marshaling](#) — the mechanism that enables objects to be used across thread, process, and network boundaries, allowing for location independence.
- [Aggregation](#) — a way in which one object can make use of another.

See also

[Introduction to COM and ATL](#)

[The Component Object Model](#)

Interfaces (ATL)

3/4/2019 • 2 minutes to read • [Edit Online](#)

An interface is the way in which an object exposes its functionality to the outside world. In COM, an interface is a table of pointers (like a C++ vtable) to functions implemented by the object. The table represents the interface, and the functions to which it points are the methods of that interface. An object can expose as many interfaces as it chooses.

Each interface is based on the fundamental COM interface, [IUnknown](#). The methods of `IUnknown` allow navigation to other interfaces exposed by the object.

Also, each interface is given a unique interface ID (IID). This uniqueness makes it easy to support interface versioning. A new version of an interface is simply a new interface, with a new IID.

NOTE

IIDs for the standard COM and OLE interfaces are predefined.

See also

[Introduction to COM](#)

[COM Objects and Interfaces](#)

IUnknown

3/12/2019 • 2 minutes to read • [Edit Online](#)

[IUnknown](#) is the base interface of every other COM interface. This interface defines three methods: [QueryInterface](#), [AddRef](#), and [Release](#). [QueryInterface](#) allows an interface user to ask the object for a pointer to another of its interfaces. [AddRef](#) and [Release](#) implement reference counting on the interface.

See also

[Introduction to COM](#)

[IUnknown and Interface Inheritance](#)

Reference Counting

3/4/2019 • 2 minutes to read • [Edit Online](#)

COM itself does not automatically try to remove an object from memory when it thinks the object is no longer being used. Instead, the object programmer must remove the unused object. The programmer determines whether an object can be removed based on a reference count.

COM uses the `IUnknown` methods, `AddRef` and `Release`, to manage the reference count of interfaces on an object. The general rules for calling these methods are:

- Whenever a client receives an interface pointer, `AddRef` must be called on the interface.
- Whenever the client has finished using the interface pointer, it must call `Release`.

In a simple implementation, each `AddRef` call increments and each `Release` call decrements a counter variable inside the object. When the count returns to zero, the interface no longer has any users and is free to remove itself from memory.

Reference counting can also be implemented so that each reference to the object (not to an individual interface) is counted. In this case, each `AddRef` and `Release` call delegates to a central implementation on the object, and `Release` frees the entire object when its reference count reaches zero.

NOTE

When a `CComObject`-derived object is constructed using the **new** operator, the reference count is 0. Therefore, a call to `AddRef` must be made after successfully creating the `CComObject`-derived object.

See also

[Introduction to COM](#)

[Managing Object Lifetimes through Reference Counting](#)

QueryInterface

3/12/2019 • 2 minutes to read • [Edit Online](#)

Although there are mechanisms by which an object can express the functionality it provides statically (before it is instantiated), the fundamental COM mechanism is to use the `IUnknown` method called `QueryInterface`.

Every interface is derived from `IUnknown`, so every interface has an implementation of `QueryInterface`. Regardless of implementation, this method queries an object using the IID of the interface to which the caller wants a pointer. If the object supports that interface, `QueryInterface` retrieves a pointer to the interface, while also calling `AddRef`. Otherwise, it returns the `E_NOINTERFACE` error code.

Note that you must obey [Reference Counting](#) rules at all times. If you call `Release` on an interface pointer to decrement the reference count to zero, you should not use that pointer again. Occasionally you may need to obtain a weak reference to an object (that is, you may wish to obtain a pointer to one of its interfaces without incrementing the reference count), but it is not acceptable to do this by calling `QueryInterface` followed by `Release`. The pointer obtained in such a manner is invalid and should not be used. This more readily becomes apparent when `_ATL_DEBUG_INTERFACES` is defined, so defining this macro is a useful way of finding reference counting bugs.

See also

[Introduction to COM](#)

[QueryInterface: Navigating in an Object](#)

Marshaling

3/4/2019 • 2 minutes to read • [Edit Online](#)

The COM technique of marshaling allows interfaces exposed by an object in one process to be used in another process. In marshaling, COM provides code (or uses code provided by the interface implementor) both to pack a method's parameters into a format that can be moved across processes (as well as, across the wire to processes running on other machines) and to unpack those parameters at the other end. Likewise, COM must perform these same steps on the return from the call.

NOTE

Marshaling is typically not necessary when an interface provided by an object is being used in the same process as the object. However, marshaling may be needed between threads.

See also

[Introduction to COM](#)

[Marshaling Details](#)

Aggregation

3/4/2019 • 2 minutes to read • [Edit Online](#)

There are times when an object's implementor would like to take advantage of the services offered by another, prebuilt object. Furthermore, it would like this second object to appear as a natural part of the first. COM achieves both of these goals through containment and aggregation.

Aggregation means that the containing (outer) object creates the contained (inner) object as part of its creation process and the interfaces of the inner object are exposed by the outer. An object allows itself to be aggregatable or not. If it is, then it must follow certain rules for aggregation to work properly.

Primarily, all `IUnknown` method calls on the contained object must delegate to the containing object.

See also

[Introduction to COM](#)

[Reusing Objects](#)

Introduction to ATL

3/4/2019 • 2 minutes to read • [Edit Online](#)

ATL is the Active Template Library, a set of template-based C++ classes with which you can easily create small, fast Component Object Model (COM) objects. It has special support for key COM features including: stock implementations of [IUnknown](#), [IClassFactory](#), [IClassFactory2](#), and `IDispatch`; dual interfaces; standard COM enumerator interfaces; connection points; tear-off interfaces; and ActiveX controls.

ATL code can be used to create single-threaded objects, apartment-model objects, free-threaded model objects, or both free-threaded and apartment-model objects.

Topics covered in this section include:

- How a [template library](#) differs from a standard library.
- What you [can and cannot do with ATL](#).
- [Recommendations for choosing between ATL and MFC](#).

See also

[Introduction to COM and ATL](#)

Using a Template Library

3/4/2019 • 2 minutes to read • [Edit Online](#)

A template is somewhat like a macro. As with a macro, invoking a template causes it to expand (with appropriate parameter substitution) to code you have written. However, a template goes further than this to allow the creation of new classes based on types that you pass as parameters. These new classes implement type-safe ways of performing the operation expressed in your template code.

Template libraries such as ATL differ from traditional C++ class libraries in that they are typically supplied only as source code (or as source code with a little, supporting run time) and are not inherently or necessarily hierarchical in nature. Rather than deriving from a class to get the functionality you desire, you instantiate a class from a template.

See also

[Introduction to ATL](#)

Scope of ATL

3/4/2019 • 2 minutes to read • [Edit Online](#)

ATL allows you to easily create COM objects, Automation servers, and ActiveX controls. ATL provides built-in support for many of the fundamental COM interfaces.

ATL is shipped as source code which you include in your application. ATL also makes a DLL available (atl90.dll), which contains code that can be shared across components. However, this DLL is not necessary.

See also

[Introduction to ATL](#)

[ATL Project Wizard](#)

Recommendations for Choosing Between ATL and MFC

3/4/2019 • 2 minutes to read • [Edit Online](#)

When developing components and applications, you can choose between two approaches — ATL and MFC (the Microsoft Foundation Class Library).

Using ATL

ATL is a fast, easy way to both create a COM component in C++ and maintain a small footprint. Use ATL to create a control if you don't need all of the built-in functionality that MFC automatically provides.

Using MFC

MFC allows you to create full applications, ActiveX controls, and active documents. If you have already created a control with MFC, you may want to continue development in MFC. When creating a new control, consider using ATL if you don't need all of MFC's built-in functionality.

Using ATL in an MFC Project

You can add support for using ATL in an existing MFC project by running a wizard. For details, see [Adding ATL Support to Your MFC Project](#).

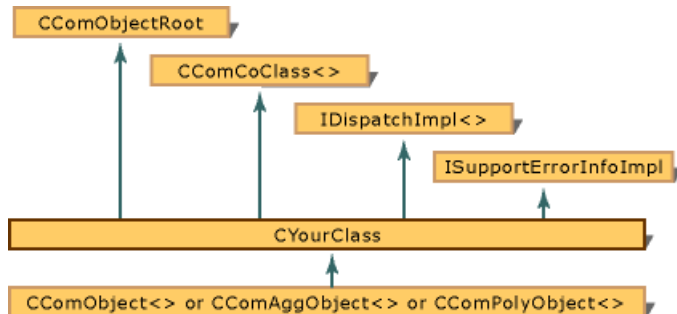
See also

[Introduction to ATL](#)

Fundamentals of ATL COM Objects

3/4/2019 • 2 minutes to read • [Edit Online](#)

The following illustration depicts the relationship among the classes and interfaces that are used to define an ATL COM object.



NOTE

This diagram shows that `CComObject` is derived from `CYourClass` whereas `CComAggObject` and `CComPolyObject` include `CYourClass` as a member variable.

There are three ways to define an ATL COM object. The standard option is to use the `CComObject` class which is derived from `CYourClass`. The second option is to create an aggregated object by using the `CComAggObject` class. The third option is to use the `CComPolyObject` class. `CComPolyObject` acts as a hybrid: it can function as a `CComObject` class or as a `CComAggObject` class, depending on how it is first created. For more information about how to use the `CComPolyObject` class, see [CComPolyObject Class](#).

When you use standard ATL COM, you use two objects: an outer object and an inner object. External clients access the functionality of the inner object through the wrapper functions that are defined in the outer object. The outer object is of type `CComObject`.

When you use an aggregated object, the outer object does not provide wrappers for the functionality of the inner object. Instead, the outer object provides a pointer that is directly accessed by external clients. In this scenario, the outer object is of type `CComAggObject`. The inner object is a member variable of the outer object, and it is of type `CYourClass`.

Because the client does not have to go through the outer object to interact with the inner object, aggregated objects are usually more efficient. Also, the outer object does not have to know the functionality of the aggregated object, given that the interface of the aggregated object is directly available to the client. However, not all objects can be aggregated. For an object to be aggregated, it needs to be designed with aggregation in mind.

ATL implements [IUnknown](#) in two phases:

- `CComObject`, `CComAggObject`, or `CComPolyObject` implements the `IUnknown` methods.
- `CComObjectRoot` or `CComObjectRootEx` manages the reference count and outer pointers of `IUnknown`.

Other aspects of your ATL COM object are handled by other classes:

- `CComCoClass` defines the object's default class factory and aggregation model.
- `IDispatchImpl` provides a default implementation of the `IDispatch` interface portion of any dual

interfaces on the object.

- [ISupportErrorInfoImpl](#) implements the `ISupportErrorInfo` interface that ensures error information can be propagated up the call chain correctly.

In This Section

[Implementing CComObjectRootEx](#)

Show example COM map entries for implementing `CComObjectRootEx`.

[Implementing CComObject, CComAggObject, and CComPolyObject](#)

Discusses how the **DECLARE_*_AGGREGATABLE** macros affect the use of `CComObject`, `CComAggObject`, and `CComPolyObject`.

[Supporting IDispatch and IErrorInfo](#)

Lists the ATL implementation classes to use for supporting the `IDispatch` and `IErrorInfo` interfaces.

[Supporting IDispEventImpl](#)

Discusses the steps to implement a connection point for your class.

[Changing the Default Class Factory and Aggregation Model](#)

Show what macros to use to change the default class factory and aggregation model.

[Creating an Aggregated Object](#)

Lists the steps for creating an aggregated object.

Related Sections

[Creating an ATL Project](#)

Provides information about creating an ATL COM object.

[ATL](#)

Provides links to conceptual topics on how to program using the Active Template Library.

See also

[Concepts](#)

Implementing CComObjectRootEx

3/12/2019 • 2 minutes to read • [Edit Online](#)

[CComObjectRootEx](#) is essential; all ATL objects must have one instance of [CComObjectRootEx](#) or [CComObjectRoot](#) in their inheritance. [CComObjectRootEx](#) provides the default [QueryInterface](#) mechanism based on COM map entries.

Through its COM map, an object's interfaces are exposed to a client when the client queries for an interface. The query is performed through [CComObjectRootEx::InternalQueryInterface](#). [InternalQueryInterface](#) only handles interfaces in the COM map table.

You can enter interfaces into the COM map table with the [COM_INTERFACE_ENTRY](#) macro or one of its variants. For example, the following code enters the interfaces [IDispatch](#), [IBeeper](#), and [ISupportErrorInfo](#) into the COM map table:

```
BEGIN_COM_MAP(CBeeper)
    COM_INTERFACE_ENTRY( IBeeper)
    COM_INTERFACE_ENTRY( IDispatch)
    COM_INTERFACE_ENTRY_TEAR_OFF( IID_ISupportErrorInfo, CBeeper2)
END_COM_MAP()
```

See also

[Fundamentals of ATL COM Objects](#)

[COM Map Macros](#)

Implementing CComObject, CComAggObject, and CComPolyObject

3/12/2019 • 2 minutes to read • [Edit Online](#)

The template classes `CComObject`, `CComAggObject`, and `CComPolyObject` are always the most derived classes in the inheritance chain. It is their responsibility to handle all of the methods in `IUnknown`: `QueryInterface`, `AddRef`, and `Release`. In addition, `CComAggObject` and `CComPolyObject` (when used for aggregated objects) provide the special reference counting and `QueryInterface` semantics required for the inner unknown.

Whether `CComObject`, `CComAggObject`, or `CComPolyObject` is used depends on whether you declare one (or none) of the following macros:

MACRO	EFFECT
<code>DECLARE_NOT_AGGREGATABLE</code>	Always uses <code>CComObject</code> .
<code>DECLARE_AGGREGATABLE</code>	Uses <code>CComAggObject</code> if the object is aggregated and <code>CComObject</code> if it is not. <code>CComCoClass</code> contains this macro so if none of the <code>DECLARE_*_AGGREGATABLE</code> macros are declared in your class, this will be the default.
<code>DECLARE_ONLY_AGGREGATABLE</code>	Always uses <code>CComAggObject</code> . Returns an error if the object is not aggregated.
<code>DECLARE_POLY_AGGREGATABLE</code>	ATL creates an instance of <code>CComPolyObject<CYourClass></code> when <code>IClassFactory::CreateInstance</code> is called. During creation, the value of the outer unknown is checked. If it is NULL, <code>IUnknown</code> is implemented for a nonaggregated object. If the outer unknown is not NULL, <code>IUnknown</code> is implemented for an aggregated object.

The advantage of using `CComAggObject` and `CComObject` is that the implementation of `IUnknown` is optimized for the kind of object being created. For instance, a nonaggregated object only needs a reference count, while an aggregated object needs both a reference count for the inner unknown and a pointer to the outer unknown.

The advantage of using `CComPolyObject` is that you avoid having both `CComAggObject` and `CComObject` in your module to handle the aggregated and nonaggregated cases. A single `CComPolyObject` object handles both cases. This means only one copy of the vtable and one copy of the functions exist in your module. If your vtable is large, this can substantially decrease your module size. However, if your vtable is small, using `CComPolyObject` can result in a slightly larger module size because it is not optimized for an aggregated or nonaggregated object, as are `CComAggObject` and `CComObject`.

See also

[Fundamentals of ATL COM Objects](#)
[Aggregation and Class Factory Macros](#)

Supporting IDispatch and IErrorInfo

3/12/2019 • 2 minutes to read • [Edit Online](#)

You can use the template class [IDispatchImpl](#) to provide a default implementation of the `IDispatch Interface` portion of any dual interfaces on your object.

If your object uses the `IErrorInfo` interface to report errors back to the client, then your object must support the `ISupportErrorInfo Interface` interface. The template class [ISupportErrorInfoImpl](#) provides an easy way to implement this if you only have a single interface that generates errors on your object.

See also

[Fundamentals of ATL COM Objects](#)

Supporting IDisEventImpl

3/12/2019 • 2 minutes to read • [Edit Online](#)

The template class `IDisEventImpl` can be used to provide support for connection point sinks in your ATL class. A connection point sink allows your class to handle events fired from external COM objects. These connection point sinks are mapped with an event sink map, provided by your class.

To properly implement a connection point sink for your class, the following steps must be completed:

- Import the type libraries for each external object
- Declare the `IDisEventImpl` interfaces
- Declare an event sink map
- Advise and unadvise the connection points

The steps involved in implementing a connection point sink are all accomplished by modifying only the header file (.h) of your class.

Importing the Type Libraries

For each external object whose events you want to handle, you must import the type library. This step defines the events that can be handled and provides information that is used when declaring the event sink map. The `#import` directive can be used to accomplish this. Add the necessary `#import` directive lines for each dispatch interface you will support to the header file (.h) of your class.

The following example imports the type library of an external COM server (`MSCAL.Calendar.7`):

```
#import "PROGID:MSCAL.Calendar.7" no_namespace, raw_interfaces_only
```

NOTE

You must have a separate `#import` statement for each external type library you will support.

Declaring the IDisEventImpl Interfaces

Now that you have imported the type libraries of each dispatch interface, you need to declare separate `IDisEventImpl` interfaces for each external dispatch interface. Modify the declaration of your class by adding an `IDisEventImpl` interface declaration for each external object. For more information on the parameters, see [IDisEventImpl](#).

The following code declares two connection point sinks, for the `DCalendarEvents` interface, for the COM object implemented by class `CMyCompositCtrl2`:

```
public IDisEventImpl<IDC_CALENDAR1, CMyCompositCtrl2, &__uuidof(DCalendarEvents), &__uuidof(__MSACAL), 7, 0>,
public IDisEventImpl<IDC_CALENDAR2, CMyCompositCtrl2, &__uuidof(DCalendarEvents), &__uuidof(__MSACAL), 7, 0>
```

Declaring an Event Sink Map

In order for the event notifications to be handled by the proper function, your class must route each event to its correct handler. This is achieved by declaring an event sink map.

ATL provides several macros, [BEGIN_SINK_MAP](#), [END_SINK_MAP](#), and [SINK_ENTRY_EX](#), that make this mapping easier. The standard format is as follows:

```
BEGIN_SINK_MAP(comClass)
    SINK_ENTRY_EX(id, iid, dispid, func)
    . . . //additional external event entries
END_SINK_MAP()
```

The following example declares an event sink map with two event handlers:

```
BEGIN_SINK_MAP(CMyCompositCtrl2)
    //Make sure the Event Handlers have __stdcall calling convention
    SINK_ENTRY_EX(IDC_CALENDAR1, __uuidof(DCalendarEvents), DISPID_CLICK,
        &CMyCompositCtrl2::ClickCalendar1)
    SINK_ENTRY_EX(IDC_CALENDAR2, __uuidof(DCalendarEvents), DISPID_CLICK,
        &CMyCompositCtrl2::ClickCalendar2)
END_SINK_MAP()
```

The implementation is nearly complete. The last step concerns the advising and unadvising of the external interfaces.

Advising and Unadvising the IDispatchImpl Interfaces

The final step is to implement a method that will advise (or unadvise) all connection points at the proper times. This advising must be done before communication between the external clients and your object can take place. Before your object becomes visible, each external dispatch interface supported by your object is queried for outgoing interfaces. A connection is established and a reference to the outgoing interface is used to handle events from the object. This procedure is referred to as "advising."

After your object is finished with the external interfaces, the outgoing interfaces should be notified that they are no longer used by your class. This process is referred to as "unadvising."

Because of the unique nature of COM objects, this procedure varies, in detail and execution, between implementations. These details are beyond the scope of this topic and are not addressed.

See also

[Fundamentals of ATL COM Objects](#)

Changing the Default Class Factory and Aggregation Model

3/4/2019 • 2 minutes to read • [Edit Online](#)

ATL uses [CCoClass](#) to define the default class factory and aggregation model for your object. `CCoClass` specifies the following two macros:

- [DECLARE_CLASSFACTORY](#) Declares the class factory to be [CCoClassFactory](#).
- [DECLARE_AGGREGATABLE](#) Declares that your object can be aggregated.

You can override either of these defaults by specifying another macro in your class definition. For example, to use [CCoClassFactory2](#) instead of `CCoClassFactory`, specify the [DECLARE_CLASSFACTORY2](#) macro:

```
class ATL_NO_VTABLE CMyClass2 :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CCoClass<CMyClass2, &CLSID_MyClass>,
    public IDispatchImpl<IMyClass, &IID_IMyClass, &LIBID_NVC_ATL_COMLib, /*wMajor =*/ 1, /*wMinor =*/ 0>,
    public IDispatchImpl<IMyDualInterface, &__uuidof(IMyDualInterface), &LIBID_NVC_ATL_COMLib, /* wMajor = */
1, /* wMinor = */ 0>
{
public:
    DECLARE_CLASSFACTORY2(CMyLicense)

    // Remainder of class declaration omitted
```

Two other macros that define a class factory are [DECLARE_CLASSFACTORY_AUTO_THREAD](#) and [DECLARE_CLASSFACTORY_SINGLETON](#).

ATL also uses the **typedef** mechanism to implement default behavior. For example, the [DECLARE_AGGREGATABLE](#) macro uses **typedef** to define a type called `_CreatorClass`, which is then referenced throughout ATL. Note that in a derived class, a **typedef** using the same name as the base class's **typedef** results in ATL using your definition and overriding the default behavior.

See also

[Fundamentals of ATL COM Objects](#)

[Aggregation and Class Factory Macros](#)

Creating an Aggregated Object

3/4/2019 • 2 minutes to read • [Edit Online](#)

Aggregation delegates `IUnknown` calls, providing a pointer to the outer object's `IUnknown` to the inner object.

To create an aggregated object

1. Add an `IUnknown` pointer to your class object and initialize it to NULL in the constructor.
2. Override `FinalConstruct` to create the aggregate.
3. Use the `IUnknown` pointer, defined in Step 1, as the second parameter for the `COM_INTERFACE_ENTRY_AGGREGATE` macros.
4. Override `FinalRelease` to release the `IUnknown` pointer.

NOTE

If you use and release an interface from the aggregated object during `FinalConstruct`, you should add the `DECLARE_PROTECT_FINAL_CONSTRUCT` macro to the definition of your class object.

See also

[Fundamentals of ATL COM Objects](#)

Dual Interfaces and ATL

3/4/2019 • 2 minutes to read • [Edit Online](#)

A dual interface allows its methods to be accessed as dispinterface methods or as vtable methods. This section covers some of the features of dual interfaces from an ATL perspective.

In This Section

[Implementing a Dual Interface](#)

Discusses the classes and wizards involved in implementing a dual interface.

[Multiple Dual Interfaces](#)

Discusses how to expose multiple dual interfaces on a single object.

[The nonextensible Attribute](#)

Discusses when to use the **nonextensible** attribute on your interface definition.

[Dual Interfaces and Events](#)

Discusses design reasons for not making an event interface a dual interface.

Related Sections

[ATL](#)

Provides links to conceptual topics on how to program using the Active Template Library.

See also

[Concepts](#)

Implementing a Dual Interface

3/4/2019 • 2 minutes to read • [Edit Online](#)

You can implement a dual interface using the [IDispatchImpl](#) class, which provides a default implementation of the `IDispatch` methods in a dual interface. For more information, see [Implementing the IDispatch Interface](#).

To use this class:

- Define your dual interface in a type library.
- Derive your class from a specialization of `IDispatchImpl` (pass information about the interface and type library as the template arguments).
- Add an entry (or entries) to the COM map to expose the dual interface through `QueryInterface`.
- Implement the vtable part of the interface in your class.
- Ensure that the type library containing the interface definition is available to your objects at run time.

ATL Simple Object Wizard

If you want to create a new interface and a new class to implement it, you can use the [ATL Add Class dialog box](#), and then the [ATL Simple Object Wizard](#).

Implement Interface Wizard

If you have an existing interface, you can use the [Implement Interface Wizard](#) to add the necessary base class, COM map entries, and skeleton method implementations to an existing class.

NOTE

You may need to adjust the generated base class so that the major and minor version numbers of the type library are passed as template arguments to your `IDispatchImpl` base class. The Implement Interface Wizard doesn't check the type library version number for you.

Implementing IDispatch

You can use an `IDispatchImpl` base class to provide an implementation of a dispinterface just by specifying the appropriate entry in the COM map (using the [COM_INTERFACE_ENTRY2](#) or [COM_INTERFACE_ENTRY_IID](#) macro) as long as you have a type library describing a corresponding dual interface. It is quite common to implement the `IDispatch` interface this way, for example. The ATL Simple Object Wizard and Implement Interface Wizard both assume that you intend to implement `IDispatch` in this way, so they will add the appropriate entry to the map.

NOTE

ATL offers the [IDispEventImpl](#) and [IDispEventSimpleImpl](#) classes to help you implement dispinterfaces without requiring a type library containing the definition of a compatible dual interface.

See also

Multiple Dual Interfaces

3/4/2019 • 2 minutes to read • [Edit Online](#)

You may want to combine the advantages of a dual interface (that is, the flexibility of both vtable and late binding, thus making the class available to scripting languages as well as C++) with the techniques of multiple inheritance.

Although it is possible to expose multiple dual interfaces on a single COM object, it is not recommended. If there are multiple dual interfaces, there must be only one `IDispatch` interface exposed. The techniques available to ensure that this is the case carry penalties such as loss of function or increased code complexity. The developer considering this approach should carefully weigh the advantages and disadvantages.

Exposing a Single IDispatch Interface

It is possible to expose multiple dual interfaces on a single object by deriving from two or more specializations of `IDispatchImpl`. However, if you allow clients to query for the `IDispatch` interface, you will need to use the `COM_INTERFACE_ENTRY2` macro (or `COM_INTERFACE_ENTRY_IID`) to specify which base class to use for the implementation of `IDispatch`.

```
COM_INTERFACE_ENTRY2(IDispatch, IMyDualInterface)
```

Because only one `IDispatch` interface is exposed, clients that can only access your objects through the `IDispatch` interface will not be able to access the methods or properties in any other interface.

Combining Multiple Dual Interfaces into a Single Implementation of IDispatch

ATL does not provide any support for combining multiple dual interfaces into a single implementation of `IDispatch`. However, there are several known approaches to manually combining the interfaces, such as creating a templated class that contains a union of the separate `IDispatch` interfaces, creating a new object to perform the `QueryInterface` function, or using a typeinfo-based implementation of nested objects to create the `IDispatch` interface.

These approaches have problems with potential namespace collisions, as well as code complexity and maintainability. It is not recommended that you create multiple dual interfaces.

See also

[Dual Interfaces and ATL](#)

nonextensible Attribute

3/12/2019 • 2 minutes to read • [Edit Online](#)

If a dual interface will not be extended at run time (that is, you won't provide methods or properties via `IDispatch::Invoke` that are not available via the vtable), you should apply the **nonextensible** attribute to your interface definition. This attribute provides information to client languages (such as Visual Basic) that can be used to enable full code verification at compile time. If this attribute is not supplied, bugs may remain hidden in the client code until run time.

For more information on the **nonextensible** attribute and an example, see [nonextensible](#).

See also

[Dual Interfaces and ATL](#)

Dual Interfaces and Events

3/4/2019 • 2 minutes to read • [Edit Online](#)

While it is possible to design an event interface as a dual, there are a number of good design reasons not to do so. The fundamental reason is that the source of the event will only fire the event via the vtable or via `Invoke`, not both. If the event source fires the event as a direct vtable method call, the `IDispatch` methods will never be used and it's clear that the interface should have been a pure vtable interface. If the event source fires the event as a call to `Invoke`, the vtable methods will never be used and it's clear that the interface should have been a dispinterface. If you define your event interfaces as duals, you'll be requiring clients to implement part of an interface that will never be used.

NOTE

This argument does not apply to dual interfaces, in general. From an implementation perspective, duals are a quick, convenient, and well-supported way of implementing interfaces that are accessible to a wide range of clients.

There are further reasons to avoid dual event interfaces; neither Visual Basic nor Internet Explorer support them.

See also

[Dual Interfaces and ATL](#)

ATL Collections and Enumerators

4/1/2019 • 2 minutes to read • [Edit Online](#)

A `collection` is a COM object that provides an interface that allows access to a group of data items (raw data or other objects). An interface that follows the standards for providing access to a group of objects is known as a *collection interface*.

At a minimum, collection interfaces must provide a `Count` property that returns the number of items in the collection, an `Item` property that returns an item from the collection based on an index, and a `_NewEnum` property that returns an enumerator for the collection. Optionally, collection interfaces can provide `Add` and `Remove` methods to allow items to be inserted into or deleted from the collection, and a `Clear` method to remove all items.

An `enumerator` is a COM object that provides an interface for iterating through items in a collection. Enumerator interfaces provide serial access to the elements of a collection via four required methods: `Next`, `Skip`, `Reset`, and `Clone`.

You can learn more about enumerator interfaces by reading reference content such as [IEnumString](#) interface.

In This Section

[ATL Collection and Enumerator Classes](#)

Briefly describes and provides links to the ATL classes that will help you implement collections and enumerators.

[Design Principles for Collection and Enumerator Interfaces](#)

Discusses the different design principles behind each type of interface.

[Implementing a C++ Standard Library-Based Collection](#)

An extended example that walks you through the implementation of a C++ Standard Library-based collection.

Related Sections

[ATL](#)

Provides links to conceptual topics on how to program using the Active Template Library.

[ATLCollections Sample](#)

A sample that demonstrates the use of `ICollectionOnSTLImpl` and `CComEnumOnSTL`, and the implementation of custom copy policy classes.

See also

[Concepts](#)

ATL Collection and Enumerator Classes

3/4/2019 • 2 minutes to read • [Edit Online](#)

ATL provides the following classes to help you implement collections and enumerators.

CLASS	DESCRIPTION
ICollectionOnSTLImpl	Collection interface implementation
IEnumOnSTLImpl	Enumerator interface implementation (assumes data stored in a C++ Standard Library-compatible container)
CComEnumImpl	Enumerator interface implementation (assumes data stored in an array)
CComEnumOnSTL	Enumerator object implementation (uses <code>IEnumOnSTLImpl</code>)
CComEnum	Enumerator object implementation (uses <code>CComEnumImpl</code>)
_Copy	Copy policy class
_CopyInterface	Copy policy class
CAdapt	Adapter class (hides operator & allowing <code>CComPtr</code> , <code>CComQIPtr</code> , and <code>CComBSTR</code> to be stored in C++ Standard Library containers)

See also

[Collections and Enumerators](#)

Design Principles for Collection and Enumerator Interfaces

3/4/2019 • 2 minutes to read • [Edit Online](#)

There are different design principles behind each type of interface:

- A collection interface provides *random* access to a *single* item in the collection via the `Item` method, it lets clients discover how many items are in the collection via the `Count` property, and often allows clients to add and remove items.
- An enumerator interface provides *serial* access to *multiple* items in a collection, it doesn't allow the client to discover how many items are in the collection (until the enumerator stops returning items), and it doesn't provide any way of adding or removing items.

Each type of interface plays a different role in providing access to the elements in a collection.

See also

[Collections and Enumerators](#)

Implementing a C++ Standard Library-Based Collection

4/1/2019 • 5 minutes to read • [Edit Online](#)

ATL provides the `ICollectionOnSTLImpl` interface to enable you to quickly implement C++ Standard Library-based collection interfaces on your objects. To understand how this class works, you will work through a simple example (below) that uses this class to implement a read-only collection aimed at Automation clients.

The sample code is from the [ATLCollections sample](#).

To complete this procedure, you will:

- [Generate a new Simple Object](#).
- [Edit the IDL file](#) for the generated interface.
- [Create five typedefs](#) describing how the collection items are stored and how they will be exposed to clients via COM interfaces.
- [Create two typedefs for copy policy classes](#).
- [Create typedefs for the enumerator and collection implementations](#).
- [Edit the wizard-generated C++ code to use the collection typedef](#).
- [Add code to populate the collection](#).

Generating a New Simple Object

Create a new project, ensuring that the Attributes box under Application Settings is cleared. Use the ATL Add Class dialog box and Add Simple Object Wizard to generate a Simple Object called `Words`. Make sure that a dual interface called `IWords` is generated. Objects of the generated class will be used to represent a collection of words (that is, strings).

Editing the IDL File

Now, open the IDL file and add the three properties necessary to turn `IWords` into a read-only collection interface, as shown below:

```

[
    object,
    uuid(7B3AC376-509F-4068-87BA-03B73ADC359B),
    dual,                                     // (1)
    nonextensible,                           // (2)
    pointer_default(unique)
]
interface IWords : IDispatch
{
    [id(DISPID_NEWENUM), propget]           // (3)
    HRESULT _NewEnum([out, retval] IUnknown** ppUnk);

    [id(DISPID_VALUE), propget]             // (4)
    HRESULT Item([in] long Index, [out, retval] BSTR* pVal); // (5)

    [id(0x00000001), propget]              // (6)
    HRESULT Count([out, retval] long* pVal);

};

```

This is the standard form for a read-only collection interface designed with Automation clients in mind. The numbered comments in this interface definition correspond to the comments below:

1. Collection interfaces are usually dual because Automation clients accesses the `_NewEnum` property via `IDispatch::Invoke`. However, Automation clients can access the remaining methods via the vtable, so dual interfaces are preferable to dispinterfaces.
2. If a dual interface or dispinterface will not be extended at run time (that is, you won't provide extra methods or properties via `IDispatch::Invoke`), you should apply the **nonextensible** attribute to your definition. This attribute enables Automation clients to perform full code verification at compile time. In this case, the interface should not be extended.
3. The correct DISPID is important if you want Automation clients to be able to use this property. (Note that there is only one underscore in `DISPID_NEWENUM`.)
4. You can supply any value as the DISPID of the `Item` property. However, `Item` typically uses `DISPID_VALUE` to make it the default property of the collection. This allows Automation clients to refer to the property without naming it explicitly.
5. The data type used for the return value of the `Item` property is the type of the item stored in the collection as far as COM clients are concerned. The interface returns strings, so you should use the standard COM string type, `BSTR`. You can store the data in a different format internally as you'll see shortly.
6. The value used for the DISPID of the `Count` property is completely arbitrary. There's no standard DISPID for this property.

Creating Typedefs for Storage and Exposure

Once the collection interface is defined, you need to decide how the data will be stored, and how the data will be exposed via the enumerator.

The answers to these questions can be provided in the form of a number of typedefs, which you can add near the top of the header file for your newly created class:

```
// Store the data in a vector of std::strings
typedef std::vector< std::string >      ContainerType;

// The collection interface exposes the data as BSTRs
typedef BSTR                          CollectionExposedType;
typedef IWords                        CollectionInterface;

// Use IEnumVARIANT as the enumerator for VB compatibility
typedef VARIANT                      EnumeratorExposedType;
typedef IEnumVARIANT                  EnumeratorInterface;
```

In this case, you will store the data as a **std::vector** of **std::strings**. **std::vector** is a C++ Standard Library container class that behaves like a managed array. **std::string** is the C++ Standard Library's string class. These classes make it easy to work with a collection of strings.

Since Visual Basic support is vital to the success of this interface, the enumerator returned by the `_NewEnum` property must support the `IEnumVARIANT` interface. This is the only enumerator interface understood by Visual Basic.

Creating Typedefs for Copy Policy Classes

The typedefs you have created so far provide all the information you need to create further typedefs for the copy classes that will be used by the enumerator and collection:

```
// Typedef the copy classes using existing typedefs
typedef VCUE::GenericCopy<EnumeratorExposedType, ContainerType::value_type> EnumeratorCopyType;
typedef VCUE::GenericCopy<CollectionExposedType, ContainerType::value_type> CollectionCopyType;
```

In this example, you can use the custom `GenericCopy` class defined in `VCUE_Copy.h` and `VCUE_CopyString.h` from the [ATL Collections](#) sample. You can use this class in other code, but you may need to define further specializations of `GenericCopy` to support data types used in your own collections. For more information, see [ATL Copy Policy Classes](#).

Creating Typedefs for Enumeration and Collection

Now all the template parameters necessary to specialize the `CCoEnumOnSTL` and `ICollectionOnSTLImpl` classes for this situation have been provided in the form of typedefs. To simplify the use of the specializations, create two more typedefs as shown below:

```
typedef CCoEnumOnSTL< EnumeratorInterface, &__uuidof(EnumeratorInterface), EnumeratorExposedType,
EnumeratorCopyType, ContainerType > EnumeratorType;
typedef ICollectionOnSTLImpl< CollectionInterface, ContainerType, CollectionExposedType, CollectionCopyType,
EnumeratorType > CollectionType;
```

Now `CollectionType` is a synonym for a specialization of `ICollectionOnSTLImpl` that implements the `IWords` interface defined earlier and provides an enumerator that supports `IEnumVARIANT`.

Editing the Wizard-Generated Code

Now you must derive `CWords` from the interface implementation represented by the `CollectionType` typedef rather than `IWords`, as shown below:

```

class ATL_NO_VTABLE CWords :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CWords, &CLSID_Words>,
    // 'CollectionType' replaces 'IWords' in next line
    public IDispatchImpl<CollectionType, &IID_IWords, &LIBID_NVC_ATL_COMLib, /*wMajor */ 1, /*wMinor */ 0>
{
public:
    DECLARE_REGISTRY_RESOURCEID(IDR_WORDS)

    BEGIN_COM_MAP(CWords)
        COM_INTERFACE_ENTRY(IWords)
        COM_INTERFACE_ENTRY(IDispatch)
    END_COM_MAP()

    // Remainder of class declaration omitted.

```

Adding Code to Populate the Collection

The only thing that remains is to populate the vector with data. In this simple example, you can add a few words to the collection in the constructor for the class:

```

CWords()
{
    m_coll.push_back("this");
    m_coll.push_back("is");
    m_coll.push_back("a");
    m_coll.push_back("test");
}

```

Now, you can test the code with the client of your choice.

See also

[Collections and Enumerators](#)

[ATLCollections Sample](#)

[ATL Copy Policy Classes](#)

ATL Copy Policy Classes

4/1/2019 • 3 minutes to read • [Edit Online](#)

Copy policy classes are [utility classes](#) used to initialize, copy, and delete data. Copy policy classes allow you to define copy semantics for any type of data, and to define conversions between different data types.

ATL uses copy policy classes in its implementations of the following templates:

- [CComEnumImpl](#)
- [IEnumOnSTLImpl](#)
- [ICollectionOnSTLImpl](#)

By encapsulating the information needed to copy or convert data in a copy policy class that can be passed as a template argument, the ATL developers have provided for extreme reusability of these classes. For example, if you need to implement a collection using any arbitrary data type, all you need to provide is the appropriate copy policy; you never have to touch the code that implements the collection.

Definition

By definition, a class that provides the following static functions is a copy policy class:

```
static void init( DestinationType * p );
```

```
static HRESULT copy( DestinationType * pTo, const SourceType * pFrom );
```

```
static void destroy( DestinationType * p );
```

You can replace the types `DestinationType` and `SourceType` with arbitrary data types for each copy policy.

NOTE

Although you can define copy policy classes for any arbitrary data types, use of the classes in ATL code should limit the types that make sense. For example, when using a copy policy class with ATL's collection or enumerator implementations, `DestinationType` must be a type that can be used as a parameter in a COM interface method.

Use **init** to initialize data, **copy** to copy data, and **destroy** to free the data. The precise meaning of initialization, copying, and destruction are the domain of the copy policy class and will vary depending on the data types involved.

There are two requirements on the use and implementation of a copy policy class:

- The first parameter to **copy** must only receive a pointer to data that you have previously initialized using **init**.
- **destroy** must only ever receive a pointer to data that you have previously initialized using **init** or copied via **copy**.

Standard Implementations

ATL provides two copy policy classes in the form of the `_Copy` and `_CopyInterface` template classes:

- The `_Copy` class allows homogeneous copying only (not conversion between data types) since it only

offers a single template parameter to specify both `DestinationType` and `SourceType`. The generic implementation of this template contains no initialization or destruction code and uses `memcpy` to copy the data. ATL also provides specializations of `_Copy` for `VARIANT`, `LPOLESTR`, `OLEVERB`, and `CONNECTDATA` data types.

- The `_CopyInterface` class provides an implementation for copying interface pointers following standard COM rules. Once again this class allows only homogeneous copying, so it uses simple assignment and a call to `AddRef` to perform the copy.

Custom Implementations

Typically, you'll need to define your own copy policy classes for heterogeneous copying (that is, conversion between data types). For some examples of custom copy policy classes, look at the files `VCUE_Copy.h` and `VCUE_CopyString.h` in the [ATLCollections](#) sample. These files contain two template copy policy classes, `GenericCopy` and `MapCopy`, plus a number of specializations of `GenericCopy` for different data types.

GenericCopy

`GenericCopy` allows you to specify the `SourceType` and `DestinationType` as template arguments. Here's the most general form of the `GenericCopy` class from `VCUE_Copy.h`:

```
template <class DestinationType, class SourceType = DestinationType>
class GenericCopy
{
public :
    typedef DestinationType destination_type;
    typedef SourceType source_type;

    static void init(destination_type* p)
    {
        _Copy<destination_type>::init(p);
    }
    static void destroy(destination_type* p)
    {
        _Copy<destination_type>::destroy(p);
    }
    static HRESULT copy(destination_type* pTo, const source_type* pFrom)
    {
        return _Copy<destination_type>::copy(pTo, const_cast<source_type*>(pFrom));
    }
}; // class GenericCopy
```

`VCUE_Copy.h` also contains the following specializations of this class: `GenericCopy<BSTR>`, `GenericCopy<VARIANT, BSTR>`, `GenericCopy<BSTR, VARIANT>`. `VCUE_CopyString.h` contains specializations for copying from **std::strings**: `GenericCopy<std::string>`, `GenericCopy<VARIANT, std::string>`, and `GenericCopy<BSTR, std::string>`. You could enhance `GenericCopy` by providing further specializations of your own.

MapCopy

`MapCopy` assumes that the data being copied is stored into a C++ Standard Library-style map, so it allows you to specify the type of map in which the data is stored and the destination type. The implementation of the class just uses the typedefs supplied by the `MapType` class to determine the type of the source data and to call the appropriate `GenericCopy` class. No specializations of this class are needed.

```

template <class MapType, class DestinationType = MapType::mapped_type>
class MapCopy
{
public :
    typedef DestinationType          destination_type;
    typedef typename MapType::value_type  source_type;

    typedef MapType                  map_type;
    typedef typename MapType::mapped_type pseudosource_type;

    static void init(destination_type* p)
    {
        GenericCopy<destination_type, pseudosource_type>::init(p);
    }
    static void destroy(destination_type* p)
    {
        GenericCopy<destination_type, pseudosource_type>::destroy(p);
    }
    static HRESULT copy(destination_type* pTo, const source_type* pFrom)
    {
        return GenericCopy<destination_type, pseudosource_type>::copy(pTo, &(pFrom->second));
    }

}; // class MapCopy

```

See also

[Implementing a C++ Standard Library-Based Collection](#)

[ATLCollections Sample](#)

ATL Composite Control Fundamentals

3/4/2019 • 2 minutes to read • [Edit Online](#)

A composite control is a type of ActiveX control that can contain (similar to a dialog box) other ActiveX controls or Windows controls. Once the composite control is built, it can be inserted anywhere an ActiveX control can be hosted.

The ATL Project Wizard and **Add Class** dialog box automate the process of creating and implementing a composite control project, similar to the result of running the Application Wizard to create an MFC application framework. The development process consists of five steps:

- [Creating an ATL project](#)
- [Inserting a composite control](#)
- [Modifying the ATL project](#)
- [Adding functionality to the composite control](#)
- [Building and testing the ATL project](#)

See also

[Concepts](#)

[Composite Control Global Functions](#)

[Composite Control Macros](#)

Inserting a Composite Control

3/4/2019 • 2 minutes to read • [Edit Online](#)

The **Add Class** dialog box allows you to insert an ATL object into a project. Access this dialog box by right-clicking the project name in Solution Explorer, pointing to **Add**, and then clicking **Add Class**.

In the **Add Class** dialog box, choose **ATL Control**. This will start the [ATL Control Wizard](#). To create a composite control, select the **Options** tab, and click the **Composite control** check box.

A default HTML page will be created for viewing the control.

See also

[Composite Control Fundamentals](#)

Modifying the ATL Project

3/4/2019 • 2 minutes to read • [Edit Online](#)

At this point, your composite control project implements the necessary objects for your composite control. The next step is to add any controls that the composite control will contain and handle any necessary events.

To add additional ActiveX or Windows controls, add a new resource script and then use the Dialog editor. For more information on adding controls (and related tasks), see [Dialog Editor](#).

To handle any necessary events from the ActiveX controls, see [Adding Functionality to the Composite Control](#).

See also

[Composite Control Fundamentals](#)

[How to: Create a Resource Script File](#)

Adding Functionality to the Composite Control

3/4/2019 • 2 minutes to read • [Edit Online](#)

Once you have inserted any necessary controls into the composite control, the next step involves adding new functionality. This new functionality usually falls into two categories:

- Supporting additional interfaces and customizing the behavior of your composite control with additional, specific features.
- Handling events from the contained ActiveX control (or controls).

For the purpose and scope of this article, the remainder of this section focuses solely on handling events from ActiveX controls.

NOTE

If you need to handle messages from Windows controls, see [Implementing a Window](#) for more information on message handling in ATL.

After inserting an ActiveX control in the dialog resource, right-click the control and click **Add Event Handler**. Select the event you want to handle and click **Add and Edit**. The event handler code will be added to the control's .h file.

Connection points for ActiveX controls on the composite control are automatically connected and disconnected via calls to [CComCompositeControl::AdviseSinkMap](#).

See also

[Composite Control Fundamentals](#)

Building and Testing the ATL Project

3/4/2019 • 2 minutes to read • [Edit Online](#)

As mentioned in [Inserting a Composite Control](#), one of the initial components of the project is a default HTML page that hosts your new composite control. After you finish modifying the composite control, click **Build Solution** or **Rebuild Solution** from the **Build** menu. Once the project successfully builds, load the HTML page, located in the root directory of your composite control project, into Internet Explorer or another browser and test the functionality of your control.

You can also test your composite control using the Test Container tool, or any other application that can host an ActiveX control. See [Testing Properties and Events with Test Container](#) for information on how to access the test container.

See also

[Composite Control Fundamentals](#)

ATL Control Containment FAQ

3/4/2019 • 9 minutes to read • [Edit Online](#)

Which ATL Classes Facilitate ActiveX Control Containment?

ATL's control-hosting code doesn't require you to use any ATL classes; you can simply create an **"AtlAxWin80"** window and use the control-hosting API if necessary (for more information, see **What Is the ATL Control-Hosting API**). However, the following classes make the containment features easier to use.

CLASS	DESCRIPTION
CAxWindow	Wraps an "AtlAxWin80" window, providing methods for creating the window, creating a control and/or attaching a control to the window, and retrieving interface pointers on the host object.
CAxWindow2T	Wraps an "AtlAxWinLic80" window, providing methods for creating the window, creating a control and/or attaching a licensed control to the window, and retrieving interface pointers on the host object.
CComCompositeControl	Acts as a base class for ActiveX control classes based on a dialog resource. Such controls can contain other ActiveX controls.
CAxDialogImpl	Acts as a base class for dialog classes based on a dialog resource. Such dialogs can contain ActiveX controls.
CWindow	Provides a method, GetDlgControl , that will return an interface pointer on a control, given the ID of its host window. In addition, the Windows API wrappers exposed by CWindow generally make window management easier.

What Is the ATL Control-Hosting API?

ATL's control-hosting API is the set of functions that allows any window to act as an ActiveX control container. These functions can be statically or dynamically linked into your project since they are available as source code and exposed by ATL90.dll. The control-hosting functions are listed in the table below.

FUNCTION	DESCRIPTION
AtlAxAttachControl	Creates a host object, connects it to the supplied window, then attaches an existing control.
AtlAxCreateControl	Creates a host object, connects it to the supplied window, then loads a control.
AtlAxCreateControlLic	Creates a licensed ActiveX control, initializes it, and hosts it in the specified window, similar to AtlAxCreateControl .
AtlAxCreateControlEx	Creates a host object, connects it to the supplied window, then loads a control (also allows event sinks to be set up).

FUNCTION	DESCRIPTION

AtlAxCreateControlLicEx	Creates a licensed ActiveX control, initializes it, and hosts it in the specified window, similar to AtlAxCreateControlLic .
AtlAxCreateDialog	Creates a modeless dialog box from a dialog resource and returns the window handle.
AtlAxDialogBox	Creates a modal dialog box from a dialog resource.
AtlAxGetControl	Returns the IUnknown interface pointer of the control hosted in a window.
AtlAxGetHost	Returns the IUnknown interface pointer of the host object connected to a window.
AtlAxWinInit	Initializes the control-hosting code.
AtlAxWinTerm	Uninitializes the control-hosting code.

The `HWND` parameters in the first three functions must be an existing window of (almost) any type. If you call any of these three functions explicitly (typically, you won't have to), do not pass a handle to a window that's already acting as a host (if you do, the existing host object won't be freed).

The first seven functions call [AtlAxWinInit](#) implicitly.

NOTE

The control-hosting API forms the foundation of ATL's support for ActiveX control containment. However, there is usually little need to call these functions directly if you take advantage of or make full use of ATL's wrapper classes. For more information, see [Which ATL Classes Facilitate ActiveX Control Containment](#).

What Is AtlAxWin100?

`AtlAxWin100` is the name of a window class that helps provide ATL's control-hosting functionality. When you create an instance of this class, the window procedure will automatically use the control-hosting API to create a host object associated with the window and load it with the control that you specify as the title of the window.

When Do I Need to Call AtlAxWinInit?

[AtlAxWinInit](#) registers the "**AtlAxWin80**" window class (plus a couple of custom window messages) so this function must be called before you try to create a host window. However, you don't always need to call this function explicitly, since the hosting APIs (and the classes that use them) often call this function for you. There is no harm in calling this function more than once.

What Is a Host Object?

A host object is a COM object that represents the ActiveX control container supplied by ATL for a particular window. The host object subclasses the container window so that it can reflect messages to the control, it provides the necessary container interfaces to be used by the control, and it exposes the [IAxWinHostWindow](#) and [IAxWinAmbientDispatch](#) interfaces to allow you to configure the environment of the control.

You can use the host object to set the ambient properties of the container.

Can I Host More Than One Control in a Single Window?

It is not possible to host more than one control in a single ATL host window. Each host window is designed to hold exactly one control at a time (this allows for a simple mechanism for handling message reflection and per-control ambient properties). However, if you need the user to see multiple controls in a single window, it's a simple matter to create multiple host windows as children of that window.

Can I Reuse a Host Window?

It is not recommended that you reuse host windows. To ensure the robustness of your code, you should tie the lifetime of your host window to the lifetime of a single control.

When Do I Need to Call AtlAxWinTerm?

[AtlAxWinTerm](#) unregisters the "**AtlAxWin80**" window class. You should call this function (if you no longer need to create host windows) after all existing host windows have been destroyed. If you don't call this function, the window class will be unregistered automatically when the process terminates.

Hosting ActiveX Controls Using ATL AXHost

The sample in this section shows how to create AXHost and how to host an ActiveX control using various ATL functions. It also shows how to access the control and sink events (using [IDispEventImpl](#)) from the control that is hosted. The sample hosts the Calendar control in a main window or in a child window.

Notice the definition of the `USE_METHOD` symbol. You can change the value of this symbol to vary between 1 and 8. The value of the symbol determines how the control will be created:

- For even-numbered values of `USE_METHOD`, the call to create the host subclasses a window and converts it into a control host. For odd-numbered values, the code creates a child window that acts as a host.
- For values of `USE_METHOD` between 1 and 4, access to the control and sinking of events are accomplished in the call that also creates the host. Values between 5 and 8 query the host for interfaces and hook the sink.

Here's a summary:

USE_METHOD	HOST	CONTROL ACCESS AND EVENT SINKING	FUNCTION DEMONSTRATED
1	Child window	One step	CreateControlLicEx
2	Main window	One step	AtlAxCreateControlLicEx
3	Child window	One step	CreateControlEx
4	Main window	One step	AtlAxCreateControlEx
5	Child window	Multiple steps	CreateControlLic
6	Main window	Multiple steps	AtlAxCreateControlLic
7	Child window	Multiple steps	CreateControl
8	Main window	Multiple steps	AtlAxCreateControl

```

// Your project must be apartment threaded or the (AtlAx)CreateControl(Lic)(Ex)
// calls will fail.
#define _ATL_APARTMENT_THREADED
#include <atlbase.h>
#include <atlwin.h>
#include <atlhost.h>

// Value of g_UseMethod determines the function used to create the control.
int g_UseMethod = 0; // 1 to 8 are valid values
bool ValidateUseMethod() { return (1 <= g_UseMethod) && (g_UseMethod <= 8); }

#import "PROGID:MSCAL.Calendar.7" no_namespace, raw_interfaces_only

// Child window class that will be subclassed for hosting Active X control
class CChildWindow : public CWindowImpl<CChildWindow>
{
public:
    BEGIN_MSG_MAP(CChildWindow)
        END_MSG_MAP()
};

class CMainWindow : public CWindowImpl<CMainWindow, CWindow, CFrameWinTraits>,
    public IDispatchImpl<1, CMainWindow, &__uuidof(DCalendarEvents), &__uuidof(__MSACAL), 7, 0>
{
public :

    CChildWindow m_wndChild;
    CAxWindow2 m_axwnd;
    CWindow m_wndEdit;

    static ATL::CWndClassInfo& GetWndClassInfo()
    {
        static ATL::CWndClassInfo wc =
        {
            {
                sizeof(WNDCLASSEX),
                CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS,
                StartWindowProc,
                0, 0, NULL, NULL, NULL,
                (HBRUSH)(COLOR_WINDOW + 1),
                0,
                _T("MainWindow"),
                NULL
            },
            NULL, NULL, IDC_ARROW, TRUE, 0, _T("")
        };
        return wc;
    }

    BEGIN_MSG_MAP(CMainWindow)
        MESSAGE_HANDLER(WM_CREATE, OnCreate)
        MESSAGE_HANDLER(WM_DESTROY, OnDestroy)
    END_MSG_MAP()

    BEGIN_SINK_MAP(CMainWindow)
        SINK_ENTRY_EX(1, __uuidof(DCalendarEvents), DISPID_CLICK, OnClick)
    END_SINK_MAP()

    // Helper to display events
    void DisplayNotification(TCHAR* pszMessage)
    {
        CWindow wnd;
        wnd.Attach(GetDlgItem(2));

        wnd.SendMessage(EM_SETSEL, (LPARAM)-1, -1);
        wnd.SendMessage(EM_REPLACESEL, 0, (LPARAM)pszMessage);
    }
}

```

```

// Event Handler for Click
STDMETHOD(OnClick)()
{
    DisplayNotification(_T("OnClick\r\n"));
    return S_OK;
}

LRESULT OnCreate(UINT, WPARAM, LPARAM, BOOL&)
{
    HRESULT hr = E_INVALIDARG;

    _AtlModule->Lock();

    RECT rect;
    GetClientRect(&rect);

    RECT rect2;
    rect2 = rect;

    rect2.bottom -= 200;

    // if g_UseMethod is odd then create AxHost directly as the child of the main window
    if (g_UseMethod & 0x1)
    {
        m_axwnd.Create(m_hWnd, rect2, NULL, WS_CHILD | WS_VISIBLE | WS_BORDER, 0, 1);
    }
    // if g_UseMethod is even then the AtlAx version is invoked.
    else
    {
        // Create a child window.
        // AtlAx functions will subclass this window.
        m_wndChild.Create(m_hWnd, rect2, NULL, WS_CHILD | WS_VISIBLE | WS_BORDER, 0, 1);
        // Attach the child window to the CAxWindow so we can access the
        // host that subclasses the child window.
        m_axwnd.Attach(m_wndChild);
    }

    if (m_axwnd.m_hWnd != NULL)
    {
        CComPtr<IUnknown> spControl;

        // The calls to (AtlAx)CreateControl(Lic)(Ex) do the following:
        // Create Calendar control. (Passing in NULL for license key.
        // Pass in valid license key to the Lic functions if the
        // control requires one.)
        // Get the IUnknown pointer for the control.
        // Sink events from the control.

        // The AtlAx versions subclass the hWnd that is passed in to them
        // to implement the host functionality.

        // The first 4 calls accomplish it in one call.
        // The last 4 calls accomplish it using multiple steps.

        switch (g_UseMethod)
        {
            case 1:
            {
                hr = m_axwnd.CreateControlLicEx(
                    OLESTR("MSCAL.Calendar.7"),
                    NULL,
                    NULL,
                    &spControl,
                    __uuidof(DCalendarEvents),
                    (IUnknown*)(IDispatchImpl<1, CMainwindow,
                        &__uuidof(DCalendarEvents), &__uuidof(__MSACAL), 7, 0>*)this
                );
                break;
            }
        }
    }
}

```

```

},
case 2:
{
    hr = AtlAxCreateControlLicEx(
        OLESTR("MSCAL.Calendar.7"),
        m_wndChild.m_hWnd,
        NULL,
        NULL,
        &spControl,
        __uuidof(DCalendarEvents),
        (IUnknown*)(IDispatchImpl<1, CMainWindow,
            &__uuidof(DCalendarEvents), &__uuidof(__MSACAL), 7, 0>*)this,
        NULL
    );
    break;
}
case 3:
{
    hr = m_axwnd.CreateControlEx(
        OLESTR("MSCAL.Calendar.7"),
        NULL,
        NULL,
        &spControl,
        __uuidof(DCalendarEvents),
        (IUnknown*)(IDispatchImpl<1, CMainWindow,
            &__uuidof(DCalendarEvents), &__uuidof(__MSACAL), 7, 0>*)this
    );
    break;
}
case 4:
{
    hr = AtlAxCreateControlEx(
        OLESTR("MSCAL.Calendar.7"),
        m_wndChild.m_hWnd,
        NULL,
        NULL,
        &spControl,
        __uuidof(DCalendarEvents),
        (IUnknown*)(IDispatchImpl<1, CMainWindow,
            &__uuidof(DCalendarEvents), &__uuidof(__MSACAL), 7, 0>*)this
    );
    break;
}
// The following calls create the control, obtain an interface to
// the control, and set up the sink in multiple steps.
case 5:
{
    hr = m_axwnd.CreateControlLic(
        OLESTR("MSCAL.Calendar.7")
    );
    break;
}
case 6:
{
    hr = AtlAxCreateControlLic(
        OLESTR("MSCAL.Calendar.7"),
        m_wndChild.m_hWnd,
        NULL,
        NULL
    );
    break;
}
case 7:
{
    hr = m_axwnd.CreateControl(
        OLESTR("MSCAL.Calendar.7")
    );
    break;
}
case 8:

```

```

        case 0:
        {
            hr = AtlAxCreateControl(
                OLESTR("MSCAL.Calendar.7"),
                m_wndChild.m_hWnd,
                NULL,
                NULL
            );
            break;
        }
    }

    // have to obtain an interface to the control and set up the sink
    if (g_UseMethod > 4)
    {
        if (SUCCEEDED(hr))
        {
            hr = m_axwnd.QueryControl(&spControl);
            if (SUCCEEDED(hr))
            {
                // Sink events form the control
                DispEventAdvise(spControl, &__uuidof(DCalendarEvents));
            }
        }
    }

    if (SUCCEEDED(hr))
    {
        // Use the returned IUnknown pointer.
        CComPtr<ICalendar> spCalendar;
        hr = spControl.QueryInterface(&spCalendar);
        if (SUCCEEDED(hr))
        {
            spCalendar->put_ShowDateSelectors(VARIANT_FALSE);
        }
    }
}

rect2 = rect;
rect2.top = rect.bottom - 200 + 1;
m_wndEdit.Create(_T("Edit"), m_hWnd, rect2, NULL, WS_CHILD | WS_VISIBLE |
    WS_BORDER | ES_AUTOHSCROLL | ES_AUTOVSCROLL | ES_MULTILINE, 0, 2);
return 0;
}

LRESULT OnDestroy(UINT, WPARAM, LPARAM, BOOL&)
{
    _pAtlModule->Unlock();
    return 0;
}
};

class CHostActiveXModule : public CATlExeModuleT<CHostActiveXModule>
{
public :

    CMainWindow m_wndMain;

    // Create the Main window
    HRESULT PreMessageLoop(int nCmdShow)
    {
        HRESULT hr = CATlExeModuleT<CHostActiveXModule>::PreMessageLoop(nCmdShow);
        if (SUCCEEDED(hr))
        {
            AtlAxWinInit();
            hr = S_OK;
            RECT rc;
            rc.top = rc.left = 100;
            rc.bottom = rc.right = 500;

```

```

        m_wndMain.Create(NULL, rc, _T("Host Calendar") );
        m_wndMain.ShowWindow(nCmdShow);
    }
    return hr;
}

// Clean up. App is exiting.
HRESULT PostMessageLoop()
{
    AtlAxWinTerm();
    return CATLExeModuleT<CHostActiveXModule>::PostMessageLoop();
}
};

CHostActiveXModule _AtlModule;

int APIENTRY _tWinMain(HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                      LPTSTR lpCmdLine,
                      int nCmdShow)
{
    UNREFERENCED_PARAMETER(hInstance);
    UNREFERENCED_PARAMETER(hPrevInstance);

    g_UseMethod = _ttoi(lpCmdLine);

    if (ValidateUseMethod())
    {
        return _AtlModule.WinMain(nCmdShow);
    }
    else
    {
        return E_INVALIDARG;
    }
}

```

See also

[Control Containment FAQ](#)

[AtlAxCreateControl](#)

[AtlAxCreateControlEx](#)

[AtlAxCreateControlLic](#)

[AtlAxCreateControlLicEx](#)

[CAXWindow2T Class](#)

[IAxWinHostWindowLic Interface](#)

ATL COM Property Pages

4/1/2019 • 2 minutes to read • [Edit Online](#)

COM property pages provide a user interface for setting the properties (or calling the methods) of one or more COM objects. Property pages are used extensively by ActiveX controls for providing rich user interfaces that allow control properties to be set at design time.

Property pages are COM objects that implement the [IPropertyPage](#) or [IPropertyPage2](#) interface. These interfaces provide methods that allow the page to be associated with a `site` (a COM object representing the container of the page) and a number of *objects* (COM objects whose methods will be called in response to changes made by the user of the property page). The property page container is responsible for calling methods on the property page interface to tell the page when to show or hide its user interface, and when to apply the changes made by the user to the underlying objects.

Each property page can be built completely independently of the objects whose properties can be set. All that a property page needs is to understand a particular interface (or set of interfaces) and to provide a user interface for calling methods on that interface.

For more information, see [Property Sheets and Property Pages](#) in the Windows SDK.

In This Section

[Specifying Property Pages](#)

Lists the steps for specifying property pages for your control and shows an example class.

[Implementing Property Pages](#)

Lists the steps for implementing property pages, including methods to override. Walks you through a complete example based on the ATLPages sample program.

Related Sections

[ATLPages Sample](#)

The sample abstract for the ATLPages sample, which implements a property page using `IPropertyPageImpl`.

[ATL](#)

Provides links to conceptual topics on how to program using the Active Template Library.

See also

[Concepts](#)

Specifying Property Pages

4/1/2019 • 2 minutes to read • [Edit Online](#)

When you create an ActiveX control, you will often want to associate it with property pages that can be used to set the properties of your control. Control containers use the `ISpecifyPropertyPages` interface to find out which property pages can be used to set your control's properties. You will need to implement this interface on your control.

To implement `ISpecifyPropertyPages` using ATL, take the following steps:

1. Derive your class from `ISpecifyPropertyPagesImpl`.
2. Add an entry for `ISpecifyPropertyPages` to your class's COM map.
3. Add a `PROP_PAGE` entry to the property map for each page associated with your control.

NOTE

When generating a standard control using the [ATL Control Wizard](#), you will only have to add the `PROP_PAGE` entries to the property map. The wizard generates the necessary code for the other steps.

Well-behaved containers will display the specified property pages in the same order as the `PROP_PAGE` entries in the property map. Generally, you should put standard property page entries after the entries for your custom pages in the property map, so that users see the pages specific to your control first.

Example

The following class for a calendar control uses the `ISpecifyPropertyPages` interface to tell containers that its properties can be set using a custom date page and the stock color page.

```
class ATL_NO_VTABLE CMyCtrl :
    OtherInterfaces
    public ISpecifyPropertyPagesImpl<CMyCtrl>
{
public:

    BEGIN_COM_MAP(CMyCtrl)
        OtherComMapEntries
        COM_INTERFACE_ENTRY(ISpecifyPropertyPages)
    END_COM_MAP()

    BEGIN_PROP_MAP(CMyCtrl)
        OtherPropMapEntries
        PROP_PAGE(CLSID_DatePage)
        PROP_PAGE(CLSID_StockColorPage)
    END_PROP_MAP()

    // Remainder of class declaration omitted.
```

See also

[Property Pages](#)

[ATLPages Sample](#)

Implementing Property Pages

5/15/2019 • 2 minutes to read • [Edit Online](#)

The ATL Property Page wizard is not available in Visual Studio 2019 and later.

Property pages are COM objects that implement the `IPropertyPage` or `IPropertyPage2` interface. ATL provides support for implementing property pages through the [ATL Property Page Wizard](#) in the [Add Class dialog box](#).

To create a property page using ATL:

- Create or open an ATL Dynamic-link library (DLL) server project.
- Open the [Add Class dialog box](#) and select **ATL Property Page**.
- Make sure your property page is apartment threaded (since it has a user interface).
- Set the title, description (Doc String), and help file to be associated with your page.
- Add controls to the generated dialog resource to act as the user interface of your property page.
- Respond to changes in your page's user interface to perform validation, update the page site, or update the objects associated with your page. In particular, call `IPropertyPageImpl::SetDirty` when the user makes changes to the property page.
- Optionally override the `IPropertyPageImpl` methods using the guidelines below.

IPROPERTYPAGEIMPL METHOD	OVERRIDE WHEN YOU WANT TO...	NOTES
SetObjects	Perform basic sanity checks on the number of objects being passed to your page and the interfaces that they support.	Execute your own code before calling the base class implementation. If the objects being set don't conform to your expectations, you should fail the call as soon as possible.
Activate	Initialize your page's user interface (for example, set dialog controls with current property values from objects, create controls dynamically, or perform other initializations).	Call the base class implementation before your code so that the base class has a chance to create the dialog window and all the controls before you try to update them.
Apply	Validate the property settings and update the objects.	There is no need to call the base class implementation since it doesn't do anything apart from trace the call.
Deactivate	Clean up window-related items.	The base class implementation destroys the dialog box representing the property page. If you need to clean up before the dialog box is destroyed, you should add your code before calling the base class.

For an example property page implementation, see [Example: Implementing a Property Page](#).

NOTE

If you want to host ActiveX controls in your property page, you will need to change the derivation of your wizard-generated class. Replace **CDialogImpl<CYourClass>** with **CAXDialogImpl<CYourClass>** in the list of base classes.

See also

[Property Pages](#)

[ATLPages Sample](#)

Example: Implementing a Property Page

5/15/2019 • 9 minutes to read • [Edit Online](#)

The ATL Property Page wizard is not available in Visual Studio 2019 and later.

This example shows how to build a property page that displays (and allows you to change) properties of the [Document Classes](#) interface.

The example is based on the [ATL Pages sample](#).

To complete this example, you will:

- [Add the ATL property page class](#) using the Add Class dialog box and the ATL Property Page Wizard.
- [Edit the dialog resource](#) by adding new controls for the interesting properties of the `Document` interface.
- [Add message handlers](#) to keep the property page site informed of changes made by the user.
- Add some `#import` statements and a typedef in the [Housekeeping](#) section.
- [Override IPropertyPageImpl::SetObjects](#) to validate the objects being passed to the property page.
- [Override IPropertyPageImpl::Activate](#) to initialize the property page's interface.
- [Override IPropertyPageImpl::Apply](#) to update the object with the latest property values.
- [Display the property page](#) by creating a simple helper object.
- [Create a macro](#) that will test the property page.

Adding the ATL Property Page Class

First, create a new ATL project for a DLL server called `ATLPages7`. Now use the [ATL Property Page Wizard](#) to generate a property page. Give the property page a **Short Name** of **DocProperties** then switch to the **Strings** page to set property-page-specific items as shown in the table below.

ITEM	VALUE
Title	TextDocument
Doc String	VCUE TextDocument Properties
Helpfile	<blank>

The values that you set on this page of the wizard will be returned to the property page container when it calls `IPropertyPage::GetPageInfo`. What happens to the strings after that is dependent on the container, but typically they will be used to identify your page to the user. The Title will usually appear in a tab above your page and the Doc String may be displayed in a status bar or ToolTip (although the standard property frame doesn't use this string at all).

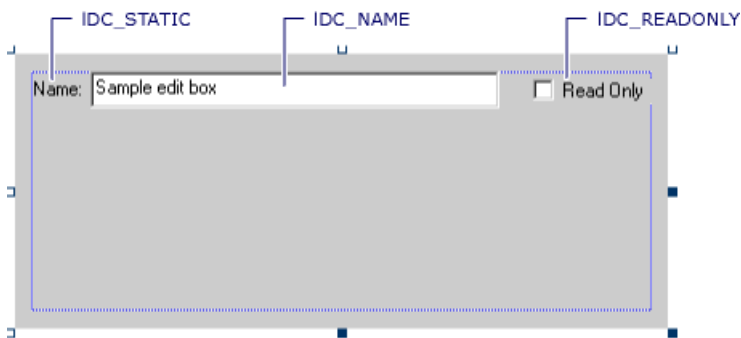
NOTE

The strings that you set here are stored as string resources in your project by the wizard. You can easily edit these strings using the resource editor if you need to change this information after the code for your page has been generated.

Click **OK** to have the wizard generate your property page.

Editing the Dialog Resource

Now that your property page has been generated, you'll need to add a few controls to the dialog resource representing your page. Add an edit box, a static text control, and a check box and set their IDs as shown below:



These controls will be used to display the file name of the document and its read-only status.

NOTE

The dialog resource does not include a frame or command buttons, nor does it have the tabbed look that you might have expected. These features are provided by a property page frame such as the one created by calling [OleCreatePropertyFrame](#).

Adding Message Handlers

With the controls in place, you can add message handlers to update the dirty status of the page when the value of either of the controls changes:

```
BEGIN_MSG_MAP(CDocProperties)
    COMMAND_HANDLER(IDC_NAME, EN_CHANGE, OnUIChange)
    COMMAND_HANDLER(IDC_READONLY, BN_CLICKED, OnUIChange)
    CHAIN_MSG_MAP(IPropertyPageImpl<CDocProperties>)
END_MSG_MAP()

// Respond to changes in the UI to update the dirty status of the page
LRESULT OnUIChange(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled)
{
    wNotifyCode; wID; hWndCtl; bHandled;
    SetDirty(true);
    return 0;
}
```

This code responds to changes made to the edit control or check box by calling [IPropertyPageImpl::SetDirty](#), which informs the page site that the page has changed. Typically the page site will respond by enabling or disabling an **Apply** button on the property page frame.

NOTE

In your own property pages, you might need to keep track of precisely which properties have been altered by the user so that you can avoid updating properties that haven't been changed. This example implements that code by keeping track of the original property values and comparing them with the current values from the UI when it's time to apply the changes.

Housekeeping

Now add a couple of `#import` statements to DocProperties.h so that the compiler knows about the `Document`

interface:

```
// MSO.dll
#import <libid:2DF8D04C-5BFA-101B-BDE5-00AA0044DE52> version("2.2") \
    rename("RGB", "Rgb") \
    rename("DocumentProperties", "documentproperties") \
    rename("ReplaceText", "replaceText") \
    rename("FindText", "findText") \
    rename("GetObject", "getObject") \
    raw_interfaces_only

// dte.olb
#import <libid:80CC9F66-E7D8-4DDD-85B6-D9E6CD0E93E2> \
    inject_statement("using namespace Office;") \
    rename("ReplaceText", "replaceText") \
    rename("FindText", "findText") \
    rename("GetObject", "getObject") \
    rename("SearchPath", "searchPath") \
    raw_interfaces_only
```

You'll also need to refer to the `IPropertyPageImpl` base class; add the following **typedef** to the `CDocProperties` class:

```
typedef IPropertyPageImpl<CDocProperties> PPGBaseClass;
```

Overriding IPropertyPageImpl::SetObjects

The first `IPropertyPageImpl` method that you need to override is [SetObjects](#). Here you'll add code to check that only a single object has been passed and that it supports the `Document` interface that you're expecting:

```
STDMETHOD(SetObjects)(ULONG nObjects, IUnknown** ppUnk)
{
    HRESULT hr = E_INVALIDARG;
    if (nObjects == 1)
    {
        CComQIPtr<EnvDTE::Document> pDoc(ppUnk[0]);
        if (pDoc)
            hr = PPGBaseClass::SetObjects(nObjects, ppUnk);
    }
    return hr;
}
```

NOTE

It makes sense to support only a single object for this page because you will allow the user to set the file name of the object — only one file can exist at any one location.

Overriding IPropertyPageImpl::Activate

The next step is to initialize the property page with the property values of the underlying object when the page is first created.

In this case you should add the following members to the class since you'll also use the initial property values for comparison when users of the page apply their changes:

```
CComBSTR m_bstrFullName; // The original name
VARIANT_BOOL m_bReadOnly; // The original read-only state
```

The base class implementation of the [Activate](#) method is responsible for creating the dialog box and its controls, so you can override this method and add your own initialization after calling the base class:

```
STDMETHOD(Activate)(HWND hWndParent, LPCRECT prc, BOOL bModal)
{
    // If we don't have any objects, this method should not be called
    // Note that OleCreatePropertyFrame will call Activate even if
    // a call to SetObjects fails, so this check is required
    if (!m_ppUnk)
        return E_UNEXPECTED;

    // Use Activate to update the property page's UI with information
    // obtained from the objects in the m_ppUnk array

    // We update the page to display the Name and ReadOnly properties
    // of the document

    // Call the base class
    HRESULT hr = PPGBaseClass::Activate(hWndParent, prc, bModal);
    if (FAILED(hr))
        return hr;

    // Get the EnvDTE::Document pointer
    CComQIPtr<EnvDTE::Document> pDoc(m_ppUnk[0]);
    if (!pDoc)
        return E_UNEXPECTED;

    // Get the FullName property
    hr = pDoc->get_FullName(&m_bstrFullName);
    if (FAILED(hr))
        return hr;

    // Set the text box so that the user can see the document name
    USES_CONVERSION;
    SetDlgItemText(IDC_NAME, CW2CT(m_bstrFullName));

    // Get the ReadOnly property
    m_bReadOnly = VARIANT_FALSE;
    hr = pDoc->get_ReadOnly(&m_bReadOnly);
    if (FAILED(hr))
        return hr;

    // Set the check box so that the user can see the document's read-only status
    CheckDlgButton(IDC_READONLY, m_bReadOnly ? BST_CHECKED : BST_UNCHECKED);

    return hr;
}
```

This code uses the COM methods of the `Document` interface to get the properties that you're interested in. It then uses the Win32 API wrappers provided by [CDialogImpl](#) and its base classes to display the property values to the user.

Overriding IPropertyPageImpl::Apply

When users want to apply their changes to the objects, the property page site will call the [Apply](#) method. This is the place to do the reverse of the code in `Activate` — whereas `Activate` took values from the object and pushed them into the controls on the property page, `Apply` takes values from the controls on the property page and pushes them into the object.


```

STDMETHOD(Apply)(void)
{
    // If we don't have any objects, this method should not be called
    if (!m_ppUnk)
        return E_UNEXPECTED;

    // Use Apply to validate the user's settings and update the objects'
    // properties

    // Check whether we need to update the object
    // Quite important since standard property frame calls Apply
    // when it doesn't need to
    if (!m_bDirty)
        return S_OK;

    HRESULT hr = E_UNEXPECTED;

    // Get a pointer to the document
    CComQIPtr<EnvDTE::Document> pDoc(m_ppUnk[0]);
    if (!pDoc)
        return hr;

    // Get the read-only setting
    VARIANT_BOOL bReadOnly = IsDlgButtonChecked(IDC_READONLY) ? VARIANT_TRUE : VARIANT_FALSE;

    // Get the file name
    CComBSTR bstrName;
    if (!GetDlgItemText(IDC_NAME, bstrName.m_str))
        return E_FAIL;

    // Set the read-only property
    if (bReadOnly != m_bReadOnly)
    {
        hr = pDoc->put_ReadOnly(bReadOnly);
        if (FAILED(hr))
            return hr;
    }

    // Save the document
    if (bstrName != m_bstrFullName)
    {
        EnvDTE::vsSaveStatus status;
        hr = pDoc->Save(bstrName, &status);
        if (FAILED(hr))
            return hr;
    }

    // Clear the dirty status of the property page
    SetDirty(false);

    return S_OK;
}

```

NOTE

The check against `m_bDirty` at the beginning of this implementation is an initial check to avoid unnecessary updates of the objects if `Apply` is called more than once. There are also checks against each of the property values to ensure that only changes result in a method call to the `Document`.

NOTE

`Document` exposes `FullName` as a read-only property. To update the file name of the document based on changes made to the property page, you have to use the `Save` method to save the file with a different name. Thus, the code in a property page doesn't have to limit itself to getting or setting properties.

Displaying the Property Page

To display this page, you need to create a simple helper object. The helper object will provide a method that simplifies the `OleCreatePropertyFrame` API for displaying a single page connected to a single object. This helper will be designed so that it can be used from Visual Basic.

Use the [Add Class dialog box](#) and the [ATL Simple Object Wizard](#) to generate a new class and use `Helper` as its short name. Once created, add a method as shown in the table below.

ITEM	VALUE
Method Name	<code>ShowPage</code>
Parameters	<code>[in] BSTR bstrCaption, [in] BSTR bstrID, [in] IUnknown* pUnk</code>

The *bstrCaption* parameter is the caption to be displayed as the title of the dialog box. The *bstrID* parameter is a string representing either a CLSID or a ProgID of the property page to display. The *pUnk* parameter will be the `IUnknown` pointer of the object whose properties will be configured by the property page.

Implement the method as shown below:

```
STDMETHODIMP CHelper::ShowPage(BSTR bstrCaption, BSTR bstrID, IUnknown* pUnk)
{
    if (!pUnk)
        return E_INVALIDARG;

    // First, assume bstrID is a string representing the CLSID
    CLSID theCLSID = {0};
    HRESULT hr = CLSIDFromString(bstrID, &theCLSID);
    if (FAILED(hr))
    {
        // Now assume bstrID is a ProgID
        hr = CLSIDFromProgID(bstrID, &theCLSID);
        if (FAILED(hr))
            return hr;
    }

    // Use the system-supplied property frame
    return OleCreatePropertyFrame(
        GetActiveWindow(), // Parent window of the property frame
        0,                 // Horizontal position of the property frame
        0,                 // Vertical position of the property frame
        bstrCaption,        // Property frame caption
        1,                 // Number of objects
        &pUnk,               // Array of IUnknown pointers for objects
        1,                 // Number of property pages
        &theCLSID,           // Array of CLSIDs for property pages
        NULL,              // Locale identifier
        0,                 // Reserved - 0
        NULL,              // Reserved - 0
    );
}
```

Creating a Macro

Once you've built the project, you can test the property page and the helper object using a simple macro that you can create and run in the Visual Studio development environment. This macro will create a helper object, then call its `ShowPage` method using the ProgID of the **DocProperties** property page and the `IUnknown` pointer of the document currently active in the Visual Studio editor. The code you need for this macro is shown below:

```
Imports EnvDTE
Imports System.Diagnostics

Public Module AtlPages

Public Sub Test()
    Dim Helper
    Helper = CreateObject("ATLPages7.Helper.1")

    On Error Resume Next
    Helper.ShowPage( ActiveDocument.Name, "ATLPages7Lib.DocumentProperties.1", DTE.ActiveDocument )
End Sub

End Module
```

When you run this macro, the property page will be displayed showing the file name and read-only status of the currently active text document. The read-only state of the document only reflects the ability to write to the document in the development environment; it doesn't affect the read-only attribute of the file on disk.

See also

[Property Pages](#)
[ATLPages Sample](#)

ATL Support for DHTML Controls

3/4/2019 • 2 minutes to read • [Edit Online](#)

Using ATL, you can create a control with Dynamic HTML (DHTML) capability. An ATL DHTML control:

- Hosts the WebBrowser control.
- Specifies, using HTML, the user interface (UI) of the DHTML control.
- Accesses the WebBrowser object and its methods through its interface, [IWebBrowser2](#).
- Manages communication between C++ code and HTML.

A DHTML control is similar to any other ATL control, except the DHTML control includes an additional dispatch interface. See the figure in [Identifying the Elements of the DHTML Control Project](#) for an illustration of the interfaces provided in the default DHTML project.

You can view the ATL DHTML control in a Web browser or other container, such as the ActiveX Control Test Container.

In This Section

[Identifying the Elements of the DHTML Control Project](#)

Describes the elements of a DHTML control project.

[Calling C++ Code from DHTML](#)

Provides an example of calling C++ code from a DHTML control.

[Creating an ATL DHTML Control](#)

Lists the steps for creating a DHTML control.

[Testing the ATL DHTML Control](#)

Shows how to build and test the initial DHTML control project.

[Modifying the ATL DHTML Control](#)

Shows how to add some functionality to the control.

[Testing the Altered ATL DHTML Control](#)

Shows how to build and test the control's added functionality.

Related Sections

[ATL](#)

Provides links to conceptual topics on how to program using the Active Template Library.

Identifying the Elements of the DHTML Control Project

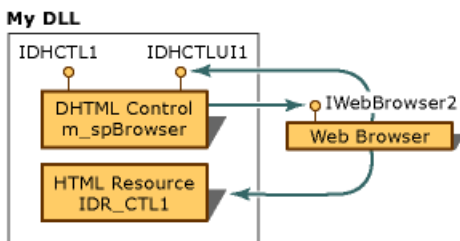
3/5/2019 • 2 minutes to read • [Edit Online](#)

Most DHTML control code is exactly like that created for any ATL control. For a basic understanding of the generic code, work through the [ATL tutorial](#), and read the sections [Creating an ATL Project](#) and [Fundamentals of ATL COM Objects](#).

A DHTML control is similar to any ATL control, except:

- In addition to the regular interfaces a control implements, it implements an additional interface that is used to communicate between the C++ code and the HTML user interface (UI). The HTML UI calls into C++ code using this interface.
- It creates an HTML resource for the control UI.
- It allows access to the DHTML object model through the member variable `m_spBrowser`, which is a smart pointer of type [IWebBrowser2](#). Use this pointer to access any part of the DHTML object model.

The following graphic illustrates the relationship between your DLL, the DHTML control, the Web browser, and the HTML resource.



NOTE

The names on this graphic are placeholders. The names of your HTML resource and the interfaces exposed on your control are based on the names you assign them in the ATL Control Wizard.

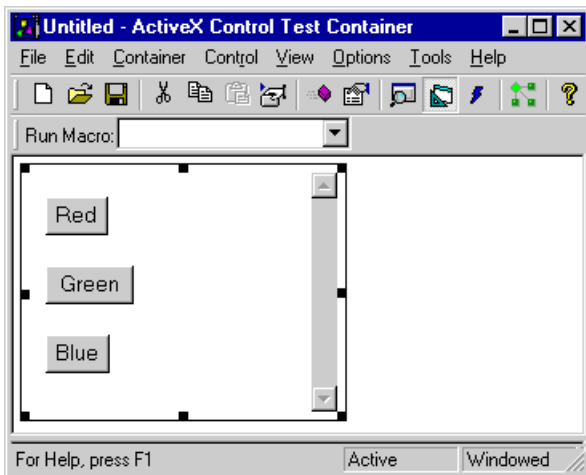
In this graphic, the elements are:

- **My DLL** The DLL created using the ATL Project Wizard.
- **DHTML Control** (`m_spBrowser`) The DHTML control, created using the ATL Object Wizard. This control accesses the Web browser object and its methods through the Web browser object's interface, `IWebBrowser2`. The control itself exposes the following two interfaces, in addition to the other standard interfaces required for a control.
 - `IDHCTL1` The interface exposed by the control for use only by the container.
 - `IDHCTUI1` The dispatch interface for communicating between the C++ code and the HTML user interface. The Web browser uses the control's dispatch interface to display the control. You can call various methods of this dispatch interface from the control's user interface by invoking `window.external`, followed by the method name on this dispatch interface that you want to invoke. You would access `window.external` from a SCRIPT tag within the HTML that makes up the UI for this control. For more information about invoking external methods in the resource file, see [Calling](#)

[C++ Code from DHTML.](#)

- **IDR_CTL1** The resource ID of the HTML resource. Its file name, in this case, is DHCTL1UI.htm. The DHTML control uses an HTML resource that contains standard HTML tags and external window dispatch commands that you can edit using the Text editor.
- **Web Browser** The Web browser displays the control's UI, based on the HTML in the HTML resource. A pointer to the Web browser's `IWebBrowser2` interface is available in the DHTML control to allow access to the DHTML object model.

The ATL Control Wizard generates a control with default code in both the HTML resource and the .cpp file. You can compile and run the control as generated by the wizard, and then view the control in either the Web browser or the ActiveX Control Test Container. The picture below shows the default ATL DHTML control with three buttons displayed in Test Container:



See [Creating an ATL DHTML Control](#) to get started building a DHTML control. See [Testing Properties and Events with Test Container](#) for information on how to access Test Container.

See also

[Support for DHTML Control](#)

Calling C++ Code from DHTML

3/4/2019 • 2 minutes to read • [Edit Online](#)

A DHTML control can be hosted in a container, such as Test Container or Internet Explorer. See [Testing Properties and Events with Test Container](#) for information on how to access Test Container.

The container hosting the control communicates with the control using the normal control interfaces. DHTML uses the dispatch interface that ends with "UI" to communicate with your C++ code and your HTML resource. In [Modifying the ATL DHTML Control](#), you can practice adding the methods to be called by these different interfaces.

To see an example of calling C++ code from DHTML, [create a DHTML control](#) using the ATL Control Wizard and examine the code in the header file and in the HTML file.

Declaring WebBrowser Methods in the Header File

To invoke C++ methods from the DHTML UI, you must add methods to your control's UI interface. For example, the header file created by the ATL Control Wizard contains the C++ method `OnClick`, which is a member of the UI interface of the wizard-generated control.

Examine `OnClick` in the control's .h file:

```
STDMETHOD(OnClick)(IDispatch* pdispBody, VARIANT varColor)
```

The first parameter, *pdispBody*, is a pointer to the body object's dispatch interface. The second parameter, *varColor*, identifies the color to apply to the control.

Calling C++ Code in the HTML File

Once you have declared the WebBrowser methods in the header file, you can invoke the methods from the HTML file. Notice in the HTML file that the ATL Control Wizard inserts three Windows dispatch methods: three `OnClick` methods that dispatch messages to change the background color of the control.

Examine one of the methods in the HTML file:

```
<BUTTON onclick='window.external.OnClick(theBody, "red");'>Red</BUTTON>
```

In the HTML code above, the window external method, `OnClick`, is called as part of the button tag. The method has two parameters: `theBody`, which references the body of the HTML document, and `"red"`, which indicates that the control's background color will be changed to red when the button is clicked. The `Red` following the tag is the button's label.

See [Modifying the ATL DHTML Control](#) for more information about providing your own methods. See [Identifying the Elements of the DHTML Control Project](#) for more information about the HTML file.

See also

[Support for DHTML Control](#)

Creating an ATL DHTML Control

3/4/2019 • 2 minutes to read • [Edit Online](#)

The ATL Control Wizard automates the process of creating a DHTML control. It generates the necessary resource files, including an HTML file containing sample code.

To create an ATL DHTML control

1. Follow the steps in [Creating an ATL Project](#).
2. In **Class View**, right-click the project node, point to **Add**, and click **Add Class** from the shortcut menu. In the **Add Class** dialog box, double-click the [ATL Control Wizard](#). In the **ATL Control Wizard**, click the **Options** tab and select **DHTML control**.

You can now [test the default control](#).

See also

[Support for DHTML Control](#)

Testing the ATL DHTML Control

3/4/2019 • 2 minutes to read • [Edit Online](#)

Once you have created your project, you can build and test the sample control. Before you do this, use **Class View** and **Solution Explorer** to examine the project. The elements of your project are described in greater detail in [Identifying the Elements of the DHTML Control Project](#).

To build and test the ATL DHTML control

1. Build the project. From the **Build** menu, click **Build Solution**.
2. When the build is completed, open **Test Container**. See [Testing Properties and Events with Test Container](#) for information on how to access **Test Container**.
3. In **Test Container**, from the **Edit** menu, click **Insert New Control**.
4. In the **Insert Control** dialog box, select your control from the list box. Remember, its name is based on the short name you indicated in the ATL Control Wizard. Click **OK**.
5. Examine the control. Note that it has a scroll bar. Use the control's handles to resize the control to activate the scrollbar.
6. Test the control's buttons. The background color changes to the color indicated by the button.
7. Close **Test Container**.

Next, try [modifying the DHTML control](#).

See also

[Support for DHTML Control](#)

Modifying the ATL DHTML Control

3/4/2019 • 2 minutes to read • [Edit Online](#)

The ATL Control Wizard provides starter code so you can build and run the control, and so you can see how the methods are written in the project files and how the DHTML calls into the control's C++ code using the dispatch methods. You can add any dispatch method to the interface. Then, you can call the methods in the HTML resource.

To modify the ATL DHTML control

1. In **Class View**, expand the control project.

Note that the interface that ends in "UI" has one method, `OnClick`. The interface that does not end in "UI" does not have any methods.

2. Add a method called `MethodInvoked` to the interface that does not end in "UI."

This method will be added to the interface that is used in the control container for container interaction, not to the interface used by DHTML to interact with the control. Only the container can invoke this method.

3. Find the stubbed-out method in the .cpp file and add code to display a message box, for example:

```
::MessageBox(NULL, _T("I'm invoked"), _T("Your Container Message"), MB_OK);
```

4. Add another method called `HelloHTML`, only this time, add it to the interface that ends in "UI." Find the stubbed-out `HelloHTML` method in the .cpp file and add code to display a message box, for example:

```
::MessageBox(NULL, _T("Here's your message"), _T("HelloHTML"), MB_OK);
```

5. Add a third method, `GoToURL`, to the interface that does not end in "UI." Implement this method by calling `IWebBrowser2::Navigate`, as follows:

```
m_spBrowser->Navigate(CComBSTR(L"www.microsoft.com"), NULL, NULL, NULL, NULL);
```

You can use the `IWebBrowser2` methods because ATL provides a pointer to that interface for you in your .h file.

Next, modify the HTML resource to invoke the methods you created. You will add three buttons for invoking these methods.

To modify the HTML resource

1. In **Solution Explorer**, double-click the .htm file to display the HTML resource.

Examine the HTML, especially the calls to the external Windows dispatch methods. The HTML calls the project's `OnClick` method, and the parameters indicate the body of the control (`theBody`) and the color to assign ("red"). The text following the method call is the label that appears on the button.

2. Add another `OnClick` method, only change the color. For example:

```
<br>
<br>
<BUTTON onclick='window.external.OnClick(theBody, "white");'>Refresh</BUTTON>
```

This method will create a button, labeled **Refresh**, that the user can click to return the control to the original, white background.

3. Add the call to the `HelloHTML` method you created. For example:

```
<br>
<br>
<BUTTON onclick='window.external>HelloHTML();'>HelloHTML</BUTTON>
```

This method will create a button, labeled **HelloHTML**, that the user can click to display the `HelloHTML` message box.

You can now build and [test the modified DHTML control](#).

See also

[Support for DHTML Control](#)

Testing the Modified ATL DHTML Control

3/4/2019 • 2 minutes to read • [Edit Online](#)

Try out your new control to see how it works now.

To build and test the modified control

1. Rebuild the project and open it in **Test Container**. See [Testing Properties and Events with Test Container](#) for information on how to access **Test Container**.

Resize the control to show all of the buttons you added.

2. Examine the two buttons that you inserted by altering the HTML. Each button bears the label you identified in [Modifying the ATL DHTML Control](#): **Refresh** and **HelloHTML**.
3. Test the two new buttons to see how they work.

Now test the methods that are not part of the UI.

1. Highlight the control, so the border is activated.
2. On the **Control** menu, choose **Invoke Methods**.

The methods in the list labeled **Method Name** are the methods that the container can call: `MethodInvoked` and `GoToURL`. All other methods are controlled by the UI.

3. Select a method to invoke and choose **Invoke** to display the method's message box or to navigate to `www.microsoft.com`.
4. In the **Invoke Methods** dialog box, choose **Close**.

To learn more about the various elements and files that make up an ATL DHTML control, see [Identifying the Elements of the DHTML Control Project](#).

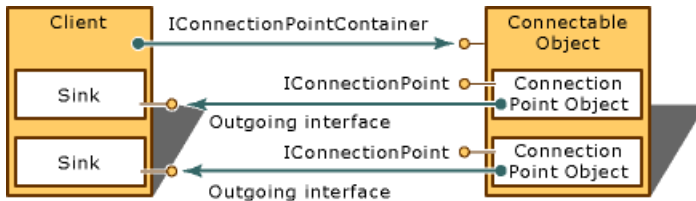
See also

[Support for DHTML Control](#)

ATL Connection Points

3/4/2019 • 2 minutes to read • [Edit Online](#)

A connectable object is one that supports outgoing interfaces. An outgoing interface allows the object to communicate with a client. For each outgoing interface, the connectable object exposes a connection point. Each outgoing interface is implemented by a client on an object called a sink.



Each connection point supports the [IConnectionPoint](#) interface. The connectable object exposes its connection points to the client through the [IConnectionPointContainer](#) interface.

In This Section

[ATL Connection Point Classes](#)

Briefly describes the ATL classes that support connection points.

[Adding Connection Points to an Object](#)

Outlines the steps used to add connection points to an object.

[ATL Connection Point Example](#)

Provides an example of declaring a connection point.

Related Sections

[ATL](#)

Provides links to conceptual topics on how to program using the Active Template Library.

See also

[Concepts](#)

ATL Connection Point Classes

3/4/2019 • 2 minutes to read • [Edit Online](#)

ATL uses the following classes to support connection points:

- [IConnectionPointImpl](#) implements a connection point. The IID of the outgoing interface it represents is passed as a template parameter.
- [IConnectionPointContainerImpl](#) implements the connection point container and manages the list of `IConnectionPointImpl` objects.
- [IPropertyNotifySinkCP](#) implements a connection point representing the [IPropertyNotifySink](#) interface.
- [CComDynamicUnkArray](#) manages an arbitrary number of connections between the connection point and its sinks.
- [CComUnkArray](#) manages a predefined number of connections as specified by the template parameter.
- [CFirePropNotifyEvent](#) notifies a client's sink that an object's property has changed or is about to change.
- [IDispEventImpl](#) provides support for connection points for an ATL COM object. These connection points are mapped with an event sink map, which is provided by your COM object.
- [IDispEventSimpleImpl](#) works in conjunction with the event sink map in your class to route events to the appropriate handler function.

See also

[Connection Point](#)

Adding Connection Points to an Object

4/1/2019 • 2 minutes to read • [Edit Online](#)

The [ATL Tutorial](#) demonstrates how to create a control with support for connection points, how to add events, and then how to implement the connection point. ATL implements connection points with the [IConnectionPointImpl](#) class.

To implement a connection point, you have two choices:

- Implement your own outgoing event source, by adding a connection point to the control or object.
- Reuse a connection point interface defined in another type library.

In either case, the Implement Connection Point Wizard uses a type library to do its work.

To add a connection point to a control or object

1. Define a dispinterface in the library block of the .idl file. If you enabled support for connection points when you created the control with the ATL Control Wizard, the dispinterface will already be created. If you did not enable support for connection points when you created the control, you must manually add a dispinterface to the .idl file. The following is an example of a dispinterface. Outgoing interfaces are not required to be dispatch interfaces but many scripting languages such as VBScript and JScript require this, so this example uses two dispinterfaces:

```
[
    uuid(3233E37D-BCC0-4871-B277-48AE6B61224A),
    helpstring("Buddy Events")
]
dispinterface DBuddyEvents
{
    properties:
    methods:
};
```

Use either the uuidgen.exe or guidgen.exe utility to generate a GUID.

2. Add the dispinterface as the `[default,source]` interface in the coclass for the object in the project's .idl file. Again, if you enabled support for connection points when you created the control, the ATL Control Wizard will create the `[default,source]` entry. To manually add this entry, add the line in bold:

```
coclass Buddy
{
    [default] interface IBuddy;
    [default,source] dispinterface DBuddyEvents;
};
```

See the .idl file in the [Circ](#) ATL sample for an example.

3. Use Class View to add methods and properties to the event interface. Right-click the class in Class View, point to **Add** on the shortcut menu, and click **Add Connection Point**.
4. In the **Source Interfaces** list box of the Implement Connection Point Wizard, select **Project's interfaces**. If you choose an interface for your control and press **OK**, you will:
 - Generate a header file with an event proxy class that implements the code that will make the

outgoing calls for the event.

- Add an entry to the connection point map.

You will also see a list of all of the type libraries on your computer. You should only use one of these other type libraries to define your connection point if you want to implement the exact same outgoing interface found in another type library.

To reuse a connection point interface defined in another type library

1. In Class View, right-click a class that implements a **BEGIN_COM_MAP** macro, point to **Add** on the shortcut menu, and click **Add Connection Point**.
2. In the Implement Connection Point Wizard, select a type library and an interface in the type library and click **Add**.
3. Edit the .idl file to either:
 - Copy the dispinterface from the .idl file for the object whose event-source is being used.
 - Use the **importlib** instruction on that type library.

See also

[Connection Point](#)

ATL Connection Point Example

3/4/2019 • 2 minutes to read • [Edit Online](#)

This example shows an object that supports [IPropertyNotifySink](#) as an outgoing interface:

```
class ATL_NO_VTABLE CConnect1 :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CConnect1, &CLSID_Connect1>,
    public IConnectionPointContainerImpl<CConnect1>,
    public IConnectionPointImpl<CConnect1, &IID_IPropertyNotifySink>,
    public IConnect1
{
public:
    CConnect1()
    {
    }

    DECLARE_REGISTRY_RESOURCEID(IDR_CONNECT1)

    BEGIN_COM_MAP(CConnect1)
        COM_INTERFACE_ENTRY(IConnect1)
        COM_INTERFACE_ENTRY(IConnectionPointContainer)
    END_COM_MAP()

    BEGIN_CONNECTION_POINT_MAP(CConnect1)
        CONNECTION_POINT_ENTRY(IID_IPropertyNotifySink)
    END_CONNECTION_POINT_MAP()

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    HRESULT FinalConstruct()
    {
        return S_OK;
    }

    void FinalRelease()
    {
    }

public:
};
```

When specifying `IPropertyNotifySink` as an outgoing interface, you can use class [IPropertyNotifySinkCP](#) instead of `IConnectionPointImpl`. For example:

```
class ATL_NO_VTABLE CConnect2 :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CConnect2, &CLSID_Connect2>,
    public IConnectionPointContainerImpl<CConnect2>,
    public IPropertyNotifySinkCP<CConnect2>
```

See also

[Connection Point](#)

Event Handling and ATL

3/4/2019 • 2 minutes to read • [Edit Online](#)

This section shows how to sink events using ATL. It covers the principles of COM event handling and the specifics of sinking events using the support provided by ATL.

For information on how to fire events and implement connection points, read [ATL Connection Points](#).

In This Section

[Event Handling Principles](#)

Discusses the steps common to all event handling.

[Implementing the Event Handling Interface](#)

Discusses the classes to use for implementing the event interface.

[Using IDisEventImpl](#)

Lists the steps for using `IDisEventImpl` and shows a code sample.

[Using IDisEventSimpleImpl](#)

Lists the steps for using `IDisEventSimpleImpl` and shows a code sample.

[ATL Event Handling Summary](#)

Summarizes, using tables, the main ways for implementing an event interface and for advising and unadvising the event source using ATL.

Related Sections

[ATL](#)

Provides links to conceptual topics on how to program using the Active Template Library.

See also

[Concepts](#)

Event Handling Principles

3/4/2019 • 2 minutes to read • [Edit Online](#)

There are three steps common to all event handling. You will need to:

- Implement the event interface on your object.
- Advise the event source that your object wants to receive events.
- Unadvise the event source when your object no longer needs to receive events.

The way that you'll implement the event interface will depend on its type. An event interface can be vtable, dual, or a dispinterface. It's up to the designer of the event source to define the interface; it's up to you to implement that interface.

NOTE

Although there are no technical reasons that an event interface can't be dual, there are a number of good design reasons to avoid the use of duals. However, this is a decision made by the designer/implementer of the event *source*. Since you're working from the perspective of the event `sink`, you need to allow for the possibility that you might not have any choice but to implement a dual event interface. For more information on dual interfaces, see [Dual Interfaces and ATL](#).

Advising the event source can be broken down into three steps:

- Query the source object for [IConnectionPointContainer](#).
- Call [IConnectionPointContainer::FindConnectionPoint](#) passing the IID of the event interface that interests you. If successful, this will return the [IConnectionPoint](#) interface on a connection point object.
- Call [IConnectionPoint::Advise](#) passing the `IUnknown` of the event sink. If successful, this will return a `DWORD` cookie representing the connection.

Once you have successfully registered your interest in receiving events, methods on your object's event interface will be called according to the events fired by the source object. When you no longer need to receive events, you can pass the cookie back to the connection point via [IConnectionPoint::Unadvise](#). This will break the connection between source and sink.

Be careful to avoid reference cycles when handling events.

See also

[Event Handling](#)

Implementing the Event Handling Interface

3/4/2019 • 2 minutes to read • [Edit Online](#)

ATL helps you with all three elements required for handling events: implementing the event interface, advising the event source, and unadvising the event source. The precise steps you'll need to take depend on the type of the event interface and the performance requirements of your application.

The most common ways of implementing an interface using ATL are:

- Deriving from a custom interface directly.
- Deriving from [IDispatchImpl](#) for dual interfaces described in a type library.
- Deriving from [IDispEventImpl](#) for dispinterfaces described in a type library.
- Deriving from [IDispEventSimpleImpl](#) for dispinterfaces not described in a type library or when you want to improve efficiency by not loading the type information at run time.

If you are implementing a custom or dual interface, you should advise the event source by calling [AtlAdvise](#) or [CComPtrBase::Advise](#). You will need to keep track of the cookie returned by the call yourself. Call [AtlUnadvise](#) to break the connection.

If you are implementing a dispinterface using `IDispEventImpl` or `IDispEventSimpleImpl`, you should advise the event source by calling [IDispEventSimpleImpl::DispEventAdvise](#). Call [IDispEventSimpleImpl::DispEventUnadvise](#) to break the connection.

If you are using `IDispEventImpl` as a base class of a composite control, the event sources listed in the sink map will be advised and unadvised automatically using [CComCompositeControl::AdviseSinkMap](#).

The `IDispEventImpl` and `IDispEventSimpleImpl` classes manage the cookie for you.

See also

[Event Handling](#)

Using IDispEventImpl

4/1/2019 • 2 minutes to read • [Edit Online](#)

When using `IDispEventImpl` to handle events, you will need to:

- Derive your class from `IDispEventImpl`.
- Add an event sink map to your class.
- Add entries to the event sink map using the `SINK_ENTRY` or `SINK_ENTRY_EX` macro.
- Implement the methods that you're interested in handling.
- Advise and unadvise the event source.

Example

The example below shows how to handle the `DocumentChange` event fired by Word's **Application** object. This event is defined as a method on the `ApplicationEvents` dispinterface.

The example is from the [ATLEventHandling sample](#).

```
[ uuid(000209F7-0000-0000-C000-000000000046), hidden ]
dispinterface ApplicationEvents {
properties:
methods:
    [id(0x00000001), restricted, hidden]
    void Startup();

    [id(0x00000002)]
    void Quit();

    [id(0x00000003)]
    void DocumentChange();
};
```

The example uses `#import` to generate the required header files from Word's type library. If you want to use this example with other versions of Word, you must specify the correct mso dll file. For example, Office 2000 provides mso9.dll and OfficeXP provides mso.dll. This code is simplified from stdafx.h:

```
#pragma warning (disable : 4146)

// Paths to required MS OFFICE files (replace "MSO.DLL" and "MSWORD.OLB" with the actual paths to those
files...)
#define _MSDLL_PATH "MSO.DLL"
// Delete the *.tlh files when changing import qualifiers
#import _MSDLL_PATH rename("RGB", "MSRGB") rename("DocumentProperties", "WordDocumentProperties")
raw_interfaces_only

#import "C:\Program Files\Common Files\Microsoft Shared\VBA\VBA6\VBE6EXT.OLB" raw_interfaces_only

#define _MSWORDOLB_PATH "MSWORD.OLB"
#import _MSWORDOLB_PATH rename("ExitWindows", "WordExitWindows") rename("FindText", "WordFindText")
raw_interfaces_only

#pragma warning (default : 4146)
```

The following code appears in NotSoSimple.h. The relevant code is noted by comments:

```
// Note #import doesn't generate a LIBID (because we don't use 'named_guids')
// so we have to do it manually
namespace Word
{
    struct __declspec(uuid("00020905-0000-0000-C000-000000000046"))
        /* library */ Library;
};

class ATL_NO_VTABLE CNotSoSimple :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CNotSoSimple, &CLSID_NotSoSimple>,
    public IDispatchImpl<ISwitch, &IID_ISwitch, &LIBID_ATLEVENTHANDLINGLib>,
    // Note inheritance from IDispatchImpl
    public IDispatchImpl</*nID*/ 1, CNotSoSimple,
        &__uuidof(Word::ApplicationEvents2),
        &__uuidof(Word::Library), /*wMajor*/ 8, /*wMinor*/ 1>

{
public:
    CNotSoSimple()
    {
    }

DECLARE_REGISTRY_RESOURCEID(IDR_NOTSOSIMPLE)

DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CNotSoSimple)
    COM_INTERFACE_ENTRY(ISwitch)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

    CComPtr<Word::_Application> m_pApp;

    // Event handlers
    // Note the __stdcall calling convention and
    // dispinterface-style signature
    void __stdcall OnQuit()
    {
        Stop();
    }

    void __stdcall OnDocChange()
    {
        ATLASSERT(m_pApp != NULL);

        // Get a pointer to the _Document interface on the active document
        CComPtr<Word::_Document> pDoc;
        m_pApp->get_ActiveDocument(&pDoc);

        // Get the name from the active document
        CComBSTR bstrName;
        if (pDoc)
            pDoc->get_Name(&bstrName);

        // Create a display string
        CComBSTR bstrDisplay(_T("New document title:\n"));
        bstrDisplay += bstrName;

        // Display the name to the user
        USES_CONVERSION;
        MessageBox(NULL, W2CT(bstrDisplay), _T("IDispatchImpl : Active Document Changed"), MB_OK);
    }

    // Note the mapping from Word events to our event handler functions.
BEGIN_SINK_MAP(CNotSoSimple)
```

```

        SINK_ENTRY_EX(*nID =*/ 1, __uuidof(Word::ApplicationEvents2), /*dispid =*/ 3, OnDocChange)
        SINK_ENTRY_EX(*nID =*/ 1, __uuidof(Word::ApplicationEvents2), /*dispid =*/ 2, OnQuit)
    END_SINK_MAP()

// ISwitch
public:

    STDMETHOD(Start)()
    {
        // If we already have an object, just return
        if (m_pApp)
            return S_OK;

        // Create an instance of Word's Application object
        HRESULT hr = m_pApp.CoCreateInstance(__uuidof(Word::Application), NULL, CLSCTX_SERVER);
        if (FAILED(hr))
            return hr;

        ATLASSERT(m_pApp != NULL);

        // Make the Word user interface visible
        m_pApp->put_Visible(true);

        // Note call to advise
        // Forge a connection to enable us to receive events
        DispEventAdvise(m_pApp);

        return S_OK;
    }

    STDMETHOD(Stop)()
    {
        // Check we have an object to unadvise on
        if (!m_pApp)
            return S_OK;

        // Note call to unadvise
        // Break the connection with the event source
        DispEventUnadvise(m_pApp);

        // Release the Word application
        m_pApp.Release();

        return S_OK;
    }
};

```

See also

[Event Handling](#)

[ATLEventHandling Sample](#)

Using IDispEventSimpleImpl

4/1/2019 • 2 minutes to read • [Edit Online](#)

When using `IDispEventSimpleImpl` to handle events, you will need to:

- Derive your class from `IDispEventSimpleImpl`.
- Add an event sink map to your class.
- Define `_ATL_FUNC_INFO` structures describing the events.
- Add entries to the event sink map using the `SINK_ENTRY_INFO` macro.
- Implement the methods that you're interested in handling.
- Advise and unadvise the event source.

Example

The example below shows you how to handle the `DocumentChange` event fired by Word's **Application** object. This event is defined as a method on the `ApplicationEvents` dispinterface.

The example is from the [ATLEventHandling sample](#).

```
[ uuid(000209F7-0000-0000-C000-000000000046), hidden ]
dispinterface ApplicationEvents {
properties:
methods:
    [id(0x00000001), restricted, hidden]
    void Startup();

    [id(0x00000002)]
    void Quit();

    [id(0x00000003)]
    void DocumentChange();
};
```

The example uses `#import` to generate the required header files from Word's type library. If you want to use this example with other versions of Word, you must specify the correct mso dll file. For example, Office 2000 provides `mso9.dll` and OfficeXP provides `mso.dll`. This code is simplified from `stdafx.h`:


```
#pragma warning (disable : 4146)

// Paths to required MS OFFICE files (replace "MSO.DLL" and "MSWORD.OLB" with the actual paths to those
files...)
#define _MSDLL_PATH "MSO.DLL"
// Delete the *.tlh files when changing import qualifiers
#import _MSDLL_PATH rename("RGB", "MSRGB") rename("DocumentProperties", "WordDocumentProperties")
raw_interfaces_only

#import "C:\Program Files\Common Files\Microsoft Shared\VBA\VBA6\VBE6EXT.OLB" raw_interfaces_only

#define _MSWORDOLB_PATH "MSWORD.OLB"
#import _MSWORDOLB_PATH rename("ExitWindows", "WordExitWindows") rename("FindText", "WordFindText")
raw_interfaces_only

#pragma warning (default : 4146)
```

The only information from the type library actually used in this example is the CLSID of the Word `Application` object and the IID of the `ApplicationEvents` interface. This information is only used at compile time.

The following code appears in Simple.h. The relevant code is noted by comments:

```
// Note declaration of structure for type information
extern _ATL_FUNC_INFO OnDocChangeInfo;
extern _ATL_FUNC_INFO OnQuitInfo;

class ATL_NO_VTABLE CSimple :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CSimple, &CLSID_Simple>,
public IDispatchImpl<ISwitch, &IID_ISwitch, &LIBID_ATLEVENTHANDLINGLib>,
// Note inheritance from IDispEventSimpleImpl
public IDispEventSimpleImpl</*nID =*/ 1, CSimple, &__uuidof(Word::ApplicationEvents)>
{
public:
    CSimple()
    {
    }

    DECLARE_REGISTRY_RESOURCEID(IDR_SIMPLE)

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    BEGIN_COM_MAP(CSimple)
        COM_INTERFACE_ENTRY(ISwitch)
        COM_INTERFACE_ENTRY(IDispatch)
    END_COM_MAP()

    CComPtr<Word::_Application> m_pApp;

    // Event handlers
    // Note the __stdcall calling convention and
    // dispinterface-style signature
    void __stdcall OnQuit()
    {
        Stop();
    }

    void __stdcall OnDocChange()
    {
        ATLASSERT(m_pApp != NULL);

        // Get a pointer to the _Document interface on the active document
        CComPtr<Word::_Document> pDoc;
        m_pApp->get_ActiveDocument(&pDoc);

        // Get the name from the active document
```

```

// Get the name from the active document
CComBSTR bstrName;
if (pDoc)
    pDoc->get_Name(&bstrName);

// Create a display string
CComBSTR bstrDisplay(_T("New document title:\n"));
bstrDisplay += bstrName;

// Display the name to the user
USES_CONVERSION;
MessageBox(NULL, W2CT(bstrDisplay), _T("IDispEventSimpleImpl : Active Document Changed"), MB_OK);
}

// Note the mapping from Word events to our event handler functions.
BEGIN_SINK_MAP(CSimple)
    SINK_ENTRY_INFO(*nID =*/ 1, __uuidof(Word::ApplicationEvents), /*dispid =*/ 3, OnDocChange,
&OnDocChangeInfo)
    SINK_ENTRY_INFO(*nID =*/ 1, __uuidof(Word::ApplicationEvents), /*dispid =*/ 2, OnQuit, &OnQuitInfo)
END_SINK_MAP()

// ISwitch
public:
    STDMETHOD(Start)()
    {
        // If we already have an object, just return
        if (m_pApp)
            return S_OK;

        // Create an instance of Word's Application object
        HRESULT hr = m_pApp.CoCreateInstance(__uuidof(Word::Application), NULL, CLSCTX_SERVER);
        if (FAILED(hr))
            return hr;

        ATLASSTERT(m_pApp != NULL);

        // Make the Word user interface visible
        m_pApp->put_Visible(true);

        // Note call to advise
        // Forge a connection to enable us to receive events
        DispEventAdvise(m_pApp);

        return S_OK;
    }

    STDMETHOD(Stop)()
    {
        // Check we have an object to unadvise on
        if (!m_pApp)
            return S_OK;

        // Note call to unadvise
        // Break the connection with the event source
        DispEventUnadvise(m_pApp);

        // Release the Word application
        m_pApp.Release();

        return S_OK;
    }
};

```

The following code is from Simple.cpp:

```
// Define type info structure
_ATL_FUNC_INFO OnDocChangeInfo = {CC_STDCALL, VT_EMPTY, 0};
_ATL_FUNC_INFO OnQuitInfo = {CC_STDCALL, VT_EMPTY, 0};
// (don't actually need two structure since they're the same)
```

See also

[Event Handling](#)

[ATLEventHandling Sample](#)

ATL Event Handling Summary

3/4/2019 • 2 minutes to read • [Edit Online](#)

In general, handling COM events is a relatively simple process. There are three main steps:

- Implement the event interface on your object.
- Advise the event source that your object wants to receive events.
- Unadvise the event source when your object no longer needs to receive events.

Implementing the Interface

There are four main ways of implementing an interface using ATL.

DERIVE FROM	SUITABLE FOR INTERFACE TYPE	REQUIRES YOU TO IMPLEMENT ALL METHODS*	REQUIRES A TYPE LIBRARY AT RUN TIME
The interface	Vtable	Yes	No
IDispatchImpl	Dual	Yes	Yes
IDispEventImpl	Dispinterface	No	Yes
IDispEventSimpleImpl	Dispinterface	No	No

* When using ATL support classes, you are never required to implement the `IUnknown` or `IDispatch` methods manually.

Advising and Unadvising the Event Source

There are three main ways of advising and unadvising an event source using ATL.

ADVISE FUNCTION	UNADVISE FUNCTION	MOST SUITABLE FOR USE WITH	REQUIRES YOU TO KEEP TRACK OF A COOKIE	COMMENTS
AtlAdvise , CComPtrBase::Advise	AtlUnadvise	Vtable or dual interfaces	Yes	AtlAdvise is a global ATL function. CComPtrBase::Advise is used by CComPtr and CComQIPtr .
IDispEventSimpleImpl::DispEventAdvise	IDispEventSimpleImpl::DispEventUnadvise	IDispEventImpl or IDispEventSimpleImpl	No	Fewer parameters than AtlAdvise since the base class does more work.
CComCompositeControl::AdviseSinkMap(TRUE)	CComCompositeControl::AdviseSinkMap(FALSE)	ActiveX controls in Composite controls	No	CComCompositeControl::Advise advises all entries in the event sink map. The same function unadvise the entries. This method is called automatically by the CComCompositeControl class.

ADVISE FUNCTION	UNADVISE FUNCTION	MOST SUITABLE FOR USE WITH	REQUIRES YOU TO KEEP TRACK OF A COOKIE	COMMENTS
CAxDialogImpl::AdviseSinkMap(TRUE)	CAxDialogImpl::AdviseSinkMap(FALSE)	ActiveX controls in a dialog box	No	CAxDialogImpl::AdviseSinkMap advises and unadvises all ActiveX controls in the dialog resource. This is done automatically for you.

See also

[Event Handling](#)

[Supporting IDispatchImpl](#)

ATL and the Free Threaded Marshaler

3/4/2019 • 2 minutes to read • [Edit Online](#)

The ATL Simple Object Wizard's Attributes page provides an option that allows your class to aggregate the free threaded marshaler (FTM).

The wizard generates code to create an instance of the free threaded marshaler in `FinalConstruct` and release that instance in `FinalRelease`. A `COM_INTERFACE_ENTRY_AGGREGATE` macro is automatically added to the COM map to ensure that `QueryInterface` requests for `IMarshal` are handled by the free threaded marshaler.

The free threaded marshaler allows direct access to interfaces on your object from any thread in the same process, speeding up cross-apartment calls. This option is intended for classes that use the Both threading model.

When using this option, classes must take responsibility for the thread-safety of their data. In addition, objects that aggregate the free threaded marshaler and need to use interface pointers obtained from other objects must take extra steps to ensure that the interfaces are correctly marshaled. Typically this involves storing the interface pointers in the global interface table (GIT) and getting the pointer from the GIT each time it is used. ATL provides the class `CComGITPtr` to help you use interface pointers stored in the GIT.

See also

[Concepts](#)

[CoCreateFreeThreadedMarshaler](#)

[IMarshal](#)

[When to Use the Global Interface Table](#)

[In-Process Server Threading Issues](#)

Specifying the Threading Model for a Project (ATL)

3/4/2019 • 2 minutes to read • [Edit Online](#)

The following macros are available to specify the threading model of an ATL project:

MACRO	GUIDELINES FOR USING
<code>_ATL_SINGLE_THREADED</code>	Define if all of your objects use the single threading model.
<code>_ATL_APARTMENT_THREADED</code>	Define if one or more of your objects use apartment threading.
<code>_ATL_FREE_THREADED</code>	Define if one or more of your objects use free or neutral threading. Existing code may contain references to the equivalent macro _ATL_MULTI_THREADED .

If you do not define any of these macros for your project, `_ATL_FREE_THREADED` will be in effect.

The macros affect run-time performance as follows:

- Specifying the macro that corresponds to the objects in your project can improve run-time performance.
- Specifying a higher level of macro, for example if you specify `_ATL_APARTMENT_THREADED` when all of your objects are single threaded, will slightly degrade run-time performance.
- Specifying a lower level of macro, for example, if you specify `_ATL_SINGLE_THREADED` when one or more of your objects use apartment threading or free threading, can cause your application to fail at run time.

See [Options, ATL Simple Object Wizard](#) for a description of the threading models available for an ATL object.

See also

[Concepts](#)

ATL Module Classes

3/4/2019 • 2 minutes to read • [Edit Online](#)

This topic discusses the module classes that were new in ATL 7.0.

CComModule Replacement Classes

Earlier versions of ATL used `CComModule`. In ATL 7.0, `CComModule` functionality is replaced by several classes:

- [CAtlBaseModule](#) Contains information required by most applications that use ATL. Contains the HINSTANCE of the module and the resource instance.
- [CAtlComModule](#) Contains information required by the COM classes in ATL.
- [CAtlWinModule](#) Contains information required by the windowing classes in ATL.
- [CAtlDebugInterfacesModule](#) Contains support for interface debugging.
- [CAtlModule](#) The following `CAtlModule`-derived classes are customized to contain information required in a particular application type. Most members in these classes can be overridden:
 - [CAtlDllModuleT](#) Used in DLL applications. Provides code for the standard exports.
 - [CAtlExeModuleT](#) Used in EXE applications. Provides code required in an EXE.
 - [CAtlServiceModuleT](#) Provides support to create Windows NT and Windows 2000 Services.

`CComModule` is still available for backward compatibility.

Reasons for Distributing CComModule Functionality

The functionality of `CComModule` was distributed into several new classes for the following reasons:

- Make the functionality in `CComModule` granular.

Support for COM, windowing, interface debugging, and application-specific (DLL or EXE) features is now in separate classes.

- Automatically declare global instance of each of these modules.

A global instance of the required module classes is linked into the project.

- Remove the necessity of calling Init and Term methods.

Init and Term methods have moved into the constructors and destructors for the module classes; there is no longer a need to call Init and Term.

See also

[Concepts](#)

[Class Overview](#)

ATL Services

3/12/2019 • 2 minutes to read • [Edit Online](#)

To create your ATL COM object so that it runs in a service, simply select Service (EXE) from the list of server options in the ATL Project Wizard. The wizard will then create a class derived from `CAtlServiceModuleT` to implement the service.

When the ATL COM object is built as a service, it will only be registered as a local server, and it will not appear in the list of services in Control Panel. This is because it is easier to debug the service as a local server than as a service. To install it as a service, run the following at the command prompt:

```
YourEXE .exe /Service
```

To uninstall it, run the following:

```
YourEXE .exe /UnregServer
```

The first four topics in this section discuss the actions that occur during execution of `CAtlServiceModuleT` member functions. These topics appear in the same sequence as the functions are typically called. To improve your understanding of these topics, it is a good idea to use the source code generated by the ATL Project Wizard as a reference. These first four topics are:

- [The CAtlServiceModuleT::Start Function](#)
- [The CAtlServiceModuleT::ServiceMain Function](#)
- [The CAtlServiceModuleT::Run Function](#)
- [The CAtlServiceModuleT::Handler Function](#)

The last three topics discuss concepts related to developing a service:

- [Registry Entries](#) for ATL services
- [DCOMCNFG](#)
- [Debugging Tips](#) for ATL services

See also

[Concepts](#)

CAtlServiceModuleT::Start Function

3/12/2019 • 2 minutes to read • [Edit Online](#)

When the service is run, `_tWinMain` calls `CAtlServiceModuleT::WinMain`, which in turn calls `CAtlServiceModuleT::Start`.

`CAtlServiceModuleT::Start` sets up an array of `SERVICE_TABLE_ENTRY` structures that map each service to its startup function. This array is then passed to the Win32 API function, [StartServiceCtrlDispatcher](#). In theory, one EXE could handle multiple services and the array could have multiple `SERVICE_TABLE_ENTRY` structures. Currently, however, an ATL-generated service supports only one service per EXE. Therefore, the array has a single entry that contains the service name and `_ServiceMain` as the startup function. `_ServiceMain` is a static member function of `CAtlServiceModuleT` that calls the non-static member function, `ServiceMain`.

NOTE

Failure of `StartServiceCtrlDispatcher` to connect to the service control manager (SCM) probably means that the program is not running as a service. In this case, the program calls `CAtlServiceModuleT::Run` directly so that the program can run as a local server. For more information about running the program as a local server, see [Debugging Tips](#).

See also

[Services](#)

[CAtlServiceModuleT::Start](#)

CAtlServiceModuleT::ServiceMain Function

3/12/2019 • 2 minutes to read • [Edit Online](#)

The service control manager (SCM) calls `ServiceMain` when you open the Services Control Panel application, select the service, and click **Start**.

After the SCM calls `ServiceMain`, a service must give the SCM a handler function. This function lets the SCM obtain the service's status and pass specific instructions (such as pausing or stopping). The SCM gets this function when the service passes `_Handler` to the Win32 API function, [RegisterServiceCtrlHandler](#). (`_Handler` is a static member function that calls the non-static member function [Handler](#).)

At startup, a service should also inform the SCM of its current status. It does this by passing `SERVICE_START_PENDING` to the Win32 API function, [SetServiceStatus](#).

`ServiceMain` then calls `CAtlExeModuleT::InitializeCom`, which calls the Win32 API function [CoInitializeEx](#). By default, `InitializeCom` passes the `COINIT_MULTITHREADED` flag to the function. This flag indicates that the program is to be a free-threaded server.

Now, `CAtlServiceModuleT::Run` is called to perform the main work of the service. `Run` continues to execute until the service is stopped.

See also

[Services](#)

[CAtlServiceModuleT::ServiceMain](#)

CAtlServiceModuleT::Run Function

3/12/2019 • 2 minutes to read • [Edit Online](#)

`Run` contains calls to `PreMessageLoop`, `RunMessageLoop`, and `PostMessageLoop`. After being called, `PreMessageLoop` first stores the service's thread ID. The service will use this ID to close itself by sending a WM_QUIT message using the Win32 API function, [PostThreadMessage](#).

`PreMessageLoop` then calls `InitializeSecurity`. By default, `InitializeSecurity` calls [CoInitializeSecurity](#) with the security descriptor set to NULL, which means that any user has access to your object.

If you do not want the service to specify its own security, override `PreMessageLoop` and don't call `InitializeSecurity`, and COM will then determine the security settings from the registry. A convenient way to configure registry settings is with the [DCOMCNFG](#) utility discussed later in this section.

Once security is specified, the object is registered with COM so that new clients can connect to the program. Finally, the program tells the service control manager (SCM) that it is running and the program enters a message loop. The program remains running until it posts a quit message upon service shutdown.

See also

[Services](#)

[CSecurityDesc Class](#)

[CSid Class](#)

[CDacl Class](#)

[CAtlServiceModuleT::Run](#)

CAtlServiceModuleT::Handler Function

3/12/2019 • 2 minutes to read • [Edit Online](#)

`CAtlServiceModuleT::Handler` is the routine that the service control manager (SCM) calls to retrieve the status of the service and give it various instructions (such as stopping or pausing). The SCM passes an operation code to `Handler` to indicate what the service should do. A default ATL-generated service only handles the stop instruction. If the SCM passes the stop instruction, the service tells the SCM that the program is about to stop. The service then calls `PostThreadMessage` to post a quit message to itself. This terminates the message loop and the service will ultimately close.

To handle more instructions, you need to change the `m_status` data member initialized in the `CAtlServiceModuleT` constructor. This data member tells the SCM which buttons to enable when the service is selected in the Services Control Panel application.

See also

[Services](#)

[CAtlServiceModuleT::Handler](#)

Registry Entries

3/4/2019 • 2 minutes to read • [Edit Online](#)

DCOM introduced the concept of Application IDs (AppIDs), which group configuration options for one or more DCOM objects into a centralized location in the registry. You specify an AppID by indicating its value in the AppID named value under the object's CLSID.

By default, an ATL-generated service uses its CLSID as the GUID for its AppID. Under `HKEY_CLASSES_ROOT\AppID`, you can specify DCOM-specific entries. Initially, two entries exist:

- `LocalService`, with a value equal to the name of the service. If this value exists, it is used instead of the `LocalServer32` key under the CLSID.
- `ServiceParameters`, with a value equal to `-Service`. This value specifies parameters that will be passed to the service when it is started. Note that these parameters are passed to the service's `ServiceMain` function, not `WinMain`.

Any DCOM service also needs to create another key under `HKEY_CLASSES_ROOT\AppID`. This key is equal to the name of the EXE and acts as a cross-reference, as it contains an AppID value pointing back to the AppID entries.

See also

[Services](#)

DCOMCNFG

3/12/2019 • 2 minutes to read • [Edit Online](#)

DCOMCNFG is a Windows NT 4.0 utility that allows you to configure various DCOM-specific settings in the registry. The DCOMCNFG window has three pages: Default Security, Default Properties, and Applications. Under Windows 2000 a fourth page, Default Protocols, is present.

Default Security Page

You can use the Default Security page to specify default permissions for objects on the system. The Default Security page has three sections: Access, Launch, and Configuration. To change a section's defaults, click the corresponding **Edit Default** button. These Default Security settings are stored in the registry under

`HKEY_LOCAL_MACHINE\Software\Microsoft\OLE`.

Default Protocols Page

This page lists the set of network protocols available to DCOM on this machine. The order reflects the priority in which they will be used; the first in the list has the highest priority. Protocols can be added or deleted from this page.

Default Properties Page

On the Default Properties page, you must select the **Enable Distributed COM on this computer** check box if you want clients on other machines to access COM objects running on this machine. Selecting this option sets the

`HKEY_LOCAL_MACHINE\Software\Microsoft\OLE\EnableDCOM` value to `Y`.

Applications Page

You change the settings for a particular object with the Applications page. Simply select the application from the list and click the **Properties** button. The Properties window has five pages:

- The General page confirms the application you are working with.
- The Location page allows you to specify where the application should run when a client calls `CoCreateInstance` on the relevant CLSID. If you select the **Run application on the following computer** check box and enter a computer name, then a `RemoteServerName` value is added under the AppID for that application. Clearing the **Run application on this computer** check box renames the `LocalService` value to `_LocalService` and, thereby, disables it.
- The Security page is similar to the Default Security page found in the DCOMCNFG window, except that these settings apply only to the current application. Again, the settings are stored under the AppID for that object.
- The Identify page identifies which user is used to run the application.
- The Endpoints page lists the set of protocols and endpoints available for use by clients of the selected DCOM server.

See also

[Services](#)

Debugging Tips

3/4/2019 • 2 minutes to read • [Edit Online](#)

The following topics outline some useful steps for debugging your service:

- [Using Task Manager](#)
- [Displaying Assertions](#)
- [Running the Program as a Local Server](#)

See also

[Services](#)

Using Task Manager

3/4/2019 • 2 minutes to read • [Edit Online](#)

One of the simplest ways to debug a service is through the use of the Task Manager. While the service is running, start the Task Manager and click the **Processes** tab. Right-click the name of the EXE and then click **Debug**. This launches Visual C++ attached to that running process. Now, click **Break** on the **Debug** menu to allow you to set breakpoints in your code. Click **Run** to run to your selected breakpoints.

See also

[Debugging Tips](#)

Displaying Assertions

5/7/2019 • 2 minutes to read • [Edit Online](#)

If the client connected to your service appears to stop responding, the service may have asserted and displayed a message box that you are not able to see. You can confirm this by using the Visual Studio debugger to debug your code (see [Using Task Manager](#) earlier in this section).

If you determine that your service is displaying a message box that you cannot see, you may want to set the **Allow Service to Interact with Desktop** option before using the service again. This option is a startup parameter that permits any message boxes displayed by the service to appear on the desktop. To set this option, open the Services Control Panel application, select the service, click **Startup**, and then select the **Allow Service to Interact with Desktop** option.

See also

[Debugging Tips](#)

Running the Program as a Local Server

3/4/2019 • 2 minutes to read • [Edit Online](#)

If running the program as a service is inconvenient, you can temporarily change the registry so that the program is run as a normal local server. Simply rename the `LocalService` value under your AppID to `_LocalService` and ensure the `LocalServer32` key under your CLSID is set correctly. (Note that using DCOMCNFG to specify that your application should be run on a different computer renames your `LocalServer32` key to `_LocalServer32`.) Running your program as a local server takes a few more seconds on startup because the call to `StartServiceCtrlDispatcher` in `CAt1ServiceModuleT::Start` takes a few seconds before it fails.

See also

[Debugging Tips](#)

ATL Window Classes

3/4/2019 • 2 minutes to read • [Edit Online](#)

ATL includes several classes that allow you to use and implement windows. These classes, like other ATL classes, provide an efficient implementation that does not impose an overhead on your code.

This section describes the ATL window classes and explains how to use them.

In This Section

[Introduction to ATL Window Classes](#)

Briefly describes each ATL window class and provides links to the reference material on them.

[Using a Window](#)

Discusses how to use `CWindow` to manipulate a window.

[Implementing a Window](#)

Discusses message handlers, message maps, and using `CWindowImpl`. Includes details on superclassing and subclassing.

[Implementing a Dialog Box](#)

Discusses the two methods for adding a dialog box class and shows a code sample.

[Using Contained Windows](#)

Discusses contained windows in ATL, which are windows that delegate their messages to a container object instead of handling them in their own class.

[Understanding Window Traits](#)

Discusses window traits classes in ATL. These classes provide a simple method for standardizing the styles used for the creation of a window object.

Related Sections

[ATL](#)

Provides links to conceptual topics on how to program using the Active Template Library.

[Windows Support Classes](#)

Lists additional ATL classes that support windows and message maps in ATL.

Introduction to ATL Window Classes

3/4/2019 • 2 minutes to read • [Edit Online](#)

The following ATL classes are designed to implement and manipulate windows:

- [CWindow](#) allows you to attach a window handle to the `CWindow` object. You then call `CWindow` methods to manipulate the window.
- [CWindowImpl](#) allows you to implement a new window and process messages with a message map. You can create a window based on a new Windows class, superclass an existing class, or subclass an existing window.
- [CDialogImpl](#) allows you to implement a modal or a modeless dialog box and process messages with a message map.
- [CContainedWindowT](#) is a prebuilt class that implements a window whose message map is contained in another class. Using `CContainedWindowT` allows you to centralize message processing in one class.
- [CAxDialogImpl](#) allows you to implement a dialog box (modal or modeless) that hosts ActiveX controls.
- [CSimpleDialog](#) allows you to implement a modal dialog box with basic functionality.
- [CAxWindow](#) allows you to implement a window that hosts an ActiveX control.
- [CAxWindow2T](#) allows you to implement a window that hosts a licensed ActiveX control.

In addition to specific window classes, ATL provides several classes designed to make the implementation of an ATL window object easier. They are as follows:

- [CWndClassInfo](#) manages the information of a new window class.
- [CWinTraits](#) and [CWinTraitsOR](#) provide a simple method of standardizing the traits of an ATL window object.

See also

[Window Classes](#)

Using a Window

3/4/2019 • 2 minutes to read • [Edit Online](#)

Class [CWindow](#) allows you to use a window. Once you attach a window to a `CWindow` object, you can then call `CWindow` methods to manipulate the window. `CWindow` also contains an `HWND` operator to convert a `CWindow` object to an `HWND`. Thus you can pass a `CWindow` object to any function that requires a handle to a window. You can easily mix `CWindow` method calls and Win32 function calls, without creating any temporary objects.

Because `CWindow` has only two data member (a window handle and the default dimensions), it does not impose an overhead on your code. In addition, many of the `CWindow` methods simply wrap corresponding Win32 API functions. By using `CWindow`, the `HWND` member is automatically passed to the Win32 function.

In addition to using `CWindow` directly, you can also derive from it to add data or code to your class. ATL itself derives three classes from `CWindow`: [CWindowImpl](#), [CDialogImpl](#), and [CContainedWindowT](#).

See also

[Window Classes](#)

Implementing a Window

3/4/2019 • 2 minutes to read • [Edit Online](#)

Class [CWindowImpl](#) allows you to implement a window and handle its messages. Message handling in ATL is based on a message map. This section explains:

- How to [add a message handler](#) to a control.
- What [message maps](#) are and how to use them.
- The syntax for [message handler functions](#).
- How to [implement a window with CWindowImpl](#).

See also

[Window Classes](#)

Adding an ATL Message Handler

3/4/2019 • 2 minutes to read • [Edit Online](#)

To add a message handler (a member function that handles Windows messages) to a control, first select the control in the Class View. Then open the **Properties** window, select the **Messages** icon, and click the drop-down control in the box opposite the required message. This will add a declaration for the message handler in the control's header file and a skeleton implementation of the handler in the control's .cpp file. It will also add the message map and add an entry for the handler.

Adding a message handler in ATL is similar to adding a message handler to an MFC class. See [Adding an MFC Message Handler](#) for more information.

The following conditions apply only to adding an ATL message handler:

- The message handlers follow the same naming convention as MFC.
- The new message map entries are added into the main message map. The wizard does not recognize alternate message maps and chaining.

See also

[Implementing a Window](#)

Message Maps (ATL)

3/4/2019 • 2 minutes to read • [Edit Online](#)

A message map associates a handler function with a particular message, command, or notification. By using ATL's [message map macros](#), you can specify a message map for a window. The window procedures in `CWindowImpl`, `CDialogImpl`, and `CContainedWindowT` direct a window's messages to its message map.

The [message handler functions](#) accept an additional argument of type `BOOL&`. This argument indicates whether a message has been processed, and it is set to TRUE by default. A handler function can then set the argument to FALSE to indicate that it has not handled a message. In this case, ATL will continue to look for a handler function further in the message map. By setting this argument to FALSE, you can first perform some action in response to a message and then allow the default processing or another handler function to finish handling the message.

Chained Message Maps

ATL also allows you to chain message maps, which directs the message handling to a message map defined in another class. For example, you can implement common message handling in a separate class to provide uniform behavior for all windows chaining to that class. You can chain to a base class or to a data member of your class.

ATL also supports dynamic chaining, which allows you to chain to another object's message map at run time. To implement dynamic chaining, you must derive your class from [CDynamicChain](#). Then declare the [CHAIN_MSG_MAP_DYNAMIC](#) macro in your message map. CHAIN_MSG_MAP_DYNAMIC requires a unique number that identifies the object and the message map to which you are chaining. You must define this unique value through a call to `CDynamicChain::SetChainEntry`.

You can chain to any class that declares a message map, provided the class derives from [CMessageMap](#).

`CMessageMap` allows an object to expose its message maps to other objects. Note that `CWindowImpl` already derives from `CMessageMap`.

Alternate Message Maps

Finally, ATL supports alternate message maps, declared with the [ALT_MSG_MAP](#) macro. Each alternate message map is identified by a unique number, which you pass to ALT_MSG_MAP. Using alternate message maps, you can handle the messages of multiple windows in one map. Note that by default, `CWindowImpl` does not use alternate message maps. To add this support, override the `WindowProc` method in your `CWindowImpl`-derived class and call `ProcessWindowMessage` with the message map identifier.

See also

[Implementing a Window](#)

Message Handler Functions

3/12/2019 • 2 minutes to read • [Edit Online](#)

ATL provides three types of message handler functions:

TYPE OF MESSAGE HANDLER	CORRESPONDING MESSAGE MACRO
MessageHandler	MESSAGE_HANDLER
CommandHandler	COMMAND_HANDLER
NotifyHandler	NOTIFY_HANDLER

See also

[Implementing a Window](#)

[Message Maps](#)

[WM_NOTIFY](#)

CommandHandler

3/12/2019 • 2 minutes to read • [Edit Online](#)

`CommandHandler` is the function identified by the third parameter of the `COMMAND_HANDLER` macro in your message map.

Syntax

```
LRESULT CommandHandler(  
    WORD wNotifyCode,  
    WORD wID,  
    HWND hWndCtl,  
    BOOL& bHandled);
```

Parameters

wNotifyCode

The notification code.

wID

The identifier of the menu item, control, or accelerator.

hWndCtl

A handle to a window control.

bHandled

The message map sets *bHandled* to TRUE before `CommandHandler` is called. If `CommandHandler` does not fully handle the message, it should set *bHandled* to FALSE to indicate the message needs further processing.

Return Value

The result of message processing. 0 if successful.

Remarks

For an example of using this message handler in a message map, see [COMMAND_HANDLER](#).

See also

[Implementing a Window](#)

[Message Maps](#)

[WM_NOTIFY](#)

MessageHandler

3/12/2019 • 2 minutes to read • [Edit Online](#)

`MessageHandler` is the name of the function identified by the second parameter of the `MESSAGE_HANDLER` macro in your message map.

Syntax

```
LRESULT MessageHandler(  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam,  
    BOOL& bHandled);
```

Parameters

uMsg

Specifies the message.

wParam

Additional message-specific information.

lParam

Additional message-specific information.

bHandled

The message map sets *bHandled* to TRUE before `MessageHandler` is called. If `MessageHandler` does not fully handle the message, it should set *bHandled* to FALSE to indicate the message needs further processing.

Return Value

The result of message processing. 0 if successful.

Remarks

For an example of using this message handler in a message map, see [MESSAGE_HANDLER](#).

See also

[Implementing a Window](#)

[Message Maps](#)

[WM_NOTIFY](#)

NotifyHandler

3/12/2019 • 2 minutes to read • [Edit Online](#)

The name of the function identified by the third parameter of the NOTIFY_HANDLER macro in your message map.

Syntax

```
LRESULT NotifyHandler(  
    int idCtrl,  
    LPMHDR pnmh,  
    BOOL& bHandled);
```

Parameters

idCtrl

The identifier of the control sending the message.

pnmh

Address of an [NMHDR](#) structure that contains the notification code and additional information. For some notification messages, this parameter points to a larger structure that has the [NMHDR](#) structure as its first member.

bHandled

The message map sets *bHandled* to TRUE before *NotifyHandler* is called. If *NotifyHandler* does not fully handle the message, it should set *bHandled* to **FALSE** to indicate the message needs further processing.

Return Value

The result of message processing. 0 if successful.

Remarks

For an example of using this message handler in a message map, see [NOTIFY_HANDLER](#)).

See also

[Implementing a Window](#)

[Message Maps](#)

[WM_NOTIFY](#)

Implementing a Window with CWindowImpl

3/12/2019 • 2 minutes to read • [Edit Online](#)

To implement a window, derive a class from `CWindowImpl`. In your derived class, declare a message map and the message handler functions. You can now use your class in three different ways:

- [Create a window based on a new Windows class](#)
- [Superclass an existing Windows class](#)
- [Subclass an existing window](#)

Creating a Window Based on a New Windows Class

`CWindowImpl` contains the `DECLARE_WND_CLASS` macro to declare Windows class information. This macro implements the `GetWndClassInfo` function, which uses `CWndClassInfo` to define the information of a new Windows class. When `CWindowImpl::Create` is called, this Windows class is registered and a new window is created.

NOTE

`CWindowImpl` passes NULL to the `DECLARE_WND_CLASS` macro, which means ATL will generate a Windows class name. To specify your own name, pass a string to `DECLARE_WND_CLASS` in your `CWindowImpl`-derived class.

Example

Following is an example of a class that implements a window based on a new Windows class:

```
class CMyCustomWnd : public CWindowImpl<CMyCustomWnd>
{
public:
    // Optionally specify name of the new Windows class
    DECLARE_WND_CLASS(_T("MyName"))
    // If this macro is not specified in your
    // class, ATL will generate a class name

    BEGIN_MSG_MAP(CMyCustomWnd)
        MESSAGE_HANDLER(WM_PAINT, OnPaint)
    END_MSG_MAP()

    LRESULT OnPaint(UINT /*nMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
        BOOL& /*bHandled*/)
    {
        // Do some painting code
        return 0;
    }
};
```

To create a window, create an instance of `CMyWindow` and then call the `Create` method.

NOTE

To override the default Windows class information, implement the `GetWndClassInfo` method in your derived class by setting the `CWndClassInfo` members to the appropriate values.

Superclassing an Existing Windows Class

The `DECLARE_WND_SUPERCLASS` macro allows you to create a window that superclasses an existing Windows class. Specify this macro in your `CWindowImpl`-derived class. Like any other ATL window, messages are handled by a message map.

When you use `DECLARE_WND_SUPERCLASS`, a new Windows class will be registered. This new class will be the same as the existing class you specify, but will replace the window procedure with `CWindowImpl::WindowProc` (or with your function that overrides this method).

Example

Following is an example of a class that superclasses the standard Edit class:

```
class CMyEdit : public CWindowImpl<CMyEdit>
{
public:
    // "Edit" is the name of the standard Windows class.
    // "MyEdit" is the name of the new Windows class
    // that will be based on the Edit class.
    DECLARE_WND_SUPERCLASS(_T("MyEdit"), _T("Edit"))

    BEGIN_MSG_MAP(CMyEdit)
        MESSAGE_HANDLER(WM_CHAR, OnChar)
    END_MSG_MAP()

    LRESULT OnChar(UINT /*nMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
        BOOL& /*bHandled*/)
    {
        // Do some character handling code
        return 0;
    }
};
```

To create the superclassed Edit window, create an instance of `CMyEdit` and then call the `Create` method.

Subclassing an Existing Window

To subclass an existing window, derive a class from `CWindowImpl` and declare a message map, as in the two previous cases. Note, however, that you do not specify any Windows class information, since you will subclass an already existing window.

Instead of calling `Create`, call `SubclassWindow` and pass it the handle to the existing window you want to subclass. Once the window is subclassed, it will use `CWindowImpl::WindowProc` (or your function that overrides this method) to direct messages to the message map. To detach a subclassed window from your object, call `UnsubclassWindow`. The window's original window procedure will then be restored.

See also

[Implementing a Window](#)

Implementing a Dialog Box

3/4/2019 • 2 minutes to read • [Edit Online](#)

There are two ways to add a dialog box to your ATL project: use the ATL Dialog Wizard or add it manually.

Adding a Dialog Box with the ATL Dialog Wizard

In the [Add Class dialog box](#), select the ATL Dialog object to add a dialog box to your ATL project. Fill in the ATL Dialog Wizard as appropriate and click **Finish**. The wizard adds a class derived from [CAxDialogImpl](#) to your project. Open **Resource View** from the **View** menu, locate your dialog, and double-click it to open it in the resource editor.

NOTE

If your dialog box is derived from `CxDialogImpl`, it can host both ActiveX and Windows controls. If you don't want the overhead of ActiveX control support in your dialog box class, use [CSimpleDialog](#) or [CDialogImpl](#) instead.

Message and event handlers can be added to your dialog class from Class View. For more information, see [Adding an ATL Message Handler](#).

Adding a Dialog Box Manually

Implementing a dialog box is similar to implementing a window. You derive a class from either [CAxDialogImpl](#), [CDialogImpl](#), or [CSimpleDialog](#) and declare a [message map](#) to handle messages. However, you must also specify a dialog template resource ID in your derived class. Your class must have a data member called `IDD` to hold this value.

NOTE

When you create a dialog box using the ATL Dialog Wizard, the wizard automatically adds the `IDD` member as an **enum** type.

`CDialogImpl` allows you to implement a modal or a modeless dialog box that hosts Windows controls.

`CAxDialogImpl` allows you to implement a modal or a modeless dialog box that hosts both ActiveX and Windows controls.

To create a modal dialog box, create an instance of your `CDialogImpl`-derived (or `CAxDialogImpl`-derived) class and then call the [DoModal](#) method. To close a modal dialog box, call the [EndDialog](#) method from a message handler. To create a modeless dialog box, call the [Create](#) method instead of `DoModal`. To destroy a modeless dialog box, call [DestroyWindow](#).

Sinking events is automatically done in [CAxDialogImpl](#). Implement the dialog box's message handlers as you would the handlers in a `CWindowImpl`-derived class. If there is a message-specific return value, return it as an `LRESULT`. The returned `LRESULT` values are mapped by ATL for proper handling by the Windows dialog manager. For details, see the source code for [CDialogImplBaseT::DialogProc](#) in `atlwin.h`.

Example

The following class implements a dialog box:


```

class CMyDialog : public CDialogImpl<CMyDialog>
{
public:
    enum { IDD = IDD_MYDLG };

    BEGIN_MSG_MAP(CMyDialog)
        MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
        COMMAND_HANDLER(IDCANCEL, BN_CLICKED, OnBnClickedCancel)
    END_MSG_MAP()

    LRESULT OnInitDialog(UINT /*uMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
        BOOL& /*bHandled*/)
    {
        // Do some initialization code
        return 1;
    }
public:
    LRESULT OnBnClickedCancel(WORD /*wNotifyCode*/, WORD /*wID*/, HWND /*hWndCtl*/, BOOL& /*bHandled*/);
};

```

See also

[Window Classes](#)

Using Contained Windows

3/4/2019 • 2 minutes to read • [Edit Online](#)

ATL implements contained windows with [CContainedWindowT](#). A contained window represents a window that delegates its messages to a container object instead of handling them in its own class.

NOTE

You do not need to derive a class from `CContainedWindowT` in order to use contained windows.

With contained windows, you can either superclass an existing Windows class or subclass an existing window. To create a window that superclasses an existing Windows class, first specify the existing class name in the constructor for the `CContainedWindowT` object. Then call `CContainedWindowT::Create`. To subclass an existing window, you don't need to specify a Windows class name (pass NULL to the constructor). Simply call the `CContainedWindowT::SubclassWindow` method with the handle to the window being subclassed.

You typically use contained windows as data members of a container class. The container does not need to be a window; however, it must derive from [CMessageMap](#).

A contained window can use alternate message maps to handle its messages. If you have more than one contained window, you should declare several alternate message maps, each corresponding to a separate contained window.

Example

Following is an example of a container class with two contained windows:

```
class CMyContainer : public CMessageMap
{
public:
    CContainedWindow m_wndEdit;
    CContainedWindow m_wndList;

    CMyContainer() : m_wndEdit(_T("Edit"), this, 1),
                   m_wndList(_T("List"), this, 2)
    {
    }

    BEGIN_MSG_MAP(CMyContainer)
    ALT_MSG_MAP(1)
        // handlers for the Edit window go here
    ALT_MSG_MAP(2)
        // handlers for the List window go here
    END_MSG_MAP()

};
```

For more information about contained windows, see the [SUBEDIT](#) sample.

See also

[Window Classes](#)

Understanding Window Traits

3/4/2019 • 2 minutes to read • [Edit Online](#)

Window traits classes provide a simple method for standardizing the styles used for the creation of an ATL window object. Window traits are accepted as template parameters by [CWindowImpl](#) and other ATL window classes as a way of providing default window styles at the class level.

If the creator of a window instance doesn't provide styles explicitly in the call to [Create](#), you can use a traits class to ensure that the window is still created with the correct styles. You can even ensure that certain styles are set for all instances of that window class while permitting other styles to be set on a per-instance basis.

ATL Window Traits Templates

ATL provides two window traits templates that allow you to set default styles at compile time using their template parameters.

CLASS	DESCRIPTION
CWinTraits	Use this template when you want to provide default window styles that will be used only when no other styles are specified in the call to <code>Create</code> . The styles provided at run time take precedence over the styles set at compile time.
CWinTraitsOR	Use this class when you want to specify styles that must always be set for the window class. The styles provided at run time are combined with the styles set at compile time using the bitwise OR operator.

In addition to these templates, ATL provides a number of predefined specializations of the `CWinTraits` template for commonly used combinations of window styles. See the [CWinTraits](#) reference documentation for full details.

Custom Window Traits

In the unlikely situation that specializing one of the templates provided by ATL isn't sufficient and you need to create your own traits class, you just need to create a class that implements two static functions: `GetWndStyle` and

`GetWndStyleEx`:

```
static DWORD GetWndStyle(DWORD dwStyle);  
static DWORD GetWndExStyle(DWORD dwExStyle);
```

Each of these functions will be passed some style value at run time which it can use to produce a new style value. If your window traits class is being used as the template argument to an ATL window class, the style values passed to these static functions will be whatever was passed as the style arguments to [Create](#).

See also

[Window Classes](#)

ATL Collection Classes

4/1/2019 • 5 minutes to read • [Edit Online](#)

ATL provides many classes for storing and accessing data. Which class you decide to use depends on several factors, including:

- The amount of data to be stored
- Efficiency versus performance in accessing the data
- The ability to access the data by index or by key
- How the data is ordered
- Personal preference

Small Collection Classes

ATL provides the following array classes for dealing with small numbers of objects. However, these classes are limited and designed for use internally by ATL. It is not recommended that you use them in your programs.

CLASS	TYPE OF DATA STORAGE
CSimpleArray	Implements an array class for dealing with small numbers of objects.
CSimpleMap	Implements a mapping class for dealing with small numbers of objects.

General Purpose Collection Classes

The follow classes implement arrays, lists, and maps and are provided as general purpose collection classes:

CLASS	TYPE OF DATA STORAGE
CAtlArray	Implements an array.
CAtlList	Implements a list.
CAtlMap	Implements a mapping structure, whereby data can be referenced by key or value.
CRBMap	Implements a mapping structure using the Red-Black algorithm.
CRBMultiMap	Implements a Red-Black multimapping structure.

These classes will trap many programming errors when used in debug builds, but for sake of performance, these checks will not be performed in retail builds.

Specialized Collection Classes

More specialized collection classes are also provided for managing memory pointers and interface pointers:

CLASS	PURPOSE
CAutoPtrArray	Provides methods useful when constructing an array of smart pointers.
CAutoPtrList	Provides methods useful when constructing a list of smart pointers.
CComUnkArray	Stores <code>IUnknown</code> pointers and is designed to be used as a parameter to the IConnectionPointImpl template class.
CHeapPtrList	Provides methods useful when constructing a list of heap pointers.
CInterfaceArray	Provides methods useful when constructing an array of COM interface pointers.
CInterfaceList	Provides methods useful when constructing a list of COM interface pointers.

Choosing a Collection Class

Each of the available collection classes offers different performance characteristics, as shown in the table below.

- Columns 2 and 3 describe each class's ordering and access characteristics. In the table, the term "ordered" means that the order in which items are inserted and deleted determines their order in the collection; it does not mean the items are sorted on their contents. The term "indexed" means that the items in the collection can be retrieved by an integer index, much like items in a typical array.
- Columns 4 and 5 describe each class's performance. In applications that require many insertions into the collection, insertion speed might be especially important; for other applications, lookup speed may be more important.
- Column 6 describes whether each shape allows duplicate elements.
- The performance of a given collection class operation is expressed in terms of the relationship between the time required to complete the operation and the number of elements in the collection. An operation taking an amount of time that increases linearly as the number of elements increases is described as an $O(n)$ algorithm. By contrast, an operation taking a period of time that increases less and less as the number of elements increases is described as an $O(\log n)$ algorithm. Therefore, in terms of performance, $O(\log n)$ algorithms outperform $O(n)$ algorithms more and more as the number of elements increases.

Collection Shape Features

SHAPE	ORDERED	INDEXED	INSERT AN ELEMENT	SEARCH FOR SPECIFIED ELEMENT	DUPLICATE ELEMENTS
List	Yes	No	Fast (constant time)	Slow $O(n)$	Yes

SHAPE	ORDERED	INDEXED	INSERT AN ELEMENT	SEARCH FOR SPECIFIED ELEMENT	DUPLICATE ELEMENTS
Array	Yes	By int (constant time)	Slow $O(n)$ except if inserting at end, in which case constant time	Slow $O(n)$	Yes
Map	No	By key (constant time)	Fast (constant time)	Fast (constant time)	No (keys) Yes (values)
Red-Black Map	Yes (by key)	By key $O(\log n)$	Fast $O(\log n)$	Fast $O(\log n)$	No
Red-Black Multimap	Yes (by key)	By key $O(\log n)$ (multiple values per key)	Fast $O(\log n)$	Fast $O(\log n)$	Yes (multiple values per key)

Using CTraits Objects

As the ATL collection classes can be used to store a wide range of user-defined data types, it can be useful to be able to override important functions such as comparisons. This is achieved using the CTraits classes.

CTraits classes are similar to, but more flexible than, the MFC collection class helper functions; see [Collection Class Helpers](#) for more information.

When constructing your collection class, you have the option of specifying a CTraits class. This class will contain the code that will perform operations such as comparisons when called by the other methods that make up the collection class. For example, if your list object contains your own user-defined structures, you may want to redefine the equality test to only compare certain member variables. In this way, the list object's Find method will operate in a more useful manner.

Example

Code

```
// Collection class / traits class example.
// This program demonstrates using a CTraits class
// to create a new comparison operator.

#define MAX_STRING 80

// Define our own data type to store in the list.

struct MyData
{
    int ID;
    TCHAR name[MAX_STRING];
    TCHAR address[MAX_STRING];
};

// Define our own traits class, making use of the
// existing traits and overriding only the comparison
// we need.

class MyTraits : public CElementTraits< MyData >
{
public:
    // Override the comparison to only compare
```

```

// override the comparison to only compare
// the ID value.

static bool CompareElements(const MyData& element1, const MyData& element2)
{
    if (element1.ID == element2.ID)
        return true;
    else
        return false;
};

};

void DoAtlCustomTraitsList()
{
    // Declare the array, with our data type and traits class

    CATllist < MyData, MyTraits > MyList;

    // Create some variables of our data type

    MyData add_item, search_item;

    // Add some elements to the list.

    add_item.ID = 1;
    _stprintf_s(add_item.name, _T("Rumpelstiltskin"));
    _stprintf_s(add_item.address, _T("One Grimm Way"));

    MyList.AddHead(add_item);

    add_item.ID = 2;
    _stprintf_s(add_item.name, _T("Rapunzel"));
    _stprintf_s(add_item.address, _T("One Grimm Way"));

    MyList.AddHead(add_item);

    add_item.ID = 3;
    _stprintf_s(add_item.name, _T("Cinderella"));
    _stprintf_s(add_item.address, _T("Two Grimm Way"));

    MyList.AddHead(add_item);

    // Create an element which will be used
    // to search the list for a match.

    search_item.ID = 2;
    _stprintf_s(search_item.name, _T("Don't care"));
    _stprintf_s(search_item.address, _T("Don't care"));

    // Perform a comparison by searching for a match
    // between any element in the list, and our
    // search item. This operation will use the
    // (overridden) comparison operator and will
    // find a match when the IDs are the same.

    POSITION i;

    i = MyList.Find(search_item);

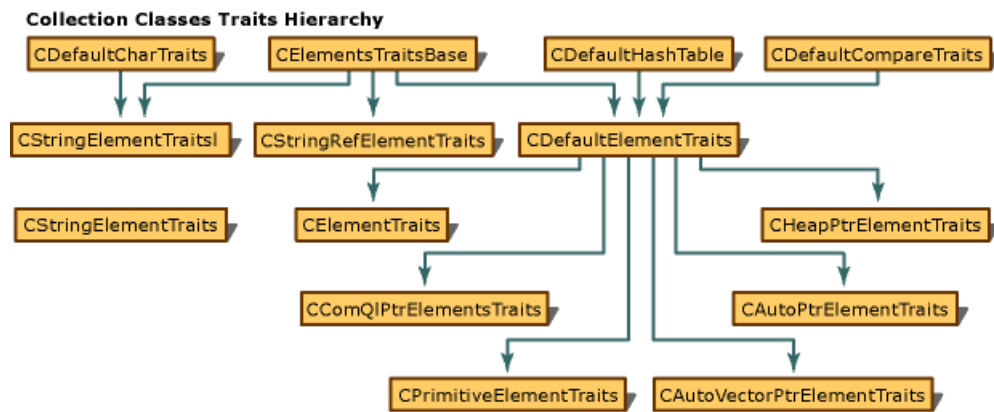
    if (i != NULL)
        _tprintf_s(_T("Item found!\n"));
    else
        _tprintf_s(_T("Item not found.\n"));
}

```

Comments

For a list of the CTraits classes, see [Collection Classes](#).

The following diagram shows the class hierarchy for the CTraits classes.



Collection Classes Samples

The following samples demonstrate the collection classes:

- [MMXSwarm Sample](#)
- [DynamicConsumer Sample](#)
- [UpdatePV Sample](#)
- [Marquee Sample](#)

See also

[Concepts](#)

[Collection Classes](#)

ATL Registry Component (Registrar)

3/4/2019 • 2 minutes to read • [Edit Online](#)

The ATL Registrar provides optimized access to the system registry through a custom interface. The Registrar is free-threaded and allows static linking of code for C++ clients.

NOTE

The source code for the ATL Registrar can be found in `atlmfcl\include\atliface.h`.

In This Section

[Creating Registrar Scripts](#)

A guide to creating registrar scripts. Includes topics on BNF syntax, parse trees, registry scripting examples, using replaceable parameters, and invoking scripts.

[Setting Up a Static Link to the Registrar Code \(C++ only\)](#)

Lists the steps to set up static linking to the Registrar.

Related Sections

[ATL](#)

Provides links to conceptual topics on how to program using the Active Template Library.

Creating Registrar scripts

5/15/2019 • 2 minutes to read • [Edit Online](#)

A registrar script provides data-driven, rather than API-driven, access to the system registry. Data-driven access is typically more efficient since it takes only one or two lines in a script to add a key to the registry.

The [ATL Control Wizard](#) automatically generates a registrar script for your COM server. You can find this script in the .rgs file associated with your object.

The ATL Registrar's Script Engine processes your registrar script at run time. ATL automatically invokes the Script Engine during server setup.

This article covers the following topics related to the registrar scripts:

- [Understanding Backus-Naur form \(BNF\) syntax](#)
- [Understanding Parse Trees](#)
- [Registry Scripting Examples](#)
- [Using Replaceable Parameters \(The Registrar's Preprocessor\)](#)
- [Invoking Scripts](#)

See also

[Registry Component \(Registrar\)](#)

2 minutes to read

Understanding Parse Trees

3/4/2019 • 2 minutes to read • [Edit Online](#)

You can define one or more parse trees in your registrar script, where each parse tree has the following form:

```
<root key>{<registry expression>}+
```

where:

```
<root key> ::= HKEY_CLASSES_ROOT | HKEY_CURRENT_USER |  
              HKEY_LOCAL_MACHINE | HKEY_USERS |  
              HKEY_PERFORMANCE_DATA | HKEY_DYN_DATA |  
              HKEY_CURRENT_CONFIG | HKCR | HKCU |  
              HKLM | HKU | HKPD | HKDD | HKCC  
<registry expression> ::= <Add Key> | <Delete Key>  
<Add Key> ::= [ForceRemove | NoRemove | val]<Key Name> [<Key Value>][{<Add Key>}]  
<Delete Key> ::= Delete<Key Name>  
<Key Name> ::= '<AlphaNumeric>+'  
<AlphaNumeric> ::= any character not NULL, i.e. ASCII 0  
<Key Value> ::= <Key Type><Key Name>  
<Key Type> ::= s | d  
<Key Value> ::= '<AlphaNumeric>'
```

NOTE

HKEY_CLASSES_ROOT and HKCR are equivalent; HKEY_CURRENT_USER and HKCU are equivalent; and so on.

A parse tree can add multiple keys and subkeys to the <root key>. In doing so, it keeps a subkey's handle open until the parser has completed parsing all of its subkeys. This approach is more efficient than operating on a single key at a time, as seen in the following example:

```
HKEY_CLASSES_ROOT  
{  
  'MyVeryOwnKey'  
  {  
    'HasASubKey'  
    {  
      'PrettyCool'  
    }  
  }  
}
```

Here, the Registrar initially opens (creates) HKEY_CLASSES_ROOT\MyVeryOwnKey. It then sees that MyVeryOwnKey has a subkey. Rather than close the key to MyVeryOwnKey, the Registrar retains the handle and opens (creates) HasASubKey using this parent handle. (The system registry can be slower when no parent handle is open.) Thus, opening HKEY_CLASSES_ROOT\MyVeryOwnKey and then opening HasASubKey with MyVeryOwnKey as the parent is faster than opening MyVeryOwnKey, closing MyVeryOwnKey, and then opening MyVeryOwnKey\HasASubKey.

See also

[Creating Registrar Scripts](#)

Registry Scripting Examples

3/4/2019 • 2 minutes to read • [Edit Online](#)

The scripting examples in this topic demonstrate how to add a key to the system registry, register the Registrar COM server, and specify multiple parse trees.

Add a Key to HKEY_CURRENT_USER

The following parse tree illustrates a simple script that adds a single key to the system registry. In particular, the script adds the key, `MyVeryOwnKey`, to `HKEY_CURRENT_USER`. It also assigns the default string value of `HowGoesIt` to the new key:

```
HKEY_CURRENT_USER
{
    'MyVeryOwnKey' = s 'HowGoesIt'
}
```

This script can easily be extended to define multiple subkeys as follows:

```
HKCU
{
    'MyVeryOwnKey' = s 'HowGoesIt'
    {
        'HasASubkey'
        {
            'PrettyCool' = d '55'
            val 'ANameValue' = s 'WithANamedValue'
        }
    }
}
```

Now, the script adds a subkey, `HasASubkey`, to `MyVeryOwnKey`. To this subkey, it adds both the `PrettyCool` subkey (with a default `DWORD` value of 55) and the `ANameValue` named value (with a string value of `WithANamedValue`).

Register the Registrar COM Server

The following script registers the Registrar COM server itself.

```

HKCR
{
    ATL.Registrar = s 'ATL Registrar Class'
    {
        CLSID = s '{44EC053A-400F-11D0-9DCD-00A0C90391D3}'
    }
    NoRemove CLSID
    {
        ForceRemove {44EC053A-400F-11D0-9DCD-00A0C90391D3} = s 'ATL Registrar Class'
        {
            ProgID = s 'ATL.Registrar'
            InprocServer32 = s '%MODULE%'
            {
                val ThreadingModel = s 'Apartment'
            }
        }
    }
}
}

```

At run time, this parse tree adds the `ATL.Registrar` key to `HKEY_CLASSES_ROOT`. To this new key, it then:

- Specifies `ATL Registrar Class` as the key's default string value.
- Adds `CLSID` as a subkey.
- Specifies `{44EC053A-400F-11D0-9DCD-00A0C90391D3}` for `CLSID`. (This value is the Registrar's CLSID for use with `CoCreateInstance`.)

Since `CLSID` is shared, it should not be removed in Unregister mode. The statement, `NoRemove CLSID`, does this by indicating that `CLSID` should be opened in Register mode and ignored in Unregister mode.

The `ForceRemove` statement provides a housekeeping function by removing a key and all of its subkeys before re-creating the key. This can be useful if the names of the subkeys have changed. In this scripting example,

`ForceRemove` checks to see if `{44EC053A-400F-11D0-9DCD-00A0C90391D3}` already exists. If it does, `ForceRemove`:

- Recursively deletes `{44EC053A-400F-11D0-9DCD-00A0C90391D3}` and all of its subkeys.
- Re-creates `{44EC053A-400F-11D0-9DCD-00A0C90391D3}`.
- Adds `ATL Registrar Class` as the default string value for `{44EC053A-400F-11D0-9DCD-00A0C90391D3}`.

The parse tree now adds two new subkeys to `{44EC053A-400F-11D0-9DCD-00A0C90391D3}`. The first key, `ProgID`, gets a default string value that is the ProgID. The second key, `InprocServer32`, gets a default string value, `%MODULE%`, that is a preprocessor value explained in the section, [Using Replaceable Parameters \(The Registrar's Preprocessor\)](#), of this article. `InprocServer32` also gets a named value, `ThreadingModel`, with a string value of `Apartment`.

Specify Multiple Parse Trees

To specify more than one parse tree in a script, simply place one tree at the end of another. For example, the following script adds the key, `MyVeryOwnKey`, to the parse trees for both `HKEY_CLASSES_ROOT` and `HKEY_CURRENT_USER`:

```

HKCR
{
    'MyVeryOwnKey' = s 'HowGoesIt'
}
HKEY_CURRENT_USER
{
    'MyVeryOwnKey' = s 'HowGoesIt'
}

```

NOTE

In a Registrar script, 4K is the maximum token size. (A token is any recognizable element in the syntax.) In the previous scripting example, `HKCR`, `HKEY_CURRENT_USER`, `'MyVeryOwnKey'`, and `'HowGoesIt'` are all tokens.

See also

[Creating Registrar Scripts](#)

Using Replaceable Parameters (The Registrar's Preprocessor)

3/12/2019 • 2 minutes to read • [Edit Online](#)

Replaceable parameters allow a Registrar's client to specify run-time data. To do this, the Registrar maintains a replacement map into which it enters the values associated with the replaceable parameters in your script. The Registrar makes these entries at run time.

Using %MODULE%

The [ATL Control Wizard](#) automatically generates a script that uses `%MODULE%`. ATL uses this replaceable parameter for the actual location of your server's DLL or EXE.

Concatenating Run-Time Data with Script Data

Another use of the preprocessor is to concatenate run-time data with script data. For example, suppose an entry is needed that contains a full path to a module with the string `", 1"` appended at the end. First, define the following expansion:

```
'MySampleKey' = s '%MODULE%, 1'
```

Then, before calling one of the script processing methods listed in [Invoking Scripts](#), add a replacement to the map:

```
TCHAR szModule[_MAX_PATH];
::GetModuleFileName(_AtlBaseModule.GetModuleInstance(), szModule, _MAX_PATH);
p->AddReplacement(OLESTR("Module"), T2OLE(szModule));
```

During the parsing of the script, the Registrar expands `'%MODULE%, 1'` to `c:\mycode\mydll.dll, 1`.

NOTE

In a Registrar script, 4K is the maximum token size. (A token is any recognizable element in the syntax.) This includes tokens that were created or expanded by the preprocessor.

NOTE

To substitute replacement values at run time, remove the call in the script to the [DECLARE_REGISTRY_RESOURCE](#) or [DECLARE_REGISTRY_RESOURCEID](#) macro. Instead, replace it with your own `UpdateRegistry` method that calls [CAtlModule::UpdateRegistryFromResourceD](#) or [CAtlModule::UpdateRegistryFromResourceS](#), and pass your array of `_ATL_REGMAP_ENTRY` structures. Your array of `_ATL_REGMAP_ENTRY` must have at least one entry that is set to `{NULL, NULL}`, and this entry should always be the last entry. Otherwise, an access violation error will be generated when `UpdateRegistryFromResource` is called.

NOTE

When building a project that outputs an executable, ATL automatically adds quotation marks around the path name created at run time with the **%MODULE%** registrar script parameter. If you do not want the path name to include the quotation marks, use the new **%MODULE_RAW%** parameter instead.

When building a project that outputs a DLL, ATL will not add quotation marks to the path name if **%MODULE%** or **%MODULE_RAW%** is used.

See also

[Creating Registrar Scripts](#)

Invoking Scripts

3/12/2019 • 2 minutes to read • [Edit Online](#)

Using [Replaceable Parameters \(The Registrar's Preprocessor\)](#) discusses replacement maps and mentions the Registrar method **AddReplacement**. The Registrar has eight other methods specific to scripting, and all are described in the following table.

METHOD	SYNTAX/DESCRIPTION
ResourceRegister	HRESULT ResourceRegister(LPCOLESTR <i>resFileName</i> , UINT <i>nID</i> , LPCOLESTR <i>szType</i>); Registers the script contained in a module's resource. <i>resFileName</i> indicates the UNC path to the module itself. <i>nID</i> and <i>szType</i> contain the resource's ID and type, respectively.
ResourceUnregister	HRESULT ResourceUnregister(LPCOLESTR <i>resFileName</i> , UINT <i>nID</i> , LPCOLESTR <i>szType</i>); Unregisters the script contained in a module's resource. <i>resFileName</i> indicates the UNC path to the module itself. <i>nID</i> and <i>szType</i> contain the resource's ID and type, respectively.
ResourceRegisterSz	HRESULT ResourceRegisterSz(LPCOLESTR <i>resFileName</i> , LPCOLESTR <i>szID</i> , LPCOLESTR <i>szType</i>); Registers the script contained in a module's resource. <i>resFileName</i> indicates the UNC path to the module itself. <i>szID</i> and <i>szType</i> contain the resource's string identifier and type, respectively.
ResourceUnregisterSz	HRESULT ResourceUnregisterSz(LPCOLESTR <i>resFileName</i> , LPCOLESTR <i>szID</i> , LPCOLESTR <i>szType</i>); Unregisters the script contained in a module's resource. <i>resFileName</i> indicates the UNC path to the module itself. <i>szID</i> and <i>szType</i> contain the resource's string identifier and type, respectively.
FileRegister	HRESULT FileRegister(LPCOLESTR <i>fileName</i>); Registers the script in a file. <i>fileName</i> is a UNC path to a file that contains (or is) a resource script.
FileUnregister	HRESULT FileUnregister(LPCOLESTR <i>fileName</i>); Unregisters the script in a file. <i>fileName</i> is a UNC path to a file that contains (or is) a resource script.
StringRegister	HRESULT StringRegister(LPCOLESTR <i>data</i>); Registers the script in a string. <i>data</i> contains the script itself.

METHOD	SYNTAX/DESCRIPTION
StringUnregister	HRESULT StringUnregister(LPCOLESTR <i>data</i>); Unregisters the script in a string. <i>data</i> contains the script itself.

ResourceRegisterSz and **ResourceUnregisterSz**, are similar to **ResourceRegister** and **ResourceUnregister**, but allow you to specify a string identifier.

The methods **FileRegister** and **FileUnregister** are useful if you do not want the script in a resource or if you want the script in its own file. The methods **StringRegister** and **StringUnregister** allow the .rgs file to be stored in a dynamically allocated string.

See also

[Creating Registrar Scripts](#)

Setting Up a Static Link to the Registrar Code (C++ Only)

3/4/2019 • 2 minutes to read • [Edit Online](#)

C++ clients can create a static link to the Registrar's code. Static linking of the Registrar's parser adds approximately 5K to a release build.

The simplest way to set up static linking assumes you have specified `DECLARE_REGISTRY_RESOURCEID` in your object's declaration. (This is the default specification used by the ATL.)

To create a static link using `DECLARE_REGISTRY_RESOURCEID`

1. Specify `/D _ATL_STATIC_REGISTRY` instead of `/D _ATL_DLL`.
2. Recompile.

See also

[Registry Component \(Registrar\)](#)

Programming with ATL and C Run-Time Code

3/4/2019 • 2 minutes to read • [Edit Online](#)

This section discusses the benefits of using the C Run-Time Library (CRT) with either static or dynamic linking.

In This Section

[Benefits and Tradeoffs of the Method Used to Link to the CRT](#)

Summarizes the benefits and tradeoffs involved in linking statically to the CRT or linking dynamically.

[Linking to the CRT in Your ATL Project](#)

Discusses the project settings and linker options for linking to the CRT; also provides details on how linking to the CRT affects your program image.

Related Sections

[ATL](#)

Provides links to conceptual topics on how to program using the Active Template Library.

[DLLs and Visual C++ run-time library behavior](#)

Provides details on how the VCRuntime and CRT startup code works.

[C Run-Time Libraries](#)

Discusses the various .lib files that comprise the C run-time libraries and lists their associated compiler options and preprocessor directives.

See also

[Concepts](#)

Benefits and Tradeoffs of the Method Used to Link to the CRT

5/7/2019 • 2 minutes to read • [Edit Online](#)

Your project can link with the CRT either dynamically or statically. The table below outlines the benefits and tradeoffs involved in choosing which method to use.

METHOD	BENEFIT	TRADEOFF
Statically linking to the CRT (Runtime Library set to Single-threaded)	The CRT DLL is not required on the system where the image will run.	About 25K of startup code is added to your image, substantially increasing its size.
Dynamically linking to the CRT (Runtime Library set to Multi-threaded)	Your image does not require the CRT startup code, so it is much smaller.	The CRT DLL must be on the system running the image.

The topic [Linking to the CRT in Your ATL Project](#) discusses how to select the manner in which to link to the CRT.

See also

[Programming with ATL and C Run-Time Code](#)
[DLLs and Visual C++ run-time library behavior](#)
[CRT Library Features](#)

Linking to the CRT in Your ATL Project

3/12/2019 • 2 minutes to read • [Edit Online](#)

The [C Run-Time Libraries](#) (CRT) provide many useful functions that can make programming much easier during ATL development. All ATL projects link to the CRT library. You can see the advantages and disadvantages of linking method in [Benefits and Tradeoffs of the Method Used to Link to the CRT](#).

Effects of Linking to the CRT on Your Program Image

If you statically link to the CRT, code from the CRT is placed in your executable image and you do not need to have the CRT DLL present on a system to run your image. If you dynamically link to the CRT, references to the code in the CRT DLL are placed in your image, but not the code itself. In order for your image to run on a given system, the CRT DLL must be present on that system. Even when you dynamically link to the CRT, you may find that some code can be statically linked (for example, `DllMainCRTStartup`).

When you link your image, you either explicitly or implicitly specify an entry point that the operating system will call into after loading the image. For a DLL, the default entry point is `DllMainCRTStartup`. For an EXE, it is `WinMainCRTStartup`. You can override the default with the `/ENTRY` linker option. The CRT provides an implementation for `DllMainCRTStartup`, `WinMainCRTStartup`, and `wWinMainCRTStartup` (the Unicode entry point for an EXE). These CRT-provided entry points call constructors on global objects and initialize other data structures that are used by some CRT functions. This startup code adds about 25K to your image if it is linked statically. If it is linked dynamically, most of the code is in the DLL, so your image size stays small.

For more information, see the linker topic [/ENTRY \(Entry-Point Symbol\)](#).

Optimization Options

Using the linker option `/OPT:NOWIN98` can further reduce a default ATL control by 10K, at the expense of increased loading time on Windows 98 systems. For more information on linking options, see [/OPT \(Optimizations\)](#).

See also

[Programming with ATL and C Run-Time Code](#)
[DLLs and Visual C++ run-time library behavior](#)

Programming with CComBSTR (ATL)

3/4/2019 • 2 minutes to read • [Edit Online](#)

The ATL class `CComBSTR` provides a wrapper around the BSTR data type. While `CComBSTR` is a useful tool, there are several situations that require caution.

- [Conversion Issues](#)
- [Scope Issues](#)
- [Explicitly Freeing the CComBSTR Object](#)
- [Using CComBSTR Objects in Loops](#)
- [Memory Leak Issues](#)

Conversion Issues

Although several `CComBSTR` methods will automatically convert an ANSI string argument into Unicode, the methods will always return Unicode format strings. To convert the output string back to ANSI, use an ATL conversion class. For more information on the ATL conversion classes, see [ATL and MFC String Conversion Macros](#).

Example

```
// Declare a CComBSTR object. Although the argument is ANSI,
// the constructor converts it into UNICODE.
CComBSTR bstrMyString("Hello World");
// Convert the string into an ANSI string
CW2A szMyString(bstrMyString);
// Display the ANSI string
MessageBoxA(NULL, szMyString, "String Test", MB_OK);
```

If you are using a string literal to modify a `CComBSTR` object, use wide character strings. This will reduce unnecessary conversions.

Example

```
// The following converts the ANSI string to Unicode
CComBSTR bstr1("Test");
// The following uses a Unicode string at compile time
CComBSTR bstr2(L"Test");
```

Scope Issues

As with any well-behaved class, `CComBSTR` will free its resources when it goes out of scope. If a function returns a pointer to the `CComBSTR` string, this can cause problems, as the pointer will reference memory that has already been freed. In these cases, use the `Copy` method, as shown below.

Example


```
// The wrong way to do it
BSTR * MyBadFunction()
{
    // Create the CComBSTR object
    CComBSTR bstrString(L"Hello World");
    // Convert the string to uppercase
    HRESULT hr;
    hr = bstrString.ToUpper();

    // Return a pointer to the BSTR. ** Bad thing to do **
    return &bstrString;
}

// The correct way to do it
HRESULT MyGoodFunction(/*[out]*/ BSTR* bstrStringPtr)
{
    // Create the CComBSTR object
    CComBSTR bstrString(L"Hello World");
    // Convert the string to uppercase
    HRESULT hr;
    hr = bstrString.ToUpper();
    if (hr != S_OK)
        return hr;
    // Return a copy of the string.
    return bstrString.CopyTo(bstrStringPtr);
}
```

Explicitly Freeing the CComBSTR Object

It is possible to explicitly free the string contained in the `CComBSTR` object before the object goes out scope. If the string is freed, the `CComBSTR` object is invalid.

Example

```
// Declare a CComBSTR object
CComBSTR bstrMyString(L"Hello World");
// Free the string explicitly
::SysFreeString(bstrMyString);
// The string will be freed a second time
// when the CComBSTR object goes out of scope,
// which is invalid.
```

Using CComBSTR Objects in Loops

As the `CComBSTR` class allocates a buffer to perform certain operations, such as the `+=` operator or `Append` method, it is not recommended that you perform string manipulation inside a tight loop. In these situations, `CStringT` provides better performance.

Example

```
// This is not an efficient way to use a CComBSTR object.
CComBSTR bstrMyString;
HRESULT hr;
while (bstrMyString.Length() < 1000)
    hr = bstrMyString.Append(L"*");
```

Memory Leak Issues

Passing the address of an initialized `CComBSTR` to a function as an **[out]** parameter causes a memory leak.

In the example below, the string allocated to hold the string `"Initialized"` is leaked when the function `MyGoodFunction` replaces the string.

```
CComBSTR bstrLeak(L"Initialized");  
HRESULT hr = MyGoodFunction(&bstrLeak);
```

To avoid the leak, call the `Empty` method on existing `CComBSTR` objects before passing the address as an **[out]** parameter.

Note that the same code would not cause a leak if the function's parameter was **[in, out]**.

See also

[Concepts](#)

[CStringT Class](#)

[wstring](#)

[String Conversion Macros](#)

ATL Encoding Reference

3/4/2019 • 2 minutes to read • [Edit Online](#)

Encoding in a range of common Internet standards such as uuencode, hexadecimal, and UTF8 is supported by the code found in atlenc.h.

Functions

AtlGetHexValue	Call this function to get the numeric value of a hexadecimal digit.
AtlHexDecode	Decodes a string of data that has been encoded as hexadecimal text such as by a previous call to AtlHexEncode .
AtlHexDecodeGetRequiredLength	Call this function to get the size in bytes of a buffer that could contain data decoded from a hex-encoded string of the specified length.
AtlHexEncode	Call this function to encode some data as a string of hexadecimal text.
AtlHexEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
AtlUnicodeToUTF8	Call this function to convert a Unicode string to UTF-8.
BEncode	Call this function to convert some data using the "B" encoding.
BEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
EscapeXML	Call this function to convert characters that are unsafe for use in XML to their safe equivalents.
GetExtendedChars	Call this function to get the number of extended characters in a string.
IsExtendedChar	Call this function to find out if a given character is an extended character (less than 32, greater than 126, and not a tab, linefeed or carriage return)
QEncode	Call this function to convert some data using the "Q" encoding.
QEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
QPDecode	Decodes a string of data that has been encoded in quoted-printable format such as by a previous call to QPEncode .

QPDecodeGetRequiredLength		Call this function to get the size in bytes of a buffer that could contain data decoded from quoted-printable-encoded string of the specified length.
QPEncode		Call this function to encode some data in quoted-printable format.
QPEncodeGetRequiredLength		Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
UUDecode		Decodes a string of data that has been uuencoded such as by a previous call to UUEncode .
UUDecodeGetRequiredLength		Call this function to get the size in bytes of a buffer that could contain data decoded from a uuencoded string of the specified length.
UUEncode		Call this function to uuencode some data.
UUEncodeGetRequiredLength		Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.

See also

[Concepts](#)

[ATL COM Desktop Components](#)

ATL Utilities Reference

3/4/2019 • 3 minutes to read • [Edit Online](#)

ATL provides code for manipulating paths and URLs in the form of [CPathT](#) and [CUrl](#). A thread pool, [CThreadPool](#), can be used in your applications. This code can be found in `atlpath.h` and `atlutil.h`.

Classes

CPathT Class	This class represents a path.
CDebugReportHook Class	Use this class to send debug reports to a named pipe.
CNonStatelessWorker Class	Receives requests from a thread pool and passes them on to a worker object that is created and destroyed on each request.
CNoWorkerThread Class	Use this class as the argument for the <code>MonitorClass</code> template parameter to cache classes if you want to disable dynamic cache maintenance.
CThreadPool Class	This class provides a pool of worker threads that process a queue of work items.
CUrl Class	This class represents a URL. It allows you to manipulate each element of the URL independently of the others whether parsing an existing URL string or building a string from scratch.
CWorkerThread Class	This class creates a worker thread or uses an existing one, waits on one or more kernel object handles, and executes a specified client function when one of the handles is signaled.

Typedefs

CPath	A specialization of CPathT using <code>CString</code> .
CPathA	A specialization of CPathT using <code>CStringA</code> .
CPathW	A specialization of CPathT using <code>CStringW</code> .
ATL_URL_PORT	The type used by CUrl for specifying a port number.

Enums

ATL_URL_SCHEME	The members of this enumeration provide constants for the schemes understood by CUrl .

Functions

AtlCanonicalizeUrl	Call this function to canonicalize a URL, which includes converting unsafe characters and spaces into escape sequences.
AtlCombineUrl	Call this function to combine a base URL and a relative URL into a single, canonical URL.
AtlEscapeUrl	Call this function to convert all unsafe characters to escape sequences.
AtlGetDefaultUrlPort	Call this function to get the default port number associated with a particular internet protocol or scheme.
AtlHexValue	Call this function to get the numeric value of a hexadecimal digit.
AtlIsUnsafeUrlChar	Call this function to find out whether a character is safe for use in a URL.
AtlUnescapeUrl	Call this function to convert escaped characters back to their original values.
SystemTimeToHttpDate	Call this function to convert a system time to a string in a format suitable for using in HTTP headers.

[|ATLPath::AddBackslash|](#)This function is an overloaded wrapper for [PathAddBackslash.](#)
[|ATLPath::AddExtension|](#)This function is an overloaded wrapper for [PathAddExtension.](#) [|ATLPath::Append|](#)This function is an overloaded wrapper for [PathAppend.](#) [|ATLPath::BuildRoot|](#)This function is an overloaded wrapper for [PathBuildRoot.](#) [|ATLPath::Canonicalize|](#)This function is an overloaded wrapper for [PathCanonicalize.](#)
[|ATLPath::Combine|](#)This function is an overloaded wrapper for [PathCombine.](#) [|ATLPath::CommonPrefix|](#)This function is an overloaded wrapper for [PathCommonPrefix.](#) [|ATLPath::CompactPath|](#)This function is an overloaded wrapper for [PathCompactPath.](#) [|ATLPath::CompactPathEx|](#)This function is an overloaded wrapper for [PathCompactPathEx.](#) [|ATLPath::FileExists|](#)This function is an overloaded wrapper for [PathFileExists.](#)
[|ATLPath::FindExtension|](#)This function is an overloaded wrapper for [PathFindExtension.](#)
[|ATLPath::FindFileName|](#)This function is an overloaded wrapper for [PathFindFileName.](#)
[|ATLPath::GetDriveNumber|](#)This function is an overloaded wrapper for [PathGetDriveNumber.](#)
[|ATLPath::IsDirectory|](#)This function is an overloaded wrapper for [PathIsDirectory.](#) [|ATLPath::IsFileSpec|](#)This function is an overloaded wrapper for [PathIsFileSpec.](#) [|ATLPath::IsPrefix|](#)This function is an overloaded wrapper for [PathIsPrefix.](#) [|ATLPath::IsRelative|](#)This function is an overloaded wrapper for [PathIsRelative.](#)
[|ATLPath::IsRoot|](#)This function is an overloaded wrapper for [PathIsRoot.](#) [|ATLPath::IsSameRoot|](#)This function is an overloaded wrapper for [PathIsSameRoot.](#) [|ATLPath::IsUNC|](#)This function is an overloaded wrapper for [PathIsUNC.](#) [|ATLPath::IsUNCServer|](#)This function is an overloaded wrapper for [PathIsUNCServer.](#)
[|ATLPath::IsUNCServerShare|](#)This function is an overloaded wrapper for [PathIsUNCServerShare.](#)
[|ATLPath::MakePretty|](#)This function is an overloaded wrapper for [PathMakePretty.](#) [|ATLPath::MatchSpec|](#)This function is an overloaded wrapper for [PathMatchSpec.](#) [|ATLPath::QuoteSpaces|](#)This function is an overloaded wrapper for [PathQuoteSpaces.](#) [|ATLPath::RelativePathTo|](#)This function is an overloaded wrapper for [PathRelativePathTo.](#) [|ATLPath::RemoveArgs|](#)This function is an overloaded wrapper for [PathRemoveArgs.](#)
[|ATLPath::RemoveBackslash|](#)This function is an overloaded wrapper for [PathRemoveBackslash.](#)
[|ATLPath::RemoveBlanks|](#)This function is an overloaded wrapper for [PathRemoveBlanks.](#)
[|ATLPath::RemoveExtension|](#)This function is an overloaded wrapper for [PathRemoveExtension.](#)
[|ATLPath::RemoveFileSpec|](#)This function is an overloaded wrapper for [PathRemoveFileSpec.](#)
[|ATLPath::RenameExtension|](#)This function is an overloaded wrapper for [PathRenameExtension.](#)

[|ATLPath::SkipRoot|](#)This function is an overloaded wrapper for [PathSkipRoot.](#) [|ATLPath::StripPath|](#)This function is an overloaded wrapper for [PathStripPath.](#) [|ATLPath::StripToRoot|](#)This function is an overloaded wrapper for [PathStripToRoot.](#) [|ATLPath::UnquoteSpaces|](#)This function is an overloaded wrapper for [PathUnquoteSpaces.](#)

See also

[Concepts](#)

[ATL COM Desktop Components](#)

Active Template Library (ATL) Tutorial

5/7/2019 • 2 minutes to read • [Edit Online](#)

ATL is designed to simplify the process of creating efficient, flexible, lightweight controls. This tutorial leads you through the creation of an ActiveX control, demonstrating many ATL and COM fundamentals.

By following this tutorial, you will learn how to add a control to an ATL project that draws a circle and a filled polygon. You will then add a property to indicate how many sides the polygon will have and create drawing code for updating the control when the property changes. The control will then be displayed on a Web page using some VBScript to make it respond to events.

The tutorial is divided into seven steps. You should perform each step in order as later steps depend on previously completed tasks. Before you begin, you should confirm that you have privileges required to register an ActiveX component on your particular computer. This is usually only a concern if you are running Visual Studio .NET over a Terminal Services connection.

- [Step 1: Creating the Project](#)
- [Step 2: Adding a Control to Your Project](#)
- [Step 3: Adding a Property to Your Control](#)
- [Step 4: Changing Your Control's Drawing Code](#)
- [Step 5: Adding an Event](#)
- [Step 6: Adding a Property Page](#)
- [Step 7: Putting Your Control on a Web Page](#)

See also

[Concepts](#)

Creating the Project (ATL Tutorial, Part 1)

5/7/2019 • 2 minutes to read • [Edit Online](#)

This tutorial walks you step-by-step through a nonattributed ATL project that creates an ActiveX object that displays a polygon. The object includes options for allowing the user to change the number of sides making up the polygon, and code to refresh the display.

NOTE

ATL and MFC are not generally supported in the Express editions of Visual Studio.

NOTE

This tutorial creates the same source code as the Polygon sample. If you want to avoid entering the source code manually, you can download it from the [Polygon sample abstract](#). You can then refer to the Polygon source code as you work through the tutorial, or use it to check for errors in your own project. To compile, open stdafx.h and replace:

```
#ifndef WINVER
#define WINVER 0x0400
#endif
```

with

```
#ifndef WINVER
#define WINVER 0x0500
#define _WIN32_WINNT 0x0500
#endif
```

The compiler will still complain about `regsvr32` not exiting correctly, but you should still have the control's DLL built and available for use.

To create the initial ATL project using the ATL Project Wizard

1. In Visual Studio 2017 and earlier: **File > New > Project**. Then open the **Visual C++** tab and select **MFC/ATL**. Select **ATL Project**.

In Visual Studio 2019: Choose **File > New > Project**, type "atl" in the search box, and choose **ATL Project**.

2. Type *Polygon* as the project name.

The location for the source code will usually default to `\Users\<username>\source\repos`, and a new folder will be created automatically.

3. Click **OK** and the **ATL Project** wizard opens.
4. Click **Application Settings** to see the options available.
5. As you are creating a control, and a control must be an in-process server, leave the **Application type** as a DLL.
6. Leave the other options at their default values, and click **OK**.

The **ATL Project Wizard** will create the project by generating several files. You can view these files in **Solution Explorer** by expanding the `Polygon` object. The files are listed below.

FILE	DESCRIPTION
Polygon.cpp	Contains the implementation of <code>DllMain</code> , <code>DllCanUnloadNow</code> , <code>DllGetClassObject</code> , <code>DllRegisterServer</code> , and <code>DllUnregisterServer</code> . Also contains the object map, which is a list of the ATL objects in your project. This is initially blank.
Polygon.def	This module-definition file provides the linker with information about the exports required by your DLL.
Polygon.idl	The interface definition language file, which describes the interfaces specific to your objects.
Polygon.rgs	This registry script contains information for registering your program's DLL.
Polygon.rc	The resource file, which initially contains the version information and a string containing the project name.
Resource.h	The header file for the resource file.
Polygonps.def	This module definition file provides the linker with information about the exports required by the proxy and stub code that support calls across apartments.
stdafx.cpp	The file that will <code>#include</code> the ATL implementation files.
stdafx.h	The file that will <code>#include</code> the ATL header files.

1. In **Solution Explorer**, right-click the `Polygon` project.
2. On the shortcut menu, click **Properties**.
3. Click on **Linker**. Change the **Per-UserRedirection** option to **Yes**.
4. Click **OK**.

In the next step, you will add a control to your project.

[On to Step 2](#)

See also

[Tutorial](#)

Adding a Control (ATL Tutorial, Part 2)

3/4/2019 • 3 minutes to read • [Edit Online](#)

In this step, you will add a control to your project, build it, and test it on a Web page.

Procedures

To add an object to an ATL project

1. In **Solution Explorer**, right-click the `Polygon` project.
2. Point to **Add** on the shortcut menu, and click **New Item** in the submenu.

The **Add New Item** dialog box appears. The different object categories are listed in the tree structure on the left.

3. Click the **ATL** folder.
4. From the list of templates on the right, select **ATL Control**. Click **Add**. The **ATL Control** wizard will open, and you can configure the control.
5. Type `PolyCtl1` as the short name and note that the other fields are automatically completed. Do not click **Finish** yet, because you have to make some changes.

The **ATL Control** wizard's **Names** page contains the following fields:

FIELD	CONTENTS
Short name	The name you entered for the control.
Class	The C++ class name created to implement the control.
.h file	The file created to contain the definition of the C++ class.
.cpp file	The file created to contain the implementation of the C++ class.
CoClass	The name of the component class for this control.
Interface	The name of the interface on which the control will implement its custom methods and properties.
Type	A description for the control.
ProgID	The readable name that can be used to look up the CLSID of the control.

You have to make several additional settings in the **ATL Control** wizard.

To enable support for rich error information and connection points

1. Click **Options** to open the **Options** page.
2. Select the **Connection points** check box. This will create support for an outgoing interface in the IDL file.

You can also add interfaces to extend the control's functionality.

To extend the control's functionality

1. Click **Interfaces** to open the **Interfaces** page.
2. Select `IProvideClassInfo2` and click the **Up** arrow to move it to the **Supported** list.
3. Select `ISpecifyPropertyPages` and click the **Up** arrow to move it to the **Supported** list.

You can also make the control insertable, which means it can be embedded into applications that support embedded objects, such as Excel or Word.

To make the control insertable

1. Click **Appearance** to open the **Appearance** page.
2. Select the **Insertable** check box.

The polygon displayed by the object will have a solid fill color, so you have to add a `Fill Color` stock property.

To add a Fill Color stock property and create the control

1. Click **Stock Properties** to open the **Stock Properties** page.
2. Under **Not supported**, scroll down the list of possible stock properties. Select `Fill Color` and click the **Up** arrow to move it to the **Supported** list.
3. This completes the options for the control. Click **Finish**.

As the wizard created the control, several code changes and file additions occurred. The following files were created:

FILE	DESCRIPTION
PolyCtl.h	Contains most of the implementation of the C++ class <code>CPolyCtl</code> .
PolyCtl.cpp	Contains the remaining parts of <code>CPolyCtl</code> .
PolyCtl.rgs	A text file that contains the registry script used to register the control.
PolyCtl.htm	A Web page containing a reference to the newly created control.

The wizard also performed the following code changes:

- Added an `#include` statement to the `stdafx.h` and `stdafx.cpp` files to include the ATL files necessary for supporting controls.
- Changed `Polygon.idl` to include details of the new control.
- Added the new control to the object map in `Polygon.cpp`.

Now you can build the control to see it in action.

Building and Testing the Control

To build and test the control

1. On the **Build** menu, click **Build Polygon**.

Once the control finishes building, right-click PolyCtl.htm in **Solution Explorer** and select **View in Browser**. The HTML Web page containing the control will be displayed. You should see a page with the title "ATL 8.0 test page for object PolyCtl" and the text PolyCtl. This is your control.

NOTE

If the control isn't visible, know that some browsers require settings adjustments to run ActiveX controls. Please refer to the browser's documentation on how to enable ActiveX controls.

NOTE

When completing this tutorial, if you receive an error message where the DLL file cannot be created, close the PolyCtl.htm file and the ActiveX Control Test container and build the solution again. If you still cannot create the DLL, reboot the computer or log off (if you are using Terminal Services).

Next, you will add a custom property to the control.

[Back to Step 1](#) | [On to Step 3](#)

See also

[Tutorial](#)

Adding a Property to the Control (ATL Tutorial, Part 3)

3/4/2019 • 2 minutes to read • [Edit Online](#)

`IPolyCtl` is the interface that contains the control's custom methods and properties, and you will add a property to it.

To add the property definitions to your project

1. In **Class View**, expand the `Polygon` branch.
2. Right-click `IPolyCtl`.
3. On the shortcut menu, click **Add**, and then click **Add Property**. The **Add Property** wizard will appear.
4. Type `Sides` as the **Property Name**.
5. In the drop-down list of **Property Type**, select `short`.
6. Click **OK** to finish adding the property.
7. From **Solution Explorer**, open `Polygon.idl` and replace the following lines at the end of the `IPolyCtl : IDispatch` interface:

```
short get_Sides();  
void set_Sides(short value);
```

with

```
[propget, id(1), helpstring("property Sides")] HRESULT Sides([out, retval] short *pVal);  
[propput, id(1), helpstring("property Sides")] HRESULT Sides([in] short newVal);
```

8. From **Solution Explorer**, open `PolyCtl.h` and add the following lines after the definition of `m_clrFillColor`:

```
short m_nSides;  
STDMETHOD(get_Sides)(short* pVal);  
STDMETHOD(put_Sides)(short newVal);
```

Although you now have skeleton functions to set and retrieve the property and a variable to store the property, you must implement the functions accordingly.

To update the get and put methods

1. Set the default value of `m_nSides`. Make the default shape a triangle by adding a line to the constructor in `PolyCtl.h`:

```
m_nSides = 3;
```

2. Implement the `Get` and `Put` methods. The `get_Sides` and `put_Sides` function declarations have been added to `PolyCtl.h`. Now add the code for `get_Sides` and `put_Sides` to `PolyCtl.cpp` with the following:

```

STDMETHODIMP CPolyCtl::get_Sides(short* pVal)
{
    *pVal = m_nSides;

    return S_OK;
}

STDMETHODIMP CPolyCtl::put_Sides(short newVal)
{
    if (2 < newVal && newVal < 101)
    {
        m_nSides = newVal;
        return S_OK;
    }
    else
    {
        return Error(_T("Shape must have between 3 and 100 sides"));
    }
}

```

The `get_Sides` method returns the current value of the `Sides` property through the `pVal` pointer. In the `put_Sides` method, the code ensures the user is setting the `Sides` property to an acceptable value. The minimum must be 3, and because an array of points will be used for each side, 100 is a reasonable limit for a maximum value.

You now have a property called `Sides`. In the next step, you will change the drawing code to use it.

[Back to Step 2](#) | [On to Step 4](#)

See also

[Tutorial](#)

Changing the Drawing Code (ATL Tutorial, Part 4)

3/4/2019 • 5 minutes to read • [Edit Online](#)

By default, the control's drawing code displays a square and the text **PolyCtl**. In this step, you will change the code to display something more interesting. The following tasks are involved:

- Modifying the Header File
- Modifying the `OnDraw` Function
- Adding a Method to Calculate the Polygon Points
- Initializing the Fill Color

Modifying the Header File

Start by adding support for the math functions `sin` and `cos`, which will be used calculate the polygon points, and by creating an array to store positions.

To modify the header file

1. Add the line `#include <math.h>` to the top of `PolyCtl.h`. The top of the file should look like this:

```
#include <math.h>
#include "resource.h"      // main symbols
```

2. Implement the `IProvideClassInfo` interface to provide method information for the control, by adding the following code to `PolyCtl.h`. In the `CPolyCtl` class, replace line:

```
public CComControl<CPolyCtl>
```

with

```
public CComControl<CPolyCtl>,
public IProvideClassInfo2Impl<&CLSID_PolyCtl, &DIID_IPolyCtlEvents, &LIBID_PolygonLib>
```

and in `BEGIN_COM_MAP(CPolyCtl)`, add the lines:

```
COM_INTERFACE_ENTRY(IProvideClassInfo)
COM_INTERFACE_ENTRY(IProvideClassInfo2)
```

3. Once the polygon points are calculated, they will be stored in an array of type `POINT`, so add the array after the definition statement `short m_nSides;` in `PolyCtl.h`:

```
POINT m_arrPoint[100];
```

Modifying the OnDraw Method

Now you should modify the `OnDraw` method in `PolyCtl.h`. The code you will add creates a new pen and brush with which to draw your polygon, and then calls the `Ellipse` and `Polygon` Win32 API functions to perform the actual

drawing.

To modify the OnDraw function

1. Replace the existing `OnDraw` method in `PolyCtl.h` with the following code:

```
HRESULT CPolyCtl::OnDraw(ATL_DRAWINFO& di)
{
    RECT& rc = *(RECT*)di.prcBounds;
    HDC hdc = di.hdcDraw;

    COLORREF colFore;
    HBRUSH hOldBrush, hBrush;
    HPEN hOldPen, hPen;

    // Translate m_colFore into a COLORREF type
    OleTranslateColor(m_clrFillColor, NULL, &colFore);

    // Create and select the colors to draw the circle
    hPen = (HPEN)GetStockObject(BLACK_PEN);
    hOldPen = (HPEN)SelectObject(hdc, hPen);
    hBrush = (HBRUSH)GetStockObject(WHITE_BRUSH);
    hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);

    Ellipse(hdc, rc.left, rc.top, rc.right, rc.bottom);

    // Create and select the brush that will be used to fill the polygon
    hBrush = CreateSolidBrush(colFore);
    SelectObject(hdc, hBrush);

    CalcPoints(rc);
    Polygon(hdc, &m_arrPoint[0], m_nSides);

    // Select back the old pen and brush and delete the brush we created
    SelectObject(hdc, hOldPen);
    SelectObject(hdc, hOldBrush);
    DeleteObject(hBrush);

    return S_OK;
}
```

Adding a Method to Calculate the Polygon Points

Add a method, called `CalcPoints`, that will calculate the coordinates of the points that make up the perimeter of the polygon. These calculations will be based on the `RECT` variable that is passed into the function.

To add the CalcPoints method

1. Add the declaration of `CalcPoints` to the `IPolyCtl` public section of the `CPolyCtl` class in `PolyCtl.h`:

```
void CalcPoints(const RECT& rc);
```

The last part of the public section of the `CPolyCtl` class will look like this:

```
void FinalRelease()
{
}
public:
    void CalcPoints(const RECT& rc);
```

2. Add this implementation of the `CalcPoints` function to the end of `PolyCtl.cpp`:

```

void CPolyCtl::CalcPoints(const RECT& rc)
{
    const double pi = 3.14159265358979;
    POINT    ptCenter;
    double   dblRadiusx = (rc.right - rc.left) / 2;
    double   dblRadiusy = (rc.bottom - rc.top) / 2;
    double   dblAngle = 3 * pi / 2;           // Start at the top
    double   dblDiff   = 2 * pi / m_nSides;   // Angle each side will make
    ptCenter.x = (rc.left + rc.right) / 2;
    ptCenter.y = (rc.top + rc.bottom) / 2;

    // Calculate the points for each side
    for (int i = 0; i < m_nSides; i++)
    {
        m_arrPoint[i].x = (long)(dblRadiusx * cos(dblAngle) + ptCenter.x + 0.5);
        m_arrPoint[i].y = (long)(dblRadiusy * sin(dblAngle) + ptCenter.y + 0.5);
        dblAngle += dblDiff;
    }
}

```

Initializing the Fill Color

Initialize `m_clrFillColor` with a default color.

To initialize the fill color

1. Use green as the default color by adding this line to the `CPolyCtl` constructor in `PolyCtl.h`:

```
m_clrFillColor = RGB(0, 0xFF, 0);
```

The constructor now looks like this:

```

CPolyCtl()
{
    m_nSides = 3;
    m_clrFillColor = RGB(0, 0xFF, 0);
}

```

Building and Testing the Control

Rebuild the control. Make sure the `PolyCtl.htm` file is closed if it is still open, and then click **Build Polygon** on the **Build** menu. You could view the control once again from the `PolyCtl.htm` page, but this time use the ActiveX Control Test Container.

To use the ActiveX Control Test Container

1. Build and start the ActiveX Control Test Container. The [TSTCON Sample: ActiveX Control Test Container](#) can be found on GitHub.

NOTE

For errors involving `ATL::CW2AEX`, in `Script.Cpp`, replace line

```
TRACE( "XActiveScriptSite::GetItemInfo( %s )\n", pszNameT ); with  
TRACE( "XActiveScriptSite::GetItemInfo( %s )\n", pszNameT.m_psz ); , and line  
TRACE( "Source Text: %s\n", COLE2CT( bstrSourceLineText ) ); with  
TRACE( "Source Text: %s\n", bstrSourceLineText ); .
```

For errors involving `HMONITOR`, open `StdAfx.h` in the `TCProps` project and replace:

```
#ifndef WINVER  
#define WINVER 0x0400  
#endif
```

with

```
#ifndef WINVER  
#define WINVER 0x0500  
#define _WIN32_WINNT 0x0500  
#endif
```

2. In **Test Container**, on the **Edit** menu, click **Insert New Control**.
3. Locate your control, which will be called `PolyCtl class`, and click **OK**. You will see a green triangle within a circle.

Try changing the number of sides by following the next procedure. To modify properties on a dual interface from within **Test Container**, use **Invoke Methods**.

To modify a control's property from within the Test Container

1. In **Test Container**, click **Invoke Methods** on the **Control** menu.

The **Invoke Method** dialog box is displayed.

2. Select the **PropPut** version of the **Sides** property from the **Method Name** drop-down list box.
3. Type `5` in the **Parameter Value** box, click **Set Value**, and click **Invoke**.

Note that the control does not change. Although you changed the number of sides internally by setting the `m_nSides` variable, this did not cause the control to repaint. If you switch to another application and then switch back to **Test Container**, you will find that the control has repainted and has the correct number of sides.

To correct this problem, add a call to the `FireViewChange` function, defined in `IViewObjectExImpl`, after you set the number of sides. If the control is running in its own window, `FireViewChange` will call the `InvalidateRect` method directly. If the control is running windowless, the `InvalidateRect` method will be called on the container's site interface. This forces the control to repaint itself.

To add a call to FireViewChange

1. Update `PolyCtl.cpp` by adding the call to `FireViewChange` to the `put_Sides` method. When you have finished, the `put_Sides` method should look like this:

```
STDMETHODIMP CPolyCtl::put_Sides(short newVal)
{
    if (2 < newVal && newVal < 101)
    {
        m_nSides = newVal;
        FireViewChange();
        return S_OK;
    }
    else
    {
        return Error(_T("Shape must have between 3 and 100 sides"));
    }
}
```

After adding `FireViewChange`, rebuild and try the control again in the ActiveX Control Test Container. This time when you change the number of sides and click `Invoke`, you should see the control change immediately.

In the next step, you will add an event.

[Back to Step 3](#) | [On to Step 5](#)

See also

[Tutorial](#)

[Testing Properties and Events with Test Container](#)

Adding an Event (ATL Tutorial, Part 5)

3/4/2019 • 6 minutes to read • [Edit Online](#)

In this step, you will add a `ClickIn` and a `ClickOut` event to your ATL control. You will fire the `ClickIn` event if the user clicks within the polygon and fire `ClickOut` if the user clicks outside. The tasks to add an event are as follows:

- Adding the `ClickIn` and `ClickOut` methods
- Generating the Type Library
- Implementing the Connection Point Interfaces

Adding the ClickIn and ClickOut methods

When you created the ATL control in step 2, you selected the **Connection points** check box. This created the `_IPolyCtlEvents` interface in the `Polygon.idl` file. Note that the interface name starts with an underscore. This is a convention to indicate that the interface is an internal interface. Thus, programs that allow you to browse COM objects can choose not to display the interface to the user. Also note that selecting **Connection points** added the following line in the `Polygon.idl` file to indicate that `_IPolyCtlEvents` is the default source interface:

```
[default, source] dispinterface _IPolyCtlEvents;
```

The source attribute indicates that the control is the source of the notifications, so it will call this interface on the container.

Now add the `ClickIn` and `ClickOut` methods to the `_IPolyCtlEvents` interface.

To add the ClickIn and ClickOut methods

1. In **Solution Explorer**, open `Polygon.idl` and add the following code under `methods:` in the `dispInterface_IPolyCtlEvents` declaration of the `PolygonLib` library:

```
[id(1), helpstring("method ClickIn")] void ClickIn([in] LONG x,[in] LONG y);  
[id(2), helpstring("method ClickOut")] void ClickOut([in] LONG x,[in] LONG y);
```

The `ClickIn` and `ClickOut` methods take the x and y coordinates of the clicked point as parameters.

Generating the Type Library

Generate the type library at this point, because the project will use it to obtain the information it needs to construct a connection point interface and a connection point container interface for your control.

To generate the type library

1. Rebuild your project.

-or-

2. Right-click the `Polygon.idl` file in **Solution Explorer** and click **Compile** on the shortcut menu.

This will create the `Polygon.tlb` file, which is your type library. The `Polygon.tlb` file is not visible from **Solution Explorer**, because it is a binary file and cannot be viewed or edited directly.

Implementing the Connection Point Interfaces

Implement a connection point interface and a connection point container interface for your control. In COM, events are implemented through the mechanism of connection points. To receive events from a COM object, a container establishes an advisory connection to the connection point that the COM object implements. Because a COM object can have multiple connection points, the COM object also implements a connection point container interface. Through this interface, the container can determine which connection points are supported.

The interface that implements a connection point is called `IConnectionPoint`, and the interface that implements a connection point container is called `IConnectionPointContainer`.

To help implement `IConnectionPoint`, you will use the Implement Connection Point Wizard. This wizard generates the `IConnectionPoint` interface by reading your type library and implementing a function for each event that can be fired.

To implement the connection points

1. In **Solution Explorer**, open `_IPolyCtlEvents_CP.h` and add the following code under the `public:` statement in the `CProxy_IPolyCtlEvents` class:

```

VOID Fire_ClickIn(LONG x, LONG y)
{
    T* pT = static_cast<T*>(this);
    int nConnectionIndex;
    CComVariant* pvars = new CComVariant[2];
    int nConnections = m_vec.GetSize();

    for (nConnectionIndex = 0; nConnectionIndex < nConnections; nConnectionIndex++)
    {
        pT->Lock();
        CComPtr<IUnknown> sp = m_vec.GetAt(nConnectionIndex);
        pT->Unlock();
        IDispatch* pDispatch = reinterpret_cast<IDispatch*>(sp.p);
        if (pDispatch != NULL)
        {
            pvars[1].vt = VT_I4;
            pvars[1].lVal = x;
            pvars[0].vt = VT_I4;
            pvars[0].lVal = y;
            DISPPARAMS disp = { pvars, NULL, 2, 0 };
            pDispatch->Invoke(0x1, IID_NULL, LOCALE_USER_DEFAULT, DISPATCH_METHOD, &disp, NULL, NULL, NULL);
        }
    }
    delete[] pvars;
}

VOID Fire_ClickOut(LONG x, LONG y)
{
    T* pT = static_cast<T*>(this);
    int nConnectionIndex;
    CComVariant* pvars = new CComVariant[2];
    int nConnections = m_vec.GetSize();

    for (nConnectionIndex = 0; nConnectionIndex < nConnections; nConnectionIndex++)
    {
        pT->Lock();
        CComPtr<IUnknown> sp = m_vec.GetAt(nConnectionIndex);
        pT->Unlock();
        IDispatch* pDispatch = reinterpret_cast<IDispatch*>(sp.p);
        if (pDispatch != NULL)
        {
            pvars[1].vt = VT_I4;
            pvars[1].lVal = x;
            pvars[0].vt = VT_I4;
            pvars[0].lVal = y;
            DISPPARAMS disp = { pvars, NULL, 2, 0 };
            pDispatch->Invoke(0x2, IID_NULL, LOCALE_USER_DEFAULT, DISPATCH_METHOD, &disp, NULL, NULL, NULL);
        }
    }
    delete[] pvars;
}

```

You will see that this file has a class called `CProxy_IPolyCtlEvents` that derives from `IConnectionPointImpl`. `_IPolyCtlEvents_CP.h` now defines the two methods `Fire_ClickIn` and `Fire_ClickOut`, which take the two coordinate parameters. You call these methods when you want to fire an event from your control.

By creating the control with **Connection points** option selected, the `_IPolyCtlEvents_CP.h` file was generated for you. It also added `CProxy_PolyEvents` and `IConnectionPointContainerImpl` to your control's multiple inheritance list and exposed `IConnectionPointContainer` for you by adding appropriate entries to the COM map.

You are finished implementing the code to support events. Now, add some code to fire the events at the appropriate moment. Remember, you are going to fire a `ClickIn` or `ClickOut` event when the user clicks the left mouse button in the control. To find out when the user clicks the button, add a handler for the `WM_LBUTTONDOWN`

message.

To add a handler for the WM_LBUTTONDOWN message

1. In **Class View**, right-click the `CPolyCtl` class and click **Properties** on the shortcut menu.
2. In the **Properties** window, click the **Messages** icon and then click `WM_LBUTTONDOWN` from the list on the left.
3. From the drop-down list that appears, click **<Add> OnLButtonDown**. The `OnLButtonDown` handler declaration will be added to `PolyCtl.h`, and the handler implementation will be added to `PolyCtl.cpp`.

Next, modify the handler.

To modify the OnLButtonDown method

1. Change the code which comprises the `OnLButtonDown` method in `PolyCtl.cpp` (deleting any code placed by the wizard) so that it looks like this:

```
LRESULT CPolyCtl::OnLButtonDown(UINT /*uMsg*/, WPARAM /*wParam*/, LPARAM lParam,
    BOOL& /*bHandled*/)
{
    HRGN hRgn;
    WORD xPos = LOWORD(lParam); // horizontal position of cursor
    WORD yPos = HIWORD(lParam); // vertical position of cursor

    CalcPoints(m_rcPos);

    // Create a region from our list of points
    hRgn = CreatePolygonRgn(&m_arrPoint[0], m_nSides, WINDING);

    // If the clicked point is in our polygon then fire the ClickIn
    // event otherwise we fire the ClickOut event
    if (PtInRegion(hRgn, xPos, yPos))
        Fire_ClickIn(xPos, yPos);
    else
        Fire_ClickOut(xPos, yPos);

    // Delete the region that we created
    DeleteObject(hRgn);
    return 0;
}
```

This code makes use of the points calculated in the `onDraw` function to create a region that detects the user's mouse clicks with the call to `PtInRegion`.

The `uMsg` parameter is the ID of the Windows message being handled. This allows you to have one function that handles a range of messages. The `wParam` and the `lParam` parameters are the standard values for the message being handled. The parameter `bHandled` allows you to specify whether the function handled the message or not. By default, the value is set to `TRUE` to indicate that the function handled the message, but you can set it to `FALSE`. This will cause ATL to continue looking for another message handler function to send the message to.

Building and Testing the Control

Now try out your events. Build the control and start the ActiveX Control Test Container again. This time, view the event log window. To route events to the output window, click **Logging** from the **Options** menu and select **Log to output window**. Insert the control and try clicking in the window. Note that `ClickIn` is fired if you click within the filled polygon, and `ClickOut` is fired when you click outside of it.

Next, you will add a property page.

[Back to Step 4](#) | [On to Step 6](#)

See also

[Tutorial](#)

Adding a Property Page (ATL Tutorial, Part 6)

5/10/2019 • 6 minutes to read • [Edit Online](#)

NOTE

The ATL OLE DB Provider wizard is not available in Visual Studio 2019 and later.

Property pages are implemented as separate COM objects, which allow them to be shared if required. In this step, you will do the following tasks to add a property page to the control:

- Creating the Property Page Resource
- Adding Code to Create and Manage the Property Page
- Adding the Property Page to the Control

Creating the Property Page Resource

To add a property page to your control, use the ATL Property Page template.

To add a Property Page

1. In **Solution Explorer**, right-click `Polygon`.
2. On the shortcut menu, click **Add** > **New Item**.
3. From the list of templates, select **ATL** > **ATL Property Page** and click **Add**.
4. When the **ATL Property Page Wizard** appears, enter *PolyProp* as the **Short** name.
5. Click **Strings** to open the **Strings** page and enter **&Polygon** as the **Title**.

The **Title** of the property page is the string that appears in the tab for that page. The **Doc string** is a description that a property frame uses to put in a status line or tool tip. Note that the standard property frame currently does not use this string, so you can leave it with the default contents. You will not generate a **Help file** at the moment, so delete the entry in that text box.

6. Click **Finish**, and the property page object will be created.

The following three files are created:

FILE	DESCRIPTION
PolyProp.h	Contains the C++ class <code>CPolyProp</code> , which implements the property page.
PolyProp.cpp	Includes the PolyProp.h file.
PolyProp.rgs	The registry script that registers the property page object.

The following code changes are also made:

- The new property page is added to the object entry map in Polygon.cpp.
- The `PolyProp` class is added to the Polygon.idl file.

- The new registry script file PolyProp.rgs is added to the project resource.
- A dialog box template is added to the project resource for the property page.
- The property strings that you specified are added to the resource string table.

Now add the fields that you want to appear on the property page.

To add fields to the Property Page

1. In **Solution Explorer**, double-click the Polygon.rc resource file. This will open **Resource View**.
2. In **Resource View**, expand the `Dialog` node and double-click `IDD_POLYPROP`. Note that the dialog box that appears is empty except for a label that tells you to insert your controls here.
3. Select that label and change it to read `Sides:` by altering the **Caption** text in the **Properties** window.
4. Resize the label box so that it fits the size of the text.
5. Drag an **Edit Control** from the **Toolbox** to the right of the label.
6. Finally, change the **ID** of the edit control to `IDC_SIDES` using the **Properties** window.

This completes the process of creating the property page resource.

Adding Code to Create and Manage the Property Page

Now that you have created the property page resource, you need to write the implementation code.

First, enable the `CPolyProp` class to set the number of sides in your object when the **Apply** button is pressed.

To modify the Apply function to set the number of sides

1. Replace the `Apply` function in PolyProp.h with the following code:

```
STDMETHOD(Apply)(void)
{
    USES_CONVERSION;
    ATLTRACE(_T("CPolyProp:Apply\n"));
    for (UINT i = 0; i < m_nObjects; i++)
    {
        CComQIPtr<IPolyCtl, &IID_IPolyCtl> pPoly(m_ppUnk[i]);
        short nSides = (short)GetDlgItemInt(IDC_SIDES);
        if FAILED(pPoly->put_Sides(nSides))
        {
            CComPtr<IErrorInfo> pError;
            CComBSTR strError;
            GetErrorInfo(0, &pError);
            pError->GetDescription(&strError);
            MessageBox(OLE2T(strError), _T("Error"), MB_ICONEXCLAMATION);
            return E_FAIL;
        }
    }
    m_bDirty = FALSE;
    return S_OK;
}
```

A property page can have more than one client attached to it at a time, so the `Apply` function loops around and calls `put_Sides` on each client with the value retrieved from the edit box. You are using the `CComQIPtr` class, which performs the `QueryInterface` on each object to obtain the `IPolyCtl` interface from the `IUnknown` interface (stored in the `m_ppUnk` array).

The code now checks that setting the `Sides` property actually worked. If it fails, the code displays a message box

displaying error details from the `IErrorInfo` interface. Typically, a container asks an object for the `ISupportErrorInfo` interface and calls `InterfaceSupportsErrorInfo` first, to determine whether the object supports setting error information. You can skip this task.

`CComPtr` helps you by automatically handling the reference counting, so you do not need to call `Release` on the interface. `CComBSTR` helps you with BSTR processing, so you do not have to perform the final `SysFreeString` call. You also use one of the various string conversion classes, so you can convert the BSTR if necessary (this is why the `USES_CONVERSION` macro is at the start of the function).

You also need to set the property page's dirty flag to indicate that the **Apply** button should be enabled. This occurs when the user changes the value in the **Sides** edit box.

To handle the Apply button

1. In **Class View**, right-click `CPolyProp` and click **Properties** on the shortcut menu.
2. In the **Properties** window, click the **Events** icon.
3. Expand the `IDC_SIDES` node in the event list.
4. Select `EN_CHANGE`, and from the drop-down menu to the right, click **<Add> OnEnChangeSides**. The `OnEnChangeSides` handler declaration will be added to `Polyprop.h`, and the handler implementation to `Polyprop.cpp`.

Next, you will modify the handler.

To modify the OnEnChangeSides method

1. Add the following code in `Polyprop.cpp` to the `OnEnChangeSides` method (deleting any code that the wizard put there):

```
LRESULT CPolyProp::OnEnChangeSides(WORD /*wNotifyCode*/, WORD /*wID*/,
    HWND /*hWndCtl*/, BOOL& /*bHandled*/)
{
    SetDirty(TRUE);

    return 0;
}
```

`OnEnChangeSides` will be called when a `WM_COMMAND` message is sent with the `EN_CHANGE` notification for the `IDC_SIDES` control. `OnEnChangeSides` then calls `SetDirty` and passes `TRUE` to indicate the property page is now dirty and the **Apply** button should be enabled.

Adding the Property Page to the Control

The ATL Property Page template and wizard do not add the property page to your control for you automatically, because there could be multiple controls in your project. You will need to add an entry to the control's property map.

To add the property page

1. Open `PolyCtl.h` and add these lines to the property map:

```
PROP_ENTRY_TYPE("Sides", 1, CLSID_PolyProp, VT_INT)
PROP_PAGE(CLSID_PolyProp)
```

The control's property map now looks like this:

```

BEGIN_PROP_MAP(CPolyCtl)
    PROP_DATA_ENTRY("_cx", m_sizeExtent.cx, VT_UI4)
    PROP_DATA_ENTRY("_cy", m_sizeExtent.cy, VT_UI4)
#ifdef _WIN32_WCE
    PROP_ENTRY_TYPE("FillColor", DISPID_FILLCOLOR, CLSID_StockColorPage, VT_UI4)
#endif
    PROP_ENTRY_TYPE("Sides", 1, CLSID_PolyProp, VT_INT)
    PROP_PAGE(CLSID_PolyProp)
    // Example entries
    // PROP_ENTRY("Property Description", dispid, clsid)
    // PROP_PAGE(CLSID_StockColorPage)
END_PROP_MAP()

```

You could have added a `PROP_PAGE` macro with the CLSID of your property page, but if you use the `PROP_ENTRY` macro as shown, the `Sides` property value is also saved when the control is saved.

The three parameters to the macro are the property description, the DISPID of the property, and the CLSID of the property page that has the property on it. This is useful if, for example, you load the control into Visual Basic and set the number of Sides at design time. Because the number of Sides is saved, when you reload your Visual Basic project, the number of Sides will be restored.

Building and Testing the Control

Now build that control and insert it into ActiveX Control Test Container. In **Test Container**, on the **Edit** menu, click **PolyCtl Class Object**. The property page appears with the information you added.

The **Apply** button is initially disabled. Start typing a value in the **Sides** box and the **Apply** button will become enabled. After you have finished entering the value, click the **Apply** button. The control display changes, and the **Apply** button is again disabled. Try entering an invalid value. You will see a message box containing the error description that you set from the `put_Sides` function.

Next, you will put your control on a Web page.

[Back to Step 5](#) | [On to Step 7](#)

See also

[Tutorial](#)

Putting the Control on a Web Page (ATL Tutorial, Part 7)

5/7/2019 • 3 minutes to read • [Edit Online](#)

Your control is now finished. To see your control work in a real-world situation, put it on a Web page. An HTML file that contains the control was created when you defined your control. Open the PolyCtl.htm file from **Solution Explorer**, and you can see your control on a Web page.

In this step, you will add functionality to the control and script the Web page to respond to events. You will also modify the control to let Internet Explorer know that the control is safe for scripting.

Adding new functionality

To add control features

1. Open PolyCtl.cpp and replace the following code:

```
if (PtInRegion(hRgn, xPos, yPos))
    Fire_ClickIn(xPos, yPos);
else
    Fire_ClickOut(xPos, yPos);
```

with

```
short temp = m_nSides;
if (PtInRegion(hRgn, xPos, yPos))
{
    Fire_ClickIn(xPos, yPos);
    put_Sides(++temp);
}
else
{
    Fire_ClickOut(xPos, yPos);
    put_Sides(--temp);
}
```

The shape will now add or remove sides depending on where you click.

Scripting the Web Page

The control does not do anything yet, so change the Web page to respond to the events that you send.

To script the Web page

1. Open PolyCtl.htm and select HTML view. Add the following lines to the HTML code. They should be added after `</OBJECT>` but before `</BODY>`.

```
<SCRIPT LANGUAGE="VBScript">
<!--
    Sub PolyCtl_ClickIn(x, y)
        MsgBox("Clicked (" & x & ", " & y & ") - adding side")
    End Sub
    Sub PolyCtl_ClickOut(x, y)
        MsgBox("Clicked (" & x & ", " & y & ") - removing side")
    End Sub
-->
</SCRIPT>
```

2. Save the HTM file.

You have added some VBScript code that gets the Sides property from the control and increases the number of sides by one if you click inside the control. If you click outside the control, you reduce the number of sides by one.

Indicating that the Control Is Safe for Scripting

You can view the Web page with the control in Internet Explorer only. Other browsers no longer support ActiveX controls because of security weaknesses.

NOTE

If the control isn't visible, know that some browsers require settings adjustments to run ActiveX controls. Please refer to the browser's documentation on how to enable ActiveX controls.

Based on your current Internet Explorer security settings, you may receive a Security Alert dialog box stating that the control may not be safe to script and could potentially do damage. For example, if you had a control that displayed a file but also had a `Delete` method that deleted a file, it would be safe if you just viewed it on a page. It would be not safe to script, however, because someone could call the `Delete` method.

IMPORTANT

For this tutorial, you can change your security settings in Internet Explorer to run ActiveX controls that are not marked as safe. In Control Panel, click **Internet Properties** and click **Security** to change the appropriate settings. When you have completed the tutorial, change your security settings back to their original state.

You can programmatically alert Internet Explorer that it does not need to display the Security Alert dialog box for this particular control. You can do this with the `IObjectSafety` interface, and ATL supplies an implementation of this interface in the class `IObjectSafetyImpl`. To add the interface to your control, add `IObjectSafetyImpl` to your list of inherited classes and add an entry for it in your COM map.

To add IObjectSafetyImpl to the control

1. Add the following line to the end of the list of inherited classes in `PolyCtl.h` and add a comma to the previous line:

```
public IObjectSafetyImpl<CPolyCtl, INTERFACESAFE_FOR_UNTRUSTED_CALLER>
```

2. Add the following line to the COM map in `PolyCtl.h`:

```
COM_INTERFACE_ENTRY(IObjectSafety)
```

Building and Testing the Control

Build the control. Once the build has finished, open PolyCtl.htm in browser view again. This time, the Web page should be displayed directly without the **Safety Alert** dialog box. Click inside the polygon; the number of sides increases by one. Click outside the polygon to reduce the number of sides.

[Back to Step 6](#)

Next Steps

This concludes the ATL tutorial. For links to more information about ATL, see the [ATL start page](#).

See also

[Tutorial](#)

ATL Class Overview

3/12/2019 • 2 minutes to read • [Edit Online](#)

Classes in the Active Template Library (ATL) can be categorized as follows:

Class Factories	Memory Management
Class Information	MMC Snap-In
Collection	Object Safety
COM Modules	Persistence
Composite Controls	Properties and Property Pages
Connection Points	Registry Support
Control Containment	Running Objects
Controls: General Support	Security
Data Transfer	Service Provider Support
Data Types	Site Information
Debugging and Exception	String and Text
Dual Interfaces	Tear-Off Interfaces
Enumerators and Collections	Thread Pooling
Error Information	Threading Models and Critical Sections
File Handling	UI Support
Interface Pointers	Windows Support
IUnknown Implementation	Utility

For additional classes that can be used in ATL projects, see [Shared Classes](#).

See also

[Classes and structs](#)

[ATL COM Desktop Components](#)

[Functions](#)

[Global Variables](#)

[Macros](#)

Class Factories Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes implement or support a class factory:

- [CComClassFactory](#) Provides a default class factory for object creation.
- [CComClassFactory2](#) Controls object creation through a license.
- [CComClassFactoryAutoThread](#) Allows objects to be created in multiple thread-pooled apartments.
- [CComClassFactorySingleton](#) Creates a single object.
- [CComCoClass](#) Defines the class factory for the object.

See also

[Class Overview](#)

[Aggregation and Class Factory Macros](#)

Class Information Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following class provides support for retrieving class information:

- [IProvideClassInfo2Impl](#) Provides access to type information. Retrieves the outgoing IID for the object's default event set.

See also

[Class Overview](#)

Collection Classes

3/4/2019 • 2 minutes to read • [Edit Online](#)

The following classes provide support for arrays, lists, maps, and also traits methods for helping with comparisons and element access.

- [CAtlArray](#) This class implements an array object.
- [CAtlList](#) This class provides methods for creating and managing a list object.
- [CAtlMap](#) This class provides methods for creating and managing a map object.
- [CAutoPtrArray](#) This class provides methods useful when constructing an array of smart pointers.
- [CAutoPtrElementTraits](#) This class provides methods, static functions, and typedefs useful when creating collections of smart pointers.
- [CAutoPtrList](#) This class provides methods useful when constructing a list of smart pointers.
- [CAutoVectorPtrElementTraits](#) This class provides methods, static functions, and typedefs useful when creating collections of smart pointers using vector new and delete operators.
- [CComQIPtrElementTraits](#) This class provides methods, static functions, and typedefs useful when creating collections of COM interface pointers.
- [CComSafeArray](#) This class is a wrapper for the [SAFEARRAY Data Type](#) structure.
- [CComSafeArrayBound](#) This class is a wrapper for a [SAFEARRAYBOUND](#) structure.
- [CComUnkArray](#) This class stores **IUnknown** pointers and is designed to be used as a parameter to the [IConnectionPointImpl](#) template class.
- [CDefaultCharTraits](#) This class provides two static functions for converting characters between uppercase and lowercase.
- [CDefaultCompareTraits](#) This class provides default element comparison functions.
- [CDefaultElementTraits](#) This class provides default methods and functions for a collection class.
- [CDefaultHashTraits](#) This class provides a static function for calculating hash values.
- [CElementTraits](#) This class is used by collection classes to provide methods and functions for moving, copying, comparison, and hashing operations.
- [CElementTraitsBase](#) This class provides default copy and move methods for a collection class.
- [CHepPtrElementTraits](#) This class provides methods, static functions, and typedefs useful when creating collections of heap pointers.
- [CHepPtrList](#) This class provides methods useful when constructing a list of heap pointers.
- [CInterfaceArray](#) This class provides methods useful when constructing an array of COM interface pointers.
- [CInterfaceList](#) This class provides methods useful when constructing a list of COM interface pointers.
- [CPrimitiveElementTraits](#) This class provides default methods and functions for a collection class composed of primitive data types.

- [CRBMap](#) This class represents a mapping structure, using a Red-Black binary tree.
- [CRBMultiMap](#) This class represents a mapping structure that allows each key to be associated with more than one value, using a Red-Black binary tree.
- [CRBTree](#) This class provides methods for creating and utilizing a Red-Black tree.
- [CSimpleArray](#) This class provides methods for managing a simple array.
- [CSimpleArrayEqualHelper](#) This class is a helper for the [CSimpleArray](#) class.
- [CSimpleArrayEqualHelperFalse](#) This class is a helper for the [CSimpleArray](#) class.
- [CSimpleMap](#) This class provides support for a simple mapping array.
- [CSimpleMapEqualHelper](#) This class is a helper for the [CSimpleMap](#) class.
- [CSimpleMapEqualHelperFalse](#) This class is a helper for the [CSimpleMap](#) class.
- [CStringElementTraits](#) This class provides static functions used by collection classes storing `CString` objects.
- [CStringElementTraitsI](#) This class provides static functions related to strings stored in collection class objects. It is similar to [CStringElementTraits](#), but performs case-insensitive comparisons.
- [CStringRefElementTraits](#) This class provides static functions related to strings stored in collection class objects. The string objects are dealt with as references.

Related Articles

[ATL Collection Classes](#)

See also

[Class Overview](#)

[Collection Classes](#)

COM Modules Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes provide support for a COM module:

- [CAtlBaseModule](#) This class is instantiated in every ATL project.
- [CAtlComModule](#) This class implements a COM server module.
- [CAtlModule](#) This class provides methods used by several ATL module classes.
- [CAtlModuleT](#) This class implements an ATL module.
- [CAtlExeModuleT](#) This class represents the module for an application.
- [CAtlServiceModuleT](#) This class implements a service.
- [CAtlWinModule](#) This class provides support for ATL windowing components.
- [CComModule](#) This class implements a DLL or EXE module. Obsolete in ATL 7.0.
- [CComAutoThreadModule](#) This class implements an EXE module, with support for multiple thread-pooled apartments. Obsolete in ATL 7.0.

Related Articles

[ATL Module Classes](#)

See also

[Class Overview](#)

[Module Classes](#)

Composite Controls Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following class provides support for creating composite controls

- [CComCompositeControl](#) ActiveX controls derived from `CComCompositeControl1` are hosted by a standard dialog box. These types of controls are called composite controls because they are able to host other controls (native Windows controls and ActiveX controls).

Related Articles

[Composite Control Fundamentals](#)

See also

[Class Overview](#)

[Composite Control Macros](#)

[Composite Control Global Functions](#)

Connection Points Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes provide support for connection points:

- [IConnectionPointContainerImpl](#) Implements a connection point container.
- [IConnectionPointImpl](#) Implements a connection point.
- [IPropertyNotifySinkCP](#) Implements a connection point representing the [IPropertyNotifySink](#) interface.
- [CComDynamicUnkArray](#) Manages unlimited connections between a connection point and its sinks.
- [CComUnkArray](#) Manages a fixed number of connections between a connection point and its sinks.
- [CFirePropNotifyEvent](#) Notifies a client's sink that an object's property has changed or is about to change.
- [IDispEventImpl](#) Provides support for connection points for an ATL COM object. These connection points are mapped with an event sink map, which is provided by your COM object.
- [IDispEventSimpleImpl](#) Works in conjunction with the event sink map in your class to route events to the appropriate handler function.

Related Articles

[Connection Points](#)

[Event Handling and ATL](#)

See also

[Class Overview](#)

[Connection Point Macros](#)

[Connection Point Global Functions](#)

Control Containment Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes provide containment support for hosting controls:

- [CAxWindow](#) Provides methods for manipulating a window that hosts an ActiveX control.
- [CAxWindow2T](#) Provides methods for manipulating a window that hosts an ActiveX control and also has support for hosting licensed ActiveX controls.
- [IAxWinAmbientDispatch](#) Call the methods on this interface to set the ambient properties available to a hosted control.
- [IAxWinHostWindow](#) Call the methods on this interface to create and/or attach a control to a host object, or to get an interface from a hosted control.

Related Articles

[ATL Control Containment FAQ](#)

See also

[Class Overview](#)

Controls: General Support Classes

4/1/2019 • 2 minutes to read • [Edit Online](#)

The following classes provide general support for ATL controls:

- [CComControl](#) Consists of helper functions and data members that are essential to ATL controls.
- [IOleControlImpl](#) Provides methods necessary for controls.
- [IOleObjectImpl](#) Provides the principal methods through which a container communicates with a control. Manages the activation and deactivation of in-place controls.
- [IQuickActivateImpl](#) Combines initialization into a single call to help containers avoid delays when loading controls.
- [IPointerInactiveImpl](#) Provides minimal mouse interaction for an otherwise inactive control.

Sample Program

[ATLFire](#)

Related Articles

[ATL Tutorial](#)

See also

[Class Overview](#)

Data Transfer Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes support various types of data transfer:

- [IDataObjectImpl](#) Supports Uniform Data Transfer by using standard formats to retrieve and set data. Handles data change notifications by managing connections to advise sinks.
- [CBindStatusCallback](#) Allows an asynchronous moniker to send and receive information about the asynchronous data transfer to and from your object.

See also

[Class Overview](#)

Data Types Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes wrap C++ data types:

- [CComBSTR](#) Wraps the `BSTR` data type.
- [CComVariant](#) Wraps the `VARIANT` data type.
- [CComCurrency](#) Includes methods and operators for creating and managing a `CURRENCY` object.

See also

[Class Overview](#)

Debugging and Exceptions Classes

3/4/2019 • 2 minutes to read • [Edit Online](#)

These classes provide support for exception handling and debugging.

- [CAtlDebugInterfacesModule](#) This class provides support for debugging interfaces.
- [CAtlException](#) This class defines an ATL exception.

See also

[Class Overview](#)

[Debugging and Error Reporting Global Functions](#)

[Debugging and Error Reporting Macros](#)

Dual Interfaces Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following class provides support for dual interfaces:

- [IDispatchImpl](#) Implements the `IDispatch` portion of a dual interface. For more information, see [Implementing the IDispatch Interface](#).

See also

[Class Overview](#)

Enumerators and Collections Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes provide support for COM collections and enumerations:

- [CComEnum](#) Defines a COM enumerator object based on an array.
- [CComEnumImpl](#) Provides the implementation for a COM enumerator interface where the items being enumerated are stored in an array.
- [CComEnumOnSTL](#) Defines a COM enumerator object based on a C++ Standard Library collection.
- [IEnumOnSTLImpl](#) Provides the implementation for a COM enumerator interface where the items being enumerated are stored in a C++ Standard Library-compatible container.
- [ICollectionOnSTLImpl](#) Provides the implementation for the `Count`, `Item`, and `_NewEnum` properties of a collection interface.

Related Articles

[ATL Collections and Enumerators](#)

See also

[Class Overview](#)

Error Information Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following class indicates how error information is handled:

- [ISupportErrorInfoImpl](#) Determines whether the object supports the [IErrorInfo](#) interface. `IErrorInfo` allows error information to be propagated back to the client.

See also

[Class Overview](#)

File Handling Classes

3/4/2019 • 2 minutes to read • [Edit Online](#)

These classes provide methods for handling files, temporary files, and memory-mapped files.

- [CAtlFile](#) This class provides a thin wrapper around the Windows file-handling API.
- [CAtlFileMapping](#) This class represents a memory-mapped file, adding a cast operator to the methods of [CAtlFileMappingBase](#).
- [CAtlFileMappingBase](#) This class represents a memory-mapped file.
- [CAtlTemporaryFile](#) This class provides methods for the creation and use of a temporary file.

See also

[Class Overview](#)

Interface Pointers Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes manage a given interface pointer:

- [CComPtr](#) Performs automatic reference counting.
- [CComQIPtr](#) Similar to `CComPtr`, but also performs automatic querying of interfaces.
- [CInterfaceArray](#) Provides methods useful when constructing an array of COM interface pointers.
- [CInterfaceList](#) Provides methods useful when constructing a list of COM interface pointers.
- [CComGITPtr](#) Provides methods for dealing with interface pointers and the global interface table (GIT).

See also

[Class Overview](#)

IUnknown Implementation Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes implement `IUnknown` and related methods:

- [CComObjectRootEx](#) Manages reference counting for both aggregated and nonaggregated objects. Allows you to specify a threading model.
- [CComObjectRoot](#) Manages reference counting for both aggregated and nonaggregated objects. Uses the default threading model of the server.
- [CComAggObject](#) Implements `IUnknown` for an aggregated object.
- [CComObject](#) Implements `IUnknown` for a nonaggregated object.
- [CComPolyObject](#) Implements `IUnknown` for aggregated and nonaggregated objects. Using `CComPolyObject` avoids having both `CComAggObject` and `CComObject` in your module. A single `CComPolyObject` object handles both aggregated and nonaggregated cases.
- [CComObjectNoLock](#) Implements `IUnknown` for a nonaggregated object, without modifying the module lock count.
- [CComTearOffObject](#) Implements `IUnknown` for a tear-off interface.
- [CComCachedTearOffObject](#) Implements `IUnknown` for a "cached" tear-off interface.
- [CComContainedObject](#) Implements `IUnknown` for the inner object of an aggregation or a tear-off interface.
- [CComObjectGlobal](#) Manages a reference count on the module to ensure your object won't be deleted.
- [CComObjectStack](#) Creates a temporary COM object, using a skeletal implementation of `IUnknown`.

Related Articles

[Fundamentals of ATL COM Objects](#)

See also

[Class Overview](#)

[Aggregation and Class Factory Macros](#)

[COM Map Macros](#)

[COM Map Global Functions](#)

Memory Management Classes

3/4/2019 • 2 minutes to read • [Edit Online](#)

These classes provide support for heap pointers, smart pointers, and other memory allocation routines.

- [CAutoPtr](#) This class represents a smart pointer object.
- [CAutoPtrArray](#) This class provides methods useful when constructing an array of smart pointers.
- [CAutoPtrList](#) This class provides methods useful when constructing a list of smart pointers.
- [CAutoVectorPtr](#) This class represents a smart pointer object using vector new and delete operators.
- [CComAllocator](#) This class provides methods for managing memory using COM memory routines.
- [CComGITPtr](#) This class provides methods for dealing with interface pointers and the global interface table (GIT).
- [CComHeap](#) This class implements [IAtlMemMgr](#) using the COM memory allocation functions.
- [CComHeapPtr](#) A smart pointer class for managing heap pointers.
- [CComPtr](#) A smart pointer class for managing COM interface pointers.
- [CComPtrBase](#) This class provides a basis for smart pointer classes using COM-based memory routines.
- [CComQIPtr](#) A smart pointer class for managing COM interface pointers.
- [CCRTAllocator](#) This class provides methods for managing memory using CRT memory routines.
- [CCRTHeap](#) This class implements [IAtlMemMgr](#) using the CRT heap functions.
- [CGlobalHeap](#) This class implements [IAtlMemMgr](#) using the Win32 global heap functions.
- [CHandle](#) This class provides methods for creating and using a handle object.
- [CHeapPtr](#) A smart pointer class for managing heap pointers.
- [CHeapPtrBase](#) This class forms the basis for several smart heap pointer classes.
- [CHeapPtrList](#) This class provides methods useful when constructing a list of heap pointers.
- [CLocalHeap](#) This class implements [IAtlMemMgr](#) using the Win32 local heap functions.
- [CWin32Heap](#) This class implements [IAtlMemMgr](#) using the Win32 heap allocation functions.
- [IAtlMemMgr](#) This class represents the interface to a memory manager.

See also

[Class Overview](#)

MMC Snap-In Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes provide support for developing Microsoft Management Console (MMC) snap-in components:

- [CSnapInItemImpl](#) Implements a snap-in node object, such as adding menu items and toolbars, and forwarding commands for the snap-in node to the appropriate handler function.
- [CSnapInPropertyPageImpl](#) Implements a snap-in property page object.

See also

[Class Overview](#)

[Snap-In Object Macros](#)

Object Safety Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following class provides support for object safety:

- [IObjectSafetyImpl](#) Allows an object to be marked as safe for initialization or safe for scripting.

Related Articles

[ATL Tutorial](#)

See also

[Class Overview](#)

Persistence Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes implement object persistence:

- [IPersistPropertyBagImpl](#) Allows a client to load and save an object's properties to a property bag.
- [IPersistStreamInitImpl](#) Allows a client to load and save an object's persistent data to a stream.
- [IPersistStorageImpl](#) Allows a client to load and save an object's persistent data to a storage.

Related Articles

[ATL Tutorial](#)

See also

[Class Overview](#)

[Property Map Macros](#)

Properties and Property Pages Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes support properties and property pages:

- [CComDispatchDriver](#) Retrieves or sets an object's properties through an `IDispatch` pointer.
- [CStockPropImpl](#) Implements the stock properties supported by ATL.
- [IPerPropertyBrowsingImpl](#) Accesses the information in an object's property pages.
- [IPersistPropertyBagImpl](#) Stores an object's properties in a client-supplied property bag.
- [IPropertyPageImpl](#) Manages a particular property page within a property sheet.
- [IPropertyPage2Impl](#) Similar to `IPropertyPageImpl`, but also allows a client to select a specific property in a property page.
- [ISpecifyPropertyPagesImpl](#) Obtains the CLSIDs for the property pages supported by an object.

Related Articles

[ATL Tutorial](#)

[ATL COM Property Pages](#)

See also

[Class Overview](#)

[Property Map Macros](#)

Registry Support Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following class provides registry support:

- [CRegKey](#) Contains methods for manipulating values in the system registry.

Related Articles

[The ATL Registry Component \(Registrar\)](#)

See also

[Class Overview](#)

[Registry Macros](#)

Running Objects Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following class provides support for running objects:

- [IRunnableObjectImpl](#) Determines if an object is running, forces it to run, or locks it into the running state.

Related Articles

[ATL Tutorial](#)

See also

[Class Overview](#)

Security Classes

3/4/2019 • 2 minutes to read • [Edit Online](#)

These classes are wrappers for common Win32 security classes and objects.

- [CAccessToken](#) This class is a wrapper for an access token.
- [CAcl](#) This class is a wrapper for an `ACL` (access-control list) structure.
- [CDacl](#) This class is a wrapper for a `DACL` (discretionary access-control list) structure.
- [CPrivateObjectSecurityDesc](#) This class represents a private object security descriptor object.
- [CSacl](#) This class is a wrapper for a `SACL` (system access-control list) structure.
- [CSecurityAttributes](#) This class is a thin wrapper for the `SECURITY_ATTRIBUTES` structure.
- [CSecurityDesc](#) This class is a wrapper for the `SECURITY_DESCRIPTOR` structure.
- [CSid](#) This class is a wrapper for a `SID` (security identifier) structure.
- [CTokenGroups](#) This class is a wrapper for the `TOKEN_GROUPS` structure.
- [CTokenPrivileges](#) This class is a wrapper for the `TOKEN_PRIVILEGES` structure.

See also

[Class Overview](#)

[Security Global Functions](#)

Service Provider Support Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following class provides support for service providers:

- [IServiceProviderImpl](#) Locates a service specified by its GUID and returns the interface pointer for the requested interface on the service.

See also

[Class Overview](#)

Site Information Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes allow an object to communicate with its site:

- [IObjectWithSiteImpl](#) Retrieves and sets a pointer to an object's site. Used for objects that are not controls.
- [IOleObjectImpl](#) Retrieves and sets a pointer to an object's site. Used for controls.

See also

[Class Overview](#)

String and Text Classes

3/4/2019 • 2 minutes to read • [Edit Online](#)

These classes provide support for strings and text string conversions.

- [CA2AEX](#) This class is used by the string conversion macros CA2TEX and CT2AEX, and the typedef CA2A.
- [CA2CAEX](#) This class is used by string conversion macros CA2CTEX and CT2CAEX, and the typedef CA2CA.
- [CA2WEX](#) This class is used by the string conversion macros CA2TEX, CA2CTEX, CT2WEX, and CT2CWEX, and the typedef CA2W.
- [CW2AEX](#) This class is used by the string conversion macros CT2AEX, CW2TEX, CW2CTEX, and CT2CAEX, and the typedef CW2A.
- [CW2CWEX](#) This class is used by the string conversion macros CW2CTEX and CT2CWEX, and the typedef CW2CW.
- [CW2WEX](#) This class is used by the string conversion macros CW2TEX and CT2WEX, and the typedef CW2W.
- [CComBSTR](#) This class is a wrapper for BSTRs.
- [_U_STRINGorID](#) This argument adapter class allows either resource names (`LPCTSTR`s) or resource IDs (UINTs) to be passed to a function without requiring the caller to convert the ID to a string using the `MAKEINTRESOURCE` macro.

See also

[Class Overview](#)

[ATL and MFC String Conversion Macros](#)

Tear-Off Interfaces Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes provide support for tear-off interfaces:

- [CComTearOffObject](#) Implements `IUnknown` for a tear-off interface.
- [CComCachedTearOffObject](#) Implements `IUnknown` for a "cached" tear-off interface.

See also

[Class Overview](#)

Thread Pooling Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes support thread pooling:

- [CComAutoThreadModule](#) Implements an EXE module, with support for multiple thread-pooled apartments.
- [CComApartment](#) Manages an apartment in a thread-pooled EXE module.
- [CComSimpleThreadAllocator](#) Manages thread selection for an EXE module.

See also

[Class Overview](#)

Threading Models and Critical Sections Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes define a threading model and critical section:

- [CAtlAutoThreadModule](#) Implements a thread-pooled, apartment-model COM server.
- [CAtlAutoThreadModuleT](#) Provides methods for implementing a thread-pooled, apartment-model COM server.
- [CComMultiThreadModel](#) Provides thread-safe methods for incrementing and decrementing a variable. Provides a critical section.
- [CComMultiThreadModelNoCS](#) Provides thread-safe methods for incrementing and decrementing a variable. Does not provide a critical section.
- [CComSingleThreadModel](#) Provides methods for incrementing and decrementing a variable. Does not provide a critical section.
- [CComObjectThreadModel](#) Determines the appropriate threading-model class for a single object class.
- [CComGlobalsThreadModel](#) Determines the appropriate threading-model class for an object that is globally available.
- [CComAutoCriticalSection](#) Contains methods for obtaining and releasing a critical section. The critical section is automatically initialized.
- [CComCriticalSection](#) Contains methods for obtaining and releasing a critical section. The critical section must be explicitly initialized.
- [CComFakeCriticalSection](#) Mirrors the methods in `CComCriticalSection` without providing a critical section. The methods in `CComFakeCriticalSection` do nothing.
- [CRTThreadTraits](#) Provides the creation function for a CRT thread. Use this class if the thread will use CRT functions.
- [Win32ThreadTraits](#) Provides the creation function for a Windows thread. Use this class if the thread will not use CRT functions.

See also

[Class Overview](#)

UI Support Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes provide general UI support:

- [IDocHostUIHandlerDispatch](#) An interface to the Microsoft HTML parsing and rendering engine.
- [IOleObjectImpl](#) Provides the principal methods through which a container communicates with a control. Manages the activation and deactivation of in-place controls.
- [IOleInPlaceObjectWindowlessImpl](#) Manages the reactivation of in-place controls. Enables a windowless control to receive messages, as well as to participate in drag-and-drop operations.
- [IOleInPlaceActiveObjectImpl](#) Assists communication between an in-place control and its container.
- [IViewObjectExImpl](#) Enables a control to display itself directly and to notify the container of changes in its display. Provides support for flicker-free drawing, non-rectangular and transparent controls, and hit testing.

Related Articles

[ATL Tutorial](#)

See also

[Class Overview](#)

Utility Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following MFC-independent utility classes are provided:

- [CImage](#) Provides enhanced bitmap support, including the ability to load and save images in JPEG, GIF, BMP, and Portable Network Graphics (PNG) formats.
- [CPoint](#) Provides an implementation for storing coordinate (x, y) pairs.
- [CRect](#) Provides an implementation for storing coordinates of rectangular areas.
- [CSize](#) Provides an implementation for storing distance, relative positions, or paired values.
- [CString](#) Provides an implementation for storing character strings.
- [CAdapt](#) A simple template used to wrap classes that redefine the address-of operator.
- [_U_RECT](#) An argument adapter class that allows either `RECT` pointers or references to be passed to a function that is implemented in terms of pointers.

See also

[Class Overview](#)

Windows Support Classes

3/12/2019 • 2 minutes to read • [Edit Online](#)

The following classes provide support for windows:

- [_U_MENUorID](#) Provides wrappers for `CreateWindow` and `CreateWindowEx`.
- [CWindow](#) Contains methods for manipulating a window. `CWindow` is the base class for `CWindowImpl`, `CDialogImpl`, and `CContainedWindow`.
- [CWindowImpl](#) Implements a window based on a new window class. Also allows you to subclass or superclass the window.
- [CDialogImpl](#) Implements a dialog box.
- [CAxDialogImpl](#) Implements a dialog box (modal or modeless) that hosts ActiveX controls.
- [CSimpleDialog](#) Implements a dialog box (modal or modeless) with basic functionality.
- [CAxWindow](#) Manipulates a window that hosts an ActiveX control.
- [CAxWindow2T](#) Provides methods for manipulating a window that hosts an ActiveX control and also has support for hosting licensed ActiveX controls.
- [CContainedWindowT](#) Implements a window contained within another object.
- [CWndClassInfo](#) Manages the information of a new window class.
- [CDynamicChain](#) Supports dynamic chaining of message maps.
- [CMessageMap](#) Allows an object to expose its message maps to other objects.
- [CWinTraits](#) Provides a simple method of standardizing the traits of an ATL window object.
- [CWinTraitsOR](#) Provides default values for window styles and extended styles used to create a window. These values are added, using the logical-OR operator, to values provided during the creation of a window.

Related Articles

[ATL Window Classes](#)

[ATL Tutorial](#)

See also

[Class Overview](#)

[Message Map Macros](#)

[Window Class Macros](#)

ATL classes and structs

3/4/2019 • 14 minutes to read • [Edit Online](#)

The Active Template Library (ATL) includes the following classes and structs. To find a particular class by category, see the [ATL Class Overview](#).

CLASS / STRUCT	DESCRIPTION	HEADER FILE
ATL_DRAWINFO	Contains information used for rendering to various targets, such as a printer, metafile, or ActiveX control.	atlctl.h
_AtlCreateWndData	Contains class instance data in windowing code in ATL.	atlbase.h
_ATL_BASE_MODULE70	Used by any project that uses ATL.	atlbase.h
_ATL_COM_MODULE70	Used by COM-related code in ATL.	atlbase.h
_ATL_FUNC_INFO	Contains type information used to describe a method or property on a dispinterface.	atlcom.h
_ATL_MODULE70	Contains data used by every ATL module.	atlbase.h
_ATL_WIN_MODULE70	Used by windowing code in ATL.	atlbase.h
CA2AEX	This class is used by the string conversion macros CA2TEX and CT2AEX, and the typedef CA2A.	atlconv.h
CA2CAEX	This class is used by string conversion macros CA2CTEX and CT2CAEX, and the typedef CA2CA.	atlconv.h
CA2WEX	This class is used by the string conversion macros CA2TEX, CA2CTEX, CT2WEX, and CT2CWEX, and the typedef CA2W.	atlconv.h
CAccessToken	This class is a wrapper for an access token.	atlsecurity.h
CAcl	This class is a wrapper for an ACL (access-control list) structure.	atlsecurity.h
CAdapt	This template is used to wrap classes that redefine the address-of operator to return something other than the address of the object.	atlcomcli.h

CLASS / STRUCT	DESCRIPTION	HEADER FILE
CAtlArray	This class implements an array object.	atlcoll.h
CAtlAutoThreadModule	This class implements a thread-pooled, apartment-model COM server.	atlbase.h
CAtlAutoThreadModuleT	This class provides methods for implementing a thread-pooled, apartment-model COM server.	atlbase.h
CAtlBaseModule	This class is instantiated in every ATL project.	atlcore.h
CAtlComModule	This class implements a COM server module.	atlbase.h
CAtlDebugInterfacesModule	This class provides support for debugging interfaces.	atlbase.h
CAtlDllModuleT	This class represents the module for a DLL.	atlbase.h
CAtlException	This class defines an ATL exception.	atlexcept.h
CAtlExeModuleT	This class represents the module for an application.	atlbase.h
CAtlFile	This class provides a thin wrapper around the Windows file-handling API.	atlfile.h
CAtlFileMapping	This class represents a memory-mapped file, adding a cast operator to the methods of CAtlFileMappingBase .	atlfile.h
CAtlFileMappingBase	This class represents a memory-mapped file.	atlfile.h
CAtlList	This class provides methods for creating and managing a list object.	atlcoll.h
CAtlMap	This class provides methods for creating and managing a map object.	atlcoll.h
CAtlModule	This class provides methods used by several ATL module classes.	atlbase.h
CAtlModuleT	This class implements an ATL module.	atlbase.h
CAtlPreviewCtrlImpl	This class is an ATL implementation of a window that is placed on a host window provided by the Shell for Rich Preview.	atlpreviewctrlimpl.h
CAtlServiceModuleT	This class implements a service.	atlbase.h

CLASS / STRUCT	DESCRIPTION	HEADER FILE
CAtlTemporaryFile	This class provides methods for the creation and use of a temporary file.	atlfile.h
CAtlTransactionManager	This class provides a wrapper to Kernel Transaction Manager (KTM) functions.	atltransactionmanager.h
CAtlWinModule	This class provides support for ATL windowing components.	atlbase.h
CAutoPtr	This class represents a smart pointer object.	atlbase.h
CAutoPtrArray	This class provides methods useful when constructing an array of smart pointers.	atlbase.h
CAutoPtrElementTraits	This class provides methods, static functions, and typedefs useful when creating collections of smart pointers.	atlcoll.h
CAutoPtrList	This class provides methods useful when constructing a list of smart pointers.	atlcoll.h
CAutoVectorPtr	This class represents a smart pointer object using vector new and delete operators.	atlbase.h
CAutoVectorPtrElementTraits	This class provides methods, static functions, and typedefs useful when creating collections of smart pointers using vector new and delete operators.	atlcoll.h
CAxDialogImpl	This class implements a dialog box (modal or modeless) that hosts ActiveX controls.	atlwin.h
CAxWindow	This class provides methods for manipulating a window hosting an ActiveX control.	atlwin.h
CAxWindow2T	This class provides methods for manipulating a window that hosts an ActiveX control and also has support for hosting licensed ActiveX controls.	atlwin.h
CBindStatusCallback	This class implements the <code>IBindStatusCallback</code> interface.	atlctl.h
CComAggObject	This class implements IUnknown for an aggregated object.	atlcom.h
CComAllocator	This class provides methods for managing memory using COM memory routines.	atlbase.h

CLASS / STRUCT	DESCRIPTION	HEADER FILE
CComApartment	This class provides support for managing an apartment in a thread-pooled EXE module.	atlbase.h
CComAutoCriticalSection	This class provides methods for obtaining and releasing ownership of a critical section object.	atlcore.h
CComAutoThreadModule	As of ATL 7.0, <code>CComAutoThreadModule</code> is obsolete: see ATL Modules for more details.	atlbase.h
CComBSTR	This class is a wrapper for BSTRs.	atlbase.h
CComCachedTearOffObject	This class implements IUnknown for a tear-off interface.	atlcom.h
CComClassFactory	This class implements the IClassFactory interface.	atlcom.h
CComClassFactory2	This class implements the IClassFactory2 interface.	atlcom.h
CComClassFactoryAutoThread	This class implements the IClassFactory interface and allows objects to be created in multiple apartments.	atlcom.h
CComClassFactorySingleton	This class derives from CComClassFactory and uses CComObjectGlobal to construct a single object.	atlcom.h
CComCoClass	This class provides methods for creating instances of a class and obtaining its properties.	atlcom.h
CComCompositeControl	This class provides the methods required to implement a composite control.	atlctl.h
CComContainedObject	This class implements IUnknown by delegating to the owner object's <code>IUnknown</code> .	atlcom.h
CComControl	This class provides methods for creating and managing ATL controls.	atlctl.h
CComControlBase	This class provides methods for creating and managing ATL controls.	atlctl.h
CComCriticalSection	This class provides methods for obtaining and releasing ownership of a critical section object.	atlcore.h

CLASS / STRUCT	DESCRIPTION	HEADER FILE
CComCritSecLock	This class provides methods for locking and unlocking a critical section object.	atlbase.h
CComCurrency	This class has methods and operators for creating and managing a CURRENCY object.	atlcur.h
CComDynamicUnkArray	This class stores an array of <code>IUnknown</code> pointers.	atlcom.h
CComEnum	This class defines a COM enumerator object based on an array.	atlcom.h
CComEnumImpl	This class provides the implementation for a COM enumerator interface where the items being enumerated are stored in an array.	atlcom.h
CComEnumOnSTL	This class defines a COM enumerator object based on a C++ Standard Library collection.	atlcom.h
CComFakeCriticalSection	This class provides the same methods as CComCriticalSection but does not provide a critical section.	atlcore.h
CComGITPtr	This class provides methods for dealing with interface pointers and the global interface table (GIT).	atlbase.h
CComHeap	This class implements IAtlMemMgr using the COM memory allocation functions.	ATLComMem.h
CComHeapPtr	A smart pointer class for managing heap pointers.	atlbase.h
CComModule	As of ATL 7.0, <code>CComModule</code> is obsolete: see ATL Modules for more details.	atlbase.h
CComMultiThreadModel	This class provides thread-safe methods for incrementing and decrementing the value of a variable.	atlbase.h
CComMultiThreadModelNoCS	This class provides thread-safe methods for incrementing and decrementing the value of a variable, without critical section locking or unlocking functionality.	atlbase.h
CComObject	This class implements <code>IUnknown</code> for a nonaggregated object.	atlcom.h

CLASS / STRUCT	DESCRIPTION	HEADER FILE
CComObjectGlobal	This class manages a reference count on the module containing your <code>Base</code> object.	atlcom.h
CComObjectNoLock	This class implements <code>IUnknown</code> for a nonaggregated object, but does not increment the module lock count in the constructor.	atlcom.h
CComObjectRoot	This typedef of CComObjectRootEx is templated on the default threading model of the server.	atlcom.h
CComObjectRootEx	This class provides methods to handle object reference count management for both nonaggregated and aggregated objects.	atlcom.h
CComObjectStack	This class creates a temporary COM object and provides it with a skeletal implementation of <code>IUnknown</code> .	atlcom.h
CComPolyObject	This class implements <code>IUnknown</code> for an aggregated or nonaggregated object.	atlcom.h
CComPtr	A smart pointer class for managing COM interface pointers.	atlcomcli.h
CComPtrBase	This class provides a basis for smart pointer classes using COM-based memory routines.	atlcomcli.h
CComQIPtr	A smart pointer class for managing COM interface pointers.	atlcomcli.h
CComQIPtrElementTraits	This class provides methods, static functions, and typedefs useful when creating collections of COM interface pointers.	atlcoll.h
CComSafeArray	This class is a wrapper for the <code>SAFEARRAY Data Type</code> structure.	atlsafe.h
CComSafeArrayBound	This class is a wrapper for a <code>SAFEARRAYBOUND</code> structure.	atlsafe.h
CComSimpleThreadAllocator	This class manages thread selection for the class CComAutoThreadModule .	atlbase.h
CComSingleThreadModel	This class provides methods for incrementing and decrementing the value of a variable.	atlbase.h

CLASS / STRUCT	DESCRIPTION	HEADER FILE
CComTearOffObject	This class implements a tear-off interface.	atlcom.h
CComUnkArray	This class stores <code>IUnknown</code> pointers and is designed to be used as a parameter to the IConnectionPointImpl template class.	atlcom.h
CComVariant	This class wraps the VARIANT type, providing a member indicating the type of data stored.	atlcomcli.h
CContainedWindowT	This class implements a window contained within another object.	atlwin.h
CCRTAllocator	This class provides methods for managing memory using CRT memory routines.	atlcore.h
CCRTHeap	This class implements IAtlMemMgr using the CRT heap functions.	atlmem.h
CDacl	This class is a wrapper for a DACL (discretionary access-control list) structure.	atlsecurity.h
CDebugReportHook Class	Use this class to send debug reports to a named pipe.	atlutil.h
CDefaultCharTraits	This class provides two static functions for converting characters between uppercase and lowercase.	atlcoll.h
CDefaultCompareTraits	This class provides default element comparison functions.	atlcoll.h
CDefaultElementTraits	This class provides default methods and functions for a collection class.	atlcoll.h
CDefaultHashTraits	This class provides a static function for calculating hash values.	atlcoll.h
CDialogImpl	This class provides methods for creating a modal or modeless dialog box.	atlwin.h
CDynamicChain	This class provides methods supporting the dynamic chaining of message maps.	atlwin.h
CElementTraits	This class is used by collection classes to provide methods and functions for moving, copying, comparison, and hashing operations.	atlcoll.h

CLASS / STRUCT	DESCRIPTION	HEADER FILE
CElementTraitsBase	This class provides default copy and move methods for a collection class.	atlcoll.h
CFirePropNotifyEvent	This class provides methods for notifying the container's sink regarding control property changes.	atlctl.h
CGlobalHeap	This class implements IAtlMemMgr using the Win32 global heap functions.	atlmem.h
CHandle	This class provides methods for creating and using a handle object.	atlbase.h
CHeapPtr	A smart pointer class for managing heap pointers.	atlc core.h
CHeapPtrBase	This class forms the basis for several smart heap pointer classes.	atlc core.h
CHeapPtrElementTraits Class	This class provides methods, static functions, and typedefs useful when creating collections of heap pointers.	atlcoll.h
CHeapPtrList	This class provides methods useful when constructing a list of heap pointers.	atlcoll.h
CImage	Provides enhanced bitmap support, including the ability to load and save images in JPEG, GIF, BMP, and Portable Network Graphics (PNG) formats.	atlimage.h
CInterfaceArray	This class provides methods useful when constructing an array of COM interface pointers.	atlcoll.h
CInterfaceList	This class provides methods useful when constructing a list of COM interface pointers.	atlcoll.h
CLocalHeap	This class implements IAtlMemMgr using the Win32 local heap functions.	atlmem.h
CMessageMap	This class allows an object's message maps to be accessed by another object.	atlwin.h
CNonStatelessWorker Class	Receives requests from a thread pool and passes them on to a worker object that is created and destroyed on each request.	atlutil.h

CLASS / STRUCT	DESCRIPTION	HEADER FILE
CNoWorkerThread Class	Use this class as the argument for the <code>MonitorClass</code> template parameter cache classes if you want to disable dynamic cache maintenance.	atlutil.h
CPathT Class	This class represents a path.	atlpath.h
CPrimitiveElementTraits	This class provides default methods and functions for a collection class composed of primitive data types.	atllcoll.h
CPrivateObjectSecurityDesc	This class represents a private object security descriptor object.	atlsecurity.h
CRBMap	This class represents a mapping structure, using a Red-Black binary tree.	atllcoll.h
CRBMultiMap	This class represents a mapping structure that allows each key to be associated with more than one value, using a Red-Black binary tree.	atllcoll.h
CRBTree	This class provides methods for creating and utilizing a Red-Black tree.	atllcoll.h
CRegKey	This class provides methods for manipulating entries in the system registry.	atlbase.h
CRTThreadTraits	This class provides the creation function for a CRT thread. Use this class if the thread will use CRT functions.	atlbase.h
CSacl	This class is a wrapper for a SACL (system access-control list) structure.	atlsecurity.h
CSecurityAttributes	This class is a thin wrapper for the <code>SECURITY_ATTRIBUTES</code> structure.	atlsecurity.h
CSecurityDesc	This class is a wrapper for the <code>SECURITY_DESCRIPTOR</code> structure.	atlsecurity.h
CSid	This class is a wrapper for a <code>SID</code> (security identifier) structure.	atlsecurity.h
CSimpleArray	This class provides methods for managing a simple array.	atlsimpcoll.h
CSimpleArrayEqualHelper	This class is a helper for the CSimpleArray class.	atlsimpcoll.h

CLASS / STRUCT	DESCRIPTION	HEADER FILE
CSimpleArrayEqualHelperFalse	This class is a helper for the CSimpleArray class.	atlsimpcoll.h
CSimpleDialog	This class implements a basic modal dialog box.	atlwin.h
CSimpleMap	This class provides support for a simple mapping array.	atlsimpcoll.h
CSimpleMapEqualHelper	This class is a helper for the CSimpleMap class.	atlsimpcoll.h
CSimpleMapEqualHelperFalse	This class is a helper for the CSimpleMap class.	atlsimpcoll.h
CSnapInItemImpl	This class provides methods for implementing a snap-in node object.	atlsnap.h
CSnapInPropertyPageImpl	This class provides methods for implementing a snap-in property page object.	atlsnap.h
CStockPropImpl	This class provides methods for supporting stock property values.	atlctl.h
CStringElementTraits	This class provides static functions used by collection classes storing <code>CString</code> objects.	cstringt.h
CStringElementTraitsI	This class provides static functions related to strings stored in collection class objects. It is similar to CStringElementTraits , but performs case-insensitive comparisons.	atlcoll.h
CStringRefElementTraits	This class provides static functions related to strings stored in collection class objects. The string objects are dealt with as references.	atlcoll.h
CThreadPool Class	This class provides a pool of worker threads that process a queue of work items.	atlutil.h
CTokenGroups	This class is a wrapper for the <code>TOKEN_GROUPS</code> structure.	atlsecurity.h
CTokenPrivileges	This class is a wrapper for the <code>TOKEN_PRIVILEGES</code> structure.	atlsecurity.h
CUrl Class	This class represents a URL. It allows you to manipulate each element of the URL independently of the others whether parsing an existing URL string or building a string from scratch.	atlutil.h

CLASS / STRUCT	DESCRIPTION	HEADER FILE
CW2AEX	This class is used by the string conversion macros CT2AEX, CW2TEX, CW2CTEX, and CT2CAEX, and the typedef CW2A.	atlconv.h
CW2CWEX	This class is used by the string conversion macros CW2CTEX and CT2CWEX, and the typedef CW2CW.	atlconv.h
CW2WEX	This class is used by the string conversion macros CW2TEX and CT2WEX, and the typedef CW2W.	atlconv.h
CWin32Heap	This class implements IAtlMemMgr using the Win32 heap allocation functions.	atlmem.h
CWindow	This class provides methods for manipulating a window.	atlwin.h
CWindowImpl	This class provides methods for creating or subclassing a window.	atlwin.h
CWinTraits	This class provides a method for standardizing the styles used when creating a window object.	atlwin.h
CWinTraitsOR	This class provides a method for standardizing the styles used when creating a window object.	atlwin.h
CWndClassInfo	This class provides methods for registering information for a window class.	atlwin.h
CWorkerThread Class	This class creates a worker thread or uses an existing one, waits on one or more kernel object handles, and executes a specified client function when one of the handles is signaled.	atlutil.h
IAtlAutoThreadModule	This class represents an interface to a <code>CreateInstance</code> method.	atlbase.h
IAtlMemMgr	This class represents the interface to a memory manager.	atlmem.h
IAxWinAmbientDispatch	This interface provides methods for specifying characteristics of the hosted control or container.	atlbase.h, ATLIFace.h
IAxWinAmbientDispatchEx	This interface implements supplemental ambient properties for a hosted control.	atlbase.h, ATLIFace.h

CLASS / STRUCT	DESCRIPTION	HEADER FILE
IAxWinHostWindow	This interface provides methods for manipulating a control and its host object.	atlbase.h, ATLIFace.h
IAxWinHostWindowLic	This interface provides methods for manipulating a licensed control and its host object.	atlbase.h, ATLIFace.h
ICollectionOnSTLImpl	This class provides methods used by a collection class.	atlcom.h
IConnectionPointContainerImpl	This class implements a connection point container to manage a collection of IConnectionPointImpl objects.	atlcom.h
IConnectionPointImpl	This class implements a connection point.	atlcom.h
IDataObjectImpl	This class provides methods for supporting Uniform Data Transfer and managing connections.	atlctl.h
IDispatchImpl	This class provides a default implementation for the <code>IDispatch</code> portion of a dual interface.	atlcom.h
IDispEventImpl	This class provides implementations of the <code>IDispatch</code> methods.	atlcom.h
IDispEventSimpleImpl	This class provides implementations of the <code>IDispatch</code> methods, without getting type information from a type library.	atlcom.h
IDocHostUIHandlerDispatch	An interface to the Microsoft HTML parsing and rendering engine.	atlbase.h, ATLIFace.h
IEnumOnSTLImpl	This class defines an enumerator interface based on a C++ Standard Library collection.	atlcom.h
IObjectSafetyImpl	This class provides a default implementation of the <code>IObjectSafety</code> interface to allow a client to retrieve and set an object's safety levels.	atlctl.h
IObjectWithSiteImpl	This class provides methods allowing an object to communicate with its site.	atlcom.h
IOleControlImpl	This class provides a default implementation of the <code>IOleControl</code> interface and implements <code>IUnknown</code> .	atlctl.h

CLASS / STRUCT	DESCRIPTION	HEADER FILE
IOleInPlaceActiveObjectImpl	This class provides methods for assisting communication between an in-place control and its container.	atlctl.h
IOleInPlaceObjectWindowlessImpl	This class implements IUnknown and provides methods that enable a windowless control to receive window messages and to participate in drag-and-drop operations.	atlctl.h
IOleObjectImpl	This class implements IUnknown and is the principal interface through which a container communicates with a control.	atlctl.h
IPropertyBrowsingImpl	This class implements IUnknown and allows a client to access the information in an object's property pages.	atlctl.h
IPersistPropertyBagImpl	This class implements IUnknown and allows an object to save its properties to a client-supplied property bag.	atlcom.h
IPersistStorageImpl	This class implements the IPersistStorage interface.	atlcom.h
IPersistStreamInitImpl	This class implements IUnknown and provides a default implementation of the IPersistStreamInit interface.	atlcom.h
IPointerInactivImpl	This class implements IUnknown and the IPointerInactive interface methods.	atlctl.h
IPropertyNotifySinkCP	This class exposes the IPropertyNotifySink interface as an outgoing interface on a connectable object.	atlctl.h
IPropertyPage2Impl	This class implements IUnknown and inherits the default implementation of IPropertyPageImpl .	atlctl.h
IPropertyPageImpl	This class implements IUnknown and provides a default implementation of the IPropertyPage interface.	atlctl.h
IProvideClassInfo2Impl	This class provides a default implementation of the IProvideClassInfo and IProvideClassInfo2 methods.	atlcom.h
IQuickActivateImpl	This class combines containers' control initialization into a single call.	atlctl.h

CLASS / STRUCT	DESCRIPTION	HEADER FILE
IRunnableObjectImpl	This class implements <code>IUnknown</code> and provides a default implementation of the IRunnableObject interface.	atlctl.h
IServiceProviderImpl	This class provides a default implementation of the <code>IServiceProvider</code> interface.	atlcom.h
ISpecifyPropertyPagesImpl	This class implements <code>IUnknown</code> and provides a default implementation of the ISpecifyPropertyPages interface.	atlcom.h
ISupportErrorInfoImpl	This class provides a default implementation of the <code>ISupportErrorInfo</code> Interface interface and can be used when only a single interface generates errors on an object.	atlcom.h
IThreadPoolConfig Interface	This interface provides methods for configuring a thread pool.	atlutil.h
IViewObjectExImpl	This class implements <code>IUnknown</code> and provides default implementations of the IViewObject , IViewObject2 , and IViewObjectEx interfaces.	atlctl.h
IWorkerThreadClient Interface	<code>IWorkerThreadClient</code> is the interface implemented by clients of the CWorkerThread class.	atlutil.h
_U_MENUorID	This class provides wrappers for <code>CreateWindow</code> and <code>CreateWindowEx</code> .	atlwin.h
_U_RECT	This argument adapter class allows either <code>RECT</code> pointers or references to be passed to a function that is implemented in terms of pointers.	atlwin.h
_U_STRINGorID	This argument adapter class allows either resource names (LPCTSTRs) or resource IDs (UINTs) to be passed to a function without requiring the caller to convert the ID to a string using the MAKEINTRESOURCE macro.	atlwin.h
Win32ThreadTraits	This class provides the creation function for a Windows thread. Use this class if the thread will not use CRT functions.	atlbase.h

See also

[ATL COM Desktop Components](#)

[Functions](#)

[Global Variables](#)

[Typedefs](#)

[Class Overview](#)

_ATL_BASE_MODULE70 Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

Used by any project that uses ATL.

Syntax

```
struct _ATL_BASE_MODULE70 {
    UINT cbSize;
    HINSTANCE m_hInst;
    HINSTANCE m_hInstResource;
    bool m_bNT5orWin98;
    DWORD dwAtlBuildVer;
    GUID* pguidVer;
    CRITICAL_SECTION m_csResource;
    CSimpleArray<HINSTANCE> m_rgResourceInstance;
};
```

Members

`cbSize`

The size of the structure, used for versioning.

`m_hInst`

The `hInstance` for this module (either exe or dll).

`m_hInstResource`

Default instance resource handle.

`m_bNT5orWin98`

Operating system version information. Used internally by ATL.

`dwAtlBuildVer`

Stores the version of ATL. Currently 0x0700.

`pguidVer`

ATL's internal GUID.

`m_csResource`

Used to synchronize access to the `m_rgResourceInstance` array. Used internally by ATL.

`m_rgResourceInstance`

Array used to search for resources in all the resource instances of which ATL is aware. Used internally by ATL.

Remarks

`_ATL_BASE_MODULE` is defined as a typedef of `_ATL_BASE_MODULE70`.

Requirements

Header: atlc core.h

See also

[Classes and structs](#)

_ATL_COM_MODULE70 Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

Used by COM-related code in ATL.

Syntax

```
struct _ATL_COM_MODULE70 {
    UINT cbSize;
    HINSTANCE m_hInstTypeLib;
    _ATL_OBJMAP_ENTRY** m_ppAutoObjMapFirst;
    _ATL_OBJMAP_ENTRY** m_ppAutoObjMapLast;
    CRITICAL_SECTION m_csObjMap;
};
```

Members

`cbSize`

The size of the structure, used for versioning.

`m_hInstTypeLib`

The handle instance to the type library for this module.

`m_ppAutoObjMapFirst`

Address of the array element indicating the beginning of the object map entries for this module.

`m_ppAutoObjMapLast`

Address of the array element indicating the end of the object map entries for this module.

`m_csObjMap`

Critical section to serialize access to the object map entries. Used internally by ATL.

Remarks

[_ATL_COM_MODULE](#) is defined as a typedef of `_ATL_COM_MODULE70`.

Requirements

Header: atlbase.h

See also

[Classes and structs](#)

_ATL_FUNC_INFO Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

Contains type information used to describe a method or property on a dispinterface.

Syntax

```
struct _ATL_FUNC_INFO {  
    CALLCONV cc;  
    VARTYPE vtReturn;  
    SHORT nParams;  
    VARTYPE pVarTypes[_ATL_MAX_VARTYPES];  
};
```

Members

`cc`

The calling convention. When using this structure with the [IDispEventSimpleImpl](#) class, this member must be `CC_STDCALL`. `CC_CDECL` is the only option supported in Windows CE for the `CALLCONV` field of the `_ATL_FUNC_INFO` structure. Any other value is unsupported thus its behavior undefined.

`vtReturn`

The variant type of the function return value.

`nParams`

The number of function parameters.

`pVarTypes`

An array of variant types of the function parameters.

Remarks

Internally, ATL uses this structure to hold information obtained from a type library. You may need to manipulate this structure directly if you provide type information for an event handler used with the [IDispEventSimpleImpl](#) class and [SINK_ENTRY_INFO](#) macro.

Example

Given a dispinterface method defined in IDL:

```
HRESULT SomeFunction([in] long Number, [in] BSTR String);
```

you would define an `_ATL_FUNC_INFO` structure:

```
_ATL_FUNC_INFO info = {CC_STDCALL, VT_EMPTY, 2, {VT_I4, VT_BSTR} };
```

Requirements

Header: `atlcom.h`

See also

[Classes and structs](#)

[IDispEventSimpleImpl Class](#)

[SINK_ENTRY_INFO](#)

_ATL_MODULE70 Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

Contains data used by every ATL module.

Syntax

```
struct _ATL_MODULE70 {
    UINT cbSize;
    LONG m_nLockCnt;
    _ATL_TERMFUNC_ELEM* m_pTermFuncs;
    CComCriticalSection m_csStaticDataInitAndTypeInfo;
};
```

Members

`cbSize`

The size of the structure, used for versioning.

`m_nLockCnt`

Reference count to determine how long the module should stay alive.

`m_pTermFuncs`

Tracks functions that have been registered to be called when ATL shuts down.

`m_csStaticDataInitAndTypeInfo`

Used to coordinate access to internal data in multithreaded situations.

Remarks

`_ATL_MODULE` is defined as a typedef of `_ATL_MODULE70`.

Requirements

Header: atlbase.h

See also

[Classes and structs](#)

_ATL_WIN_MODULE70 Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

Used by windowing code in ATL.

Syntax

```
struct _ATL_WIN_MODULE70 {  
    UNIT cbSize;  
    CRITICAL_SECTION m_csWindowCreate;  
    _AtlCreateWndData* m_pCreateWndList;  
    CSimpleArray<ATOM> m_rgWindowClassAtoms;  
};
```

Members

`cbSize`

The size of the structure, used for versioning.

`m_csWindowCreate`

Used to serialize access to window registration code. Used internally by ATL.

`m_pCreateWndList`

Used to bind windows to their objects. Used internally by ATL.

`m_rgWindowClassAtoms`

Used to track window class registrations so that they can be properly unregistered at termination. Used internally by ATL.

Remarks

`_ATL_WIN_MODULE` is defined as a typedef of `_ATL_WIN_MODULE70`.

Requirements

Header: atlbase.h

See also

[Classes and structs](#)

_AtlCreateWndData Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

This structure contains class instance data in windowing code in ATL.

Syntax

```
struct _AtlCreateWndData{  
    void* m_pThis;  
    DWORD m_dwThreadID;  
    _AtlCreateWndData* m_pNext;  
};
```

Members

`m_pThis`

The **this** pointer used to get access to the class instance in window procedures.

`m_dwThreadID`

The thread ID of the current class instance.

`m_pNext`

Pointer to the next `_AtlCreateWndData` object.

Requirements

Header: atlbase.h

See also

[Classes and structs](#)

ATL_DRAWINFO Structure

3/4/2019 • 3 minutes to read • [Edit Online](#)

Contains information used for rendering to various targets, such as a printer, metafile, or ActiveX control.

Syntax

```
struct ATL_DRAWINFO {
    UINT cbSize;
    DWORD dwDrawAspect;
    LONG lindex;
    DVTARGETDEVICE* ptd;
    HDC hicTargetDev;
    HDC hdcDraw;
    LPCRECTL prcBounds;
    LPCRECTL prcWBounds;
    BOOL bOptimize;
    BOOL bZoomed;
    BOOL bRectInHimetric;
    SIZEL ZoomNum;
    SIZEL ZoomDen;
};
```

Members

cbSize

The size of the structure, in bytes.

dwDrawAspect

Specifies how the target is to be represented. Representations can include content, an icon, a thumbnail, or a printed document. For a list of possible values, see [DVASPECT](#) and [DVASPECT2](#).

lindex

Portion of the target that is of interest for the draw operation. Its interpretation varies depending on the value in the **dwDrawAspect** member.

ptd

Pointer to a [DVTARGETDEVICE](#) structure that enables drawing optimizations depending on the aspect specified. Note that newer objects and containers that support optimized drawing interfaces support this member as well. Older objects and containers that do not support optimized drawing interfaces always specify NULL for this member.

hicTargetDev

Information context for the target device pointed to by **ptd** from which the object can extract device metrics and test the device's capabilities. If **ptd** is NULL, the object should ignore the value in the **hicTargetDev** member.

hdcDraw

The device context on which to draw. For a windowless object, the **hdcDraw** member is in the [MM_TEXT](#) mapping mode with its logical coordinates matching the client coordinates of the containing window. In addition, the device context should be in the same state as the one normally passed by a [WM_PAINT](#) message.

prcBounds

Pointer to a [RECTL](#) structure specifying the rectangle on **hdcDraw** and in which the object should be drawn. This

member controls the positioning and stretching of the object. This member should be NULL to draw a windowless in-place active object. In every other situation, NULL is not a legal value and should result in an `E_INVALIDARG` error code. If the container passes a non-NULL value to a windowless object, the object should render the requested aspect into the specified device context and rectangle. A container can request this from a windowless object to render a second, non-active view of the object or to print the object.

`prcWBounds`

If `hdcDraw` is a metafile device context (see [GetDeviceCaps](#) in the Windows SDK), this is a pointer to a `RECTL` structure specifying the bounding rectangle in the underlying metafile. The rectangle structure contains the window extent and window origin. These values are useful for drawing metafiles. The rectangle indicated by `prcBounds` is nested inside this `prcWBounds` rectangle; they are in the same coordinate space.

`bOptimize`

Nonzero if the drawing of the control is to be optimized, otherwise 0. If the drawing is optimized, the state of the device context is automatically restored when you are finished rendering.

`bZoomed`

Nonzero if the target has a zoom factor, otherwise 0. The zoom factor is stored in `ZoomNum`.

`bRectInHimetric`

Nonzero if the dimensions of `prcBounds` are in HIMETRIC, otherwise 0.

`ZoomNum`

The width and height of the rectangle into which the object is rendered. The zoom factor along the x-axis (the proportion of the object's natural size to its current extent) of the target is the value of `ZoomNum.cx` divided by the value of `ZoomDen.cx`. The zoom factor along the y-axis is achieved in a similar fashion.

`ZoomDen`

The actual width and height of the target.

Remarks

Typical usage of this structure would be the retrieval of information during the rendering of the target object. For example, you could retrieve values from `ATL_DRAWINFO` inside your overload of [CComControlBase::OnDrawAdvanced](#).

This structure stores pertinent information used to render the appearance of an object for the target device. The information provided can be used in drawing to the screen, a printer, or even a metafile.

Requirements

Header: `atlctl.h`

See also

[Classes and structs](#)

[IViewObject::Draw](#)

[CComControlBase::OnDrawAdvanced](#)

_U_MENUorID Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides wrappers for `CreateWindow` and `CreateWindowEx`.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class _U_MENUorID
```

Members

Public Constructors

NAME	DESCRIPTION
_U_MENUorID::_U_MENUorID	The constructor.

Public Data Members

NAME	DESCRIPTION
_U_MENUorID::m_hMenu	A handle to a menu.

Remarks

This argument adapter class allows either IDs (UINTs) or menu handles (HMENUs) to be passed to a function without requiring an explicit cast on the part of the caller.

This class is designed for implementing wrappers to the Windows API, particularly the [CreateWindow](#) and [CreateWindowEx](#) functions, both of which accept an HMENU argument that may be a child window identifier (UINT) rather than a menu handle. For example, you can see this class in use as a parameter to [CWindowImpl::Create](#).

The class defines two constructor overloads: one accepts a UINT argument and the other accepts an HMENU argument. The UINT argument is just cast to an HMENU in the constructor and the result stored in the class's single data member, [m_hMenu](#). The argument to the HMENU constructor is stored directly without conversion.

Requirements

Header: `atlwin.h`

_U_MENUorID::m_hMenu

The class holds the value passed to either of its constructors as a public HMENU data member.

```
HMENU m_hMenu;
```

`_U_MENUorID::_U_MENUorID`

The `UINT` argument is just cast to an `HMENU` in the constructor and the result stored in the class's single data member, [m_hMenu](#).

```
_U_MENUorID(UINT nID);  
_U_MENUorID(HMENU hMenu);
```

Parameters

nID

A child window identifier.

hMenu

A menu handle.

Remarks

The argument to the `HMENU` constructor is stored directly without conversion.

See also

[Class Overview](#)

_U_RECT Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This argument adapter class allows either `RECT` pointers or references to be passed to a function that is implemented in terms of pointers.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class _U_RECT
```

Members

Public Constructors

NAME	DESCRIPTION
<code>_U_RECT::_U_RECT</code>	The constructor.

Public Data Members

NAME	DESCRIPTION
<code>_U_RECT::m_lpRect</code>	Pointer to a <code>RECT</code> .

Remarks

The class defines two constructor overloads: one accepts a **RECT&** argument and the other accepts an `LPRECT` argument. The first constructor stores the address of the reference argument in the class's single data member, `m_lpRect`. The argument to the pointer constructor is stored directly without conversion.

Requirements

Header: `atlwin.h`

`_U_RECT::m_lpRect`

The class holds the value passed to either of its constructors as a public `LPRECT` data member.

```
LPRECT m_lpRect;
```

`_U_RECT::_U_RECT`

The address of the reference argument is stored in the class's single data member, `m_lpRect`.

```
_U_RECT(RECT& rc);  
_U_RECT(LPRECT lpRect);
```

Parameters

rc

A `RECT` reference.

lpRect

A `RECT` pointer.

Remarks

The argument to the pointer constructor is stored directly without conversion.

See also

[Class Overview](#)

_U_STRINGorID Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This argument adapter class allows either resource names (LPCTSTRs) or resource IDs (UINTs) to be passed to a function without requiring the caller to convert the ID to a string using the MAKEINTRESOURCE macro.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class _U_STRINGorID
```

Members

Public Constructors

NAME	DESCRIPTION
_U_STRINGorID::_U_STRINGorID	The constructor.

Public Data Members

NAME	DESCRIPTION
_U_STRINGorID::m_lpstr	The resource identifier.

Remarks

This class is designed for implementing wrappers to the Windows resource management API such as the [FindResource](#), [LoadIcon](#), and [LoadMenu](#) functions, which accept an LPCTSTR argument that may be either the name of a resource or its ID.

The class defines two constructor overloads: one accepts a LPCTSTR argument and the other accepts a UINT argument. The UINT argument is converted to a resource type compatible with Windows resource-management functions using the MAKEINTRESOURCE macro and the result stored in the class's single data member, [m_lpstr](#). The argument to the LPCTSTR constructor is stored directly without conversion.

Requirements

Header: atlwin.h

_U_STRINGorID::m_lpstr

The class holds the value passed to either of its constructors as a public LPCTSTR data member.

```
LPCTSTR m_lpstr;
```

_U_STRINGorID::_U_STRINGorID

The UINT constructor converts its argument to a resource type compatible with Windows resource-management functions using the MAKEINTRESOURCE macro and the result is stored in the class's single data member, [m_lpstr](#).

```
_U_STRINGorID(UINT nID);  
_U_STRINGorID(LPCTSTR lpString);
```

Parameters

nID

A resource ID.

lpString

A resource name.

Remarks

The argument to the LPCTSTR constructor is stored directly without conversion.

See also

[Class Overview](#)

CA2AEX Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class is used by the string conversion macros CA2TEX and CT2AEX, and the typedef CA2A.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <int t_nBufferLength = 128>
class CA2AEX
```

Parameters

t_nBufferLength

The size of the buffer used in the translation process. The default length is 128 bytes.

Members

Public Constructors

NAME	DESCRIPTION
CA2AEX::CA2AEX	The constructor.
CA2AEX::~~CA2AEX	The destructor.

Public Operators

NAME	DESCRIPTION
CA2AEX::operator LPSTR	Conversion operator.

Public Data Members

NAME	DESCRIPTION
CA2AEX::m_psz	The data member that stores the source string.
CA2AEX::m_szBuffer	The static buffer, used to store the converted string.

Remarks

Unless extra functionality is required, use CA2TEX, CT2AEX, or CA2A in your own code.

This class contains a fixed-size static buffer which is used to store the result of the conversion. If the result is too large to fit into the static buffer, the class allocates memory using **malloc**, freeing the memory when the object goes out of scope. This ensures that, unlike text conversion macros available in previous versions of ATL, this class

is safe to use in loops and that it won't overflow the stack.

If the class tries to allocate memory on the heap and fails, it will call `AtlThrow` with an argument of `E_OUTOFMEMORY`.

By default, the ATL conversion classes and macros use the current thread's ANSI code page for the conversion.

The following macros are based on this class:

- `CA2TEX`
- `CT2AEX`

The following typedef is based on this class:

- `CA2A`

For a discussion of these text conversion macros, see [ATL and MFC String Conversion Macros](#).

Example

See [ATL and MFC String Conversion Macros](#) for an example of using these string conversion macros.

Requirements

Header: `atlconv.h`

CA2AEX::CA2AEX

The constructor.

```
CA2AEX(LPCSTR psz, UINT nCodePage) throw(...);  
CA2AEX(LPCSTR psz) throw(...);
```

Parameters

psz

The text string to be converted.

nCodePage

Unused in this class.

Remarks

Creates the buffer required for the translation.

CA2AEX::~CA2AEX

The destructor.

```
~CA2AEX() throw();
```

Remarks

Frees the allocated buffer.

CA2AEX::m_psz

The data member that stores the source string.

```
LPSTR m_psz;
```

CA2AEX::m_szBuffer

The static buffer, used to store the converted string.

```
char m_szBuffer[ t_nBufferLength];
```

CA2AEX::operator LPSTR

Conversion operator.

```
operator LPSTR() const throw();
```

Return Value

Returns the text string as type LPSTR.

See also

[CA2CAEX Class](#)

[CA2WEX Class](#)

[CW2AEX Class](#)

[CW2CWEX Class](#)

[CW2WEX Class](#)

[Class Overview](#)

CA2CAEX Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class is used by string conversion macros CA2CTEX and CT2CAEX, and the typedef CA2CA.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<int t_nBufferLength = 128>
class CA2CAEX
```

Parameters

t_nBufferLength

The size of the buffer used in the translation process. The default length is 128 bytes.

Members

Public Constructors

NAME	DESCRIPTION
CA2CAEX::CA2CAEX	The constructor.
CA2CAEX::~~CA2CAEX	The destructor.

Public Operators

NAME	DESCRIPTION
CA2CAEX::operator LPCSTR	Conversion operator.

Public Data Members

NAME	DESCRIPTION
CA2CAEX::m_psz	The data member that stores the source string.

Remarks

Unless extra functionality is required, use CA2CTEX, CT2CAEX, or CA2CA in your own code.

This class is safe to use in loops and won't overflow the stack. By default, the ATL conversion classes and macros will use the current thread's ANSI code page for the conversion.

The following macros are based on this class:

- CA2CTEX

- CT2CAEX

The following typedef is based on this class:

- CA2CA

For a discussion of these text conversion macros, see [ATL and MFC String Conversion Macros](#).

Example

See [ATL and MFC String Conversion Macros](#) for an example of using these string conversion macros.

Requirements

Header: atlconv.h

CA2CAEX::CA2CAEX

The constructor.

```
CA2CAEX(LPCSTR psz, UINT nCodePage) throw(...);  
CA2CAEX(LPCSTR psz) throw(...);
```

Parameters

psz

The text string to be converted.

nCodePage

Unused in this class.

Remarks

Creates the buffer required for the translation.

CA2CAEX::~CA2CAEX

The destructor.

```
~CA2CAEX() throw();
```

Remarks

Frees the allocated buffer.

CA2CAEX::m_psz

The data member that stores the source string.

```
LPCSTR m_psz;
```

CA2CAEX::operator LPCSTR

Conversion operator.

```
operator LPCSTR() const throw();
```

Return Value

Returns the text string as type LPCSTR.

See also

[CA2AEX Class](#)

[CA2WEX Class](#)

[CW2AEX Class](#)

[CW2CWEX Class](#)

[CW2WEX Class](#)

[Class Overview](#)

CA2WEX Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class is used by the string conversion macros CA2TEX, CA2CTEX, CT2WEX, and CT2CWEX, and the typedef CA2W.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <int t_nBufferLength = 128>
class CA2WEX
```

Parameters

t_nBufferLength

The size of the buffer used in the translation process. The default length is 128 bytes.

Members

Public Constructors

NAME	DESCRIPTION
CA2WEX::CA2WEX	The constructor.
CA2WEX::~~CA2WEX	The destructor.

Public Operators

NAME	DESCRIPTION
CA2WEX::operator LPWSTR	Conversion operator.

Public Data Members

NAME	DESCRIPTION
CA2WEX::m_psz	The data member that stores the source string.
CA2WEX::m_szBuffer	The static buffer, used to store the converted string.

Remarks

Unless extra functionality is required, use CA2TEX, CA2CTEX, CT2WEX, CT2CWEX, or CA2W in your code.

This class contains a fixed-size static buffer which is used to store the result of the conversion. If the result is too large to fit into the static buffer, the class allocates memory using **malloc**, freeing the memory when the object

goes out of scope. This ensures that, unlike text conversion macros available in previous versions of ATL, this class is safe to use in loops and that it won't overflow the stack.

If the class tries to allocate memory on the heap and fails, it will call `AtlThrow` with an argument of `E_OUTOFMEMORY`.

By default, the ATL conversion classes and macros use the current thread's ANSI code page for the conversion. If you want to override that behavior for a specific conversion, specify the code page as the second parameter to the constructor for the class.

The following macros are based on this class:

- `CA2TEX`
- `CA2CTEX`
- `CT2WEX`
- `CT2CWEX`

The following typedef is based on this class:

- `CA2W`

For a discussion of these text conversion macros, see [ATL and MFC String Conversion Macros](#).

Example

See [ATL and MFC String Conversion Macros](#) for an example of using these string conversion macros.

Requirements

Header: `atlconv.h`

CA2WEX::CA2WEX

The constructor.

```
CA2WEX(LPCSTR psz, UINT nCodePage) throw(...);
CA2WEX(LPCSTR psz) throw(...);
```

Parameters

psz

The text string to be converted.

nCodePage

The code page used to perform the conversion. See the code page parameter discussion for the Windows SDK function [MultiByteToWideChar](#) for more details.

Remarks

Allocates the buffer used in the translation process.

CA2WEX::~CA2WEX

The destructor.

```
~CA2WEX() throw();
```

Remarks

Frees the allocated buffer.

CA2WEX::m_psz

The data member that stores the source string.

```
LPWSTR m_psz;
```

CA2WEX::m_szBuffer

The static buffer, used to store the converted string.

```
wchar_t m_szBuffer[t_nBufferLength];
```

CA2WEX::operator LPWSTR

Conversion operator.

```
operator LPWSTR() const throw();
```

Return Value

Returns the text string as type LPWSTR.

See also

[CA2AEX Class](#)

[CA2CAEX Class](#)

[CW2AEX Class](#)

[CW2CWEX Class](#)

[CW2WEX Class](#)

[Class Overview](#)

CAccessToken Class

4/1/2019 • 24 minutes to read • [Edit Online](#)

This class is a wrapper for an access token.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CAccessToken
```

Members

Public Constructors

NAME	DESCRIPTION
CAccessToken::~CAccessToken	The destructor.

Public Methods

NAME	DESCRIPTION
CAccessToken::Attach	Call this method to take ownership of the given access token handle.
CAccessToken::CheckTokenMembership	Call this method to determine if a specified SID is enabled in the <code>CAccessToken</code> object.
CAccessToken::CreateImpersonationToken	Call this method to create a new impersonation access token.
CAccessToken::CreatePrimaryToken	Call this method to create a new primary token.
CAccessToken::CreateProcessAsUser	Call this method to create a new process running in the security context of the user represented by the <code>CAccessToken</code> object.
CAccessToken::CreateRestrictedToken	Call this method to create a new, restricted <code>CAccessToken</code> object.
CAccessToken::Detach	Call this method to revoke ownership of the access token.
CAccessToken::DisablePrivilege	Call this method to disable a privilege in the <code>CAccessToken</code> object.

NAME	DESCRIPTION
CAccessToken::DisablePrivileges	Call this method to disable one or more privileges in the <code>CAccessToken</code> object.
CAccessToken::EnablePrivilege	Call this method to enable a privilege in the <code>CAccessToken</code> object.
CAccessToken::EnablePrivileges	Call this method to enable one or more privileges in the <code>CAccessToken</code> object.
CAccessToken::GetDefaultDacl	Call this method to return the <code>CAccessToken</code> object's default DACL.
CAccessToken::GetEffectiveToken	Call this method to get the <code>CAccessToken</code> object equal to the access token in effect for the current thread.
CAccessToken::GetGroups	Call this method to return the <code>CAccessToken</code> object's token groups.
CAccessToken::GetHandle	Call this method to retrieve a handle to the access token.
CAccessToken::GetImpersonationLevel	Call this method to get the impersonation level from the access token.
CAccessToken::GetLogonSessionId	Call this method to get the Logon Session ID associated with the <code>CAccessToken</code> object.
CAccessToken::GetLogonSid	Call this method to get the Logon SID associated with the <code>CAccessToken</code> object.
CAccessToken::GetOwner	Call this method to get the owner associated with the <code>CAccessToken</code> object.
CAccessToken::GetPrimaryGroup	Call this method to get the primary group associated with the <code>CAccessToken</code> object.
CAccessToken::GetPrivileges	Call this method to get the privileges associated with the <code>CAccessToken</code> object.
CAccessToken::GetProcessToken	Call this method to initialize the <code>CAccessToken</code> with the access token from the given process.
CAccessToken::GetProfile	Call this method to get the handle pointing to the user profile associated with the <code>CAccessToken</code> object.
CAccessToken::GetSource	Call this method to get the source of the <code>CAccessToken</code> object.
CAccessToken::GetStatistics	Call this method to get information associated with the <code>CAccessToken</code> object.
CAccessToken::GetTerminalServicesSessionId	Call this method to get the Terminal Services Session ID associated with the <code>CAccessToken</code> object.

NAME	DESCRIPTION
CAccessToken::GetThreadToken	Call this method to initialize the <code>CAccessToken</code> with the token from the given thread.
CAccessToken::GetTokenId	Call this method to get the Token ID associated with the <code>CAccessToken</code> object.
CAccessToken::GetType	Call this method to get the token type of the <code>CAccessToken</code> object.
CAccessToken::GetUser	Call this method to identify the user associated with the <code>CAccessToken</code> object.
CAccessToken::HKeyCurrentUser	Call this method to get the handle pointing to the user profile associated with the <code>CAccessToken</code> object.
CAccessToken::Impersonate	Call this method to assign an impersonation <code>CAccessToken</code> to a thread.
CAccessToken::ImpersonateLoggedOnUser	Call this method to allow the calling thread to impersonate the security context of a logged-on user.
CAccessToken::IsTokenRestricted	Call this method to test if the <code>CAccessToken</code> object contains a list of restricted SIDs.
CAccessToken::LoadUserProfile	Call this method to load the user profile associated with the <code>CAccessToken</code> object.
CAccessToken::LogonUser	Call this method to create a logon session for the user associated with the given credentials.
CAccessToken::OpenCOMClientToken	Call this method from within a COM server handling a call from a client to initialize the <code>CAccessToken</code> with the access token from the COM client.
CAccessToken::OpenNamedPipeClientToken	Call this method from within a server taking requests over a named pipe to initialize the <code>CAccessToken</code> with the access token from the client.
CAccessToken::OpenRPCClientToken	Call this method from within a server handling a call from an RPC client to initialize the <code>CAccessToken</code> with the access token from the client.
CAccessToken::OpenThreadToken	Call this method to set the impersonation level and then initialize the <code>CAccessToken</code> with the token from the given thread.
CAccessToken::PrivilegeCheck	Call this method to determine whether a specified set of privileges are enabled in the <code>CAccessToken</code> object.
CAccessToken::Revert	Call this method to stop a thread that is using an impersonation token.

NAME	DESCRIPTION
CAccessToken::SetDefaultDacl	Call this method to set the default DACL of the <code>CAccessToken</code> object.
CAccessToken::SetOwner	Call this method to set the owner of the <code>CAccessToken</code> object.
CAccessToken::SetPrimaryGroup	Call this method to set the primary group of the <code>CAccessToken</code> object.

Remarks

An [access token](#) is an object that describes the security context of a process or thread and is allocated to each user logged onto a Windows system.

For an introduction to the access control model in Windows, see [Access Control](#) in the Windows SDK.

Requirements

Header: atlsecurity.h

CAccessToken::Attach

Call this method to take ownership of the given access token handle.

```
void Attach(HANDLE hToken) throw();
```

Parameters

hToken

A handle to the access token.

Remarks

In debug builds, an assertion error will occur if the `CAccessToken` object already has ownership of an access token.

CAccessToken::~CAccessToken

The destructor.

```
virtual ~CAccessToken() throw();
```

Remarks

Frees all allocated resources.

CAccessToken::CheckTokenMembership

Call this method to determine if a specified SID is enabled in the `CAccessToken` object.

```
bool CheckTokenMembership(  
    const CSid& rSid,  
    bool* pbIsMember) const throw(...);
```

Parameters

rSid

Reference to a [CSid Class](#) object.

pbIsMember

Pointer to a variable that receives the results of the check.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The `CheckTokenMembership` method checks for the presence of the SID in the user and group SIDs of the access token. If the SID is present and has the SE_GROUP_ENABLED attribute, *pbIsMember* is set to TRUE; otherwise, it is set to FALSE.

In debug builds, an assertion error will occur if *pbIsMember* is not a valid pointer.

NOTE

The `CAccessToken` object must be an impersonation token and not a primary token.

CAccessToken::CreateImpersonationToken

Call this method to create an impersonation access token.

```
bool CreateImpersonationToken(  
    CAccessToken* pImp,  
    SECURITY_IMPERSONATION_LEVEL sil = SecurityImpersonation) const throw(...);
```

Parameters

pImp

Pointer to the new `CAccessToken` object.

sil

Specifies a [SECURITY_IMPERSONATION_LEVEL](#) enumerated type that supplies the impersonation level of the new token.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

`CreateImpersonationToken` calls [DuplicateToken](#) to create a new impersonation token.

CAccessToken::CreatePrimaryToken

Call this method to create a new primary token.

```
bool CreatePrimaryToken(
    CAccessToken* pPri,
    DWORD dwDesiredAccess = MAXIMUM_ALLOWED,
    const CSecurityAttributes* pTokenAttributes = NULL) const throw(...);
```

Parameters

pPri

Pointer to the new `CAccessToken` object.

dwDesiredAccess

Specifies the requested access rights for the new token. The default, `MAXIMUM_ALLOWED`, requests all access rights that are valid for the caller. See [Access Rights and Access Masks](#) for more on access rights.

pTokenAttributes

Pointer to a `SECURITY_ATTRIBUTES` structure that specifies a security descriptor for the new token and determines whether child processes can inherit the token. If *pTokenAttributes* is `NULL`, the token gets a default security descriptor and the handle cannot be inherited.

Return Value

Returns `TRUE` on success, `FALSE` on failure.

Remarks

`CreatePrimaryToken` calls [DuplicateTokenEx](#) to create a new primary token.

CAccessToken::CreateProcessAsUser

Call this method to create a new process running in the security context of the user represented by the

`CAccessToken` object.

```
bool CreateProcessAsUser(
    LPCTSTR pApplicationName,
    LPTSTR pCommandLine,
    LPPROCESS_INFORMATION pProcessInformation,
    LPSTARTUPINFO pStartupInfo,
    DWORD dwCreationFlags = NORMAL_PRIORITY_CLASS,
    bool bLoadProfile = false,
    const CSecurityAttributes* pProcessAttributes = NULL,
    const CSecurityAttributes* pThreadAttributes = NULL,
    bool bInherit = false,
    LPCTSTR pCurrentDirectory = NULL) throw();
```

Parameters

pApplicationName

Pointer to a null-terminated string that specifies the module to execute. This parameter may not be `NULL`.

pCommandLine

Pointer to a null-terminated string that specifies the command line to execute.

pProcessInformation

Pointer to a `PROCESS_INFORMATION` structure that receives identification information about the new process.

pStartupInfo

Pointer to a `STARTUPINFO` structure that specifies how the main window for the new process should appear.

dwCreationFlags

Specifies additional flags that control the priority class and the creation of the process. See the Win32 function

[CreateProcessAsUser](#) for a list of flags.

bLoadProfile

If TRUE, the user's profile is loaded with [LoadUserProfile](#).

pProcessAttributes

Pointer to a [SECURITY_ATTRIBUTES](#) structure that specifies a security descriptor for the new process and determines whether child processes can inherit the returned handle. If *pProcessAttributes* is NULL, the process gets a default security descriptor and the handle cannot be inherited.

pThreadAttributes

Pointer to a [SECURITY_ATTRIBUTES](#) structure that specifies a security descriptor for the new thread and determines whether child processes can inherit the returned handle. If *pThreadAttributes* is NULL, the thread gets a default security descriptor and the handle cannot be inherited.

bInherit

Indicates whether the new process inherits handles from the calling process. If TRUE, each inheritable open handle in the calling process is inherited by the new process. Inherited handles have the same value and access privileges as the original handles.

pCurrentDirectory

Pointer to a null-terminated string that specifies the current drive and directory for the new process. The string must be a full path that includes a drive letter. If this parameter is NULL, the new process will have the same current drive and directory as the calling process.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

`CreateProcessAsUser` uses the `CreateProcessAsUser` Win32 function to create a new process that runs in the security context of the user represented by the `CAccessToken` object. See the description of the [CreateProcessAsUser](#) function for a full discussion of the parameters required.

For this method to succeed, the `CAccessToken` object must hold `AssignPrimaryToken` (unless it is a restricted token) and `IncreaseQuota` privileges.

CAccessToken::CreateRestrictedToken

Call this method to create a new, restricted `CAccessToken` object.

```
bool CreateRestrictedToken(  
    CAccessToken* pRestrictedToken,  
    const CTokenGroups& SidsToDisable,  
    const CTokenGroups& SidsToRestrict,  
    const CTokenPrivileges& PrivilegesToDelete = CTokenPrivileges()) const throw(...);
```

Parameters

pRestrictedToken

The new, restricted `CAccessToken` object.

SidsToDisable

A `CTokenGroups` object that specifies the deny-only SIDs.

SidsToRestrict

A `CTokenGroups` object that specifies the restricting SIDs.

PrivilegesToDelete

A `CTokenPrivileges` object that specifies the privileges to delete in the restricted token. The default creates an empty object.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

`CreateRestrictedToken` uses the `CreateRestrictedToken` Win32 function to create a new `CAccessToken` object, with restrictions.

IMPORTANT

When using `CreateRestrictedToken`, ensure the following: the existing token is valid (and not entered by the user) and *SidsToDisable* and *PrivilegesToDelete* are both valid (and not entered by the user). If the method returns FALSE, deny functionality.

CAccessToken::Detach

Call this method to revoke ownership of the access token.

```
HANDLE Detach() throw();
```

Return Value

Returns the handle to the `CAccessToken` which has been detached.

Remarks

This method revokes the `CAccessToken`'s ownership of the access token.

CAccessToken::DisablePrivilege

Call this method to disable a privilege in the `CAccessToken` object.

```
bool DisablePrivilege(  
    LPCTSTR pszPrivilege,  
    CTokenPrivileges* pPreviousState = NULL) throw(...);
```

Parameters

pszPrivilege

Pointer to a string containing the privilege to disable in the `CAccessToken` object.

pPreviousState

Pointer to a `CTokenPrivileges` object which will contain the previous state of the privileges.

Return Value

Returns TRUE on success, FALSE on failure.

CAccessToken::DisablePrivileges

Call this method to disable one or more privileges in the `CAccessToken` object.

```
bool DisablePrivileges(  
    const CAtlArray<LPCTSTR>& rPrivileges,  
    CTokenPrivileges* pPreviousState = NULL) throw(...);
```

Parameters

rPrivileges

Pointer to an array of strings containing the privileges to disable in the `CAccessToken` object.

pPreviousState

Pointer to a `CTokenPrivileges` object which will contain the previous state of the privileges.

Return Value

Returns TRUE on success, FALSE on failure.

CAccessToken::EnablePrivilege

Call this method to enable a privilege in the `CAccessToken` object.

```
bool EnablePrivilege(  
    LPCTSTR pszPrivilege,  
    CTokenPrivileges* pPreviousState = NULL) throw(...);
```

Parameters

pszPrivilege

Pointer to a string containing the privilege to enable in the `CAccessToken` object.

pPreviousState

Pointer to a `CTokenPrivileges` object which will contain the previous state of the privileges.

Return Value

Returns TRUE on success, FALSE on failure.

CAccessToken::EnablePrivileges

Call this method to enable one or more privileges in the `CAccessToken` object.

```
bool EnablePrivileges(  
    const CAtlArray<LPCTSTR>& rPrivileges,  
    CTokenPrivileges* pPreviousState = NULL) throw(...);
```

Parameters

rPrivileges

Pointer to an array of strings containing the privileges to enable in the `CAccessToken` object.

pPreviousState

Pointer to a `CTokenPrivileges` object which will contain the previous state of the privileges.

Return Value

Returns TRUE on success, FALSE on failure.

CAccessToken::GetDefaultDacl

Call this method to return the `CAccessToken` object's default DACL.

```
bool GetDefaultDacl(CDacl* pDacl) const throw(...);
```

Parameters

pDacl

Pointer to the [CDacl Class](#) object which will receive the `CAccessToken` object's default DACL.

Return Value

Returns TRUE if the default DACL has been recovered, FALSE otherwise.

CAccessToken::GetEffectiveToken

Call this method to get the `CAccessToken` object equal to the access token in effect for the current thread.

```
bool GetEffectiveToken(DWORD dwDesiredAccess) throw();
```

Parameters

dwDesiredAccess

Specifies an access mask that specifies the requested types of access to the access token. These requested access types are compared with the token's DACL to determine which accesses are granted or denied.

Return Value

Returns TRUE on success, FALSE on failure.

CAccessToken::GetGroups

Call this method to return the `CAccessToken` object's token groups.

```
bool GetGroups(CTokenGroups* pGroups) const throw(...);
```

Parameters

pGroups

Pointer to the [CTokenGroups Class](#) object which will receive the group information.

Return Value

Returns TRUE on success, FALSE on failure.

CAccessToken::GetHandle

Call this method to retrieve a handle to the access token.

```
HANDLE GetHandle() const throw();
```

Return Value

Returns a handle to the `CAccessToken` object's access token.

CAccessToken::GetImpersonationLevel

Call this method to get the impersonation level from the access token.

```
bool GetImpersonationLevel(
    SECURITY_IMPERSONATION_LEVEL* pImpersonationLevel) const throw(...);
```

Parameters

pImpersonationLevel

Pointer to a [SECURITY_IMPERSONATION_LEVEL](#) enumeration type which will receive the impersonation level information.

Return Value

Returns TRUE on success, FALSE on failure.

CAccessToken::GetLogonSessionId

Call this method to get the Logon Session ID associated with the `CAccessToken` object.

```
bool GetLogonSessionId(LUID* pluid) const throw(...);
```

Parameters

pluid

Pointer to a [LUID](#) which will receive the Logon Session ID.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

In debug builds, an assertion error will occur if *pluid* is an invalid value.

CAccessToken::GetLogonSid

Call this method to get the Logon SID associated with the `CAccessToken` object.

```
bool GetLogonSid(CSid* pSid) const throw(...);
```

Parameters

pSid

Pointer to a [CSid Class](#) object.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

In debug builds, an assertion error will occur if *pSid* is an invalid value.

CAccessToken::GetOwner

Call this method to get the owner associated with the `CAccessToken` object.

```
bool GetOwner(CSid* pSid) const throw(...);
```

Parameters

pSid

Pointer to a [CSid Class](#) object.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The owner is set by default on any objects created while this access token is in effect.

CAccessToken::GetPrimaryGroup

Call this method to get the primary group associated with the `CAccessToken` object.

```
bool GetPrimaryGroup(CSid* pSid) const throw(...);
```

Parameters

pSid

Pointer to a [CSid Class](#) object.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The group is set by default on any objects created while this access token is in effect.

CAccessToken::GetPrivileges

Call this method to get the privileges associated with the `CAccessToken` object.

```
bool GetPrivileges(CTokenPrivileges* pPrivileges) const throw(...);
```

Parameters

pPrivileges

Pointer to a [CTokenPrivileges Class](#) object which will receive the privileges.

Return Value

Returns TRUE on success, FALSE on failure.

CAccessToken::GetProcessToken

Call this method to initialize the `CAccessToken` with the access token from the given process.

```
bool GetProcessToken(DWORD dwDesiredAccess, HANDLE hProcess = NULL) throw();
```

Parameters

dwDesiredAccess

Specifies an access mask that specifies the requested types of access to the access token. These requested access types are compared with the token's DACL to determine which accesses are granted or denied.

hProcess

Handle to the process whose access token is opened. If the default value of NULL is used, the current process is used.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Calls the [OpenProcessToken](#) Win32 function.

CAccessToken::GetProfile

Call this method to get the handle pointing to the user profile associated with the `CAccessToken` object.

```
HANDLE GetProfile() const throw();
```

Return Value

Returns a handle pointing to the user profile, or NULL if no profile exists.

CAccessToken::GetSource

Call this method to get the source of the `CAccessToken` object.

```
bool GetSource(TOKEN_SOURCE* pSource) const throw(...);
```

Parameters

pSource

Pointer to a [TOKEN_SOURCE](#) structure.

Return Value

Returns TRUE on success, FALSE on failure.

CAccessToken::GetStatistics

Call this method to get information associated with the `CAccessToken` object.

```
bool GetStatistics(TOKEN_STATISTICS* pStatistics) const throw(...);
```

Parameters

pStatistics

Pointer to a [TOKEN_STATISTICS](#) structure.

Return Value

Returns TRUE on success, FALSE on failure.

CAccessToken::GetTerminalServicesSessionId

Call this method to get the Terminal Services Session ID associated with the `CAccessToken` object.

```
bool GetTerminalServicesSessionId(DWORD* pdwSessionId) const throw(...);
```

Parameters

pdwSessionId

The Terminal Services Session ID.

Return Value

Returns TRUE on success, FALSE on failure.

CAccessToken::GetThreadToken

Call this method to initialize the `CAccessToken` with the token from the given thread.

```
bool GetThreadToken(  
    DWORD dwDesiredAccess,  
    HANDLE hThread = NULL,  
    bool bOpenAsSelf = true) throw();
```

Parameters

dwDesiredAccess

Specifies an access mask that specifies the requested types of access to the access token. These requested access types are compared with the token's DACL to determine which accesses are granted or denied.

hThread

Handle to the thread whose access token is opened.

bOpenAsSelf

Indicates whether the access check is to be made against the security context of the thread calling the `GetThreadToken` method or against the security context of the process for the calling thread.

If this parameter is FALSE, the access check is performed using the security context for the calling thread. If the thread is impersonating a client, this security context can be that of a client process. If this parameter is TRUE, the access check is made using the security context of the process for the calling thread.

Return Value

Returns TRUE on success, FALSE on failure.

CAccessToken::GetTokenId

Call this method to get the Token ID associated with the `CAccessToken` object.

```
bool GetTokenId(LUID* pluid) const throw(...);
```

Parameters

pluid

Pointer to a [LUID](#) which will receive the Token ID.

Return Value

Returns TRUE on success, FALSE on failure.

CAccessToken::GetType

Call this method to get the token type of the `CAccessToken` object.

```
bool GetType(TOKEN_TYPE* pType) const throw(...);
```

Parameters

pType

Address of the [TOKEN_TYPE](#) variable that, on success, receives the type of the token.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The TOKEN_TYPE enumeration type contains values that differentiate between a primary token and an impersonation token.

CAccessToken::GetUser

Call this method to identify the user associated with the `CAccessToken` object.

```
bool GetUser(CSid* pSid) const throw(...);
```

Parameters

pSid

Pointer to a [CSid Class](#) object.

Return Value

Returns TRUE on success, FALSE on failure.

CAccessToken::HKeyCurrentUser

Call this method to get the handle pointing to the user profile associated with the `CAccessToken` object.

```
HKEY HKeyCurrentUser() const throw();
```

Return Value

Returns a handle pointing to the user profile, or NULL if no profile exists.

CAccessToken::Impersonate

Call this method to assign an impersonation `CAccessToken` to a thread.

```
bool Impersonate(HANDLE hThread = NULL) const throw(...);
```

Parameters

hThread

Handle to the thread to assign the impersonation token to. This handle must have been opened with TOKEN_IMPERSONATE access rights. If *hThread* is NULL, the method causes the thread to stop using an impersonation token.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

In debug builds, an assertion error will occur if `CAccessToken` does not have a valid pointer to a token.

The [CAutoRevertImpersonation class](#) can be used to automatically revert impersonated access tokens.

CAccessToken::ImpersonateLoggedOnUser

Call this method to allow the calling thread to impersonate the security context of a logged-on user.

```
bool ImpersonateLoggedOnUser() const throw(...);
```

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

IMPORTANT

If a call to an impersonation function fails for any reason, the client is not impersonated and the client request is made in the security context of the process from which the call was made. If the process is running as a highly privileged account, or as a member of an administrative group, the user might be able to perform actions he or she would otherwise be disallowed. Therefore, the return value for this function should always be confirmed.

CAccessToken::IsTokenRestricted

Call this method to test if the `CAccessToken` object contains a list of restricted SIDs.

```
bool IsTokenRestricted() const throw();
```

Return Value

Returns TRUE if the object contains a list of restricting SIDs, FALSE if there are no restricting SIDs or if the method fails.

CAccessToken::LoadUserProfile

Call this method to load the user profile associated with the `CAccessToken` object.

```
bool LoadUserProfile() throw(...);
```

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

In debug builds, an assertion error will occur if the `CAccessToken` does not contain a valid token, or if a user profile already exists.

CAccessToken::LogonUser

Call this method to create a logon session for the user associated with the given credentials.

```
bool LogonUser(  
    LPCTSTR pszUserName,  
    LPCTSTR pszDomain,  
    LPCTSTR pszPassword,  
    DWORD dwLogonType = LOGON32_LOGON_INTERACTIVE,  
    DWORD dwLogonProvider = LOGON32_PROVIDER_DEFAULT) throw();
```

Parameters

pszUserName

Pointer to a null-terminated string that specifies the user name. This is the name of the user account to log on to.

pszDomain

Pointer to a null-terminated string that specifies the name of the domain or server whose account database contains the *pszUserName* account.

pszPassword

Pointer to a null-terminated string that specifies the clear-text password for the user account specified by *pszUserName*.

dwLogonType

Specifies the type of logon operation to perform. See [LogonUser](#) for more details.

dwLogonProvider

Specifies the logon provider. See [LogonUser](#) for more details.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The access token resulting from the logon will be associated with the `CAccessToken`. For this method to succeed, the `CAccessToken` object must hold SE_TCB_NAME privileges, identifying the holder as part of the trusted computer base. See [LogonUser](#) for more information regarding the privileges required.

CAccessToken::OpenCOMClientToken

Call this method from within a COM server handling a call from a client to initialize the `CAccessToken` with the access token from the COM client.

```
bool OpenCOMClientToken(  
    DWORD dwDesiredAccess,  
    bool bImpersonate = false,  
    bool bOpenAsSelf = true) throw(...);
```

Parameters

dwDesiredAccess

Specifies an access mask that specifies the requested types of access to the access token. These requested access types are compared with the token's DACL to determine which accesses are granted or denied.

bImpersonate

If TRUE, the current thread will impersonate the calling COM client if this call completes successfully. If FALSE, the access token will be opened, but the thread will not have an impersonation token when this call completes.

bOpenAsSelf

Indicates whether the access check is to be made against the security context of the thread calling the [GetThreadToken](#) method or against the security context of the process for the calling thread.

If this parameter is FALSE, the access check is performed using the security context for the calling thread. If the thread is impersonating a client, this security context can be that of a client process. If this parameter is TRUE, the access check is made using the security context of the process for the calling thread.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The [CAutoRevertImpersonation Class](#) can be used to automatically revert impersonated access tokens created by setting the *bImpersonate* flag to TRUE.

CAccessToken::OpenNamedPipeClientToken

Call this method from within a server taking requests over a named pipe to initialize the `CAccessToken` with the access token from the client.

```
bool OpenNamedPipeClientToken(  
    HANDLE hPipe,  
    DWORD dwDesiredAccess,  
    bool bImpersonate = false,  
    bool bOpenAsSelf = true) throw(...);
```

Parameters

hPipe

Handle to a named pipe.

dwDesiredAccess

Specifies an access mask that specifies the requested types of access to the access token. These requested access types are compared with the token's DACL to determine which accesses are granted or denied.

bImpersonate

If TRUE, the current thread will impersonate the calling pipe client if this call completes successfully. If FALSE, the access token will be opened, but the thread will not have an impersonation token when this call completes.

bOpenAsSelf

Indicates whether the access check is to be made against the security context of the thread calling the [GetThreadToken](#) method or against the security context of the process for the calling thread.

If this parameter is FALSE, the access check is performed using the security context for the calling thread. If the thread is impersonating a client, this security context can be that of a client process. If this parameter is TRUE, the access check is made using the security context of the process for the calling thread.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The [CAutoRevertImpersonation Class](#) can be used to automatically revert impersonated access tokens created by setting the *bImpersonate* flag to TRUE.

CAccessToken::OpenRPCClientToken

Call this method from within a server handling a call from an RPC client to initialize the `CAccessToken` with the access token from the client.

```
bool OpenRPCClientToken(  
    RPC_BINDING_HANDLE BindingHandle,  
    DWORD dwDesiredAccess,  
    bool bImpersonate = false,  
    bool bOpenAsSelf = true) throw(...);
```

Parameters

BindingHandle

Binding handle on the server that represents a binding to a client.

dwDesiredAccess

Specifies an access mask that specifies the requested types of access to the access token. These requested access

types are compared with the token's DACL to determine which accesses are granted or denied.

bImpersonate

If TRUE, the current thread will impersonate the calling RPC client if this call completes successfully. If FALSE, the access token will be opened, but the thread will not have an impersonation token when this call completes.

bOpenAsSelf

Indicates whether the access check is to be made against the security context of the thread calling the [GetThreadToken](#) method or against the security context of the process for the calling thread.

If this parameter is FALSE, the access check is performed using the security context for the calling thread. If the thread is impersonating a client, this security context can be that of a client process. If this parameter is TRUE, the access check is made using the security context of the process for the calling thread.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The [CAutoRevertImpersonation Class](#) can be used to automatically revert impersonated access tokens created by setting the *bImpersonate* flag to TRUE.

CAccessToken::OpenThreadToken

Call this method to set the impersonation level and then initialize the `CAccessToken` with the token from the given thread.

```
bool OpenThreadToken(  
    DWORD dwDesiredAccess,  
    bool bImpersonate = false,  
    bool bOpenAsSelf = true,  
    SECURITY_IMPERSONATION_LEVEL sil = SecurityImpersonation) throw(...);
```

Parameters

dwDesiredAccess

Specifies an access mask that specifies the requested types of access to the access token. These requested access types are compared with the token's DACL to determine which accesses are granted or denied.

bImpersonate

If TRUE, the thread will be left at the requested impersonation level after this method completes. If FALSE, the thread will revert to its original impersonation level.

bOpenAsSelf

Indicates whether the access check is to be made against the security context of the thread calling the [GetThreadToken](#) method or against the security context of the process for the calling thread.

If this parameter is FALSE, the access check is performed using the security context for the calling thread. If the thread is impersonating a client, this security context can be that of a client process. If this parameter is TRUE, the access check is made using the security context of the process for the calling thread.

sil

Specifies a [SECURITY_IMPERSONATION_LEVEL](#) enumerated type that supplies the impersonation level of the token.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

`OpenThreadToken` is similar to [CAccessToken::GetThreadToken](#), but sets the impersonation level before initializing the `CAccessToken` from the thread's access token.

The [CAutoRevertImpersonation Class](#) can be used to automatically revert impersonated access tokens created by setting the *bImpersonate* flag to TRUE.

CAccessToken::PrivilegeCheck

Call this method to determine whether a specified set of privileges are enabled in the `CAccessToken` object.

```
bool PrivilegeCheck(  
    PPRIVILEGE_SET RequiredPrivileges,  
    bool* pbResult) const throw();
```

Parameters

RequiredPrivileges

Pointer to a [PRIVILEGE_SET](#) structure.

pbResult

Pointer to a value the method sets to indicate whether any or all of the specified privilege are enabled in the `CAccessToken` object.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

When `PrivilegeCheck` returns, the `Attributes` member of each [LUID_AND_ATTRIBUTES](#) structure is set to SE_PRIVILEGE_USED_FOR_ACCESS if the corresponding privilege is enabled. This method calls the [PrivilegeCheck](#) Win32 function.

CAccessToken::Revert

Call this method to stop a thread from using an impersonation token.

```
bool Revert(HANDLE hThread = NULL) const throw();
```

Parameters

hThread

Handle to the thread to revert from impersonation. If *hThread* is NULL, the current thread is assumed.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The reversion of impersonation tokens can be performed automatically with the [CAutoRevertImpersonation Class](#).

CAccessToken::SetDefaultDacl

Call this method to set the default DACL of the `CAccessToken` object.

```
bool SetDefaultDacl(const CDacl& rDacl) throw(...);
```

Parameters

rDacl

The new default [CDacl Class](#) information.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The default DACL is the DACL that is used by default when new objects are created with this access token in effect.

CAccessToken::SetOwner

Call this method to set the owner of the `CAccessToken` object.

```
bool SetOwner(const CSid& rSid) throw(...);
```

Parameters

rSid

The [CSid Class](#) object containing the owner information.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The owner is the default owner that is used for new objects created while this access token is in effect.

CAccessToken::SetPrimaryGroup

Call this method to set the primary group of the `CAccessToken` object.

```
bool SetPrimaryGroup(const CSid& rSid) throw(...);
```

Parameters

rSid

The [CSid Class](#) object containing the primary group information.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The primary group is the default group for new objects created while this access token is in effect.

See also

[ATLSecurity Sample](#)

[Access Tokens](#)

[Class Overview](#)

CAcl Class

3/4/2019 • 5 minutes to read • [Edit Online](#)

This class is a wrapper for an `ACL` (access-control list) structure.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CAcl
```

Members

Public Typedefs

NAME	DESCRIPTION
CAcl::CAccessMaskArray	An array of ACCESS_MASKs.
CAcl::CAceFlagArray	An array of BYTES.
CAcl::CAceTypeArray	An array of BYTES.

Public Constructors

NAME	DESCRIPTION
CAcl::CAcl	The constructor.
CAcl::~~CAcl	The destructor.

Public Methods

NAME	DESCRIPTION
CAcl::GetAceCount	Returns the number of access-control entry (ACE) objects.
CAcl::GetAdEntries	Retrieves the access-control list (ACL) entries from the <code>CAcl</code> object.
CAcl::GetAdEntry	Retrieves all of the information about an entry in a <code>CAcl</code> object.
CAcl::GetLength	Returns the length of the ACL.
CAcl::GetPACL	Returns a PACL (pointer to an ACL).

NAME	DESCRIPTION
CAcl::IsEmpty	Tests the <code>CAcl</code> object for entries.
CAcl::IsNull	Returns the status of the <code>CAcl</code> object.
CAcl::RemoveAce	Removes a specific ACE (access-control entry) from the <code>CAcl</code> object.
CAcl::RemoveAces	Removes all ACEs (access-control entries) from the <code>CAcl</code> that apply to the given <code>CSid</code> .
CAcl::SetEmpty	Marks the <code>CAcl</code> object as empty.
CAcl::SetNull	Marks the <code>CAcl</code> object as NULL.

Public Operators

NAME	DESCRIPTION
CAcl::operator const ACL *	Casts a <code>CAcl</code> object to an <code>ACL</code> structure.
CAcl::operator =	Assignment operator.

Remarks

The `ACL` structure is the header of an ACL (access-control list). An ACL includes a sequential list of zero or more [ACEs](#) (access-control entries). The individual ACEs in an ACL are numbered from 0 to $n-1$, where n is the number of ACEs in the ACL. When editing an ACL, an application refers to an access-control entry (ACE) within the ACL by its index.

There are two ACL types:

- Discretionary
- System

A discretionary ACL is controlled by the owner of an object or anyone granted `WRITE_DAC` access to the object. It specifies the access particular users and groups can have to an object. For example, the owner of a file can use a discretionary ACL to control which users and groups can and cannot have access to the file.

An object can also have system-level security information associated with it, in the form of a system ACL controlled by a system administrator. A system ACL can allow the system administrator to audit any attempts to gain access to an object.

For more details, see the [ACL](#) discussion in the Windows SDK.

For an introduction to the access control model in Windows, see [Access Control](#) in the Windows SDK.

Requirements

Header: `atlsecurity.h`

CAcl::CAccessMaskArray

An array of ACCESS_MASK objects.

```
typedef CAtlArray<ACCESS_MASK> CAccessMaskArray;
```

Remarks

This typedef specifies the array type that can be used to store access rights used in access-control entries (ACEs).

CAcl::CAceFlagArray

An array of BYTEs.

```
typedef CAtlArray<BYTE> CAceFlagArray;
```

Remarks

This typedef specifies the array type used to define the access-control entry (ACE) type-specific control flags. See the [ACE_HEADER](#) definition for the complete list of possible flags.

CAcl::CAceTypeArray

An array of BYTEs.

```
typedef CAtlArray<BYTE> CAceTypeArray;
```

Remarks

This typedef specifies the array type used to define the nature of the access-control entry (ACE) objects, such as ACCESS_ALLOWED_ACE_TYPE or ACCESS_DENIED_ACE_TYPE. See the [ACE_HEADER](#) definition for the complete list of possible types.

CAcl::CAcl

The constructor.

```
CAcl() throw();  
CAcl(const CAcl& rhs) throw(...);
```

Parameters

rhs

An existing `CAcl` object.

Remarks

The `CAcl` object can be optionally created using an existing `CAcl` object.

CAcl::~CAcl

The destructor.

```
virtual ~CAcl() throw();
```

Remarks

The destructor frees any resources acquired by the object.

CAcl::GetAceCount

Returns the number of access-control entry (ACE) objects.

```
virtual UINT GetAceCount() const throw() = 0;
```

Return Value

Returns the number of ACE entries in the `CAcl` object.

CAcl::GetAclEntries

Retrieves the access-control list (ACL) entries from the `CAcl` object.

```
void GetAclEntries(  
    CSid::CSidArray* pSids,  
    CAccessMaskArray* pAccessMasks = NULL,  
    CAceTypeArray* pAceTypes = NULL,  
    CAceFlagArray* pAceFlags = NULL) const throw(...);
```

Parameters

pSids

A pointer to an array of `CSid` objects.

pAccessMasks

The access masks.

pAceTypes

The access-control entry (ACE) types.

pAceFlags

The ACE flags.

Remarks

This method fills the array parameters with the details of every ACE object contained in the `CAcl` object. Use NULL when the details for that particular array are not required.

The contents of each array correspond to each other, that is, the first element of the `CAccessMaskArray` array corresponds to the first element in the `CSidArray` array, and so on.

See [ACE_HEADER](#) for more details on ACE types and flags.

CAcl::GetAclEntry

Retrieves all of the information about an entry in an access-control list (ACL).

```
void GetAclEntry(  
    UINT nIndex,  
    CSid* pSid,  
    ACCESS_MASK* pMask = NULL,  
    BYTE* pType = NULL,  
    BYTE* pFlags = NULL,  
    GUID* pObjectType = NULL,  
    GUID* pInheritedObjectType = NULL) const throw(...);
```

Parameters

nIndex

Index to the ACL entry to retrieve.

pSid

The [CSid](#) object to which the ACL entry applies.

pMask

The mask specifying permissions to grant or deny access.

pType

The ACE type.

pFlags

The ACE flags.

pObjectType

The object type. This will be set to GUID_NULL if the object type is not specified in the ACE, or if the ACE is not an OBJECT ACE.

pInheritedObjectType

The inherited object type. This will be set to GUID_NULL if the inherited object type is not specified in the ACE, or if the ACE is not an OBJECT ACE.

Remarks

This method will retrieve all of the information about an individual ACE, providing more information than [CAcl::GetAclEntries](#) alone makes available.

See [ACE_HEADER](#) for more details on ACE types and flags.

CAcl::GetLength

Returns the length of the access-control list (ACL).

```
UINT GetLength() const throw();
```

Return Value

Returns the required length in bytes necessary to hold the [ACL](#) structure.

CAcl::GetPACL

Returns a pointer to an access-control list (ACL).

```
const ACL* GetPACL() const throw(...);
```

Return Value

Returns a pointer to the [ACL](#) structure.

CAcl::IsEmpty

Tests the [CAcl](#) object for entries.

```
bool IsEmpty() const throw();
```

Remarks

Returns TRUE if the `CACl` object is not NULL, and contains no entries. Returns FALSE if the `CACl` object is either NULL, or contains at least one entry.

`CACl::IsNull`

Returns the status of the `CACl` object.

```
bool IsNull() const throw();
```

Return Value

Returns TRUE if the `CACl` object is NULL, FALSE otherwise.

`CACl::operator const ACL *`

Casts a `CACl` object to an `ACL` (access-control list) structure.

```
operator const ACL *() const throw(...);
```

Remarks

Returns the address of the `ACL` structure.

`CACl::operator =`

Assignment operator.

```
CACl& operator= (const CACl& rhs) throw(...);
```

Parameters

rhs

The `CACl` to assign to the existing object.

Return Value

Returns a reference to the updated `CACl` object.

`CACl::RemoveAce`

Removes a specific ACE (access-control entry) from the `CACl` object.

```
void RemoveAce(UINT nIndex) throw();
```

Parameters

nIndex

Index to the ACE entry to remove.

Remarks

This method is derived from [CAclArray::RemoveAt](#).

`CACl::RemoveAces`

Removes alls ACEs (access-control entries) from the `CACl` that apply to the given `CSid`.


```
bool RemoveAces(const CSid& rSid) throw(...)
```

Parameters

rSid

A reference to a `CSid` object.

CAcl::SetEmpty

Marks the `CAcl` object as empty.

```
void SetEmpty() throw();
```

Remarks

The `CAcl` can be set to empty or to NULL: the two states are distinct.

CAcl::SetNull

Marks the `CAcl` object as NULL.

```
void SetNull() throw();
```

Remarks

The `CAcl` can be set to empty or to NULL: the two states are distinct.

See also

[Class Overview](#)

[Security Global Functions](#)

CAdapt Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

This template is used to wrap classes that redefine the address-of operator to return something other than the address of the object.

Syntax

```
template <class T>
class CAdapt
```

Parameters

T

The adapted type.

Members

Public Constructors

NAME	DESCRIPTION
CAdapt::CAdapt	The constructor.

Public Operators

NAME	DESCRIPTION
CAdapt::operator const T&	Returns a const reference to <code>m_T</code> .
CAdapt::operator T&	Returns a reference to <code>m_T</code> .
CAdapt::operator <	Compares an object of the adapted type with <code>m_T</code> .
CAdapt::operator =	Assigns an object of the adapted type to <code>m_T</code> .
CAdapt::operator ==	Compares an object of the adapted type with <code>m_T</code> .

Public Data Members

NAME	DESCRIPTION
CAdapt::m_T	The data being adapted.

Remarks

`CAdapt` is a simple template used to wrap classes that redefine the address-of operator (`operator &`) to return something other than the address of the object. Examples of such classes include ATL's `CComBSTR`, `CComPtr`, and `CComQIPtr` classes, and the compiler COM support class, `_com_ptr_t`. These classes all redefine the address-of operator to return the address of one of their data members (a `BSTR` in the case of `CComBSTR`, and an interface

pointer in the case of the other classes).

`CAdapt`'s primary role is to hide the address-of operator defined by class `T`, yet still retain the characteristics of the adapted class. `CAdapt` fulfils this role by holding a public member, `m_T`, of type `T`, and by defining conversion operators, comparison operators, and a copy constructor to allow specializations of `CAdapt` to be treated as if they are objects of type `T`.

The adapter class `CAdapt` is useful because some container-style classes expect to be able to obtain the addresses of their contained objects using the address-of operator. The redefinition of the address-of operator can confound this requirement, typically causing compilation errors and preventing the use of the non-adapted type with classes that expect it to "just work". `CAdapt` provides a way around those problems.

Typically, you will use `CAdapt` when you want to store `CComBSTR`, `CComPtr`, `CComQIPtr`, or `_com_ptr_t` objects in a container-style class. This was most commonly necessary for C++ Standard Library containers prior to support for the C++11 Standard, but C++11 Standard Library containers automatically work with types that have overloaded `operator&()`. The Standard Library achieves this by internally using `std::addressof` to get the true addresses of objects.

Requirements

Header: `atlcomcli.h`

CAdapt::CAdapt

The constructors allow an adapter object to be default constructed, copied from an object of the adapted type, or copied from another adapter object.

```
CAdapt();
CAdapt(const T& rSrc);
CAdapt(const CAdapt& rSrCA);
CAdapt(T&& rSrcCA); // (Visual Studio 2017)
CAdapt(CAdapt<T>&& rSrCA) noexcept; // (Visual Studio 2017)
```

Parameters

rSrc

A variable of the type being adapted to be copied into the newly constructed adapter object.

rSrCA

An adapter object whose contained data should be copied (or moved) into the newly constructed adapter object.

CAdapt::m_T

Holds the data being adapted.

```
T m_T;
```

Remarks

This **public** data member can be accessed directly or indirectly with `operator const T&` and `operator T&`.

CAdapt::operator const T&

Returns a **const** reference to the `m_T` member, allowing the adapter object to be treated as if it were an object of type `T`.

```
operator const T&() const;
```

Return Value

A **const** reference to `m_T`.

CAdapt::operator T&

Returns a reference to the `m_T` member, allowing the adapter object to be treated as if it were an object of type *T*.

```
operator T&();
```

Return Value

A reference to `m_T`.

CAdapt::operator <

Compares an object of the adapted type with `m_T`.

```
bool operator<(const T& rSrc) const;
```

Parameters

rSrc

A reference to the object to be compared.

Return Value

The result of the comparison between `m_T` and *rSrc*.

CAdapt::operator =

The assignment operator assigns the argument, *rSrc*, to the data member `m_T` and returns the current adapter object.

```
CAdapt& operator= (const T& rSrc);  
CAdapt& operator= (T&& rSrcA); // (Visual Studio 2017)  
CAdapt& operator= (CAdapt<T>&& rSrcA) noexcept; // (Visual Studio 2017)
```

Parameters

rSrc

A reference to an object of the adapted type to be copied.

rSrcA

A reference to an object to be moved.

Return Value

A reference to the current object.

CAdapt::operator ==

Compares an object of the adapted type with `m_T`.

```
bool operator== (const T& rSrc) const;
```

Parameters

rSrc

A reference to the object to be compared.

Return Value

The result of the comparison between *m_T* and *rSrc*.

See also

[Class Overview](#)

CAtlArray Class

4/1/2019 • 11 minutes to read • [Edit Online](#)

This class implements an array object.

Syntax

```
template<typename E, class ETraits = CElementTraits<E>>
class CAtlArray
```

Parameters

E

The type of data to be stored in the array.

ETraits

The code used to copy or move elements.

Members

Methods

Add	Call this method to add an element to the array object.
Append	Call this method to add the contents of one array to the end of another.
AssertValid	Call this method to confirm that the array object is valid.
CAtlArray	The constructor.
~CAtlArray	The destructor.
Copy	Call this method to copy the elements of one array to another.
FreeExtra	Call this method to remove any empty elements from the array.
GetAt	Call this method to retrieve a single element from the array object.
GetCount	Call this method to return the number of elements stored in the array.
GetData	Call this method to return a pointer to the first element in the array.
InsertArrayAt	Call this method to insert one array into another.

InsertAt	Call this method to insert a new element (or multiple copies of an element) into the array object.
IsEmpty	Call this method to test if the array is empty.
RemoveAll	Call this method to remove all elements from the array object.
RemoveAt	Call this method to remove one or more elements from the array.
SetAt	Call this method to set the value of an element in the array object.
SetAtGrow	Call this method to set the value of an element in the array object, expanding the array as required.
SetCount	Call this method to set the size of the array object.

Operators

operator []	Call this operator to return a reference to an element in the array.

Typedefs

INARGTYPE	The data type to use for adding elements to the array.
OUTARGTYPE	The data type to use for retrieving elements from the array.

Remarks

`CAtlArray` provides methods for creating and managing an array of elements of a user-defined type. Although similar to standard C arrays, the `CAtlArray` object can dynamically shrink and grow as necessary. The array index always starts at position 0, and the upper bound can be fixed, or allowed to expand as new elements are added.

For arrays with a small number of elements, the ATL class [CSimpleArray](#) can be used.

`CAtlArray` is closely related to MFC's `CArray` class and will work in an MFC project, albeit without serialization support.

For more information, see [ATL Collection Classes](#).

Requirements

Header: atlcoll.h

CAtlArray::Add

Call this method to add an element to the array object.

```
size_t Add(INARGTYPE element);  
size_t Add();
```

Parameters

element

The element to be added to the array.

Return Value

Returns the index of the added element.

Remarks

The new element is added to the end of the array. If no element is provided, an empty element is added; that is, the array is increased in size as though a real element has been added. If the operation fails, [AtlThrow](#) is called with the argument `E_OUTOFMEMORY`.

Example

```
// Declare an array of integers  
CAtlArray<int> iArray;  
  
iArray.Add(1);    // element 0  
iArray.Add(2);    // element 1  
iArray.Add();     // element 2  
  
ATLASSERT(iArray.GetCount() == 3);
```

CAtlArray::Append

Call this method to add the contents of one array to the end of another.

```
size_t Append(const CAtlArray<E, ETraits>& aSrc);
```

Parameters

aSrc

The array to append.

Return Value

Returns the index of the first appended element.

Remarks

The elements in the supplied array are added to the end of the existing array. If necessary, memory will be allocated to accommodate the new elements.

The arrays must be of the same type, and it is not possible to append an array to itself.

In debug builds, an `ATLASSERT` will be raised if the `CAtlArray` argument is not a valid array or if *aSrc* refers to the same object. In release builds, invalid arguments may lead to unpredictable behavior.

Example


```
// Declare two integer arrays
CATlArray<int> iArray1,iArray2;

iArray1.Add(1); // element 0
iArray1.Add(2); // element 1

iArray2.Add(3); // element 0
iArray2.Add(4); // element 1

// Append iArray2 to iArray1
iArray1.Append(iArray2);

ATLASSERT(iArray1.GetCount() == 4);
```

CATlArray::AssertValid

Call this method to confirm that the array object is valid.

```
void AssertValid() const;
```

Remarks

If the array object is not valid, ATLASSERT will throw an assertion. This method is available only if `_DEBUG` is defined.

Example

```
CATlArray<float> fArray;
// AssertValid only exists in debug builds
#ifdef _DEBUG
fArray.AssertValid();
#endif
```

CATlArray::CATlArray

The constructor.

```
CATlArray() throw();
```

Remarks

Initializes the array object.

Example

```
CATlArray<int> iArray;
```

CATlArray::~CATlArray

The destructor.

```
~CATlArray() throw();
```

Remarks

Frees up any resources used by the array object.

CAtlArray::Copy

Call this method to copy the elements of one array to another.

```
void Copy(const CAtlArray<E, ETraits>& aSrc);
```

Parameters

aSrc

The source of the elements to copy to an array.

Remarks

Call this method to overwrite elements of one array with the elements of another array. If necessary, memory will be allocated to accommodate the new elements. It is not possible to copy elements of an array to itself.

If the existing contents of the array are to be retained, use [CAtlArray::Append](#) instead.

In debug builds, an ATLASSERT will be raised if the existing `CAtlArray` object is not valid, or if *aSrc* refers to the same object. In release builds, invalid arguments may lead to unpredictable behavior.

NOTE

`CAtlArray::Copy` does not support arrays consisting of elements created with the [CAutoPtr](#) class.

Example

```
CAtlArray<int> iArrayS, iArrayT;

iArrayS.Add(1);
iArrayS.Add(2);

iArrayT.Add(3);
iArrayT.Add(4);

iArrayT.Copy(iArrayS);

ATLASSERT(iArrayT.GetCount() == 2);
ATLASSERT(iArrayT[0] == 1);
ATLASSERT(iArrayT[1] == 2);
```

CAtlArray::FreeExtra

Call this method to remove any empty elements from the array.

```
void FreeExtra() throw();
```

Remarks

Any empty elements are removed, but the size and upper bound of the array remain unchanged.

In debug builds, an ATLASSERT will be raised if the `CAtlArray` object is not valid, or if the array would exceed its maximum size.

CAtlArray::GetAt

Call this method to retrieves a single element from the array object.

```
const E& GetAt(size_t iElement) const throw();
E& GetAt(size_t iElement) throw();
```

Parameters

iElement

The index value of the array element to return.

Return Value

Returns a reference to the required array element.

Remarks

In debug builds, an ATLASSERT will be raised if *iElement* exceeds the number of elements in the array. In release builds, an invalid argument may lead to unpredictable behavior.

Example

```
// Declare an array of integers

CAtlArray<int> iMyArray;
int element;

// Add ten elements to the array
for (int i = 0; i < 10; i++)
{
    iMyArray.Add(i);
}

// Use GetAt and SetAt to modify
// every element in the array

for (size_t i = 0; i < iMyArray.GetCount(); i++)
{
    element = iMyArray.GetAt(i);
    element *= 10;
    iMyArray.SetAt(i, element);
}
```

CAtlArray::GetCount

Call this method to return the number of elements stored in the array.

```
size_t GetCount() const throw();
```

Return Value

Returns the number of elements stored in the array.

Remarks

As the first element in the array is at position 0, the value returned by `GetCount` is always 1 greater than the largest index.

Example

See the example for [CAtlArray::GetAt](#).

CAtlArray::GetData

Call this method to return a pointer to the first element in the array.

```
E* GetData() throw();  
const E* GetData() const throw();
```

Return Value

Returns a pointer to the memory location storing the first element in the array. If no elements are available, NULL is returned.

Example

```
// Define an array of integers  
CAtlArray<int> MyArray;  
  
// Define a pointer  
int* pData;  
  
// Allocate enough space for 32 elements  
// with buffer increase to be calculated  
// automatically  
MyArray.SetCount(32, -1);  
  
// Set the pointer to the first element  
pData = MyArray.GetData();  
  
// Set array values directly  
for (int j = 0; j < 32; j++, pData++)  
{  
    *pData = j * 10;  
}
```

CAtlArray::INARGTYPE

The data type to use for adding elements to the array.

```
typedef ETraits::INARGTYPE INARGTYPE;
```

CAtlArray::InsertArrayAt

Call this method to insert one array into another.

```
void InsertArrayAt(size_t iStart, const CAtlArray<E, ETraits>* paNew);
```

Parameters

iStart

The index at which the array is to be inserted.

paNew

The array to be inserted.

Remarks

Elements from the array *paNew* are copied into the array object, beginning at element *iStart*. The existing array elements are moved to avoid being overwritten.

In debug builds, an ATLASSERT will be raised if the `CAtlArray` object is not valid, or if the *paNew* pointer is

NULL or invalid.

NOTE

`CAtlArray::InsertArrayAt` does not support arrays consisting of elements created with the `CAutoPtr` class.

Example

```
// Define two integer arrays
CAtlArray<int> iTargetArray, iSourceArray;

// Add elements to first array
for (int x = 0; x < 10; x++)
{
    iTargetArray.Add(x);
}

// Add elements to the second array
for (int x = 0; x < 10; x++)
{
    iSourceArray.Add(x * 10);
}

// Insert the Source array into the Target
// array, starting at the 5th element.
iTargetArray.InsertArrayAt(5, &iSourceArray);
```

CAtlArray::InsertAt

Call this method to insert a new element (or multiple copies of an element) into the array object.

```
void InsertAt(size_t iElement, INARGTYPE element, size_t nCount = 1);
```

Parameters

iElement

The index where the element or elements are to be inserted.

element

The value of the element or elements to be inserted.

nCount

The number of elements to add.

Remarks

Inserts one or more elements into the array, starting at index *iElement*. Existing elements are moved to avoid being overwritten.

In debug builds, an ATLASSERT will be raised if the `CAtlArray` object is invalid, the number of elements to be added is zero, or the combined number of elements is too large for the array to contain. In retail builds, passing invalid parameters may cause unpredictable results.

Example

```
// Declare an array of integers
CAtlArray<int> iBuffer;

// Add elements to the array
for (int b = 0; b < 10; b++)
{
    iBuffer.Add(0);
}

// Insert ten 1's into the array
// at position 5
iBuffer.InsertAt(5, 1, 10);
```

CAtlArray::IsEmpty

Call this method to test if the array is empty.

```
bool IsEmpty() const throw();
```

Return Value

Returns true if the array is empty, false otherwise.

Remarks

The array is said to be empty if it contains no elements. Therefore, even if the array contains empty elements, it is not empty.

Example

```
// Define an array of chars
CAtlArray<char> cArray;

// Add an element
cArray.Add('a');

// Confirm array is not empty
ATLASSERT(!cArray.IsEmpty());

// Remove all elements
cArray.RemoveAll();

// Confirm array is empty
ATLASSERT(cArray.IsEmpty());
```

CAtlArray::operator []

Call this operator to return a reference to an element in the array.

```
E& operator[](size_t ielement) throw();
const E& operator[](size_t ielement) const throw();
```

Parameters

iElement

The index value of the array element to return.

Return Value

Returns a reference to the required array element.

Remarks

Performs a similar function to [CAtlArray::GetAt](#). Unlike the MFC class [CArray](#), this operator cannot be used as a substitute for [CAtlArray::SetAt](#).

In debug builds, an ATLASSERT will be raised if *iElement* exceeds the total number of elements in the array. In retail builds, an invalid parameter may cause unpredictable results.

CAtlArray::OUTARGTYPE

The data type to use for retrieving elements from the array.

```
typedef ETraits::OUTARGTYPE OUTARGTYPE;
```

CAtlArray::RemoveAll

Call this method to remove all elements from the array object.

```
void RemoveAll() throw();
```

Remarks

Removes all of the elements from the array object.

This method calls [CAtlArray::SetCount](#) to resize the array and subsequently frees any allocated memory.

Example

See the example for [CAtlArray::IsEmpty](#).

CAtlArray::RemoveAt

Call this method to remove one or more elements from the array.

```
void RemoveAt(size_t iElement, size_t nCount = 1);
```

Parameters

iElement

The index of the first element to remove.

nCount

The number of elements to remove.

Remarks

Removes one or more elements from the array. Any remaining elements are shifted down. The upper bound is decremented, but memory is not freed until a call to [CAtlArray::FreeExtra](#) is made.

In debug builds, an ATLASSERT will be raised if the `CAtlArray` object is not valid, or if the combined total of *iElement* and *nCount* exceeds the total number of elements in the array. In retail builds, invalid parameters may cause unpredictable results.

Example

```
// Declare an array of chars
CAtlArray<char> cMyArray;

// Add ten elements to the array
for (int a = 0; a < 10; a++)
{
    cMyArray.Add('*');
}

// Remove five elements starting with
// the element at position 1
cMyArray.RemoveAt(1, 5);

// Free memory
cMyArray.FreeExtra();

// Confirm size of array
ATLASSERT(cMyArray.GetCount() == 5);
```

CAtlArray::SetAt

Call this method to set the value of an element in the array object.

```
void SetAt(size_t iElement, INARGTYPE element);
```

Parameters

iElement

The index pointing to the array element to set.

element

The new value of the specified element.

Remarks

In debug builds, an ATLASSERT will be raised if *iElement* exceeds the number of elements in the array. In retail builds, an invalid parameter may result in unpredictable results.

Example

See the example for [CAtlArray::GetAt](#).

CAtlArray::SetCount

Call this method to set the size of the array object.

```
bool SetCount(size_t nNewSize, int nGrowBy = - 1);
```

Parameters

nNewSize

The required size of the array.

nGrowBy

A value used to determine how large to make the buffer. A value of -1 causes an internally calculated value to be used.

Return Value

Returns true if the array is successfully resized, false otherwise.

Remarks

The array can be increased or decreased in size. If increased, extra empty elements are added to the array. If decreased, the elements with the largest indices will be deleted and memory freed.

Use this method to set the size of the array before using it. If `SetCount` is not used, the process of adding elements — and the subsequent memory allocation performed — will reduce performance and fragment memory.

Example

See the example for [CAtlArray::GetData](#).

CAtlArray::SetAtGrow

Call this method to set the value of an element in the array object, expanding the array as required.

```
void SetAtGrow(size_t iElement, INARGTYPE element);
```

Parameters

iElement

The index pointing to the array element to set.

element

The new value of the specified element.

Remarks

Replaces the value of the element pointed to by the index. If *iElement* is larger than the current size of the array, the array is automatically increased using a call to [CAtlArray::SetCount](#). In debug builds, an ATLASSERT will be raised if the `CAtlArray` object is not valid. In retail builds, invalid parameters may cause unpredictable results.

Example

```
// Declare an array of integers
CAtlArray<int> iGrowArray;

// Add an element
iGrowArray.Add(0);

// Add an extra element at position 19.
// This will grow the array to accommodate.
iGrowArray.SetAtGrow(19, 0);

// Confirm size of new array
ATLASSERT(iGrowArray.GetCount() == 20);

// Note: the values at position 1 to 18
// are undefined.
```

See also

[MMXSwarm Sample](#)

[DynamicConsumer Sample](#)

[UpdatePV Sample](#)

[Marquee Sample](#)

[CArray Class](#)

[Class Overview](#)

CAtlAutoThreadModule Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements a thread-pooled, apartment-model COM server.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CAtlAutoThreadModule : public CAtlAutoThreadModuleT<CAtlAutoThreadModule>
```

Remarks

`CAtlAutoThreadModule` derives from `CAtlAutoThreadModuleT` and implements a thread-pooled, apartment-model COM server. `CAtlAutoThreadModule` uses `CComApartment` to manage an apartment for each thread in the module.

You must use the `DECLARE_CLASSFACTORY_AUTO_THREAD` macro in your object's class definition to specify `CComClassFactoryAutoThread` as the class factory. You should then add a single instance of a class derived from

`CAtlAutoThreadModuleT` such as `CAtlAutoThreadModule`. For example:

```
CAtlAutoThreadModule _AtlAutoModule; // name is immaterial.
```

NOTE

This class replaces the obsolete `CComAutoThreadModule` class.

Inheritance Hierarchy

`IAtlAutoThreadModule`

`CAtlAutoThreadModuleT`

`CAtlAutoThreadModule`

Requirements

Header: atlbase.h

See also

[CAtlAutoThreadModuleT Class](#)

[IAtlAutoThreadModule Class](#)

[Class Overview](#)

[Module Classes](#)

CAtlAutoThreadModuleT Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods for implementing a thread-pooled, apartment-model COM server.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class T,
    class ThreadAllocator = CComSimpleThreadAllocator,
    DWORD dwWait = INFINITE>
class ATL_NO_VTABLE CAtlAutoThreadModuleT : public IAtlAutoThreadModule
```

Parameters

T

The class which will implement the COM server.

ThreadAllocator

The class managing thread selection. The default value is [CComSimpleThreadAllocator](#).

dwWait

Specifies the time-out interval, in milliseconds. The default is INFINITE, which means the method's time-out interval never elapses.

Members

Public Methods

NAME	DESCRIPTION
CAtlAutoThreadModuleT::GetDefaultThreads	This static function dynamically calculates and returns the maximum number of threads for the EXE module, based on the number of processors.

Remarks

The class [CAtlAutoThreadModule](#) derives from `CAtlAutoThreadModuleT` in order to implement a thread-pooled, apartment-model COM server. It replaces the obsolete class [CComAutoThreadModule](#).

NOTE

This class should not be used in a DLL, as the default *dwWait* value of INFINITE will cause a deadlock when the DLL is unloaded.

Inheritance Hierarchy

[IAtlAutoThreadModule](#)

[CAtlAutoThreadModuleT](#)

Requirements

Header: atlbase.h

CAtlAutoThreadModuleT::GetDefaultThreads

This static function dynamically calculates and returns the maximum number of threads for the EXE module, based on the number of processors.

```
static int GetDefaultThreads();
```

Return Value

The number of threads to be created in the EXE module.

Remarks

Override this method if you want to use a different method for calculating the number of threads. By default, the number of threads is based on the number of processors.

See also

[IAtlAutoThreadModule Class](#)

[Class Overview](#)

[IAtlAutoThreadModule Class](#)

[Module Classes](#)

CAtlBaseModule Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class is instantiated in every ATL project.

Syntax

```
class CAtlBaseModule : public _ATL_BASE_MODULE
```

Members

Public Constructors

NAME	DESCRIPTION
CAtlBaseModule::CAtlBaseModule	The constructor.

Public Methods

NAME	DESCRIPTION
CAtlBaseModule::AddResourceInstance	Adds a resource instance to the list of stored handles.
CAtlBaseModule::GetHInstanceAt	Returns a handle to a specified resource instance.
CAtlBaseModule::GetModuleInstance	Returns the module instance from a <code>CAtlBaseModule</code> object.
CAtlBaseModule::GetResourceInstance	Returns the resource instance from a <code>CAtlBaseModule</code> object.
CAtlBaseModule::RemoveResourceInstance	Removes a resource instance from the list of stored handles.
CAtlBaseModule::SetResourceInstance	Sets the resource instance of a <code>CAtlBaseModule</code> object.

Public Data Members

NAME	DESCRIPTION
CAtlBaseModule::m_bInitFailed	A variable that indicates if the module initialization has failed.

Remarks

An instance of `CAtlBaseModule` named `_AtlBaseModule` is present in every ATL project, containing a handle to the module instance, a handle to the module containing resources (which by default, are one and the same), and an array of handles to modules providing primary resources. `CAtlBaseModule` can be safely accessed from multiple threads.

This class replaces the obsolete `CComModule` class used in earlier versions of ATL.

Inheritance Hierarchy

[_ATL_BASE_MODULE](#)

CAtlBaseModule

Requirements

Header: atlcore.h

CAtlBaseModule::AddResourceInstance

Adds a resource instance to the list of stored handles.

```
bool AddResourceInstance(HINSTANCE hInst) throw();
```

Parameters

hInst

The resource instance to add.

Return Value

Returns true if the resource was successfully added, false otherwise.

CAtlBaseModule::CAtlBaseModule

The constructor.

```
CAtlBaseModule() throw();
```

Remarks

Creates the `CAtlBaseModule`.

CAtlBaseModule::GetHInstanceAt

Returns a handle to a specified resource instance.

```
HINSTANCE GetHInstanceAt(int i) throw();
```

Parameters

i

The number of the resource instance.

Return Value

Returns the handle to the resource instance, or NULL if no corresponding resource instance exists.

CAtlBaseModule::GetModuleInstance

Returns the module instance from a `CAtlBaseModule` object.

```
HINSTANCE GetModuleInstance() throw();
```

Return Value

Returns the module instance.

CAtlBaseModule::GetResourceInstance

Returns the resource instance.

```
HINSTANCE GetResourceInstance() throw();
```

Return Value

Returns the resource instance.

CAtlBaseModule::m_bInitFailed

A variable that indicates if the module initialization has failed.

```
static bool m_bInitFailed;
```

Remarks

True if the module initialized, false if it failed to initialize.

CAtlBaseModule::RemoveResourceInstance

Removes a resource instance from the list of stored handles.

```
bool RemoveResourceInstance(HINSTANCE hInst) throw();
```

Parameters

hInst

The resource instance to remove.

Return Value

Returns true if the resource was successfully removed, false otherwise.

CAtlBaseModule::SetResourceInstance

Sets the resource instance of a `CAtlBaseModule` object.

```
HINSTANCE SetResourceInstance(HINSTANCE hInst) throw();
```

Parameters

hInst

The new resource instance.

Return Value

Returns the updated resource instance.

See also

[Class Overview](#)

CAtlComModule Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements a COM server module.

Syntax

```
class CAtlComModule : public _ATL_COM_MODULE
```

Members

Public Constructors

NAME	DESCRIPTION
CAtlComModule::CAtlComModule	The constructor.
CAtlComModule::~~CAtlComModule	The destructor.

Public Methods

NAME	DESCRIPTION
CAtlComModule::RegisterServer	Call this method to update the system registry for each object in the object map.
CAtlComModule::RegisterTypeLib	Call this method to register a type library.
CAtlComModule::UnregisterServer	Call this method to unregister each object in the object map.
CAtlComModule::UnRegisterTypeLib	Call this method to unregister a type library.

Remarks

`CAtlComModule` implements a COM server module, allowing a client to access the module's components.

This class replaces the obsolete [CComModule](#) class used in earlier versions of ATL. See [ATL Module Classes](#) for more details.

Inheritance Hierarchy

[_ATL_COM_MODULE](#)

`CAtlComModule`

Requirements

Header: `atlbase.h`

CAtlComModule::CAtlComModule

The constructor.

```
CAtlComModule() throw();
```

Remarks

Initializes the module.

CAtlComModule::~~CAtlComModule

The destructor.

```
~CAtlComModule();
```

Remarks

Frees all class factories.

CAtlComModule::RegisterServer

Call this method to update the system registry for each object in the object map.

```
HRESULT RegisterServer(BOOL bRegTypeLib = FALSE, const CLSID* pCLSID = NULL);
```

Parameters

bRegTypeLib

TRUE if the type library is to be registered. The default value is FALSE.

pCLSID

Points to the CLSID of the object to be registered. If NULL (the default value), all objects in the object map will be registered.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls the global function [AtlComModuleRegisterServer](#).

CAtlComModule::RegisterTypeLib

Call this method to register a type library.

```
HRESULT RegisterTypeLib(LPCTSTR lpszIndex);  
HRESULT RegisterTypeLib();
```

Parameters

lpszIndex

String in the format "\\N", where N is the integer index of the TYPELIB resource.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Adds information about a type library to the system registry. If the module instance contains multiple type libraries, use the first version of this method to specify which type library should be used.

CAtlComModule::UnregisterServer

Call this method to unregister each object in the object map.

```
HRESULT UnregisterServer(  
    BOOL bRegTypeLib = FALSE,  
    const CLSID* pCLSID = NULL);
```

Parameters

bRegTypeLib

TRUE if the type library is to be unregistered. The default value is FALSE.

pCLSID

Points to the CLSID of the object to be unregistered. If NULL (the default value), all objects in the object map will be unregistered.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls the global function [AtlComModuleUnregisterServer](#).

CAtlComModule::UnRegisterTypeLib

Call this method to unregister a type library.

```
HRESULT UnRegisterTypeLib(LPCTSTR lpszIndex);  
HRESULT UnRegisterTypeLib();
```

Parameters

lpszIndex

String in the format "\\N", where N is the integer index of the TYPELIB resource.

Remarks

Removes information about a type library from the system registry. If the module instance contains multiple type libraries, use the first version of this method to specify which type library should be used.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See also

[_ATL_COM_MODULE](#)

[Class Overview](#)

CAtlDebugInterfacesModule Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides support for debugging interfaces.

Syntax

```
class CAtlDebugInterfacesModule
```

Remarks

`CAtlDebugInterfacesModule` provides the support required for debugging interfaces. It is included in any project that defines the symbol `_ATL_DEBUG_QI`.

Requirements

Header: atlbase.h

See also

[Class Overview](#)
[Module Classes](#)

CAtDllModuleT Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class represents the module for a DLL.

Syntax

```
template <class T>
class ATL_NO_VTABLE CAtDllModuleT : public CAtlModuleT<T>
```

Parameters

T

Your class derived from `CAtDllModuleT`.

Members

Public Constructors

NAME	DESCRIPTION
CAtDllModuleT::CAtDllModuleT	The constructor.
CAtDllModuleT::~~CAtDllModuleT	The destructor.

Public Methods

NAME	DESCRIPTION
CAtDllModuleT::DllCanUnloadNow	Tests if the DLL can be unloaded.
CAtDllModuleT::DllGetClassObject	Returns a class factory.
CAtDllModuleT::DllMain	The optional entry point into a dynamic-link library (DLL).
CAtDllModuleT::DllRegisterServer	Adds entries to the system registry for objects in the DLL.
CAtDllModuleT::DllUnregisterServer	Removes entries in the system registry for objects in the DLL.
CAtDllModuleT::GetClassObject	Returns a class factory. Invoked by DllGetClassObject .

Remarks

`CAtDllModuleT` represents the module for a dynamic-link library (DLL) and provides functions used by all DLL projects. This specialization of [CAtlModuleT](#) class includes support for registration.

For more information on modules in ATL, see [ATL Module Classes](#).

Inheritance Hierarchy

[_ATL_MODULE](#)

[CAtlModule](#)

[CAtlModuleT](#)

`CAtlDllModuleT`

Requirements

Header: atlbase.h

CAtlDllModuleT::CAtlDllModuleT

The constructor.

```
CAtlDllModuleT() throw();
```

CAtlDllModuleT::~~CAtlDllModuleT

The destructor.

```
~CAtlDllModuleT() throw();
```

CAtlDllModuleT::DllCanUnloadNow

Tests if the DLL can be unloaded.

```
HRESULT DllCanUnloadNow() throw();
```

Return Value

Returns S_OK if the DLL can be unloaded, or S_FALSE if it cannot.

CAtlDllModuleT::DllGetClassObject

Returns the class factory.

```
HRESULT DllGetClassObject(  
    REFCLSID rclsid,  
    REFIID riid,  
    LPVOID* ppv) throw();
```

Parameters

rclsid

The CLSID of the object to be created.

riid

The IID of the requested interface.

ppv

A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppv* is set to NULL.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CAtlDllModuleT::DllMain

The optional entry point into a dynamic-link library (DLL).

```
BOOL WINAPI DllMain(DWORD dwReason, LPVOID /* lpReserved*/) throw();
```

Parameters

dwReason

If set to DLL_PROCESS_ATTACH, the DLL_THREAD_ATTACH and DLL_THREAD_DETACH notification calls are disabled.

lpReserved

Reserved.

Return Value

Always returns TRUE.

Remarks

Disabling the DLL_THREAD_ATTACH and DLL_THREAD_DETACH notification calls can be a useful optimization for multithreaded applications that have many DLLs, that frequently create and delete threads, and whose DLLs do not need these thread-level notifications of attachment/detachment.

CAtlDllModuleT::DllRegisterServer

Adds entries to the system registry for objects in the DLL.

```
HRESULT DllRegisterServer(BOOL bRegTypeLib = TRUE) throw();
```

Parameters

bRegTypeLib

TRUE if the type library is to be registered. The default value is TRUE.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CAtlDllModuleT::DllUnregisterServer

Removes entries in the system registry for objects in the DLL.

```
HRESULT DllUnregisterServer(BOOL bUnRegTypeLib = TRUE) throw();
```

Parameters

bUnRegTypeLib

TRUE if the type library is to be removed from the registry. The default value is TRUE.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CAtlDllModuleT::GetClassObject

Creates an object of the specified CLSID.

```
HRESULT GetClassObject(  
    REFCLSID rclsid,  
    REFIID riid,  
    LPVOID* ppv) throw();
```

Parameters

rclsid

The CLSID of the object to be created.

riid

The IID of the requested interface.

ppv

A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppv* is set to NULL.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method is called by [CAtdIIModuleT::DllGetClassObject](#) and is included for backward compatibility.

See also

[CAtdModuleT Class](#)

[CAtdExeModuleT Class](#)

[Class Overview](#)

[Module Classes](#)

CAtlException Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class defines an ATL exception.

Syntax

```
class CAtlException
```

Members

Public Constructors

NAME	DESCRIPTION
CAtlException::CAtlException	The constructor.

Public Operators

NAME	DESCRIPTION
CAtlException::operator HRESULT	Casts the current object to an HRESULT value.

Public Data Members

NAME	DESCRIPTION
CAtlException::m_hr	The variable of type HRESULT created by the object and used to store the error condition.

Remarks

A `CAtlException` object represents an exception condition related to an ATL operation. The `CAtlException` class includes a public data member that stores the status code indicating the reason for the exception and a cast operator that allows you to treat the exception as if it were an HRESULT.

In general, you will call `AtlThrow` rather than creating a `CAtlException` object directly.

Requirements

Header: atlexcept.h

CAtlException::CAtlException

The constructor.

```
CAtlException(HRESULT hr) throw();  
CAtlException() throw();
```

Parameters

hr

The HRESULT error code.

CAtlException::operator HRESULT

Casts the current object to an HRESULT value.

```
operator HRESULT() const throw ();
```

CAtlException::m_hr

The HRESULT data member.

```
HRESULT m_hr;
```

Remarks

The data member that stores the error condition. The HRESULT value is set by the constructor, [CAtlException::CAtlException](#).

See also

[AtlThrow](#)

[Class Overview](#)

CAtlExeModuleT Class

4/1/2019 • 5 minutes to read • [Edit Online](#)

This class represents the module for an application.

Syntax

```
template <class T>
class ATL_NO_VTABLE CAtlExeModuleT : public CAtlModuleT<T>
```

Parameters

T

Your class derived from `CAtlExeModuleT`.

Members

Public Constructors

NAME	DESCRIPTION
CAtlExeModuleT::CAtlExeModuleT	The constructor.
CAtlExeModuleT::~~CAtlExeModuleT	The destructor.

Public Methods

NAME	DESCRIPTION
CAtlExeModuleT::InitializeCom	Initializes COM.
CAtlExeModuleT::ParseCommandLine	Parses the command line and performs registration if necessary.
CAtlExeModuleT::PostMessageLoop	This method is called immediately after the message loop exits.
CAtlExeModuleT::PreMessageLoop	This method is called immediately before entering the message loop.
CAtlExeModuleT::RegisterClassObjects	Registers the class object.
CAtlExeModuleT::RevokeClassObjects	Revokes the class object.
CAtlExeModuleT::Run	This method executes code in the EXE module to initialize, run the message loop, and clean up.
CAtlExeModuleT::RunMessageLoop	This method executes the message loop.
CAtlExeModuleT::UninitializeCom	Uninitializes COM.

NAME	DESCRIPTION
CAtlExeModuleT::Unlock	Decrements the module's lock count.
CAtlExeModuleT::WinMain	This method implements the code required to run an EXE.

Public Data Members

NAME	DESCRIPTION
CAtlExeModuleT::m_bDelayShutdown	A flag indicating that there should be a delay shutting down the module.
CAtlExeModuleT::m_dwPause	A pause value used to ensure all objects are released before shutdown.
CAtlExeModuleT::m_dwTimeOut	A time-out value used to delay the unloading of the module.

Remarks

`CAtlExeModuleT` represents the module for an application (EXE) and contains code that supports creating an EXE, processing the command line, registering class objects, running the message loop, and cleaning up on exit.

This class is designed to improve performance when COM objects in the EXE server are continually created and destroyed. After the last COM object is released, the EXE waits for a duration specified by the [CAtlExeModuleT::m_dwTimeOut](#) data member. If there is no activity during this period (that is, no COM objects are created), the shutdown process is initiated.

The [CAtlExeModuleT::m_bDelayShutdown](#) data member is a flag used to determine if the EXE should use the mechanism defined above. If it is set to false, then the module will terminate immediately.

For more information on modules in ATL, see [ATL Module Classes](#).

Inheritance Hierarchy

[_ATL_MODULE](#)

[CAtlModule](#)

[CAtlModuleT](#)

`CAtlExeModuleT`

Requirements

Header: atlbase.h

CAtlExeModuleT::CAtlExeModuleT

The constructor.

```
CAtlExeModuleT() throw();
```

Remarks

If the EXE module could not be initialized, WinMain will immediately return without further processing.

CAtlExeModuleT::~~CAtlExeModuleT

The destructor.

```
~CAtlExeModuleT() throw();
```

Remarks

Frees all allocated resources.

CAtlExeModuleT::InitializeCom

Initializes COM.

```
static HRESULT InitializeCom() throw();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method is called from the constructor and can be overridden to initialize COM in a manner different from the default implementation. The default implementation either calls `CoInitializeEx(NULL, COINIT_MULTITHREADED)` or `CoInitialize(NULL)` depending on the project configuration.

Overriding this method normally requires overriding [CAtlExeModuleT::UninitializeCom](#).

CAtlExeModuleT::m_bDelayShutdown

A flag indicating that there should be a delay shutting down the module.

```
bool m_bDelayShutdown;
```

Remarks

See the [CAtlExeModuleT Overview](#) for details.

CAtlExeModuleT::m_dwPause

A pause value used to ensure all objects are gone before shutdown.

```
DWORD m_dwPause;
```

Remarks

Change this value after calling [CAtlExeModuleT::InitializeCom](#) to set the number of milliseconds used as the pause value for shutting down the server. The default value is 1000 milliseconds.

CAtlExeModuleT::m_dwTimeOut

A time-out value used to delay the unloading of the module.

```
DWORD m_dwTimeOut;
```

Remarks

Change this value after calling [CAtlExeModuleT::InitializeCom](#) to define the number of milliseconds used as the time-out value for shutting down the server. The default value is 5000 milliseconds. See the [CAtlExeModuleT Overview](#) for more details.

CAtlExeModuleT::ParseCommandLine

Parses the command line and performs registration if necessary.

```
bool ParseCommandLine(LPCTSTR lpCmdLine, HRESULT* pnRetCode) throw();
```

Parameters

lpCmdLine

The command line passed to the application.

pnRetCode

The HRESULT corresponding to the registration (if it took place).

Return Value

Return true if the application should continue to run, otherwise false.

Remarks

This method is called from [CAtlExeModuleT::WinMain](#) and can be overridden to handle command-line switches. The default implementation checks for **/RegServer** and **/UnRegServer** command-line arguments and performs registration or unregistration.

CAtlExeModuleT::PostMessageLoop

This method is called immediately after the message loop exits.

```
HRESULT PostMessageLoop() throw();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Override this method to perform custom application cleanup. The default implementation calls [CAtlExeModuleT::RevokeClassObjects](#).

CAtlExeModuleT::PreMessageLoop

This method is called immediately before entering the message loop.

```
HRESULT PreMessageLoop(int nShowCmd) throw();
```

Parameters

nShowCmd

The value passed as the *nShowCmd* parameter in WinMain.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Override this method to add custom initialization code for the application. The default implementation registers the class objects.

CAtlExeModuleT::RegisterClassObjects

Registers the class object with OLE so other applications can connect to it.

```
HRESULT RegisterClassObjects(DWORD dwClsContext, DWORD dwFlags) throw();
```

Parameters

dwClsContext

Specifies the context in which the class object is to be run. Possible values are CLSCTX_INPROC_SERVER, CLSCTX_INPROC_HANDLER, or CLSCTX_LOCAL_SERVER.

dwFlags

Determines the connection types to the class object. Possible values are REGCLS_SINGLEUSE, REGCLS_MULTIPLEUSE, or REGCLS_MULTI_SEPARATE.

Return Value

Returns S_OK on success, S_FALSE if there were no classes to register, or an error HRESULT on failure.

CAtlExeModuleT::RevokeClassObjects

Removes the class object.

```
HRESULT RevokeClassObjects() throw();
```

Return Value

Returns S_OK on success, S_FALSE if there were no classes to register, or an error HRESULT on failure.

CAtlExeModuleT::Run

This method executes code in the EXE module to initialize, run the message loop, and clean up.

```
HRESULT Run(int nShowCmd = SW_HIDE) throw();
```

Parameters

nShowCmd

Specifies how the window is to be shown. This parameter can be one of the values discussed in the [WinMain](#) section. Defaults to SW_HIDE.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method can be overridden. However, in practice it is better to override [CAtlExeModuleT::PreMessageLoop](#), [CAtlExeModuleT::RunMessageLoop](#), or [CAtlExeModuleT::PostMessageLoop](#) instead.

CAtlExeModuleT::RunMessageLoop

This method executes the message loop.

```
void RunMessageLoop() throw();
```

Remarks

This method can be overridden to change the behavior of the message loop.

CAtlExeModuleT::UninitializeCom

Uninitializes COM.

```
static void UninitializeCom() throw();
```

Remarks

By default this method simply calls [CoUninitialize](#) and is called from the destructor. Override this method if you override [CAtlExeModuleT::InitializeCom](#).

CAtlExeModuleT::Unlock

Decrements the module's lock count.

```
LONG Unlock() throw();
```

Return Value

Returns a value which may be useful for diagnostics or testing.

CAtlExeModuleT::WinMain

This method implements the code required to run an EXE.

```
int WinMain(int nShowCmd) throw();
```

Parameters

nShowCmd

Specifies how the window is to be shown. This parameter can be one of the values discussed in the [WinMain](#) section.

Return Value

Returns the executable's return value.

Remarks

This method can be overridden. If overriding [CAtlExeModuleT::PreMessageLoop](#), [CAtlExeModuleT::PostMessageLoop](#), or [CAtlExeModuleT::RunMessageLoop](#) doesn't provide enough flexibility, it's possible to override the `WinMain` function using this method.

See also

[ATLDuck Sample](#)

[CAtlModuleT Class](#)

[CAtlDllModuleT Class](#)

[Class Overview](#)

CAtlFile Class

4/1/2019 • 6 minutes to read • [Edit Online](#)

This class provides a thin wrapper around the Windows file-handling API.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CAtlFile : public CHandle
```

Members

Public Constructors

NAME	DESCRIPTION
CAtlFile::CAtlFile	The constructor.

Public Methods

NAME	DESCRIPTION
CAtlFile::Create	Call this method to create or open a file.
CAtlFile::Flush	Call this method to clear the buffers for the file and cause all buffered data to be written to the file.
CAtlFile::GetOverlappedResult	Call this method to get the results of an overlapped operation on the file.
CAtlFile::GetPosition	Call this method to get the current file pointer position from the file.
CAtlFile::GetSize	Call this method to get the size in bytes of the file.
CAtlFile::LockRange	Call this method to lock a region in the file to prevent other processes from accessing it.
CAtlFile::Read	Call this method to read data from a file starting at the position indicated by the file pointer.
CAtlFile::Seek	Call this method to move the file pointer of the file.
CAtlFile::SetSize	Call this method to set the size of the file.

NAME	DESCRIPTION
CAtlFile::UnlockRange	Call this method to unlock a region of the file.
CAtlFile::Write	Call this method to write data to the file starting at the position indicated by the file pointer.

Protected Data Members

NAME	DESCRIPTION
CAtlFile::m_pTM	Pointer to <code>CAtlTransactionManager</code> object

Remarks

Use this class when file-handling needs are relatively simple, but more abstraction than the Windows API provides is required, without including MFC dependencies.

Inheritance Hierarchy

[CHandle](#)

`CAtlFile`

Requirements

Header: atfile.h

CAtlFile::CAtlFile

The constructor.

```
CAtlFile() throw();
CAtlFile(CAtlTransactionManager* pTM = NULL) throw();
CAtlFile(CAtlFile& file) throw();
explicit CAtlFile(HANDLE hFile) throw();
```

Parameters

file

The file object.

hFile

The file handle.

pTM

Pointer to `CAtlTransactionManager` object

Remarks

The copy constructor transfers ownership of the file handle from the original `CAtlFile` object to the newly constructed object.

CAtlFile::Create

Call this method to create or open a file.

```
HRESULT Create(  
    LPCTSTR szFilename,  
    DWORD dwDesiredAccess,  
    DWORD dwShareMode,  
    DWORD dwCreationDisposition,  
    DWORD dwFlagsAndAttributes = FILE_ATTRIBUTE_NORMAL,  
    LPSECURITY_ATTRIBUTES lpsa = NULL,  
    HANDLE hTemplateFile = NULL) throw();
```

Parameters

szFilename

The file name.

dwDesiredAccess

The desired access. See *dwDesiredAccess* in [CreateFile](#) in the Windows SDK.

dwShareMode

The share mode. See *dwShareMode* in [CreateFile](#) .

dwCreationDisposition

The creation disposition. See *dwCreationDisposition* in [CreateFile](#) .

dwFlagsAndAttributes

The flags and attributes. See *dwFlagsAndAttributes* in [CreateFile](#) .

lpsa

The security attributes. See *lpSecurityAttributes* in [CreateFile](#) .

hTemplateFile

The template file. See *hTemplateFile* in [CreateFile](#) .

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls [CreateFile](#) to create or open the file.

CAtlFile::Flush

Call this method to clear the buffers for the file and cause all buffered data to be written to the file.

```
HRESULT Flush() throw();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls [FlushFileBuffers](#) to flush buffered data to the file.

CAtlFile::GetOverlappedResult

Call this method to get the results of an overlapped operation on the file.

```
HRESULT GetOverlappedResult(  
    LPOVERLAPPED pOverlapped,  
    DWORD& dwBytesTransferred,  
    BOOL bWait) throw();
```

Parameters

pOverlapped

The overlapped structure. See *lpOverlapped* in [GetOverlappedResult](#) in the Windows SDK.

dwBytesTransferred

The bytes transferred. See *lpNumberOfBytesTransferred* in `GetOverlappedResult`.

bWait

The wait option. See *bWait* in `GetOverlappedResult`.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls [GetOverlappedResult](#) to get the results of an overlapped operation on the file.

CAtlFile::GetPosition

Call this method to get the current file pointer position.

```
HRESULT GetPosition(ULONGLONG& nPos) const throw();
```

Parameters

nPos

The position in bytes.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls [SetFilePointer](#) to get the current file pointer position.

CAtlFile::GetSize

Call this method to get the size in bytes of the file.

```
HRESULT GetSize(ULONGLONG& nLen) const throw();
```

Parameters

nLen

The number of bytes in the file.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls [GetFileSize](#) to get the size in bytes of the file.

CAtlFile::LockRange

Call this method to lock a region in the file to prevent other processes from accessing it.

```
HRESULT LockRange(ULONGLONG nPos, ULONGLONG nCount) throw();
```

Parameters

nPos

The position in the file where the lock should begin.

nCount

The length of the byte range to be locked.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls [LockFile](#) to lock a region in the file. Locking bytes in a file prevents access to those bytes by other processes. You can lock more than one region of a file, but no overlapping regions are allowed. When you unlock a region, using [CAtlFile::UnlockRange](#), the byte range must correspond exactly to the region that was previously locked.

`LockRange` does not merge adjacent regions; if two locked regions are adjacent, you must unlock each separately.

CAtlFile::m_pTM

Pointer to a `CAtlTransactionManager` object.

```
CAtlTransactionManager* m_pTM;
```

Remarks

CAtlFile::Read

Call this method to read data from a file starting at the position indicated by the file pointer.

```
HRESULT Read(
    LPVOID pBuffer,
    DWORD nBufSize) throw();

HRESULT Read(
    LPVOID pBuffer,
    DWORD nBufSize,
    DWORD& nBytesRead) throw();

HRESULT Read(
    LPVOID pBuffer,
    DWORD nBufSize,
    LPOVERLAPPED pOverlapped) throw();

HRESULT Read(
    LPVOID pBuffer,
    DWORD nBufSize,
    LPOVERLAPPED pOverlapped,
    LPOVERLAPPED_COMPLETION_ROUTINE pfnCompletionRoutine) throw();
```

Parameters

pBuffer

Pointer to the buffer that will receive the data read from the file.

nBufSize

The buffer size in bytes.

nBytesRead

The number of bytes read.

pOverlapped

The overlapped structure. See *lpOverlapped* in [ReadFile](#) in the Windows SDK.

pfnCompletionRoutine

The completion routine. See *lpCompletionRoutine* in [ReadFileEx](#) in the Windows SDK.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The first three forms call [ReadFile](#), the last [ReadFileEx](#) to read data from the file. Use [CAtlFile::Seek](#) to move the file pointer.

CAtlFile::Seek

Call this method to move the file pointer of the file.

```
HRESULT Seek(  
    LONGLONG nOffset,  
    DWORD dwFrom = FILE_CURRENT) throw();
```

Parameters

nOffset

The offset from the starting point given by *dwFrom*.

dwFrom

The starting point (FILE_BEGIN, FILE_CURRENT, or FILE_END).

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls [SetFilePointer](#) to move the file pointer.

CAtlFile::SetSize

Call this method to set the size of the file.

```
HRESULT SetSize(ULONGLONG nNewLen) throw();
```

Parameters

nNewLen

The new length of the file in bytes.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls [SetFilePointer](#) and [SetEndOfFile](#) to set the size of the file. On return, the file pointer is positioned at the end of the file.

CAtlFile::UnlockRange

Call this method to unlock a region of the file.

```
HRESULT UnlockRange(ULONGLONG nPos, ULONGLONG nCount) throw();
```

Parameters

nPos

The position in the file where the unlock should begin.

nCount

The length of the byte range to be unlocked.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls [UnlockFile](#) to unlock a region of the file.

CAtlFile::Write

Call this method to write data to the file starting at the position indicated by the file pointer.

```
HRESULT Write(
    LPCVOID pBuffer,
    DWORD nBufSize,
    LPOVERLAPPED pOverlapped,
    LPOVERLAPPED_COMPLETION_ROUTINE pfnCompletionRoutine) throw();

HRESULT Write(
    LPCVOID pBuffer,
    DWORD nBufSize,
    DWORD* pnBytesWritten = NULL) throw();

HRESULT Write(
    LPCVOID pBuffer,
    DWORD nBufSize,
    LPOVERLAPPED pOverlapped) throw();
```

Parameters

pBuffer

The buffer containing the data to be written to the file.

nBufSize

The number of bytes to be transferred from the buffer.

pOverlapped

The overlapped structure. See *lpOverlapped* in [WriteFile](#) in the Windows SDK.

pfnCompletionRoutine

The completion routine. See *lpCompletionRoutine* in [WriteFileEx](#) in the Windows SDK.

pnBytesWritten

The bytes written.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The first three forms call [WriteFile](#), the last calls [WriteFileEx](#) to write data to the file. Use [CAtlFile::Seek](#) to move the file pointer.

See also

[Marquee Sample](#)

[Class Overview](#)

[CHandle Class](#)

CAtlFileMapping Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class represents a memory-mapped file, adding a cast operator to the methods of [CAtlFileMappingBase](#).

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <typename T = char>
class CAtlFileMapping : public CAtlFileMappingBase
```

Parameters

T

The type of data used for the cast operator.

Members

Public Operators

NAME	DESCRIPTION
CAtlFileMapping::operator T*	Allows implicit conversion of <code>CAtlFileMapping</code> objects to <code>T*</code> .

Remarks

This class adds a single cast operator to allow implicit conversion of `CAtlFileMapping` objects to `T*`. Other members are supplied by the base class, [CAtlFileMappingBase](#).

Inheritance Hierarchy

[CAtlFileMappingBase](#)

`CAtlFileMapping`

Requirements

Header: atlfile.h

CAtlFileMapping::operator T*

Allows implicit conversion of `CAtlFileMapping` objects to `T*`.

```
operator T*() const throw();
```

Return Value

Returns a `T*` pointer to the start of the memory-mapped file.

Remarks

Calls [CAtlFileMappingBase::GetData](#) and reinterprets the returned pointer as a `T*` where T is the type used as the template parameter of this class.

See also

[CAtlFileMappingBase Class](#)

[Class Overview](#)

CAtlFileMappingBase Class

3/4/2019 • 5 minutes to read • [Edit Online](#)

This class represents a memory-mapped file.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CAtlFileMappingBase
```

Members

Public Constructors

NAME	DESCRIPTION
CAtlFileMappingBase::CAtlFileMappingBase	The constructor.
CAtlFileMappingBase::~~CAtlFileMappingBase	The destructor.

Public Methods

NAME	DESCRIPTION
CAtlFileMappingBase::CopyFrom	Call this method to copy from a file-mapping object.
CAtlFileMappingBase::GetData	Call this method to get the data from a file-mapping object.
CAtlFileMappingBase::GetHandle	Call this method to return the file handle.
CAtlFileMappingBase::GetMappingSize	Call this method to get the mapping size from a file-mapping object.
CAtlFileMappingBase::MapFile	Call this method to create a file-mapping object.
CAtlFileMappingBase::MapSharedMem	Call this method to create a file-mapping object that permits full access to all processes.
CAtlFileMappingBase::OpenMapping	Call this method to return a handle to the file-mapping object.
CAtlFileMappingBase::Unmap	Call this method to unmap a file-mapping object.

Public Operators

NAME	DESCRIPTION
CAtlFileMappingBase::operator =	Sets the current file-mapping object to another file-mapping object.

Remarks

File mapping is the association of a file's contents with a portion of the virtual address space of a process. This class provides methods for creating file-mapping objects that permit programs to easily access and share data.

For more information, see [File Mapping](#) in the Windows SDK.

Requirements

Header: `atlfile.h`

CAtlFileMappingBase::CAtlFileMappingBase

The constructor.

```
CAtlFileMappingBase(CAtlFileMappingBase& orig);
CAtlFileMappingBase() throw();
```

Parameters

orig

The original file-mapping object to copy to create the new object.

Remarks

Creates a new file-mapping object, optionally using an existing object. It is still necessary to call [CAtlFileMappingBase::MapFile](#) to open or create the file-mapping object for a particular file.

Example

```

int OpenMyFileMap()
{
    // Create the file-mapping object.
    CAtlFileMappingBase myFileMap;

    // Create a file.
    CAtlFile myFile;
    myFile.Create(_T("myMapTestFile"),
        GENERIC_READ|GENERIC_WRITE|STANDARD_RIGHTS_ALL,
        FILE_SHARE_READ|FILE_SHARE_WRITE,
        OPEN_ALWAYS);

    // The file handle.
    HANDLE hFile = (HANDLE)myFile;

    // Test the file has opened successfully.
    ATLASSERT(hFile != INVALID_HANDLE_VALUE);

    // Open the file for file-mapping.
    // Must give a size as the file is zero by default.
    if (myFileMap.MapFile(hFile,
        1024,
        0,
        PAGE_READWRITE,
        FILE_MAP_READ) != S_OK)
    {
        CloseHandle(hFile);
        return 0;
    }

    // Confirm the size of the mapping file.
    ATLASSERT(myFileMap.GetMappingSize() == 1024);

    // Now the file-mapping object is open, a second
    // process could access the filemap object to exchange
    // data.

    return 0;
}

```

CAtlFileMappingBase::~~CAtlFileMappingBase

The destructor.

```
~CAtlFileMappingBase() throw();
```

Remarks

Frees any resources allocated by the class and calls the [CAtlFileMappingBase::Unmap](#) method.

CAtlFileMappingBase::CopyFrom

Call this method to copy from a file-mapping object.

```
HRESULT CopyFrom(CAtlFileMappingBase& orig) throw();
```

Parameters

orig

The original file-mapping object to copy from.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CAtlFileMappingBase::GetData

Call this method to get the data from a file-mapping object.

```
void* GetData() const throw();
```

Return Value

Returns a pointer to the data.

CAtlFileMappingBase::GetHandle

Call this method to return a handle to the file-mapping object.

```
HANDLE GetHandle() throw ();
```

Return Value

Returns a handle to the file-mapping object.

CAtlFileMappingBase::GetMappingSize

Call this method to get the mapping size from a file-mapping object.

```
SIZE_T GetMappingSize() throw();
```

Return Value

Returns the mapping size.

Example

See the example for [CAtlFileMappingBase::CAtlFileMappingBase](#).

CAtlFileMappingBase::MapFile

Call this method to open or create a file-mapping object for the specified file.

```
HRESULT MapFile(  
    HANDLE hFile,  
    SIZE_T nMappingSize = 0,  
    ULONGLONG nOffset = 0,  
    DWORD dwMappingProtection = PAGE_READONLY,  
    DWORD dwViewDesiredAccess = FILE_MAP_READ) throw();
```

Parameters

hFile

Handle to the file from which to create a mapping object. *hFile* must be valid and cannot be set to INVALID_HANDLE_VALUE.

nMappingSize

The mapping size. If 0, the maximum size of the file-mapping object is equal to the current size of the file identified by *hFile*.

nOffset

The file offset where mapping is to begin. The offset value must be a multiple of the system's memory allocation granularity.

dwMappingProtection

The protection desired for the file view when the file is mapped. See *flProtect* in [CreateFileMapping](#) in the Windows SDK.

dwViewDesiredAccess

Specifies the type of access to the file view and, therefore, the protection of the pages mapped by the file. See *dwDesiredAccess* in [MapViewOfFileEx](#) in the Windows SDK.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

After a file-mapping object has been created, the size of the file must not exceed the size of the file-mapping object; if it does, not all of the file's contents will be available for sharing. For more details, see [CreateFileMapping](#) and [MapViewOfFileEx](#) in the Windows SDK.

Example

See the example for [CAtlFileMappingBase::CAtlFileMappingBase](#).

CAtlFileMappingBase::MapSharedMem

Call this method to create a file-mapping object that permits full access to all processes.

```
HRESULT MapSharedMem(  
    SIZE_T nMappingSize,  
    LPCTSTR szName,  
    BOOL* pbAlreadyExisted = NULL,  
    LPSECURITY_ATTRIBUTES lpsa = NULL,  
    DWORD dwMappingProtection = PAGE_READWRITE,  
    DWORD dwViewDesiredAccess = FILE_MAP_ALL_ACCESS) throw();
```

Parameters

nMappingSize

The mapping size. If 0, the maximum size of the file-mapping object is equal to the current size of the file-mapping object identified by *szName*.

szName

The name of the mapping object.

pbAlreadyExisted

Points to a BOOL value that is set to TRUE if the mapping object already existed.

lpsa

The pointer to a `SECURITY_ATTRIBUTES` structure that determines whether the returned handle can be inherited by child processes. See *lpAttributes* in [CreateFileMapping](#) in the Windows SDK.

dwMappingProtection

The protection desired for the file view, when the file is mapped. See *flProtect* in `CreateFileMapping` in the Windows SDK.

dwViewDesiredAccess

Specifies the type of access to the file view and, therefore, the protection of the pages mapped by the file. See *dwDesiredAccess* in [MapViewOfFileEx](#) in the Windows SDK.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

`MapShareMem` allows an existing file-mapping object, created by [CreateFileMapping](#), to be shared between processes.

CAtlFileMappingBase::OpenMapping

Call this method to open a named file-mapping object for the specified file.

```
HRESULT OpenMapping(  
    LPCTSTR szName,  
    SIZE_T nMappingSize,  
    ULONGLONG nOffset = 0,  
    DWORD dwViewDesiredAccess = FILE_MAP_ALL_ACCESS) throw();
```

Parameters

szName

The name of the mapping object. If there is an open handle to a file-mapping object by this name and the security descriptor on the mapping object does not conflict with the *dwViewDesiredAccess* parameter, the open operation succeeds.

nMappingSize

The mapping size. If 0, the maximum size of the file-mapping object is equal to the current size of the file-mapping object identified by *szName*.

nOffset

The file offset where mapping is to begin. The offset value must be a multiple of the system's memory allocation granularity.

dwViewDesiredAccess

Specifies the type of access to the file view and, therefore, the protection of the pages mapped by the file. See *dwDesiredAccess* in [MapViewOfFileEx](#) in the Windows SDK.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

In debug builds, an assertion error will occur if the input parameters are invalid.

CAtlFileMappingBase::operator =

Sets the current file-mapping object to another file-mapping object.

```
CAtlFileMappingBase& operator=(CAtlFileMappingBase& orig);
```

Parameters

orig

The current file-mapping object.

Return Value

Returns a reference to the current object.

CAtlFileMappingBase::Unmap

Call this method to unmap a file-mapping object.

```
HRESULT Unmap() throw();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

See [UnmapViewOfFile](#) in the Windows SDK for more details.

See also

[CAtlFileMapping Class](#)

[Class Overview](#)

CatList Class

3/4/2019 • 18 minutes to read • [Edit Online](#)

This class provides methods for creating and managing a list object.

Syntax

```
template<typename E, class ETraits = CElementTraits<E>>
class CatList
```

Parameters

E

The element type.

ETraits

The code used to copy or move elements. See [CElementTraits Class](#) for more details.

Members

Public Typedefs

NAME	DESCRIPTION
CatList::INARGTYPE	

Public Constructors

NAME	DESCRIPTION
CatList::CatList	The constructor.
CatList::~~CatList	The destructor.

Public Methods

NAME	DESCRIPTION
CatList::AddHead	Call this method to add an element to the head of the list.
CatList::AddHeadList	Call this method to add an existing list to the head of the list.
CatList::AddTail	Call this method to add an element to the tail of this list.
CatList::AddTailList	Call this method to add an existing list to the tail of this list.
CatList::AssertValid	Call this method to confirm the list is valid.
CatList::Find	Call this method to search the list for the specified element.

NAME	DESCRIPTION
CAtlList::FindIndex	Call this method to obtain the position of an element, given an index value.
CAtlList::GetAt	Call this method to return the element at a specified position in the list.
CAtlList::GetCount	Call this method to return the number of objects in the list.
CAtlList::GetHead	Call this method to return the element at the head of the list.
CAtlList::GetHeadPosition	Call this method to obtain the position of the head of the list.
CAtlList::GetNext	Call this method to return the next element from the list.
CAtlList::GetPrev	Call this method to return the previous element from the list.
CAtlList::GetTail	Call this method to return the element at the tail of the list.
CAtlList::GetTailPosition	Call this method to obtain the position of the tail of the list.
CAtlList::InsertAfter	Call this method to insert a new element into the list after the specified position.
CAtlList::InsertBefore	Call this method to insert a new element into the list before the specified position.
CAtlList::IsEmpty	Call this method to determine if the list is empty.
CAtlList::MoveToHead	Call this method to move the specified element to the head of the list.
CAtlList::MoveToTail	Call this method to move the specified element to the tail of the list.
CAtlList::RemoveAll	Call this method to remove all of the elements from the list.
CAtlList::RemoveAt	Call this method to remove a single element from the list.
CAtlList::RemoveHead	Call this method to remove the element at the head of the list.
CAtlList::RemoveHeadNoReturn	Call this method to remove the element at the head of the list without returning a value.
CAtlList::RemoveTail	Call this method to remove the element at the tail of the list.
CAtlList::RemoveTailNoReturn	Call this method to remove the element at the tail of the list without returning a value.
CAtlList::SetAt	Call this method to set the value of the element at a given position in the list.

NAME	DESCRIPTION
CAtlList::SwapElements	Call this method to swap elements in the list.

Remarks

The `CAtlList` class supports ordered lists of nonunique objects accessible sequentially or by value. `CAtlList` lists behave like doubly linked lists. Each list has a head and a tail, and new elements (or lists in some cases) can be added to either end of the list, or inserted before or after specific elements.

Most of the `CAtlList` methods make use of a position value. This value is used by the methods to reference the actual memory location where the elements are stored, and should not be calculated or predicted directly. If it is necessary to access the n th element in the list, the method [CAtlList::FindIndex](#) will return the corresponding position value for a given index. The methods [CAtlList::GetNext](#) and [CAtlList::GetPrev](#) can be used to iterate through the objects in the list.

For more information regarding the collection classes available with ATL, see [ATL Collection Classes](#).

Requirements

Header: atlcoll.h

CAtlList::AddHead

Call this method to add an element to the head of the list.

```
POSITION AddHead();
POSITION AddHead(INARGTYPE element);
```

Parameters

element

The new element.

Return Value

Returns the position of the newly added element.

Remarks

If the first version is used, an empty element is created using its default constructor, rather than its copy constructor.

Example

```
// Declare a list of integers
CATlList<int> myList;

// Add some elements, each to the head of the list.
// As each new element is added, the previous head is
// pushed down the list.
myList.AddHead(42);
myList.AddHead(49);

// Confirm the value currently at the head of the list
ATLASSERT(myList.GetHead() == 49);

// Confirm the value currently at the tail of the list
ATLASSERT(myList.GetTail() == 42);
```

CATlList::AddHeadList

Call this method to add an existing list to the head of the list.

```
void AddHeadList(const CATlList<E, ETraits>* pNew);
```

Parameters

pNew

The list to be added.

Remarks

The list pointed to by *pNew* is inserted at the start of the existing list. In debug builds, an assertion failure will occur if *pNew* is equal to NULL.

Example

```
// Define two lists of integers
CATlList<int> myList1;
CATlList<int> myList2;

// Fill up the first list
myList1.AddTail(1);
myList1.AddTail(2);
myList1.AddTail(3);

// Add an element to the second list
myList2.AddTail(4);

// Insert the first list into the second
myList2.AddHeadList(&myList1);

// The second list now contains:
// 1, 2, 3, 4
```

CATlList::AddTail

Call this method to add an element to the tail of this list.

```
POSITION AddTail();
POSITION AddTail(INARGTYPE element);
```

Parameters

element

The element to add.

Return Value

Returns the POSITION of the newly added element.

Remarks

If the first version is used, an empty element is created using its default constructor, rather than its copy constructor. The element is added to the end of the list, and so it now becomes the tail. This method can be used with an empty list.

Example

```
// Define the list
CATList<int> myList;

// Add elements to the tail
myList.AddTail(1);
myList.AddTail(2);
myList.AddTail(3);

// Confirm the current head of the list
ATLASSERT(myList.GetHead() == 1);

// Confirm the current tail of the list
ATLASSERT(myList.GetTail() == 3);
```

CATList::AddTailList

Call this method to add an existing list to the tail of this list.

```
void AddTailList(const CATList<E, ETraits>* pNew);
```

Parameters

pNew

The list to be added.

Remarks

The list pointed to by *pNew* is inserted after the last element (if any) in the list object. The last element in the *pNew* list therefore becomes the tail. In debug builds, an assertion failure will occur if *pNew* is equal to NULL.

Example

```
// Define two integer lists
CATlList<int> myList1;
CATlList<int> myList2;

// Fill up the first list
myList1.AddTail(1);
myList1.AddTail(2);
myList1.AddTail(3);

// Add an element to the second list
myList2.AddTail(4);

// Insert the first list into the second
myList2.AddTailList(&myList1);

// The second list now contains:
// 4, 1, 2, 3
```

CATlList::AssertValid

Call this method to confirm the list is valid.

```
void AssertValid() const;
```

Remarks

In debug builds, an assertion failure will occur if the list object is not valid. To be valid, an empty list must have both the head and tail pointing to NULL, and a list that is not empty must have both the head and tail pointing to valid addresses.

Example

```
// Define the list
CATlList<int> myList;

// AssertValid only exists in debug builds
#ifdef _DEBUG
myList.AssertValid();
#endif
```

CATlList::CATlList

The constructor.

```
CATlList(UINT nBlockSize = 10) throw();
```

Parameters

nBlockSize

The block size.

Remarks

The constructor for the `CATlList` object. The block size is a measure of the amount of memory allocated when a new element is required. Larger block sizes reduce calls to memory allocation routines, but use more resources.

Example

```
// Define two lists
CATlList<int> myList1;
CATlList<double> myList2;
```

CATlList::~~CATlList

The destructor.

```
~CATlList() throw();
```

Remarks

Frees all allocated resources, including a call to [CATlList::RemoveAll](#) to remove all elements from the list.

In debug builds, an assertion failure will occur if the list still contains some elements after the call to `RemoveAll`.

CATlList::Find

Call this method to search the list for the specified element.

```
POSITION Find(INARGTYPE element, POSITION posStartAfter = NULL) const throw();
```

Parameters

element

The element to be found in the list.

posStartAfter

The start position for the search. If no value is specified, the search begins with the head element.

Return Value

Returns the POSITION value of the element if found, otherwise returns NULL.

Remarks

In debug builds, an assertion failure will occur if the list object is not valid, or if the *posStartAfter* value is out of range.

Example

```
// Define the integer list
CATlList<int> myList;

// Populate the list
myList.AddTail(100);
myList.AddTail(200);
myList.AddTail(300);
myList.AddTail(400);

// Find the '300' element in the list,
// starting from the list head.
POSITION myPos = myList.Find(300);

// Confirm that the element was found
ATLASSERT(myList.GetAt(myPos) == 300);
```

CATlList::FindIndex

Call this method to obtain the position of an element, given an index value.

```
POSITION FindIndex(size_t iElement) const throw();
```

Parameters

iElement

The zero-based index of the required list element.

Return Value

Returns the corresponding POSITION value, or NULL if *iElement* is out of range.

Remarks

This method returns the POSITION corresponding to a given index value, allowing access to the *n*th element in the list.

In debug builds, an assertion failure will occur if the list object is not valid.

Example

```
// Define the integer list
CATList<int> myList;

// Populate the list
for (int i = 0; i < 100; i++)
{
    myList.AddTail(i);
}

// Iterate through the entire list
for (size_t j = 0; j < myList.GetCount(); j++)
{
    size_t i = myList.GetAt(myList.FindIndex(j));
    ATLASSERT(i == j);
}
```

CATList::GetAt

Call this method to return the element at a specified position in the list.

```
E& GetAt(POSITION pos) throw();
const E& GetAt(POSITION pos) const throw();
```

Parameters

pos

The POSITION value specifying a particular element.

Return Value

A reference to, or copy of, the element.

Remarks

If the list is **const**, `GetAt` returns a copy of the element. This allows the method to be used only on the right side of an assignment statement and protects the list from modification.

If the list is not **const**, `GetAt` returns a reference to the element. This allows the method to be used on either side of an assignment statement and thus allows the list entries to be modified.

In debug builds, an assertion failure will occur if *pos* is equal to NULL.

Example

See the example for [CAtlList::FindIndex](#).

CAtlList::GetCount

Call this method to return the number of objects in the list.

```
size_t GetCount() const throw();
```

Return Value

Returns the number of elements in the list.

Example

See the example for [CAtlList::Find](#).

CAtlList::GetHead

Call this method to return the element at the head of the list.

```
E& GetHead() throw();  
const E& GetHead() const throw();
```

Return Value

Returns a reference to, or a copy of, the element at the head of the list.

Remarks

If the list is **const**, `GetHead` returns a copy of the element at the head of the list. This allows the method to be used only on the right side of an assignment statement and protects the list from modification.

If the list is not **const**, `GetHead` returns a reference to the element at the head of the list. This allows the method to be used on either side of an assignment statement and thus allows the list entries to be modified.

In debug builds, an assertion failure will occur if the head of the list points to NULL.

Example

See the example for [CAtlList::AddHead](#).

CAtlList::GetHeadPosition

Call this method to obtain the position of the head of the list.

```
POSITION GetHeadPosition() const throw();
```

Return Value

Returns the POSITION value corresponding to the element at the head of the list.

Remarks

If the list is empty, the value returned is NULL.

Example

```
// Define the integer list
CAtlList<int> myList;
int i;

// Populate the list
for (i = 0; i < 100; i++)
{
    myList.AddTail(i);
}

// Get the starting position value
POSITION myPos = myList.GetHeadPosition();

// Iterate through the entire list
i = 0;
int j;

do {
    j = myList.GetNext(myPos);
    ATLASSERT(i == j);
    i++;
} while (myPos != NULL);
```

CAtlList::GetNext

Call this method to return the next element from the list.

```
E& GetNext(POSITION& pos) throw();
const E& GetNext(POSITION& pos) const throw();
```

Parameters

pos

A POSITION value, returned by a previous call to `GetNext`, [CAtlList::GetHeadPosition](#), or other `CAtlList` method.

Return Value

If the list is **const**, `GetNext` returns a copy of the next element of the list. This allows the method to be used only on the right side of an assignment statement and protects the list from modification.

If the list is not **const**, `GetNext` returns a reference to the next element of the list. This allows the method to be used on either side of an assignment statement and thus allows the list entries to be modified.

Remarks

The POSITION counter, *pos*, is updated to point to the next element in the list, or NULL if there are no more elements. In debug builds, an assertion failure will occur if *pos* is equal to NULL.

Example

See the example for [CAtlList::GetHeadPosition](#).

CAtlList::GetPrev

Call this method to return the previous element from the list.

```
E& GetPrev(POSITION& pos) throw();
const E& GetPrev(POSITION& pos) const throw();
```

Parameters

pos

A POSITION value, returned by a previous call to `GetPrev`, [CAtlList::GetTailPosition](#), or other `CAtlList` method.

Return Value

If the list is **const**, `GetPrev` returns a copy of an element of the list. This allows the method to be used only on the right side of an assignment statement and protects the list from modification.

If the list is not **const**, `GetPrev` returns a reference to an element of the list. This allows the method to be used on either side of an assignment statement and thus allows the list entries to be modified.

Remarks

The POSITION counter, *pos*, is updated to point to the previous element in the list, or NULL if there are no more elements. In debug builds, an assertion failure will occur if *pos* is equal to NULL.

Example

See the example for [CAtlList::GetTailPosition](#).

CAtlList::GetTail

Call this method to return the element at the tail of the list.

```
E& GetTail() throw();  
const E& GetTail() const throw();
```

Return Value

Returns a reference to, or a copy of, the element at the tail of the list.

Remarks

If the list is **const**, `GetTail` returns a copy of the element at the head of the list. This allows the method to be used only on the right side of an assignment statement and protects the list from modification.

If the list is not **const**, `GetTail` returns a reference to the element at the head of the list. This allows the method to be used on either side of an assignment statement and thus allows the list entries to be modified.

In debug builds, an assertion failure will occur if the tail of the list points to NULL.

Example

See the example for [CAtlList::AddTail](#).

CAtlList::GetTailPosition

Call this method to obtain the position of the tail of the list.

```
POSITION GetTailPosition() const throw();
```

Return Value

Returns the POSITION value corresponding to the element at the tail of the list.

Remarks

If the list is empty, the value returned is NULL.

Example

```

// Define the integer list
CAtlList<int> myList;
int i;

// Populate the list
for (i = 0; i < 100; i++)
{
    myList.AddHead(i);
}

// Get the starting position value
POSITION myP = myList.GetTailPosition();

// Iterate through the entire list
i = 0;
int j;

do {
    j = myList.GetPrev(myP);
    ATLASSERT(i == j);
    i++;
} while (myP != NULL);

```

CAtlList::INARGTYPE

Type used when an element is passed as an input argument.

```
typedef ETraits::INARGTYPE INARGTYPE;
```

CAtlList::InsertAfter

Call this method to insert a new element into the list after the specified position.

```
POSITION InsertAfter(POSITION pos, INARGTYPE element);
```

Parameters

pos

The POSITION value after which the new element will be inserted.

element

The element to be inserted.

Return Value

Returns the POSITION value of the new element.

Remarks

In debug builds, an assertion failure will occur if the list isn't valid, if the insert fails, or if an attempt is made to insert the element after the tail.

Example

```
// Define the integer list
CATlList<int> myList;

// Populate the list
POSITION myPos = myList.AddHead(1);
myPos = myList.InsertAfter(myPos, 2);
myPos = myList.InsertAfter(myPos, 3);

// Confirm the tail value is as expected
ATLASSERT(myList.GetTail() == 3);
```

CATlList::InsertBefore

Call this method to insert a new element into the list before the specified position.

```
POSITION InsertBefore(POSITION pos, INARGTYPE element);
```

Parameters

pos

The new element will be inserted into the list before this POSITION value.

element

The element to be inserted.

Return Value

Returns the POSITION value of the new element.

Remarks

In debug builds, an assertion failure will occur if the list isn't valid, if the insert fails, or if an attempt is made to insert the element before the head.

Example

```
// Define the integer list
CATlList<int> myList;

// Populate the list
POSITION myPos = myList.AddHead(1);
myPos = myList.InsertBefore(myPos, 2);
myPos = myList.InsertBefore(myPos, 3);

// Confirm the head value is as expected
ATLASSERT(myList.GetHead() == 3);
```

CATlList::IsEmpty

Call this method to determine if the list is empty.

```
bool IsEmpty() const throw();
```

Return Value

Returns true if the list contains no objects, otherwise false.

Example

```
// Define the integer list
CATList<int> myList;

// Populate the list
myList.AddTail(1);
myList.AddTail(2);
myList.AddTail(3);
myList.AddTail(4);

// Confirm not empty
ATLASSERT(myList.IsEmpty() == false);

// Remove the tail element
myList.RemoveTailNoReturn();

// Confirm not empty
ATLASSERT(myList.IsEmpty() == false);

// Remove the head element
myList.RemoveHeadNoReturn();

// Confirm not empty
ATLASSERT(myList.IsEmpty() == false);

// Remove all remaining elements
myList.RemoveAll();

// Confirm empty
ATLASSERT(myList.IsEmpty() == true);
```

CATList::MoveToHead

Call this method to move the specified element to the head of the list.

```
void MoveToHead(POSITION pos) throw();
```

Parameters

pos

The POSITION value of the element to move.

Remarks

The specified element is moved from its current position to the head of the list. In debug builds, an assertion failure will occur if *pos* is equal to NULL.

Example

```
// Define the integer list
CAtlList<int> myList;

// Populate the list
myList.AddTail(1);
myList.AddTail(2);
myList.AddTail(3);
myList.AddTail(4);

// Move the tail element to the head
myList.MoveToHead(myList.GetTailPosition());

// Confirm the head is as expected
ATLASSERT(myList.GetHead() == 4);

// Move the head element to the tail
myList.MoveToTail(myList.GetHeadPosition());

// Confirm the tail is as expected
ATLASSERT(myList.GetTail() == 4);
```

CAtlList::MoveToTail

Call this method to move the specified element to the tail of the list.

```
void MoveToTail(POSITION pos) throw();
```

Parameters

pos

The POSITION value of the element to move.

Remarks

The specified element is moved from its current position to the tail of the list. In debug builds, an assertion failure will occur if *pos* is equal to NULL.

Example

See the example for [CAtlList::MoveToHead](#).

CAtlList::RemoveAll

Call this method to remove all of the elements from the list.

```
void RemoveAll() throw();
```

Remarks

This method removes all of the elements from the list and frees the allocated memory. In debugs builds, an ATLASSERT will be raised if all elements aren't deleted or if the list structure has become corrupted.

Example

See the example for [CAtlList::IsEmpty](#).

CAtlList::RemoveAt

Call this method to remove a single element from the list.


```
void RemoveAt(POSITION pos) throw();
```

Parameters

pos

The POSITION value of the element to remove.

Remarks

The element referenced by *pos* is removed, and memory is freed. It is acceptable to use `RemoveAt` to remove the head or tail of the list.

In debug builds, an assertion failure will occur if the list is not valid or if removing the element causes the list to access memory which isn't part of the list structure.

Example

```
// Define the integer list
CATList<int> myList;

// Populate the list
myList.AddTail(100);
myList.AddTail(200);
myList.AddTail(300);

// Use RemoveAt to remove elements one by one
myList.RemoveAt(myList.Find(100));
myList.RemoveAt(myList.Find(200));
myList.RemoveAt(myList.Find(300));

// Confirm all have been deleted
ATLASSERT(myList.IsEmpty() == true);
```

CATList::RemoveHead

Call this method to remove the element at the head of the list.

```
E RemoveHead();
```

Return Value

Returns the element at the head of the list.

Remarks

The head element is deleted from the list, and memory is freed. A copy of the element is returned. In debug builds, an assertion failure will occur if the list is empty.

Example

```
// Define the integer list
CAtlList<int> myList;

// Populate the list
myList.AddTail(100);
myList.AddTail(200);
myList.AddTail(300);

// Confirm the head of the list
ATLASSERT(myList.GetHead() == 100);

// Remove the head of the list
ATLASSERT(myList.RemoveHead() == 100);

// Confirm the new head of the list
ATLASSERT(myList.GetHead() == 200);
```

CAtlList::RemoveHeadNoReturn

Call this method to remove the element at the head of the list without returning a value.

```
void RemoveHeadNoReturn() throw();
```

Remarks

The head element is deleted from the list, and memory is freed. In debug builds, an assertion failure will occur if the list is empty.

Example

See the example for [CAtlList::IsEmpty](#).

CAtlList::RemoveTail

Call this method to remove the element at the tail of the list.

```
E RemoveTail();
```

Return Value

Returns the element at the tail of the list.

Remarks

The tail element is deleted from the list, and memory is freed. A copy of the element is returned. In debug builds, an assertion failure will occur if the list is empty.

Example

```
// Define the integer list
CAtlList<int> myList;

// Populate the list
myList.AddTail(100);
myList.AddTail(200);
myList.AddTail(300);

// Confirm the tail of the list
ATLASSERT(myList.GetTail() == 300);

// Remove the tail of the list
ATLASSERT(myList.RemoveTail() == 300);

// Confirm the new tail of the list
ATLASSERT(myList.GetTail() == 200);
```

CAtlList::RemoveTailNoReturn

Call this method to remove the element at the tail of the list without returning a value.

```
void RemoveTailNoReturn() throw();
```

Remarks

The tail element is deleted from the list, and memory is freed. In debug builds, an assertion failure will occur if the list is empty.

Example

See the example for [CAtlList::IsEmpty](#).

CAtlList::SetAt

Call this method to set the value of the element at a given position in the list.

```
void SetAt(POSITION pos, INARGTYPE element);
```

Parameters

pos

The POSITION value corresponding to the element to change.

element

The new element value.

Remarks

Replaces the existing value with *element*. In debug builds, an assertion failure will occur if *pos* is equal to NULL.

Example

```
// Define the integer list
CAtlList<int> myList;

// Populate the list
myList.AddTail(100);
myList.AddTail(200);

// Use SetAt to change the values stored in the head and
// tail of the list
myList.SetAt(myList.GetHeadPosition(), myList.GetHead() * 10);
myList.SetAt(myList.GetTailPosition(), myList.GetTail() * 10);

// Confirm the values
ATLASSERT(myList.GetHead() == 1000);
ATLASSERT(myList.GetTail() == 2000);
```

CAtlList::SwapElements

Call this method to swap elements in the list.

```
void SwapElements(POSITION pos1, POSITION pos2) throw();
```

Parameters

pos1

The first POSITION value.

pos2

The second POSITION value.

Remarks

Swaps the elements at the two positions specified. In debug builds, an assertion failure will occur if either position value is equal to NULL.

Example

```

// Define the integer list
CAtLList<int> myList;

// Populate the list
for (int i = 0; i < 100; i++)
{
    myList.AddHead(i);
}

// Order is: 99, 98, 97, 96...
ATLASSERT(myList.GetHead() == 99);
ATLASSERT(myList.GetTail() == 0);

// Perform a crude bubble sort
for (int j = 0; j < 100; j++)
{
    for(int i = 0; i < 99; i++)
    {
        if (myList.GetAt(myList.FindIndex(i)) >
            myList.GetAt(myList.FindIndex(i+1)))
        {
            myList.SwapElements(myList.FindIndex(i), myList.FindIndex(i+1));
        }
    }
}

// Order is: 0, 1, 2, 3...
ATLASSERT(myList.GetHead() == 0);
ATLASSERT(myList.GetTail() == 99);

```

See also

[CList Class](#)

[Class Overview](#)

CAtIMap Class

4/1/2019 • 15 minutes to read • [Edit Online](#)

This class provides methods for creating and managing a map object.

Syntax

```
template <typename K,  
         typename V,  
         class KTraits = CElementTraits<K>,  
         class VTraits = CElementTraits<V>>  
class CAtIMap
```

Parameters

K

The key element type.

V

The value element type.

KTraits

The code used to copy or move key elements. See [CElementTraits Class](#) for more details.

VTraits

The code used to copy or move value elements.

Members

Public Typedefs

NAME	DESCRIPTION
CAtIMap::KINARGTYPE	Type used when a key is passed as an input argument
CAtIMap::KOUTARGTYPE	Type used when a key is returned as an output argument.
CAtIMap::VINARGTYPE	Type used when a value is passed as an input argument.
CAtIMap::VOUTARGTYPE	Type used when a value is passed as an output argument.

Public Classes

NAME	DESCRIPTION
CAtIMap::CPair Class	A class containing the key and value elements.

CPair Data Members

NAME	DESCRIPTION
CPair::m_key	The data member storing the key element.

NAME	DESCRIPTION
CPair::m_value	The data member storing the value element.

Public Constructors

NAME	DESCRIPTION
CAtIMap::CAtIMap	The constructor.
CAtIMap::~~CAtIMap	The destructor.

Public Methods

NAME	DESCRIPTION
CAtIMap::AssertValid	Call this method to cause an ASSERT if the <code>CAtIMap</code> is not valid.
CAtIMap::DisableAutoRehash	Call this method to disable automatic rehashing of the <code>CAtIMap</code> object.
CAtIMap::EnableAutoRehash	Call this method to enable automatic rehashing of the <code>CAtIMap</code> object.
CAtIMap::GetAt	Call this method to return the element at a specified position in the map.
CAtIMap::GetCount	Call this method to retrieve the number of elements in the map.
CAtIMap::GetHashTableSize	Call this method to determine the number of bins in the map's hash table.
CAtIMap::GetKeyAt	Call this method to retrieve the key stored at the given position in the <code>CAtIMap</code> object.
CAtIMap::GetNext	Call this method to obtain a pointer to the next element pair stored in the <code>CAtIMap</code> object.
CAtIMap::GetNextAssoc	Gets the next element for iterating.
CAtIMap::GetNextKey	Call this method to retrieve the next key from the <code>CAtIMap</code> object.
CAtIMap::GetNextValue	Call this method to get the next value from the <code>CAtIMap</code> object.
CAtIMap::GetStartPosition	Call this method to start a map iteration.
CAtIMap::GetValueAt	Call this method to retrieve the value stored at a given position in the <code>CAtIMap</code> object.
CAtIMap::InitHashTable	Call this method to initialize the hash table.

NAME	DESCRIPTION
CAtlMap::IsEmpty	Call this method to test for an empty map object.
CAtlMap::Lookup	Call this method to look up keys or values in the <code>CAtlMap</code> object.
CAtlMap::Rehash	Call this method to rehash the <code>CAtlMap</code> object.
CAtlMap::RemoveAll	Call this method to remove all elements from the <code>CAtlMap</code> object.
CAtlMap::RemoveAtPos	Call this method to remove the element at the given position in the <code>CAtlMap</code> object.
CAtlMap::RemoveKey	Call this method to remove an element from the <code>CAtlMap</code> object, given the key.
CAtlMap::SetAt	Call this method to insert an element pair into the map.
CAtlMap::SetOptimalLoad	Call this method to set the optimal load of the <code>CAtlMap</code> object.
CAtlMap::SetValueAt	Call this method to change the value stored at a given position in the <code>CAtlMap</code> object.

Public Operators

NAME	DESCRIPTION
CAtlMap::operator[]	Replaces or adds a new element to the <code>CAtlMap</code> .

Remarks

`CAtlMap` provides support for a mapping array of any given type, managing an unordered array of key elements and their associated values. Elements (consisting of a key and a value) are stored using a hashing algorithm, allowing a large amount of data to be efficiently stored and retrieved.

The *KTraits* and *VTraits* parameters are traits classes that contain any supplemental code needed to copy or move elements.

An alternative to `CAtlMap` is offered by the `CRBMap` class. `CRBMap` also stores key/value pairs, but exhibits different performance characteristics. The time taken to insert an item, look up a key, or delete a key from a `CRBMap` object is of order $\log(n)$, where n is the number of elements. For `CAtlMap`, all of these operations typically take a constant time, although worst-case scenarios might be of order n . Therefore, in a typical case, `CAtlMap` is faster.

The other difference between `CRBMap` and `CAtlMap` becomes apparent when iterating through the stored elements. In a `CRBMap`, the elements are visited in a sorted order. In a `CAtlMap`, the elements are not ordered, and no order can be inferred.

When a small number of elements need to be stored, consider using the `CSimpleMap` class instead.

For more information, see [ATL Collection Classes](#).

Requirements

Header: atlcoll.h

CAtlMap::AssertValid

Call this method to cause an ASSERT if the `CAtlMap` object is not valid.

```
void AssertValid() const;
```

Remarks

In debug builds, this method will cause an ASSERT if the `CAtlMap` object is not valid.

Example

See the example for [CAtlMap::CAtlMap](#).

CAtlMap::CAtlMap

The constructor.

```
CAtlMap(  
    UINT nBins = 17,  
    float fOptimalLoad = 0.75f,  
    float fLoThreshold = 0.25f,  
    float fHiThreshold = 2.25f,  
    UINT nBlockSize = 10) throw ();
```

Parameters

nBins

The number of bins providing pointers to the stored elements. See Remarks later in this topic for an explanation of bins.

fOptimalLoad

The optimal load ratio.

fLoThreshold

The lower threshold for the load ratio.

fHiThreshold

The upper threshold for the load ratio.

nBlockSize

The block size.

Remarks

`CAtlMap` references all of its stored elements by first creating an index using a hashing algorithm on the key. This index references a "bin" which contains a pointer to the stored elements. If the bin is already in use, a linked-list is created to access the subsequent elements. Traversing a list is slower than directly accessing the correct element, and so the map structure needs to balance storage requirements against performance. The default parameters have been chosen to give good results in most cases.

The load ratio is the ratio of the number of bins to the number of elements stored in the map object. When the map structure is recalculated, the *fOptimalLoad* parameter value will be used to calculate the number of bins required. This value can be changed using the [CAtlMap::SetOptimalLoad](#) method.

The *fLoThreshold* parameter is the lower value that the load ratio can reach before `CAtlMap` will recalculate the optimal size of the map.

The *fHiThreshold* parameter is the upper value that the load ratio can reach before the `CAtlMap` object will recalculate the optimal size of the map.

This recalculation process (known as rehashing) is enabled by default. If you want to disable this process, perhaps when entering a lot of data at one time, call the `CAtlMap::DisableAutoRehash` method. Reactivate it with the `CAtlMap::EnableAutoRehash` method.

The *nBlockSize* parameter is a measure of the amount of memory allocated when a new element is required. Larger block sizes reduce calls to memory allocation routines, but use more resources.

Before any data can be stored, it is necessary to initialize the hash table with a call to `CAtlMap::InitHashTable`.

Example

```
// Create a map which stores a double
// value using an integer key

CAtlMap<int, double> mySinTable;
int i;

// Initialize the Hash Table
mySinTable.InitHashTable(257);

// Add items to the map
for (i = 0; i < 90; i++)
    mySinTable[i] = sin((double)i);

// Confirm the map is valid
mySinTable.AssertValid();

// Confirm the number of elements in the map
ATLASSERT(mySinTable.GetCount() == 90);

// Remove elements with even key values
for (i = 0; i < 90; i += 2)
    mySinTable.RemoveKey(i);

// Confirm the number of elements in the map
ATLASSERT(mySinTable.GetCount() == 45);

// Walk through all the elements in the map.
// First, get start position.
POSITION pos;
int key;
double value;
pos = mySinTable.GetStartPosition();

// Now iterate the map, element by element
while (pos != NULL)
{
    key = mySinTable.GetKeyAt(pos);
    value = mySinTable.GetNextValue(pos);
}
```

CAtlMap::~~CAtlMap

The destructor.

```
~CAtlMap() throw();
```

Remarks

Frees any allocated resources.

CAtIMap::CPair Class

A class containing the key and value elements.

```
class CPair : public __POSITION
```

Remarks

This class is used by the methods [CAtIMap::GetNext](#) and [CAtIMap::Lookup](#) to access the key and value elements stored in the mapping structure.

CAtIMap::DisableAutoRehash

Call this method to disable automatic rehashing of the `CAtIMap` object.

```
void DisableAutoRehash() throw();
```

Remarks

When automatic rehashing is enabled (which it is by default), the number of bins in the hash table will automatically be recalculated if the load value (the ratio of the number of bins to the number of elements stored in the array) exceeds the maximum or minimum values specified at the time the map was created.

`DisableAutoRehash` is most useful when a large number of elements will be added to the map at once. Instead of triggering the rehashing process every time the limits are exceeded, it is more efficient to call `DisableAutoRehash`, add the elements, and finally call [CAtIMap::EnableAutoRehash](#).

CAtIMap::EnableAutoRehash

Call this method to enable automatic rehashing of the `CAtIMap` object.

```
void EnableAutoRehash() throw();
```

Remarks

When automatic rehashing is enabled (which it is by default), the number of bins in the hash table will automatically be recalculated if the load value (the ratio of the number of bins to the number of elements stored in the array) exceeds the maximum or minimum values specified at the time the map is created.

`EnableAutoRefresh` is most often used after a call to [CAtIMap::DisableAutoRehash](#).

CAtIMap::GetAt

Call this method to return the element at a specified position in the map.

```
void GetAt(  
    POSITION pos,  
    KOUTARGTYPE key,  
    VOUTARGTYPE value) const;  
  
CPair* GetAt(POSITION& pos) throw();
```

Parameters

pos

The position counter, returned by a previous call to [CAtlMap::GetNextAssoc](#) or [CAtlMap::GetStartPosition](#).

key

Template parameter specifying the type of the map's key.

value

Template parameter specifying the type of the map's value.

Return Value

Returns a pointer to the current pair of key/value elements stored in the map.

Remarks

In debug builds, an assertion error will occur if *pos* is equal to NULL.

CAtlMap::GetCount

Call this method to retrieve the number of elements in the map.

```
size_t GetCount() const throw();
```

Return Value

Returns the number of elements in the map object. A single element is a key/value pair.

Example

See the example for [CAtlMap::CAtlMap](#).

CAtlMap::GetHashTableSize

Call this method to determine the number of bins in the map's hash table.

```
UINT GetHashTableSize() const throw();
```

Return Value

Returns the number of bins in the hash table. See [CAtlMap::CAtlMap](#) for an explanation.

CAtlMap::GetKeyAt

Call this method to retrieve the key stored at the given position in the `CAtlMap` object.

```
const K& GetKeyAt(POSITION pos) const throw();
```

Parameters

pos

The position counter, returned by a previous call to [CAtlMap::GetNextAssoc](#) or [CAtlMap::GetStartPosition](#).

Return Value

Returns a reference to the key stored at the given position in the `CAtlMap` object.

Example

See the example for [CAtlMap::CAtlMap](#).

CAtIMap::GetNext

Call this method to obtain a pointer to the next element pair stored in the `CAtIMap` object.

```
CPair* GetNext(POSITION& pos) throw();  
const CPair* GetNext(POSITION& pos) const throw();
```

Parameters

pos

The position counter, returned by a previous call to [CAtIMap::GetNextAssoc](#) or [CAtIMap::GetStartPosition](#).

Return Value

Returns a pointer to the next pair of key/value elements stored in the map. The *pos* position counter is updated after each call. If the retrieved element is the last in the map, *pos* is set to NULL.

CAtIMap::GetNextAssoc

Gets the next element for iterating.

```
void GetNextAssoc(  
    POSITION& pos,  
    KOUTARGTYPE key,  
    VOUTARGTYPE value) const;
```

Parameters

pos

The position counter, returned by a previous call to [CAtIMap::GetNextAssoc](#) or [CAtIMap::GetStartPosition](#).

key

Template parameter specifying the type of the map's key.

value

Template parameter specifying the type of the map's value.

Remarks

The *pos* position counter is updated after each call. If the retrieved element is the last in the map, *pos* is set to NULL.

CAtIMap::GetNextKey

Call this method to retrieve the next key from the `CAtIMap` object.

```
const K& GetNextKey(POSITION& pos) const throw();
```

Parameters

pos

The position counter, returned by a previous call to [CAtIMap::GetNextAssoc](#) or [CAtIMap::GetStartPosition](#).

Return Value

Returns a reference to the next key in the map.

Remarks

Updates the current position counter, *pos*. If there are no more entries in the map, the position counter is set to

NULL.

CAtIMap::GetNextValue

Call this method to get the next value from the `CAtIMap` object.

```
V& GetNextValue(POSITION& pos) throw();  
const V& GetNextValue(POSITION& pos) const throw();
```

Parameters

pos

The position counter, returned by a previous call to [CAtIMap::GetNextAssoc](#) or [CAtIMap::GetStartPosition](#).

Return Value

Returns a reference to the next value in the map.

Remarks

Updates the current position counter, *pos*. If there are no more entries in the map, the position counter is set to NULL.

Example

See the example for [CAtIMap::CAtIMap](#).

CAtIMap::GetStartPosition

Call this method to start a map iteration.

```
POSITION GetStartPosition() const throw();
```

Return Value

Returns the start position, or NULL is returned if the map is empty.

Remarks

Call this method to start a map iteration by returning a POSITION value that can be passed to the `GetNextAssoc` method.

NOTE

The iteration sequence is not predictable

Example

See the example for [CAtIMap::CAtIMap](#).

CAtIMap::GetValueAt

Call this method to retrieve the value stored at a given position in the `CAtIMap` object.

```
V& GetValueAt(POSITION pos) throw();  
const V& GetValueAt(POSITION pos) const throw();
```

Parameters

pos

The position counter, returned by a previous call to [CAtlMap::GetNextAssoc](#) or [CAtlMap::GetStartPosition](#).

Return Value

Returns a reference to the value stored at the given position in the `CAtlMap` object.

CAtlMap::InitHashTable

Call this method to initialize the hash table.

```
bool InitHashTable(  
    UINT nBins,  
    bool bAllocNow = true);
```

Parameters

nBins

The number of bins used by the hash table. See [CAtlMap::CAtlMap](#) for an explanation.

bAllocNow

A flag indication when memory should be allocated.

Return Value

Returns TRUE on successful initialization, FALSE on failure.

Remarks

`InitHashTable` must be called before any elements are stored in the hash table. If this method is not called explicitly, it will be called automatically the first time an element is added using the bin count specified by the `CAtlMap` constructor. Otherwise, the map will be initialized using the new bin count specified by the *nBins* parameter.

If the *bAllocNow* parameter is false, the memory required by the hash table will not be allocated until it is first required. This can be useful if it is uncertain if the map will be used.

Example

See the example for [CAtlMap::CAtlMap](#).

CAtlMap::IsEmpty

Call this method to test for an empty map object.

```
bool IsEmpty() const throw();
```

Return Value

Returns TRUE if the map is empty, FALSE otherwise.

CAtlMap::KINARGTYPE

Type used when a key is passed as an input argument.

```
typedef KTraits::INARGTYPE KINARGTYPE;
```

CAtlMap::KOUTARGTYPE

Type used when a key is returned as an output argument.

```
typedef KTraits::OUTARGTYPE KOUTARGTYPE;
```

CAtIMap::Lookup

Call this method to look up keys or values in the `CAtIMap` object.

```
bool Lookup(KINARGTYPE key, VOUTARGTYPE value) const;  
const CPair* Lookup(KINARGTYPE key) const throw();  
CPair* Lookup(KINARGTYPE key) throw();
```

Parameters

key

Specifies the key that identifies the element to be looked up.

value

Variable that receives the looked-up value.

Return Value

The first form of the method returns true if the key is found, otherwise false. The second and third forms return a pointer to a `CPair` which can be used as a position for calls to `CAtIMap::GetNext` and so on.

Remarks

`Lookup` uses a hashing algorithm to quickly find the map element containing a key that exactly matches the given key parameter.

CAtIMap::operator []

Replaces or adds a new element to the `CAtIMap`.

```
V& operator[](kinargtype key) throw();
```

Parameters

key

The key of the element to add or replace.

Return Value

Returns a reference to the value associated with the given key.

Example

If the key already exists, the element is replaced. If the key does not exist, a new element is added. See the example for `CAtIMap::CAtIMap`.

CAtIMap::Rehash

Call this method to rehash the `CAtIMap` object.

```
void Rehash(UINT nBins = 0);
```

Parameters

nBins

The new number of bins to use in the hash table. See `CAtIMap::CAtIMap` for an explanation.

Remarks

If *nBins* is 0, `CAtIMap` calculates a reasonable number based on the number of elements in the map and the optimal load setting. Normally the rehashing process is automatic, but if `CAtIMap::DisableAutoRehash` has been called, this method will perform the necessary resizing.

CAtIMap::RemoveAll

Call this method to remove all elements from the `CAtIMap` object.

```
void RemoveAll() throw();
```

Remarks

Clears out the `CAtIMap` object, freeing the memory used to store the elements.

CAtIMap::RemoveAtPos

Call this method to remove the element at the given position in the `CAtIMap` object.

```
void RemoveAtPos(POSITION pos) throw();
```

Parameters

pos

The position counter, returned by a previous call to `CAtIMap::GetNextAssoc` or `CAtIMap::GetStartPosition`.

Remarks

Removes the key/value pair stored at the specified position. The memory used to store the element is freed. The POSITION referenced by *pos* becomes invalid, and while the POSITION of any other elements in the map remains valid, they do not necessarily retain the same order.

CAtIMap::RemoveKey

Call this method to remove an element from the `CAtIMap` object, given the key.

```
bool RemoveKey(KINARGTYPE key) throw();
```

Parameters

key

The key corresponding to the element pair you want to remove.

Return Value

Returns TRUE if the key is found and removed, FALSE on failure.

Example

See the example for `CAtIMap::CAtIMap`.

CAtIMap::SetAt

Call this method to insert an element pair into the map.

```
POSITION SetAt(  
    KINARGTYPE key,  
    VINARGTYPE value);
```

Parameters

key

The key value to add to the `CAt1Map` object.

value

The value to add to the `CAt1Map` object.

Return Value

Returns the position of the key/value element pair in the `CAt1Map` object.

Remarks

`SetAt` replaces an existing element if a matching key is found. If the key is not found, a new key/value pair is created.

CAt1Map::SetOptimalLoad

Call this method to set the optimal load of the `CAt1Map` object.

```
void SetOptimalLoad(  
    float fOptimalLoad,  
    float fLoThreshold,  
    float fHiThreshold,  
    bool bRehashNow = false);
```

Parameters

fOptimalLoad

The optimal load ratio.

fLoThreshold

The lower threshold for the load ratio.

fHiThreshold

The upper threshold for the load ratio.

bRehashNow

Flag indicating if the hash table should be recalculated.

Remarks

This method redefines the optimal load value for the `CAt1Map` object. See [CAt1Map::CAt1Map](#) for a discussion of the various parameters. If *bRehashNow* is true, and the number of elements is outside the minimum and maximum values, the hash table is recalculated.

CAt1Map::SetValueAt

Call this method to change the value stored at a given position in the `CAt1Map` object.

```
void SetValueAt(  
    POSITION pos,  
    VINARGTYPE value);
```

Parameters

pos

The position counter, returned by a previous call to [CAtlMap::GetNextAssoc](#) or [CAtlMap::GetStartPosition](#).

value

The value to add to the `CAtlMap` object.

Remarks

Changes the value element stored at the given position in the `CAtlMap` object.

CAtlMap::VINARGTYPE

Type used when a value is passed as an input argument.

```
typedef VTraits::INARGTYPE VINARGTYPE;
```

CAtlMap::VOUTARGTYPE

Type used when a value is passed as an output argument.

```
typedef VTraits::OUTARGTYPE VOUTARGTYPE;
```

CAtlMap::CPair::m_key

The data member storing the key element.

```
const K m_key;
```

Parameters

K

The key element type.

CAtlMap::CPair::m_value

The data member storing the value element.

```
V m_value;
```

Parameters

V

The value element type.

See also

[Marquee Sample](#)

[UpdatePV Sample](#)

[Class Overview](#)

CAtlModule Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

This class provides methods used by several ATL module classes.

Syntax

```
class ATL_NO_VTABLE CAtlModule : public _ATL_MODULE
```

Members

Public Constructors

NAME	DESCRIPTION
CAtlModule::CAtlModule	The constructor.
CAtlModule::~~CAtlModule	The destructor.

Public Methods

NAME	DESCRIPTION
CAtlModule::AddCommonRGSReplacements	Override this method to add parameters to the ATL Registry Component (Registrar) replacement map.
CAtlModule::AddTermFunc	Adds a new function to be called when the module terminates.
CAtlModule::GetGIPtr	Returns the Global Interface Pointer.
CAtlModule::GetLockCount	Returns the lock count.
CAtlModule::Lock	Increments the lock count.
CAtlModule::Term	Releases all data members.
CAtlModule::Unlock	Decrements the lock count.
CAtlModule::UpdateRegistryFromResourceD	Runs the script contained in a specified resource to register or unregister an object.
CAtlModule::UpdateRegistryFromResourceDHelper	This method is called by <code>UpdateRegistryFromResourceD</code> to perform the registry update.
CAtlModule::UpdateRegistryFromResourceS	Runs the script contained in a specified resource to register or unregister an object. This method statically links to the ATL Registry Component.

Public Data Members

NAME	DESCRIPTION
CAtlModule::m_libid	Contains the GUID of the current module.
CAtlModule::m_pGIT	Pointer to the Global Interface Table.

Remarks

This class is used by [CAtlDllModuleT Class](#), [CAtlExeModuleT Class](#), and [CAtlServiceModuleT Class](#) to provide support for DLL applications, EXE applications, and Windows services, respectively.

For more information on modules in ATL, see [ATL Module Classes](#).

This class replaces the obsolete [CComModule Class](#) used in earlier versions of ATL.

Inheritance Hierarchy

[_ATL_MODULE](#)

CAtlModule

Requirements

Header: atlbase.h

CAtlModule::AddCommonRGSReplacements

Override this method to add parameters to the ATL Registry Component (Registrar) replacement map.

```
virtual HRESULT AddCommonRGSReplacements(IRegistrarBase* /* pRegistrar */) throw() = 0;
```

Parameters

pRegistrar

Reserved.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Replaceable parameters allow a Registrar's client to specify run-time data. To do this, the Registrar maintains a replacement map into which it enters the values associated with the replaceable parameters in your script. The Registrar makes these entries at run time.

See the topic [Using Replaceable Parameters \(The Registrar's Preprocessor\)](#) for more details.

CAtlModule::AddTermFunc

Adds a new function to be called when the module terminates.

```
HRESULT AddTermFunc(_ATL_TERMFUNC* pFunc, DWORD_PTR dw) throw();
```

Parameters

pFunc

Pointer to the function to add.

dw

User-defined data, passed to the function.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CAtlModule::CAtlModule

The constructor.

```
CAtlModule() throw();
```

Remarks

Initializes data members and initiates a critical section around the module's thread.

CAtlModule::~~CAtlModule

The destructor.

```
~CAtlModule() throw();
```

Remarks

Releases all data members.

CAtlModule::GetGITPtr

Retrieves a pointer to the Global Interface Table.

```
virtual HRESULT GetGITPtr(IGlobalInterfaceTable** ppGIT) throw();
```

Parameters

ppGIT

Pointer to the variable which will receive the pointer to the Global Interface Table.

Return Value

Returns S_OK on success, or an error code on failure. E_POINTER is returned if *ppGIT* is equal to NULL.

Remarks

If the Global Interface Table object does not exist, it is created, and its address is stored in the member variable [CAtlModule::m_pGIT](#).

In debug builds, an assertion error will occur if *ppGIT* is equal to NULL, or if the Global Interface Table pointer cannot be obtained.

See [IGlobalInterfaceTable](#) for information on the Global Interface Table.

CAtlModule::GetLockCount

Returns the lock count.

```
virtual LONG GetLockCount() throw();
```

Return Value

Returns the lock count. This value may be useful for diagnostics and debugging.

CAtlModule::Lock

Increments the lock count.

```
virtual LONG Lock() throw();
```

Return Value

Increments the lock count and returns the updated value. This value may be useful for diagnostics and debugging.

CAtlModule::m_libid

Contains the GUID of the current module.

```
static GUID m_libid;
```

CAtlModule::m_pGIT

Pointer to the Global Interface Table.

```
IGlobalInterfaceTable* m_pGIT;
```

CAtlModule::Term

Releases all data members.

```
void Term() throw();
```

Remarks

Releases all data members. This method is called by the destructor.

CAtlModule::Unlock

Decrements the lock count.

```
virtual LONG Unlock() throw();
```

Return Value

Decrements the lock count and returns the updated value. This value may be useful for diagnostics and debugging.

CAtlModule::UpdateRegistryFromResourceD

Runs the script contained in a specified resource to register or unregister an object.

```
HRESULT WINAPI UpdateRegistryFromResourceD(
    UINT nResID,
    BOOL bRegister,
    struct _ATL_REGMAP_ENTRY* pMapEntries = NULL) throw();

HRESULT WINAPI UpdateRegistryFromResourceD(
    LPCTSTR lpszRes,
    BOOL bRegister,
    struct _ATL_REGMAP_ENTRY* pMapEntries = NULL) throw();
```

Parameters

lpszRes

A resource name.

nResID

A resource ID.

bRegister

TRUE if the object should be registered; FALSE otherwise.

pMapEntries

A pointer to the replacement map storing values associated with the script's replaceable parameters. ATL automatically uses %MODULE%. To use additional replaceable parameters, see [CAtlModule::AddCommonRGSReplacements](#). Otherwise, use the NULL default value.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Runs the script contained in the resource specified by *lpszRes* or *nResID*. If *bRegister* is TRUE, this method registers the object in the system registry; otherwise it removes the object from the registry.

To statically link to the ATL Registry Component (Registrar), see [CAtlModule::UpdateRegistryFromResourceS](#).

This method calls [CAtlModule::UpdateRegistryFromResourceDHelper](#) and [IRegistrar::ResourceUnregister](#).

CAtlModule::UpdateRegistryFromResourceDHelper

This method is called by `UpdateRegistryFromResourceD` to perform the registry update.

```
inline HRESULT WINAPI UpdateRegistryFromResourceDHelper(
    LPCOLESTR lpszRes,
    BOOL bRegister,
    struct _ATL_REGMAP_ENTRY* pMapEntries = NULL) throw();
```

Parameters

lpszRes

A resource name.

bRegister

Indicates whether the object should be registered.

pMapEntries

A pointer to the replacement map storing values associated with the script's replaceable parameters. ATL automatically uses %MODULE%. To use additional replaceable parameters, see

[CAtlModule::AddCommonRGSReplacements](#). Otherwise, use the NULL default value.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method provides the implementation of [CAtlModule::UpdateRegistryFromResourceD](#).

CAtlModule::UpdateRegistryFromResourceS

Runs the script contained in a specified resource to register or unregister an object. This method statically links to the ATL Registry Component.

```
HRESULT WINAPI UpdateRegistryFromResourceS(
    UINT nResID,
    BOOL bRegister,
    struct _ATL_REGMAP_ENTRY* pMapEntries = NULL) throw();

HRESULT WINAPI UpdateRegistryFromResourceS(
    LPCTSTR lpszRes,
    BOOL bRegister,
    struct _ATL_REGMAP_ENTRY* pMapEntries = NULL) throw();
```

Parameters

nResID

A resource ID.

lpszRes

A resource name.

bRegister

Indicates whether the resource script should be registered.

pMapEntries

A pointer to the replacement map storing values associated with the script's replaceable parameters. ATL automatically uses %MODULE%. To use additional replaceable parameters, see [CAtlModule::AddCommonRGSReplacements](#). Otherwise, use the NULL default value.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Similar to [CAtlModule::UpdateRegistryFromResourceD](#) except `CAtlModule::UpdateRegistryFromResourceS` creates a static link to the ATL Registry Component (Registrar).

See also

[_ATL_MODULE](#)

[Class Overview](#)

[Module Classes](#)

[Registry Component \(Registrar\)](#)

CAtlModuleT Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements an ATL module.

Syntax

```
template <class T>
class ATL_NO_VTABLE CAtlModuleT : public CAtlModule
```

Parameters

T

Your class derived from `CAtlModuleT`.

Members

Public Constructors

NAME	DESCRIPTION
CAtlModuleT::CAtlModuleT	The constructor.

Public Methods

NAME	DESCRIPTION
CAtlModuleT::InitLibId	Initializes the data member containing the GUID of the current module.
CAtlModuleT::RegisterApplId	Adds the EXE to the registry.
CAtlModuleT::RegisterServer	Adds the service to the registry.
CAtlModuleT::UnregisterApplId	Removes the EXE from the registry.
CAtlModuleT::UnregisterServer	Removes the service from the registry.
CAtlModuleT::UpdateRegistryApplId	Updates the EXE information in the registry.

Remarks

`CAtlModuleT`, derived from [CAtlModule](#), implements an Executable (EXE) or a Service (EXE) ATL module. An Executable module is a local, out-of-process server, whereas a Service module is a Windows application that runs in the background when Windows starts.

`CAtlModuleT` provides support for initializing, registering, and unregistering of the module.

Inheritance Hierarchy

[_ATL_MODULE](#)

Requirements

Header: atlbase.h

CAtlModuleT::CAtlModuleT

The constructor.

```
CAtlModuleT() throw();
```

Remarks

Calls [CAtlModuleT::InitLibId](#).

CAtlModuleT::InitLibId

Initializes the data member containing the GUID of the current module.

```
static void InitLibId() throw();
```

Remarks

Called by the constructor [CAtlModuleT::CAtlModuleT](#).

CAtlModuleT::RegisterAppId

Adds the EXE to the registry.

```
HRESULT RegisterAppId() throw();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CAtlModuleT::RegisterServer

Adds the service to the registry.

```
HRESULT RegisterServer(  
    BOOL bRegTypeLib = FALSE,  
    const CLSID* pCLSID = NULL) throw();
```

Parameters

bRegTypeLib

TRUE if the type library is to be registered. The default value is FALSE.

pCLSID

Points to the CLSID of the object to be registered. If NULL (the default value), all objects in the object map will be registered.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CAtlModuleT::UnregisterAppId

Removes the EXE from the registry.

```
HRESULT UnregisterAppId() throw();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CAtlModuleT::UnregisterServer

Removes the service from the registry.

```
HRESULT UnregisterServer(  
    BOOL bUnRegTypeLib,  
    const CLSID* pCLSID = NULL) throw();
```

Parameters

bUnRegTypeLib

TRUE if the type library is also to be unregistered.

pCLSID

Points to the CLSID of the object to be unregistered. If NULL (the default value), all objects in the object map will be unregistered.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CAtlModuleT::UpdateRegistryAppId

Updates the EXE information in the registry.

```
static HRESULT WINAPI UpdateRegistryAppId(BOOL /* bRegister*/) throw();
```

Parameters

bRegister

Reserved.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See also

[CAtlModule Class](#)

[Class Overview](#)

[Module Classes](#)

CAtlPreviewCtrlImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class is an ATL implementation of a window that is placed on a host window provided by the Shell for Rich Preview.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CAtlPreviewCtrlImpl : public CWindowImpl<CAtlPreviewCtrlImpl>, public IPreviewCtrl;
```

Members

Public Constructors

NAME	DESCRIPTION
CAtlPreviewCtrlImpl::~~CAtlPreviewCtrlImpl	Destructs a preview control object.
CAtlPreviewCtrlImpl::CAtlPreviewCtrlImpl	Constructs a preview control object.

Public Methods

NAME	DESCRIPTION
CAtlPreviewCtrlImpl::Create	Called by a Rich Preview handler to create the Windows window.
CAtlPreviewCtrlImpl::Destroy	Called by a Rich Preview handler when it needs to destroy this control.
CAtlPreviewCtrlImpl::Focus	Sets input focus to this control.
CAtlPreviewCtrlImpl::OnPaint	Handles the WM_PAINT message.
CAtlPreviewCtrlImpl::Redraw	Tells this control to redraw.
CAtlPreviewCtrlImpl::SetHost	Sets a new parent for this control.
CAtlPreviewCtrlImpl::SetPreviewVisuals	Called by a Rich Preview handler when it needs to set visuals of rich preview content.
CAtlPreviewCtrlImpl::SetRect	Sets a new bounding rectangle for this control.

Protected Methods

NAME	DESCRIPTION
CAtlPreviewCtrlImpl::DoPaint	Called by the framework to render the preview.

Protected Constants

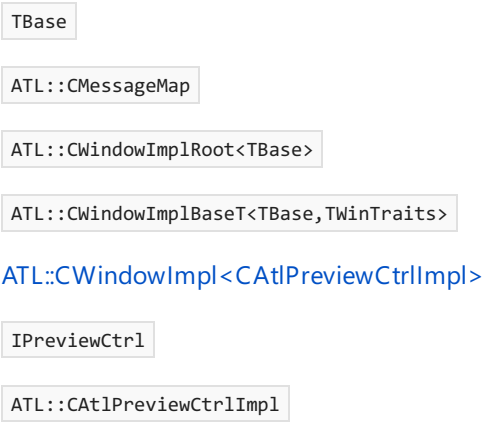
NAME	DESCRIPTION
CAtlPreviewCtrlImpl::m_plf	Font used to display text in the preview window.

Protected Data Members

NAME	DESCRIPTION
CAtlPreviewCtrlImpl::m_clrBack	Background color of the preview window.
CAtlPreviewCtrlImpl::m_clrText	Text color of preview window.

Remarks

Inheritance Hierarchy



Requirements

Header: atlpreviewctrlimpl.h

CAtlPreviewCtrlImpl::CAtlPreviewCtrlImpl

Constructs a preview control object.

```
CAtlPreviewCtrlImpl(void) : m_clrText(0),
    m_clrBack(RGB(255, 255, 255)), m_plf(NULL);
```

Remarks

CAtlPreviewCtrlImpl::~~CAtlPreviewCtrlImpl

Destructs a preview control object.

```
virtual ~CAtlPreviewCtrlImpl(void);
```

Remarks

CAtlPreviewCtrlImpl::Create

Called by a Rich Preview handler to create the Windows window.

```
virtual BOOL Create(HWND hWndParent, const RECT* prc);
```

Parameters

hWndParent

A handle to the host window supplied by the Shell for Rich Preview.

prc

Specifies the initial size and position of the window.

Return Value

TRUE if successful; otherwise FALSE.

Remarks

CAtlPreviewCtrlImpl::Destroy

Called by a Rich Preview handler when it needs to destroy this control.

```
virtual void Destroy();
```

Remarks

CAtlPreviewCtrlImpl::DoPaint

Called by the framework to render the preview.

```
virtual void DoPaint(HDC hdc);
```

Parameters

hdc

A handle to a device context for painting.

Remarks

CAtlPreviewCtrlImpl::Focus

Sets input focus to this control.

```
virtual void Focus();
```

Remarks

CAtlPreviewCtrlImpl::m_clrBack

Background color of the preview window.

```
COLORREF m_clrBack;
```

Remarks

CAtlPreviewCtrlImpl::m_clrText

Text color of the preview window.

```
COLORREF m_clrText;
```

Remarks

CAtlPreviewCtrlImpl::m_plf

Font used to display text in the preview window.

```
const LOGFONT* m_plf;
```

Remarks

CAtlPreviewCtrlImpl::OnPaint

Handles the WM_PAINT message.

```
LRESULT OnPaint(  
    UINT nMsg,  
    WPARAM wParam,  
    LPARAM lParam,  
    BOOL& bHandled);
```

Parameters

nMsg

Set to WM_PAINT.

wParam

This parameter is not used.

lParam

This parameter is not used.

bHandled

When this function returns, it contains TRUE.

Return Value

Always returns 0.

Remarks

CAtlPreviewCtrlImpl::Redraw

Tells this control to redraw.

```
virtual void Redraw();
```


Remarks

CAtlPreviewCtrlImpl::SetHost

Sets a new parent for this control.

```
virtual void SetHost(HWND hWndParent);
```

Parameters

hWndParent

A handle to the new parent window.

Remarks

CAtlPreviewCtrlImpl::SetPreviewVisuals

Called by a Rich Preview handler when it needs to set visuals of rich preview content.

```
virtual void SetPreviewVisuals(  
    COLORREF clrBack,  
    COLORREF clrText,  
    const LOGFONT* plf);
```

Parameters

clrBack

Background color of the preview window.

clrText

Text color of the preview window.

plf

Font used to display text in the preview window.

Remarks

CAtlPreviewCtrlImpl::SetRect

Sets a new bounding rectangle for this control.

```
virtual void SetRect(const RECT* prc, BOOL bRedraw);
```

Parameters

prc

Specifies the new size and position of the preview control.

bRedraw

Specifies whether the control should be redrawn.

Remarks

See also

[ATL COM Desktop Components](#)

CAtlServiceModuleT Class

5/7/2019 • 8 minutes to read • [Edit Online](#)

This class implements a service.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class T, UINT nServiceNameID>
class ATL_NO_VTABLE CAtlServiceModuleT : public CAtlExeModuleT<T>
```

Parameters

T

Your class derived from `CAtlServiceModuleT`.

nServiceNameID

The resource identifier of the service.

Members

Public Constructors

NAME	DESCRIPTION
CAtlServiceModuleT::CAtlServiceModuleT	The constructor.

Public Methods

NAME	DESCRIPTION
CAtlServiceModuleT::Handler	The handler routine for the service.
CAtlServiceModuleT::InitializeSecurity	Provides the default security settings for the service.
CAtlServiceModuleT::Install	Installs and creates the service.
CAtlServiceModuleT::IsInstalled	Confirms that the service has been installed.
CAtlServiceModuleT::LogEvent	Writes to the event log.
CAtlServiceModuleT::OnContinue	Override this method to continue the service.
CAtlServiceModuleT::OnInterrogate	Override this method to interrogate the service.
CAtlServiceModuleT::OnPause	Override this method to pause the service.

NAME	DESCRIPTION
CAtlServiceModuleT::OnShutdown	Override this method to shut down the service
CAtlServiceModuleT::OnStop	Override this method to stop the service
CAtlServiceModuleT::OnUnknownRequest	Override this method to handle unknown requests to the service
CAtlServiceModuleT::ParseCommandLine	Parses the command line and performs registration if necessary.
CAtlServiceModuleT::PreMessageLoop	This method is called immediately before entering the message loop.
CAtlServiceModuleT::RegisterAppId	Registers the service in the registry.
CAtlServiceModuleT::Run	Runs the service.
CAtlServiceModuleT::ServiceMain	The method called by the Service Control Manager.
CAtlServiceModuleT::SetServiceStatus	Updates the service status.
CAtlServiceModuleT::Start	Called by <code>CAtlServiceModuleT::WinMain</code> when the service starts.
CAtlServiceModuleT::Uninstall	Stops and removes the service.
CAtlServiceModuleT::Unlock	Decrements the service's lock count.
CAtlServiceModuleT::UnregisterAppId	Removes the service from the registry.
CAtlServiceModuleT::WinMain	This method implements the code required to run the service.

Public Data Members

NAME	DESCRIPTION
CAtlServiceModuleT::m_bService	Flag indicating the program is running as a service.
CAtlServiceModuleT::m_dwThreadId	Member variable storing the thread identifier.
CAtlServiceModuleT::m_hServiceStatus	Member variable storing a handle to the status information structure for the current service.
CAtlServiceModuleT::m_status	Member variable storing the status information structure for the current service.
CAtlServiceModuleT::m_szServiceName	The name of the service being registered.

Remarks

`CAtlServiceModuleT`, derived from [CAtlExeModuleT](#), implements a ATL Service module. `CAtlServiceModuleT` provides methods for command-line processing, installation, registering, and removal. If extra functionality is

required, these and other methods can be overridden.

This class replaces the obsolete [CComModule Class](#) used in earlier versions of ATL. See [ATL Module Classes](#) for more details.

Inheritance Hierarchy

[_ATL_MODULE](#)

[CAtlModule](#)

[CAtlModuleT](#)

[CAtlExeModuleT](#)

[CAtlServiceModuleT](#)

Requirements

Header: atlbase.h

CAtlServiceModuleT::CAtlServiceModuleT

The constructor.

```
CAtlServiceModuleT() throw();
```

Remarks

Initializes the data members and sets the initial service status.

CAtlServiceModuleT::Handler

The handler routine for the service.

```
void Handler(DWORD dwOpcode) throw();
```

Parameters

dwOpcode

A switch that defines the handler operation. For details, see the Remarks.

Remarks

This is the code that the Service Control Manager (SCM) calls to retrieve the status of the service and issue instructions such as stop or pause. The SCM passes an operation code, shown below, to [Handler](#) to indicate what the service should do.

OPERATION CODE	MEANING
SERVICE_CONTROL_STOP	Stops the service. Override the method CAtlServiceModuleT::OnStop in atlbase.h to change the behavior.
SERVICE_CONTROL_PAUSE	User implemented. Override the empty method CAtlServiceModuleT::OnPause in atlbase.h to pause the service.

OPERATION CODE	MEANING
SERVICE_CONTROL_CONTINUE	User implemented. Override the empty method CAtlServiceModuleT::OnContinue in atlbase.h to continue the service.
SERVICE_CONTROL_INTERROGATE	User implemented. Override the empty method CAtlServiceModuleT::OnInterrogate in atlbase.h to interrogate the service.
SERVICE_CONTROL_SHUTDOWN	User implemented. Override the empty method CAtlServiceModuleT::OnShutdown in atlbase.h to shutdown the service.

If the operation code isn't recognized, the method [CAtlServiceModuleT::OnUnknownRequest](#) is called.

A default ATL-generated service only handles the stop instruction. If the SCM passes the stop instruction, the service tells the SCM that the program is about to stop. The service then calls `PostThreadMessage` to post a quit message to itself. This terminates the message loop and the service will ultimately close.

CAtlServiceModuleT::InitializeSecurity

Provides the default security settings for the service.

```
HRESULT InitializeSecurity() throw();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Any class that derives from `CAtlServiceModuleT` must implement this method in the derived class.

Use PKT-level authentication, impersonation level of RPC_C_IMP_LEVEL_IDENTIFY and an appropriate non-null security descriptor in the call to `CoInitializeSecurity`.

For wizard-generated nonattributed service projects, this would be in

```
class CNonAttribServiceModule : public CAtlServiceModuleT< CNonAttribServiceModule, IDS_SERVICENAME >
{
public :
    DECLARE_LIBID(LIBID_NonAttribServiceLib)
    DECLARE_REGISTRY_APPID_RESOURCEID(IDR_NONATTRIBSERVICE, "{29160736-339F-4A1C-ABEF-C320CE103E12}")
    HRESULT InitializeSecurity() throw()
    {
        // TODO : Call CoInitializeSecurity and provide the appropriate security settings for
        // your service
        // Suggested - PKT Level Authentication,
        // Impersonation Level of RPC_C_IMP_LEVEL_IDENTIFY
        // and an appropriate Non NULL Security Descriptor.

        return S_OK;
    }
};
```

For attributed service projects, this would be in

```
[ module(SERVICE, uuid = "{D3103322-7B70-4581-8E59-12769BD9A62B}",
    name = "AttribService",
    helpstring = "AttribService 1.0 Type Library",
    resource_name="IDS_SERVICENAME") ]
class CAttribServiceModule
{
public:
    HRESULT InitializeSecurity() throw()
    {
        // TODO : Call CoInitializeSecurity and provide the appropriate security settings for
        // your service
        // Suggested - PKT Level Authentication,
        // Impersonation Level of RPC_C_IMP_LEVEL_IDENTIFY
        // and an appropriate Non NULL Security Descriptor.

        return S_OK;
    }
};
```

CAtlServiceModuleT::Install

Installs and creates the service.

```
BOOL Install() throw();
```

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Installs the service into the Service Control Manager (SCM) database and then creates the service object. If the service could not be created, a message box is displayed and the method returns FALSE.

CAtlServiceModuleT::IsInstalled

Confirms that the service has been installed.

```
BOOL IsInstalled() throw();
```

Return Value

Returns TRUE if the service is installed, FALSE otherwise.

CAtlServiceModuleT::LogEvent

Writes to the event log.

```
void __cdecl LogEvent(LPCTSTR pszFormat, ...) throw();
```

Parameters

pszFormat

The string to write to the event log.

...

Optional extra strings to be written to the event log.

Remarks

This method writes details out to an event log, using the function [ReportEvent](#). If no service is running, the string is sent to the console.

CAtlServiceModuleT::m_bService

Flag indicating the program is running as a service.

```
BOOL m_bService;
```

Remarks

Used to distinguish a Service EXE from an Application EXE.

CAtlServiceModuleT::m_dwThreadId

Member variable storing the thread identifier of the Service.

```
DWORD m_dwThreadId;
```

Remarks

This variable stores the thread identifier of the current thread.

CAtlServiceModuleT::m_hServiceStatus

Member variable storing a handle to the status information structure for the current service.

```
SERVICE_STATUS_HANDLE m_hServiceStatus;
```

Remarks

The [SERVICE_STATUS](#) structure contains information about a service.

CAtlServiceModuleT::m_status

Member variable storing the status information structure for the current service.

```
SERVICE_STATUS m_status;
```

Remarks

The [SERVICE_STATUS](#) structure contains information about a service.

CAtlServiceModuleT::m_szServiceName

The name of the service being registered.

```
TCHAR [256] m_szServiceName;
```

Remarks

A null-terminated string which stores the name of the service.

CAtlServiceModuleT::OnContinue

Override this method to continue the service.

```
void OnContinue() throw();
```

CAtlServiceModuleT::OnInterrogate

Override this method to interrogate the service.

```
void OnInterrogate() throw();
```

CAtlServiceModuleT::OnPause

Override this method to pause the service.

```
void OnPause() throw();
```

CAtlServiceModuleT::OnShutdown

Override this method to shut down the service.

```
void OnShutdown() throw();
```

CAtlServiceModuleT::OnStop

Override this method to stop the service.

```
void OnStop() throw();
```

CAtlServiceModuleT::OnUnknownRequest

Override this method to handle unknown requests to the service.

```
void OnUnknownRequest(DWORD /* dwOpcode */) throw();
```

Parameters

dwOpcode

Reserved.

CAtlServiceModuleT::ParseCommandLine

Parses the command line and performs registration if necessary.

```
bool ParseCommandLine(LPCTSTR lpCmdLine, HRESULT* pnRetCode) throw();
```

Parameters

lpCmdLine

The command line.

pnRetCode

The HRESULT corresponding to the registration (if it took place).

Return Value

Returns true on success, or false if the RGS file supplied in the command line could not be registered.

Remarks

Parses the command line and registers or unregisters the supplied RGS file if necessary. This method calls [CAtlExeModuleT::ParseCommandLine](#) to check for **/RegServer** and **/UnregServer**. Adding the argument **/Service** will register the service.

CAtlServiceModuleT::PreMessageLoop

This method is called immediately before entering the message loop.

```
HRESULT PreMessageLoop(int nShowCmd) throw();
```

Parameters

nShowCmd

This parameter is passed to [CAtlExeModuleT::PreMessageLoop](#).

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Override this method to add custom initialization code for the Service.

CAtlServiceModuleT::RegisterAppId

Registers the service in the registry.

```
inline HRESULT RegisterAppId(bool bService = false) throw();
```

Parameters

bService

Must be true to register as a service.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CAtlServiceModuleT::Run

Runs the service.

```
HRESULT Run(int nShowCmd = SW_HIDE) throw();
```

Parameters

nShowCmd

Specifies how the window is to be shown. This parameter can be one of the values discussed in the [WinMain](#)

section. The default value is SW_HIDE.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

After being called, `Run` calls [CAtlServiceModuleT::PreMessageLoop](#), [CAtlExeModuleT::RunMessageLoop](#), and [CAtlExeModuleT::PostMessageLoop](#).

CAtlServiceModuleT::ServiceMain

This method is called by the Service Control Manager.

```
void ServiceMain(DWORD dwArgc, LPTSTR* lpszArgv) throw();
```

Parameters

dwArgc

The argc argument.

lpszArgv

The argv argument.

Remarks

The Service Control Manager (SCM) calls `ServiceMain` when you open the Services application in the Control Panel, select the service, and click Start.

After the SCM calls `ServiceMain`, a service must give the SCM a handler function. This function lets the SCM obtain the service's status and pass specific instructions (such as pausing or stopping). Subsequently, [CAtlServiceModuleT::Run](#) is called to perform the main work of the service. `Run` continues to execute until the service is stopped.

CAtlServiceModuleT::SetServiceStatus

This method updates the service status.

```
void SetServiceStatus(DWORD dwState) throw();
```

Parameters

dwState

The new status. See [SetServiceStatus](#) for possible values.

Remarks

Updates the Service Control Manager's status information for the service. It is called by [CAtlServiceModuleT::Run](#), [CAtlServiceModuleT::ServiceMain](#) and other handler methods. The status is also stored in the member variable [CAtlServiceModuleT::m_status](#).

CAtlServiceModuleT::Start

Called by `CAtlServiceModuleT::WinMain` when the service starts.

```
HRESULT Start(int nShowCmd) throw();
```

Parameters

nShowCmd

Specifies how the window is to be shown. This parameter can be one of the values discussed in the [WinMain](#) section.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The [CAtlServiceModuleT::WinMain](#) method handles both registration and installation, as well as tasks involved in removing registry entries and uninstalling the module. When the service is run, `WinMain` calls `Start`.

CAtlServiceModuleT::Uninstall

Stops and removes the service.

```
BOOL Uninstall() throw();
```

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Stops the service from running and removes it from the Service Control Manager database.

CAtlServiceModuleT::Unlock

Decrements the service's lock count.

```
LONG Unlock() throw();
```

Return Value

Returns the lock count, which may be useful for diagnostics and debugging.

CAtlServiceModuleT::UnregisterAppId

Removes the service from the registry.

```
HRESULT UnregisterAppId() throw();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CAtlServiceModuleT::WinMain

This method implements the code required to start the service.

```
int WinMain(int nShowCmd) throw();
```

Parameters

nShowCmd

Specifies how the window is to be shown. This parameter can be one of the values discussed in the [WinMain](#)

section.

Return Value

Returns the service's return value.

Remarks

This method processes the command line (with [CAtlServiceModuleT::ParseCommandLine](#)) and then starts the service (using [CAtlServiceModuleT::Start](#)).

See also

[CAtlExeModuleT Class](#)

[Class Overview](#)

CAtlTemporaryFile Class

3/4/2019 • 7 minutes to read • [Edit Online](#)

This class provides methods for the creation and use of a temporary file.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CAtlTemporaryFile
```

Members

Public Constructors

NAME	DESCRIPTION
CAtlTemporaryFile::CAtlTemporaryFile	The constructor.
CAtlTemporaryFile::~~CAtlTemporaryFile	The destructor.

Public Methods

NAME	DESCRIPTION
CAtlTemporaryFile::Close	Call this method to close a temporary file and either delete its contents or store them under the specified file name.
CAtlTemporaryFile::Create	Call this method to create a temporary file.
CAtlTemporaryFile::Flush	Call this method to force any data remaining in the file buffer to be written to the temporary file.
CAtlTemporaryFile::GetPosition	Call this method to get the current file pointer position.
CAtlTemporaryFile::GetSize	Call this method to get the size in bytes of the temporary file.
CAtlTemporaryFile::HandsOff	Call this method to disassociate the file from the <code>CAtlTemporaryFile</code> object.
CAtlTemporaryFile::HandsOn	Call this method to open an existing temporary file and position the pointer at the end of the file.
CAtlTemporaryFile::LockRange	Call this method to lock a region in the file to prevent other processes from accessing it.

NAME	DESCRIPTION
CAtlTemporaryFile::Read	Call this method to read data from the temporary file starting at the position indicated by the file pointer.
CAtlTemporaryFile::Seek	Call this method to move the file pointer of the temporary file.
CAtlTemporaryFile::SetSize	Call this method to set the size of the temporary file.
CAtlTemporaryFile::TempFileName	Call this method to return the name of the temporary file.
CAtlTemporaryFile::UnlockRange	Call this method to unlock a region of the temporary file.
CAtlTemporaryFile::Write	Call this method to write data to the temporary file starting at the position indicated by the file pointer.

Public Operators

NAME	DESCRIPTION
CAtlTemporaryFile::operator HANDLE	Returns a handle to the temporary file.

Remarks

`CAtlTemporaryFile` makes it easy to create and use a temporary file. The file is automatically named, opened, closed, and deleted. If the file contents are required after the file is closed, they can be saved to a new file with a specified name.

Requirements

Header: atfile.h

Example

See the example for [CAtlTemporaryFile::CAtlTemporaryFile](#).

CAtlTemporaryFile::CAtlTemporaryFile

The constructor.

```
CAtlTemporaryFile() throw();
```

Remarks

A file is not actually opened until a call is made to [CAtlTemporaryFile::Create](#).

Example

```

// Declare the temporary file object
CAtlTemporaryFile myTempFile;

// Create the temporary file, without caring where it
// will be created, but with both read and write access.
ATLVERIFY (myTempFile.Create(NULL, GENERIC_READ|GENERIC_WRITE) == S_OK);

// Create some data to write to the file

int nBuffer[100];
DWORD bytes_written = 0, bytes_read = 0;
int i;

for (i = 0; i < 100; i++)
    nBuffer[i] = i;

// Write some data to the file
myTempFile.Write(&nBuffer, sizeof(nBuffer), &bytes_written);

// Confirm it was written ok
ATLASSERT(bytes_written == sizeof(nBuffer));

// Flush the data to disk
ATLVERIFY(myTempFile.Flush() == S_OK);

// Reset the file pointer to the beginning of the file
ATLVERIFY(myTempFile.Seek(0, FILE_BEGIN) == S_OK);

// Read in the data
myTempFile.Read(&nBuffer, sizeof(nBuffer), bytes_read);

// Confirm it was read ok
ATLASSERT(bytes_read == sizeof(nBuffer));

// Close the file, making a copy of it at another location
ATLVERIFY(myTempFile.Close(_T("c:\\temp\\mydata.tmp")) == S_OK);

```

CAtlTemporaryFile::~CAtlTemporaryFile

The destructor.

```
~CAtlTemporaryFile() throw();
```

Remarks

The destructor calls [CAtlTemporaryFile::Close](#).

CAtlTemporaryFile::Close

Call this method to close a temporary file and either delete its contents or store them under the specified file name.

```
HRESULT Close(LPCTSTR szNewName = NULL) throw();
```

Parameters

szNewName

The name for the new file to store the contents of the temporary file in. If this argument is NULL, the contents of the temporary file are deleted.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See the example for [CAtlTemporaryFile::CAtlTemporaryFile](#).

CAtlTemporaryFile::Create

Call this method to create a temporary file.

```
HRESULT Create(LPCTSTR pszDir = NULL, DWORD dwDesiredAccess = GENERIC_WRITE) throw();
```

Parameters

pszDir

The path for the temporary file. If this is NULL, [GetTempPath](#) will be called to assign a path.

dwDesiredAccess

The desired access. See *dwDesiredAccess* in [CreateFile](#) in the Windows SDK.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See the example for [CAtlTemporaryFile::CAtlTemporaryFile](#).

CAtlTemporaryFile::Flush

Call this method to force any data remaining in the file buffer to be written to the temporary file.

```
HRESULT Flush() throw();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Similar to [CAtlTemporaryFile::HandsOff](#), except that the file is not closed.

Example

See the example for [CAtlTemporaryFile::CAtlTemporaryFile](#).

CAtlTemporaryFile::GetPosition

Call this method to get the current file pointer position.

```
HRESULT GetPosition(ULONGLONG& nPos) const throw();
```

Parameters

nPos

The position in bytes.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

To change the file pointer position, use [CAtlTemporaryFile::Seek](#).

CAtlTemporaryFile::GetSize

Call this method to get the size in bytes of the temporary file.

```
HRESULT GetSize(ULONGLONG& nLen) const throw();
```

Parameters

nLen

The number of bytes in the file.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CAtlTemporaryFile::HandsOff

Call this method to disassociate the file from the `CAtlTemporaryFile` object.

```
HRESULT HandsOff() throw();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

`HandsOff` and [CAtlTemporaryFile::HandsOn](#) are used to disassociate the file from the object, and reattach it if needed. `HandsOff` will force any data remaining in the file buffer to be written to the temporary file, and then close the file. If you want to close and delete the file permanently, or if you want to close and retain the contents of the file with a given name, use [CAtlTemporaryFile::Close](#).

CAtlTemporaryFile::HandsOn

Call this method to open an existing temporary file and position the pointer at the end of the file.

```
HRESULT HandsOn() throw();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

[CAtlTemporaryFile::HandsOff](#) and `HandsOn` are used to disassociate the file from the object, and reattach it if needed.

CAtlTemporaryFile::LockRange

Call this method to lock a region in the temporary file to prevent other processes from accessing it.

```
HRESULT LockRange(ULONGLONG nPos, ULONGLONG nCount) throw();
```

Parameters

nPos

The position in the file where the lock should begin.

nCount

The length of the byte range to be locked.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Locking bytes in a file prevents access to those bytes by other processes. You can lock more than one region of a file, but no overlapping regions are allowed. To successfully unlock a region, use [CAtlTemporaryFile::UnlockRange](#), ensuring the byte range corresponds exactly to the region that was previously locked. `LockRange` does not merge adjacent regions; if two locked regions are adjacent, you must unlock each separately.

CAtlTemporaryFile::operator HANDLE

Returns a handle to the temporary file.

```
operator HANDLE() throw();
```

CAtlTemporaryFile::Read

Call this method to read data from the temporary file starting at the position indicated by the file pointer.

```
HRESULT Read(  
    LPVOID pBuffer,  
    DWORD nBufSize,  
    DWORD& nBytesRead) throw();
```

Parameters

pBuffer

Pointer to the buffer that will receive the data read from the file.

nBufSize

The buffer size in bytes.

nBytesRead

The number of bytes read.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls [CAtlFile::Read](#). To change the position of the file pointer, call [CAtlTemporaryFile::Seek](#).

Example

See the example for [CAtlTemporaryFile::CAtlTemporaryFile](#).

CAtlTemporaryFile::Seek

Call this method to move the file pointer of the temporary file.

```
HRESULT Seek(LONGLONG nOffset, DWORD dwFrom = FILE_CURRENT) throw();
```

Parameters

nOffset

The offset, in bytes, from the starting point given by *dwFrom*.

dwFrom

The starting point (FILE_BEGIN, FILE_CURRENT, or FILE_END).

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls [CAtlFile::Seek](#). To obtain the current file pointer position, call [CAtlTemporaryFile::GetPosition](#).

Example

See the example for [CAtlTemporaryFile::CAtlTemporaryFile](#).

CAtlTemporaryFile::SetSize

Call this method to set the size of the temporary file.

```
HRESULT SetSize(ULONGLONG nNewLen) throw();
```

Parameters

nNewLen

The new length of the file in bytes.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls [CAtlFile::SetSize](#). On return, the file pointer is positioned at the end of the file.

CAtlTemporaryFile::TempFileName

Call this method to return the name of temporary file.

```
LPCTSTR TempFileName() throw();
```

Return Value

Returns the LPCTSTR pointing to the file name.

Remarks

The file name is generated in [CAtlTemporaryFile::CAtlTemporaryFile](#) with a call to the [GetTempFileWindows](#) SDK function. The file extension will always be "TFR" for the temporary file.

CAtlTemporaryFile::UnlockRange

Call this method to unlock a region of the temporary file.

```
HRESULT UnlockRange(ULONGLONG nPos, ULONGLONG nCount) throw();
```

Parameters

nPos

The position in the file where the unlock should begin.

nCount

The length of the byte range to be unlocked.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls [CAtlFile::UnlockRange](#).

CAtlTemporaryFile::Write

Call this method to write data to the temporary file starting at the position indicated by the file pointer.

```
HRESULT Write(  
    LPCVOID pBuffer,  
    DWORD nBufSize,  
    DWORD* pnBytesWritten = NULL) throw();
```

Parameters

pBuffer

The buffer containing the data to be written to the file.

nBufSize

The number of bytes to be transferred from the buffer.

pnBytesWritten

The number of bytes written.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls [CAtlFile::Write](#).

Example

See the example for [CAtlTemporaryFile::CAtlTemporaryFile](#).

See also

[Class Overview](#)

[CAtlFile Class](#)

CAtlTransactionManager Class

3/4/2019 • 8 minutes to read • [Edit Online](#)

CAtlTransactionManager class provides a wrapper to Kernel Transaction Manager (KTM) functions.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CAtlTransactionManager;
```

Members

Public Constructors

NAME	DESCRIPTION
~CAtlTransactionManager	CAtlTransactionManager destructor.
CAtlTransactionManager	CAtlTransactionManager constructor.

Public Methods

NAME	DESCRIPTION
Close	Closes one the transaction handle.
Commit	Requests that the transaction be committed.
Create	Creates the transaction handle.
CreateFile	Creates or opens a file, file stream, or directory as a transacted operation.
DeleteFile	Deletes an existing file as a transacted operation.
FindFirstFile	Searches a directory for a file or subdirectory as a transacted operation.
GetFileAttributes	Retrieves file system attributes for a specified file or directory as a transacted operation.
GetFileAttributesEx	Retrieves file system attributes for a specified file or directory as a transacted operation.
GetHandle	Returns the transaction handle.

NAME	DESCRIPTION
IsFallback	Determines whether the fallback calls are enabled.
MoveFile	Moves an existing file or a directory, including its children, as a transacted operation.
RegCreateKeyEx	Creates the specified registry key and associates it with a transaction. If the key already exists, the function opens it.
RegDeleteKey	Deletes a subkey and its values from the specified platform-specific view of the registry as a transacted operation.
RegOpenKeyEx	Opens the specified registry key and associates it with a transaction.
Rollback	Requests that the transaction be rolled back.
SetFileAttributes	Sets the attributes for a file or directory as a transacted operation.

Protected Data Members

NAME	DESCRIPTION
m_bFallback	TRUE if the fallback is supported; FALSE otherwise.
m_hTransaction	The transaction handle.

Remarks

Inheritance Hierarchy

[ATL::CAtlTransactionManager](#)

Requirements

Header: atltransactionmanager.h

~CAtlTransactionManager

CAtlTransactionManager destructor.

```
virtual ~CAtlTransactionManager();
```

Remarks

In normal processing, the transaction is automatically committed and closed. If the destructor is called during an exception unwind, the transaction is rolled back and closed.

CAtlTransactionManager

CAtlTransactionManager constructor.

```
Cat1TransactionManager(BOOL bFallback = TRUE, BOOL bAutoCreateTransaction = TRUE);
```

Parameters

bFallback

TRUE indicates support fallback. If transacted function fails, the class automatically calls the "non-transacted" function. FALSE indicates no "fallback" calls.

bAutoCreateTransaction

TRUE indicates that the transaction handler is created automatically in the constructor. FALSE indicates that it is not.

Remarks

Close

Closes the transaction handle.

```
inline BOOL Close();
```

Return Value

TRUE if successful; otherwise FALSE.

Remarks

This wrapper calls the `CloseHandle` function. The method is automatically called in the destructor.

Commit

Requests that the transaction be committed.

```
inline BOOL Commit();
```

Return Value

TRUE if successful; otherwise FALSE.

Remarks

This wrapper calls the `CommitTransaction` function. The method is automatically called in the destructor.

Create

Creates the transaction handle.

```
inline BOOL Create();
```

Return Value

TRUE if successful; otherwise FALSE.

Remarks

This wrapper calls the `CreateTransaction` function. Check it for

CreateFile

Creates or opens a file, file stream, or directory as a transacted operation.

```
inline HANDLE CreateFile(  
    LPCTSTR lpFileName,  
    DWORD dwDesiredAccess,  
    DWORD dwShareMode,  
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
    DWORD dwCreationDisposition,  
    DWORD dwFlagsAndAttributes,  
    HANDLE hTemplateFile);
```

Parameters

lpFileName

The name of an object to be created or opened.

dwDesiredAccess

The access to the object, which can be summarized as read, write, both, or neither (zero). The most commonly used values are `GENERIC_READ`, `GENERIC_WRITE`, or both: `GENERIC_READ | GENERIC_WRITE`.

dwShareMode

The sharing mode of an object, which can be read, write, both, delete, all of these, or none: 0, `FILE_SHARE_DELETE`, `FILE_SHARE_READ`, `FILE_SHARE_WRITE`.

lpSecurityAttributes

A pointer to a `SECURITY_ATTRIBUTES` structure that contains an optional security descriptor and also determines whether or not the returned handle can be inherited by child processes. The parameter can be `NULL`.

dwCreationDisposition

An action to take on files that exist and do not exist. This parameter must be one of the following values, which cannot be combined: `CREATE_ALWAYS`, `CREATE_NEW`, `OPEN_ALWAYS`, `OPEN_EXISTING`, or `TRUNCATE_EXISTING`.

dwFlagsAndAttributes

The file attributes and flags. This parameter can include any combination of the available file attributes (`FILE_ATTRIBUTE_*`). All other file attributes override `FILE_ATTRIBUTE_NORMAL`. This parameter can also contain combinations of flags (`FILE_FLAG_*`) for control of buffering behavior, access modes, and other special-purpose flags. These combine with any `FILE_ATTRIBUTE_*` values.

hTemplateFile

A valid handle to a template file with the `GENERIC_READ` access right. The template file supplies file attributes and extended attributes for the file that is being created. This parameter can be `NULL`.

Return Value

Returns a handle that can be used to access the object.

Remarks

This wrapper calls the `CreateFileTransacted` function.

DeleteFile

Deletes an existing file as a transacted operation.

```
inline BOOL DeleteFile(LPCTSTR lpFileName);
```

Parameters

lpFileName

The name of the file to be deleted.

Remarks

This wrapper calls the `DeleteFileTransacted` function.

FindFirstFile

Searches a directory for a file or subdirectory as a transacted operation.

```
inline HANDLE FindFirstFile(  
    LPCTSTR lpFileName,  
    WIN32_FIND_DATA* pNextInfo);
```

Parameters

lpFileName

The directory or path, and the file name to search for. This parameter can include wildcard characters, such as an asterisk (*) or a question mark (?).

pNextInfo

A pointer to the WIN32_FIND_DATA structure that receives information about a found file or subdirectory.

Return Value

If the function succeeds, the return value is a search handle used in a subsequent call to `FindNextFile` or `FindClose`. If the function fails or fails to locate files from the search string in the *lpFileName* parameter, the return value is INVALID_HANDLE_VALUE.

Remarks

This wrapper calls the `FindFirstFileTransacted` function.

GetFileAttributes

Retrieves file system attributes for a specified file or directory as a transacted operation.

```
inline DWORD GetFileAttributes(LPCTSTR lpFileName);
```

Parameters

lpFileName

The name of the file or directory.

Remarks

This wrapper calls the `GetFileAttributesTransacted` function.

GetFileAttributesEx

Retrieves file system attributes for a specified file or directory as a transacted operation.

```
inline BOOL GetFileAttributesEx(  
    LPCTSTR lpFileName,  
    GET_FILEEX_INFO_LEVELS fInfoLevelId,  
    LPVOID lpFileInformation);
```

Parameters

lpFileName

The name of the file or directory.

flnfoLevelld

The level of attribute information to retrieve.

lpFileInformation

A pointer to a buffer that receives the attribute information. The type of attribute information that is stored into this buffer is determined by the value of *flnfoLevelld*. If the *flnfoLevelld* parameter is `GetFileExInfoStandard` then this parameter points to a `WIN32_FILE_ATTRIBUTE_DATA` structure.

Remarks

This wrapper calls the `GetFileAttributesTransacted` function.

GetHandle

Returns the transaction handle.

```
HANDLE GetHandle() const;
```

Return Value

Returns the transaction handle for a class. Returns NULL if the `CAtlTransactionManager` is not attached to a handle.

Remarks

IsFallback

Determines whether the fallback calls are enabled.

```
BOOL IsFallback() const;
```

Return Value

Returns TRUE if the class supports fallback calls. FALSE otherwise.

Remarks

m_bFallback

TRUE if the fallback is supported; FALSE otherwise.

```
BOOL m_bFallback;
```

Remarks

m_hTransaction

The transaction handle.

```
HANDLE m_hTransaction;
```

Remarks

MoveFile

Moves an existing file or a directory, including its children, as a transacted operation.

```
inline BOOL MoveFile(LPCTSTR lpOldFileName, LPCTSTR lpNewFileName);
```

Parameters

lpOldFileName

The current name of the existing file or directory on the local computer.

lpNewFileName

The new name for the file or directory. This name must not already exist. A new file may be on a different file system or drive. A new directory must be on the same drive.

Remarks

This wrapper calls the `MoveFileTransacted` function.

RegCreateKeyEx

Creates the specified registry key and associates it with a transaction. If the key already exists, the function opens it.

```
inline LSTATUS RegCreateKeyEx(  
    HKEY hKey,  
    LPCTSTR lpSubKey,  
    DWORD dwReserved,  
    LPTSTR lpClass,  
    DWORD dwOptions,  
    REGSAM samDesired,  
    CONST LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
    PHKEY phkResult,  
    LPDWORD lpdwDisposition);
```

Parameters

hKey

A handle to an open registry key.

lpSubKey

The name of a subkey that this function opens or creates.

dwReserved

This parameter is reserved and must be zero.

lpClass

The user-defined class of this key. This parameter may be ignored. This parameter can be NULL.

dwOptions

This parameter can be one of the following values: REG_OPTION_BACKUP_RESTORE, REG_OPTION_NON_VOLATILE, or REG_OPTION_VOLATILE.

samDesired

A mask that specifies the access rights for the key.

lpSecurityAttributes

Pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes. If *lpSecurityAttributes* is NULL, the handle cannot be inherited.

phkResult

A pointer to a variable that receives a handle to the opened or created key. If the key is not one of the predefined registry keys, call the `RegCloseKey` function after you have finished using the handle.

lpdwDisposition

A pointer to a variable that receives one of the following disposition values: REG_CREATED_NEW_KEY or REG_OPENED_EXISTING_KEY.

Return Value

If the function succeeds, the return value is ERROR_SUCCESS. If the function fails, the return value is a nonzero error code defined in Winerror.h.

Remarks

This wrapper calls the `RegCreateKeyTransacted` function.

RegDeleteKey

Deletes a subkey and its values from the specified platform-specific view of the registry as a transacted operation.

```
inline LSTATUS RegDeleteKeyEx(HKEY hKey, LPCTSTR lpSubKey);
```

Parameters

PARAMETER	DESCRIPTION
<i>hKey</i>	A handle to an open registry key.
<i>lpSubKey</i>	The name of the key to be deleted.

Return Value

If the function succeeds, the return value is ERROR_SUCCESS. If the function fails, the return value is a nonzero error code defined in Winerror.h.

Remarks

This wrapper calls the `RegDeleteKeyTransacted` function.

RegOpenKeyEx

Opens the specified registry key and associates it with a transaction.

```
inline LSTATUS RegOpenKeyEx(  
    HKEY hKey,  
    LPCTSTR lpSubKey,  
    DWORD ulOptions,  
    REGSAM samDesired,  
    PHKEY phkResult);
```

Parameters

hKey

A handle to an open registry key.

lpSubKey

The name of the registry subkey to be opened.

ulOptions

This parameter is reserved and must be zero.

samDesired

A mask that specifies the access rights for the key.

phkResult

A pointer to a variable that receives a handle to the opened or created key. If the key is not one of the predefined registry keys, call the `RegCloseKey` function after you have finished using the handle.

Return Value

If the function succeeds, the return value is `ERROR_SUCCESS`. If the function fails, the return value is a nonzero error code defined in `Winerror.h`.

Remarks

This wrapper calls the `RegOpenKeyTransacted` function.

Rollback

Requests that the transaction be rolled back.

```
inline BOOL Rollback();
```

Return Value

TRUE if successful; otherwise FALSE.

Remarks

This wrapper calls the `RollbackTransaction` function.

SetFileAttributes

Sets the attributes for a file or directory as a transacted operation.

```
inline BOOL SetFileAttributes(LPCTSTR lpFileName, DWORD dwAttributes);
```

Parameters

lpFileName

The name of the file or directory.

dwAttributes

The file attributes to set for the file. For more information, see [SetFileAttributesTransacted](#).

Remarks

This wrapper calls the `SetFileAttributesTransacted` function.

See also

[ATL COM Desktop Components](#)

CAtlWinModule Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides support for ATL windowing components.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CAtlWinModule : public _ATL_WIN_MODULE
```

Members

Public Constructors

NAME	DESCRIPTION
CAtlWinModule::CAtlWinModule	The constructor.
CAtlWinModule::~~CAtlWinModule	The destructor.

Public Methods

NAME	DESCRIPTION
CAtlWinModule::AddCreateWndData	Adds a data object.
CAtlWinModule::ExtractCreateWndData	Returns a pointer to the window module data object.

Remarks

This class provides support for all ATL classes which require windowing features.

Inheritance Hierarchy

[_ATL_WIN_MODULE](#)

CAtlWinModule

Requirements

Header: atlbase.h

CAtlWinModule::AddCreateWndData

This method initializes and adds an `_AtlCreateWndData` structure.

```
void AddCreateWndData(_AtlCreateWndData* pData, void* pObject);
```

Parameters

pData

Pointer to the `_AtlCreateWndData` structure to be initialized and added to the current module.

pObject

Pointer to an object's **this** pointer.

Remarks

This method calls [AtlWinModuleAddCreateWndData](#) which initializes an `_AtlCreateWndData` structure. This structure will store the **this** pointer, used to obtain the class instance in window procedures.

CAtlWinModule::CAtlWinModule

The constructor.

```
CAtlWinModule();
```

Remarks

If initialization fails, an **EXCEPTION_NONCONTINUABLE** exception is raised.

CAtlWinModule::~~CAtlWinModule

The destructor.

```
~CAtlWinModule();
```

Remarks

Frees all allocated resources.

CAtlWinModule::ExtractCreateWndData

This method returns a pointer to an `_AtlCreateWndData` structure.

```
void* ExtractCreateWndData();
```

Return Value

Returns a pointer to the `_AtlCreateWndData` structure previously added with [CAtlWinModule::AddCreateWndData](#), or NULL if no object is available.

See also

[_ATL_WIN_MODULE](#)

[Class Overview](#)

[Module Classes](#)

CAutoPtr Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

This class represents a smart pointer object.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <typename T>
class CAutoPtr
```

Parameters

T

The pointer type.

Members

Public Constructors

NAME	DESCRIPTION
CAutoPtr::CAutoPtr	The constructor.
CAutoPtr::~~CAutoPtr	The destructor.

Public Methods

NAME	DESCRIPTION
CAutoPtr::Attach	Call this method to take ownership of an existing pointer.
CAutoPtr::Detach	Call this method to release ownership of a pointer.
CAutoPtr::Free	Call this method to delete an object pointed to by a <code>CAutoPtr</code> .

Public Operators

NAME	DESCRIPTION
CAutoPtr::operator T*	The cast operator.
CAutoPtr::operator =	The assignment operator.
CAutoPtr::operator ->	The pointer-to-member operator.

Public Data Members

NAME	DESCRIPTION
CAutoPtr::m_p	The pointer data member variable.

Remarks

This class provides methods for creating and managing a smart pointer, which will help protect against memory leaks by automatically freeing resources when it falls out of scope.

Further, `CAutoPtr`'s copy constructor and assignment operator transfer ownership of the pointer, copying the source pointer to the destination pointer and setting the source pointer to NULL. It is therefore impossible to have two `CAutoPtr` objects each storing the same pointer, and this reduces the possibility of deleting the same pointer twice.

`CAutoPtr` also simplifies the creation of collections of pointers. Instead of deriving a collection class and overriding the destructor, it's simpler to make a collection of `CAutoPtr` objects. When the collection is deleted, the `CAutoPtr` objects will go out of scope and automatically delete themselves.

[CHeapPtr](#) and variants work in the same way as `CAutoPtr`, except that they allocate and free memory using different heap functions instead of the C++ **new** and **delete** operators. [CAutoVectorPtr](#) is similar to `CAutoPtr`, the only difference being that it uses **vector new[]** and **vector delete[]** to allocate and free memory.

See also [CAutoPtrArray](#) and [CAutoPtrList](#) when arrays or lists of smart pointers are required.

Requirements

Header: `atlbase.h`

Example

```

// A simple class for demonstration purposes

class MyClass
{
    int iA;
    int iB;
public:
    MyClass(int a, int b);
    void Test();
};

MyClass::MyClass(int a, int b)
{
    iA = a;
    iB = b;
}

void MyClass::Test()
{
    ATLASSTERT(iA == iB);
}

// A simple function

void MyFunction(MyClass* c)
{
    c->Test();
}

int UseMyClass()
{
    // Create an object of MyClass.
    MyClass *pMyC = new MyClass(1, 1);

    // Create a CAutoPtr object and have it take
    // over the pMyC pointer by calling Attach.
    CAutoPtr<MyClass> apMyC;
    apMyC.Attach(pMyC);

    // The overloaded -> operator allows the
    // CAutoPtr object to be used in place of the pointer.
    apMyC->Test();

    // Assign a second CAutoPtr, using the = operator.
    CAutoPtr<MyClass> apMyC2;
    apMyC2 = apMyC;

    // The casting operator allows the
    // object to be used in place of the pointer.
    MyFunction(pMyC);
    MyFunction(apMyC2);

    // Detach breaks the association, so after this
    // call, pMyC is controlled only by apMyC.
    apMyC2.Detach();

    // CAutoPtr destroys any object it controls when it
    // goes out of scope, so apMyC destroys the object
    // pointed to by pMyC here.
    return 0;
}

```

CAutoPtr::Attach

Call this method to take ownership of an existing pointer.

```
void Attach(T* p) throw();
```

Parameters

p

The `CAutoPtr` object will take ownership of this pointer.

Remarks

When a `CAutoPtr` object takes ownership of a pointer, it will automatically delete the pointer and any allocated data when it goes out of scope. If `CAutoPtr::Detach` is called, the programmer is again given responsibility for freeing any allocated resources.

In debug builds, an assertion failure will occur if the `CAutoPtr::m_p` data member currently points to an existing value; that is, it is not equal to `NULL`.

Example

See the example in the [CAutoPtr Overview](#).

CAutoPtr::CAutoPtr

The constructor.

```
CAutoPtr() throw();  
explicit CAutoPtr(T* p) throw();  
  
template<typename TSrc>  
CAutoPtr(CAutoPtr<TSrc>& p) throw();  
  
template<>  
CAutoPtr(CAutoPtr<T>& p) throw();
```

Parameters

p

An existing pointer.

TSrc

The type being managed by another `CAutoPtr`, used to initialize the current object.

Remarks

The `CAutoPtr` object can be created using an existing pointer, in which case it transfers ownership of the pointer.

Example

See the example in the [CAutoPtr Overview](#).

CAutoPtr::~CAutoPtr

The destructor.

```
~CAutoPtr() throw();
```

Remarks

Frees any allocated resources. Calls `CAutoPtr::Free`.

CAutoPtr::Detach

Call this method to release ownership of a pointer.

```
T* Detach() throw();
```

Return Value

Returns a copy of the pointer.

Remarks

Releases ownership of a pointer, sets the [CAutoPtr::m_p](#) data member variable to NULL, and returns a copy of the pointer. After calling `Detach`, it is up to the programmer to free any allocated resources over which the `CAutoPtr` object may have previously assumed responsibility.

Example

See the example in the [CAutoPtr Overview](#).

CAutoPtr::Free

Call this method to delete an object pointed to by a `CAutoPtr`.

```
void Free() throw();
```

Remarks

The object pointed to by the `CAutoPtr` is freed, and the [CAutoPtr::m_p](#) data member variable is set to NULL.

CAutoPtr::m_p

The pointer data member variable.

```
T* m_p;
```

Remarks

This member variable holds the pointer information.

CAutoPtr::operator =

The assignment operator.

```
template<>
CAutoPtr<T>& operator= (CAutoPtr<T>& p);

template<typename TSrc>
CAutoPtr<T>& operator= (CAutoPtr<TSrc>& p);
```

Parameters

p

A pointer.

TSrc

A class type.

Return Value

Returns a reference to a **CAutoPtr< T >**.

Remarks

The assignment operator detaches the `CAutoPtr` object from any current pointer and attaches the new pointer, *p*, in its place.

Example

See the example in the [CAutoPtr Overview](#).

CAutoPtr::operator ->

The pointer-to-member operator.

```
T* operator->() const throw();
```

Return Value

Returns the value of the `CAutoPtr::m_p` data member variable.

Remarks

Use this operator to call a method in a class pointed to by the `CAutoPtr` object. In debug builds, an assertion failure will occur if the `CAutoPtr` points to NULL.

Example

See the example in the [CAutoPtr Overview](#).

CAutoPtr::operator T*

The cast operator.

```
operator T* () const throw();
```

Return Value

Returns a pointer to the object data type defined in the class template.

Example

See the example in the [CAutoPtr Overview](#).

See also

[CHepPtr Class](#)

[CAutoVectorPtr Class](#)

[Class Overview](#)

CAutoPtrArray Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods useful when constructing an array of smart pointers.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <typename E>
class CAutoPtrArray : public CATlArray<
    ATL::CAutoPtr<E>,
    CAutoPtrElementTraits<E>>
```

Parameters

E

The pointer type.

Members

Public Constructors

NAME	DESCRIPTION
CAutoPtrArray::CAutoPtrArray	The constructor.

Remarks

This class provides a constructor and derives methods from [CATlArray](#) and [CAutoPtrElementTraits](#) to aid the creation of a collection class object storing smart pointers.

For more information, see [ATL Collection Classes](#).

Inheritance Hierarchy

CATlArray

CAutoPtrArray

Requirements

Header: atlcoll.h

CAutoPtrArray::CAutoPtrArray

The constructor.

```
CAutoPtrArray() throw();
```

Remarks

Initializes the smart pointer array.

See also

[CAtlArray Class](#)

[CAutoPtrElementTraits Class](#)

[CAutoPtrList Class](#)

[Class Overview](#)

CAutoPtrElementTraits Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods, static functions, and typedefs useful when creating collections of smart pointers.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<typename T>
class CAutoPtrElementTraits
: public CDefaultElementTraits<ATL::CAutoPtr<T>>
```

Parameters

T

The pointer type.

Members

Public Typedefs

NAME	DESCRIPTION
CAutoPtrElementTraits::INARGTYPE	The data type to use for adding elements to the collection class object.
CAutoPtrElementTraits::OUTARGTYPE	The data type to use for retrieving elements from the collection class object.

Remarks

This class provides methods, static functions, and typedefs for aiding the creation of collection class objects containing smart pointers. The classes [CAutoPtrArray](#) and [CAutoPtrList](#) derive from `CAutoPtrElementTraits`. If building a collection of smart pointers that requires vector new and delete operators, use [CAutoVectorPtrElementTraits](#) instead.

Inheritance Hierarchy

[CDefaultCompareTraits](#)

[CDefaultHashTraits](#)

[CElementTraitsBase](#)

[CDefaultElementTraits](#)

`CAutoPtrElementTraits`

Requirements

Header: atlcoll.h

CAutoPtrElementTraits::INARGTYPE

The data type to use for adding elements to the collection class object.

```
typedef CAutoPtr<T>& INARGTYPE;
```

CAutoPtrElementTraits::OUTARGTYPE

The data type to use for retrieving elements from the collection class object.

```
typedef T *& OUTARGTYPE;
```

See also

[CDefaultElementTraits Class](#)

[Class Overview](#)

CAutoPtrList Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods useful when constructing a list of smart pointers.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<typename E>
class CAutoPtrList :
    public CATLList<ATL::CAutoPtr<E>, CAutoPtrElementTraits<E>>
```

Parameters

E

The pointer type.

Members

Public Constructors

NAME	DESCRIPTION
CAutoPtrList::CAutoPtrList	The constructor.

Remarks

This class provides a constructor and derives methods from [CATLList](#) and [CAutoPtrElementTraits](#) to aid the creation of a list object storing smart pointers. The class [CAutoPtrArray](#) provides a similar function for an array object.

For more information, see [ATL Collection Classes](#).

Inheritance Hierarchy

[CATLList](#)

CAutoPtrList

Requirements

Header: atlcoll.h

CAutoPtrList::CAutoPtrList

The constructor.

```
CAutoPtrList(UINT nBlockSize = 10) throw();
```

Parameters

nBlockSize

The block size, with a default of 10.

Remarks

The block size is a measure of the amount of memory allocated when a new element is required. Larger block sizes reduce calls to memory allocation routines, but use more resources.

See also

[CAtlList Class](#)

[CAutoPtrElementTraits Class](#)

[Class Overview](#)

CAutoRevertImpersonation Class

4/1/2019 • 2 minutes to read • [Edit Online](#)

This class reverts [CAccessToken](#) objects to a nonimpersonating state when it goes out of scope.

Syntax

```
class CAutoRevertImpersonation
```

Members

Public Constructors

NAME	DESCRIPTION
CAutoRevertImpersonation::CAutoRevertImpersonation	Constructs an <code>CAutoRevertImpersonation</code> object
CAutoRevertImpersonation::~~CAutoRevertImpersonation	Destroys the object and reverts access token impersonation.

Public Methods

NAME	DESCRIPTION
CAutoRevertImpersonation::Attach	Automates the impersonation reversion of an access token.
CAutoRevertImpersonation::Detach	Cancels the automatic impersonation reversion.
CAutoRevertImpersonation::GetAccessToken	Retrieves the access token current associated with this object.

Remarks

An [access token](#) is an object that describes the security context of a process or thread and is allocated to each user logged onto a Windows NT or Windows 2000 system. These access tokens can be represented with the

`CAccessToken` class.

It is sometimes necessary to impersonate access tokens. This class is provided as a convenience, but it does not perform the impersonation of access tokens; it only performs the automatic reversion to a nonimpersonated state. This is because token access impersonation can be performed several different ways.

For an introduction to the access control model in Windows, see [Access Control](#) in the Windows SDK.

Requirements

Header: atlsecurity.h

CAutoRevertImpersonation::Attach

Automates the impersonation reversion of an access token.

```
void Attach(const CAccessToken* pAT) throw();
```

Parameters

pAT

The address of the [CAccessToken](#) object to be reverted automatically

Remarks

This method should only be used if the [CAutoRevertImpersonation](#) object was created with a NULL

`CAccessToken` pointer, or if [Detach](#) was called previously. For simple cases, it is not necessary to use this method.

CAutoRevertImpersonation::CAutoRevertImpersonation

Constructs a `CAutoRevertImpersonation` object.

```
CAutoRevertImpersonation(const CAccessToken* pAT) throw();
```

Parameters

pAT

The address of the [CAccessToken](#) object to be reverted automatically.

Remarks

The actual impersonation of the access token should be performed separately from and preferably before the creation of a `CAutoRevertImpersonation` object. This impersonation will be reverted automatically when the

`CAutoRevertImpersonation` object goes out of scope.

CAutoRevertImpersonation::~CAutoRevertImpersonation

Destroys the object and reverts access token impersonation.

```
~CAutoRevertImpersonation() throw();
```

Remarks

Reverts any impersonation currently in effect for the [CAccessToken](#) object provided either at construction or through the [Attach](#) method. If no `CAccessToken` is associated, the destructor has no effect.

CAutoRevertImpersonation::Detach

Cancels the automatic impersonation reversion.

```
const CAccessToken* Detach() throw();
```

Return Value

The address of the previously associated [CAccessToken](#), or NULL if no association existed.

Remarks

Calling **Detach** prevents the `CAutoRevertImpersonation` object from reverting any impersonation currently in effect for the [CAccessToken](#) object associated with this object. `CAutoRevertImpersonation` can then be destroyed with no effect or reassociated to the same or another `CAccessToken` object using [Attach](#).

CAutoRevertImpersonation::GetAccessToken

Retrieves the access token current associated with this object.

```
const CAccessToken* GetAccessToken() throw();
```

Return Value

The address of the previously associated [CAccessToken](#), or NULL if no association existed.

Remarks

If this method is called for the purposes that include the reversion of an impersonation of the `CAccessToken` object, the [Detach](#) method should be used instead.

See also

[ATLSecurity Sample](#)

[Access Tokens](#)

[Class Overview](#)

CAutoVectorPtr Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

This class represents a smart pointer object using vector new and delete operators.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<typename T>
class CAutoVectorPtr
```

Parameters

T

The pointer type.

Members

Public Constructors

NAME	DESCRIPTION
CAutoVectorPtr::CAutoVectorPtr	The constructor.
CAutoVectorPtr::~~CAutoVectorPtr	The destructor.

Public Methods

NAME	DESCRIPTION
CAutoVectorPtr::Allocate	Call this method to allocate the memory required by the array of objects pointed to by <code>CAutoVectorPtr</code> .
CAutoVectorPtr::Attach	Call this method to take ownership of an existing pointer.
CAutoVectorPtr::Detach	Call this method to release ownership of a pointer.
CAutoVectorPtr::Free	Call this method to delete an object pointed to by a <code>CAutoVectorPtr</code> .

Public Operators

NAME	DESCRIPTION
CAutoVectorPtr::operator T *	The cast operator.
CAutoVectorPtr::operator =	The assignment operator.

Public Data Members

NAME	DESCRIPTION
CAutoVectorPtr::m_p	The pointer data member variable.

Remarks

This class provides methods for creating and managing a smart pointer, which will help protect against memory leaks by automatically freeing resources when it falls out of scope. `CAutoVectorPtr` is similar to `CAutoPtr`, the only difference being that `CAutoVectorPtr` uses [vector new\[\]](#) and [vector delete\[\]](#) to allocate and free memory instead of the C++ **new** and **delete** operators. See [CAutoVectorPtrElementTraits](#) if collection classes of `CAutoVectorPtr` are required.

See [CAutoPtr](#) for an example of using a smart pointer class.

Requirements

Header: `atlbases.h`

CAutoVectorPtr::Allocate

Call this method to allocate the memory required by the array of objects pointed to by `CAutoVectorPtr`.

```
bool Allocate(size_t nElements) throw();
```

Parameters

nElements

The number of elements in the array.

Return Value

Returns true if the memory is successfully allocated, false on failure.

Remarks

In debug builds, an assertion failure will occur if the [CAutoVectorPtr::m_p](#) member variable currently points to an existing value; that is, it is not equal to NULL.

CAutoVectorPtr::Attach

Call this method to take ownership of an existing pointer.

```
void Attach(T* p) throw();
```

Parameters

p

The `CAutoVectorPtr` object will take ownership of this pointer.

Remarks

When a `CAutoVectorPtr` object takes ownership of a pointer, it will automatically delete the pointer and any allocated data when it goes out of scope. If [CAutoVectorPtr::Detach](#) is called, the programmer is again given responsibility for freeing any allocated resources.

In debug builds, an assertion failure will occur if the [CAutoVectorPtr::m_p](#) member variable currently points to an

existing value; that is, it is not equal to NULL.

CAutoVectorPtr::CAutoVectorPtr

The constructor.

```
CAutoVectorPtr() throw();  
explicit CAutoVectorPtr(T* p) throw();  
CAutoVectorPtr(CAutoVectorPtr<T>& p) throw();
```

Parameters

p

An existing pointer.

Remarks

The `CAutoVectorPtr` object can be created using an existing pointer, in which case it transfers ownership of the pointer.

CAutoVectorPtr::~~CAutoVectorPtr

The destructor.

```
~CAutoVectorPtr() throw();
```

Remarks

Frees any allocated resources. Calls [CAutoVectorPtr::Free](#).

CAutoVectorPtr::Detach

Call this method to release ownership of a pointer.

```
T* Detach() throw();
```

Return Value

Returns a copy of the pointer.

Remarks

Releases ownership of a pointer, sets the [CAutoVectorPtr::m_p](#) member variable to NULL, and returns a copy of the pointer. After calling `Detach`, it is up to the programmer to free any allocated resources over which the `CAutoVectorPtr` object may have previously assumed responsibility.

CAutoVectorPtr::Free

Call this method to delete an object pointed to by a `CAutoVectorPtr`.

```
void Free() throw();
```

Remarks

The object pointed to by the `CAutoVectorPtr` is freed, and the [CAutoVectorPtr::m_p](#) member variable is set to NULL.

CAutoVectorPtr::m_p

The pointer data member variable.

```
T* m_p;
```

Remarks

This member variable holds the pointer information.

CAutoVectorPtr::operator =

The assignment operator.

```
CAutoVectorPtr<T>& operator= (CAutoVectorPtr<T>& p) throw();
```

Parameters

p

A pointer.

Return Value

Returns a reference to a **CAutoVectorPtr< T >**.

Remarks

The assignment operator detaches the `CAutoVectorPtr` object from any current pointer and attaches the new pointer, *p*, in its place.

CAutoVectorPtr::operator T *

The cast operator.

```
operator T*() const throw();
```

Remarks

Returns a pointer to the object data type defined in the class template.

See also

[CAutoPtr Class](#)

[Class Overview](#)

CAutoVectorPtrElementTraits Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods, static functions, and typedefs useful when creating collections of smart pointers using vector new and delete operators.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <typename T>
class CAutoVectorPtrElementTraits :
    public CDefaultElementTraits<ATL::CAutoVectorPtr<T>>
```

Parameters

T

The pointer type.

Members

Public Typedefs

NAME	DESCRIPTION
CAutoVectorPtrElementTraits::INARGTYPE	The data type to use for adding elements to the collection class object.
CAutoVectorPtrElementTraits::OUTARGTYPE	The data type to use for retrieving elements from the collection class object.

Remarks

This class provides methods, static functions, and typedefs for aiding the creation of collection class objects containing smart pointers. Unlike [CAutoPtrElementTraits](#), this class uses vector new and delete operators.

Inheritance Hierarchy

[CDefaultCompareTraits](#)

[CDefaultHashTraits](#)

[CElementTraitsBase](#)

[CDefaultElementTraits](#)

[CAutoVectorPtrElementTraits](#)

Requirements

Header: atlcoll.h

CAutoVectorPtrElementTraits::INARGTYPE

The data type to use for adding elements to the collection class object.

```
typedef CAutoVectorPtr<T>& INARGTYPE;
```

CAutoVectorPtrElementTraits::OUTARGTYPE

The data type to use for retrieving elements from the collection class object.

```
typedef T*& OUTARGTYPE;
```

See also

[CDefaultElementTraits Class](#)

[CAutoVectorPtr Class](#)

[Class Overview](#)

CxDialogImpl Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

This class implements a dialog box (modal or modeless) that hosts ActiveX controls.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class T, class TBase = CWindow>
class ATL_NO_VTABLE CxDialogImpl : public CDialogImplBaseT<TBase>
```

Parameters

T

Your class, derived from `CxDialogImpl`.

TBase

The base window class for `CDialogImplBaseT`.

Members

Public Methods

NAME	DESCRIPTION
CxDialogImpl::AdviseSinkMap	Call this method to advise or unadvise all entries in the object's sink map event map.
CxDialogImpl::Create	Call this method to create a modeless dialog box.
CxDialogImpl::DestroyWindow	Call this method to destroy a modeless dialog box.
CxDialogImpl::DoModal	Call this method to create a modal dialog box.
CxDialogImpl::EndDialog	Call this method to destroy a modal dialog box.
CxDialogImpl::GetDialogProc	Call this method to get a pointer to the <code>DialogProc</code> callback function.
CxDialogImpl::GetIDD	Call this method to get the dialog template resource ID
CxDialogImpl::IsDialogMessage	Call this method to determine whether a message is intended for this dialog box and, if it is, process the message.

Protected Data Members

NAME	DESCRIPTION
CAxDialogImpl::m_bModal	A variable that exists only in debug builds and is set to true if the dialog box is modal.

Remarks

[CAxDialogImpl](#) allows you to create a modal or modeless dialog box. [CAxDialogImpl](#) provides the dialog box procedure, which uses the default message map to direct messages to the appropriate handlers.

[CAxDialogImpl](#) derives from [CDialogImplBaseT](#), which in turn derives from *TBase* (by default, [CWindow](#)) and [CMessageMap](#).

Your class must define an IDD member that specifies the dialog template resource ID. For example, adding an ATL Dialog object using the **Add Class** dialog box automatically adds the following line to your class:

```
enum { IDD = IDD_MYDLG };
```

where [MyDialog](#) is the **Short name** entered in the ATL Dialog Wizard.

See [Implementing a Dialog Box](#) for more information.

Note that an ActiveX control on a modal dialog box created with [CAxDialogImpl](#) will not support accelerator keys. To support accelerator keys on a dialog box created with [CAxDialogImpl](#), create a modeless dialog box and, using your own message loop, use [CAxDialogImpl::IsDialogMessage](#) after getting a message from the queue to handle an accelerator key.

For more information on [CAxDialogImpl](#), see [ATL Control Containment FAQ](#).

Inheritance Hierarchy

[CMessageMap](#)

[TBase](#)

[CWindowImplRoot](#)

[CDialogImplBaseT](#)

[CAxDialogImpl](#)

Requirements

Header: atlwin.h

CAxDialogImpl::AdviseSinkMap

Call this method to advise or unadvise all entries in the object's sink map event map.

```
HRESULT AdviseSinkMap(bool bAdvise);
```

Parameters

bAdvise

Set to true if all sink entries are to be advised; false if all sink entries are to be unadvised.

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure.

CxDialogImpl::Create

Call this method to create a modeless dialog box.

```
HWND Create(HWND hWndParent, LPARAM dwInitParam = NULL);
HWND Create(HWND hWndParent, RECT&, LPARAM dwInitParam = NULL);
```

Parameters

hWndParent

[in] The handle to the owner window.

dwInitParam

[in] Specifies the value to pass to the dialog box in the *lParam* parameter of the `WM_INITDIALOG` message.

RECT&

This parameter is not used. This parameter is passed in by `CComControl`.

Return Value

The handle to the newly created dialog box.

Remarks

This dialog box is automatically attached to the `CxDialogImpl` object. To create a modal dialog box, call [DoModal](#).

The second override is provided only so dialog boxes can be used with [CComControl](#).

CxDialogImpl::DestroyWindow

Call this method to destroy a modeless dialog box.

```
BOOL DestroyWindow();
```

Return Value

TRUE if the window is successfully destroyed; otherwise FALSE.

Remarks

Do not call `DestroyWindow` to destroy a modal dialog box. Call [EndDialog](#) instead.

CxDialogImpl::DoModal

Call this method to create a modal dialog box.

```
INT_PTR DoModal(
    HWND hWndParent = ::GetActiveWindow(),
    LPARAM dwInitParam = NULL);
```

Parameters

hWndParent

[in] The handle to the owner window. The default value is the return value of the [GetActiveWindow](#) Win32 function.

dwInitParam

[in] Specifies the value to pass to the dialog box in the *lParam* parameter of the WM_INITDIALOG message.

Return Value

If successful, the value of the *nRetCode* parameter specified in the call to [EndDialog](#); otherwise, -1.

Remarks

This dialog box is automatically attached to the `CxDialogImpl` object.

To create a modeless dialog box, call [Create](#).

CxDialogImpl::EndDialog

Call this method to destroy a modal dialog box.

```
BOOL EndDialog(int nRetCode);
```

Parameters

nRetCode

[in] The value to be returned by [DoModal](#).

Return Value

TRUE if the dialog box is destroyed; otherwise, FALSE.

Remarks

`EndDialog` must be called through the dialog box procedure. After the dialog box is destroyed, Windows uses the value of *nRetCode* as the return value for `DoModal`, which created the dialog box.

NOTE

Do not call `EndDialog` to destroy a modeless dialog box. Call [DestroyWindow](#) instead.

CxDialogImpl::GetDialogProc

Call this method to get a pointer to the `DialogProc` callback function.

```
virtual DLGPROC GetDialogProc();
```

Return Value

Returns a pointer to the `DialogProc` callback function.

Remarks

The `DialogProc` function is an application-defined callback function.

CxDialogImpl::GetIDD

Call this method to get the dialog template resource ID.

```
int GetIDD();
```

Return Value

Returns the dialog template resource ID.

CxDialogImpl::IsDialogMessage

Call this method to determine whether a message is intended for this dialog box and, if it is, process the message.

```
BOOL IsDialogMessage(LPMSG pMsg);
```

Parameters

pMsg

Pointer to a [MSG](#) structure that contains the message to be checked.

Return Value

Returns TRUE if the message has been processed, FALSE otherwise.

Remarks

This method is intended to be called from within a message loop.

CxDialogImpl::m_bModal

A variable that exists only in debug builds and is set to true if the dialog box is modal.

```
bool m_bModal;
```

See also

[CDialogImpl Class](#)

[Class Overview](#)

CxWindow Class

4/1/2019 • 6 minutes to read • [Edit Online](#)

This class provides methods for manipulating a window hosting an ActiveX control.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CxWindow : public CWindow
```

Members

Methods

AttachControl	Attaches an existing ActiveX control to the <code>CxWindow</code> object.
CxWindow	Constructs a <code>CxWindow</code> object.
CreateControl	Creates an ActiveX control, initializes it, and hosts it in the <code>CxWindow</code> window.
CreateControlEx	Creates an ActiveX control and retrieves an interface pointer (or pointers) from the control.
GetWndClassName	(Static) Retrieves the predefined class name of the <code>CxWindow</code> object.
QueryControl	Retrieves the <code>IUnknown</code> of the hosted ActiveX control.
QueryHost	Retrieves the <code>IUnknown</code> pointer of the <code>CxWindow</code> object.
SetExternalDispatch	Sets the external dispatch interface used by the <code>CxWindow</code> object.
SetExternalUIHandler	Sets the external <code>IDocHostUIHandler</code> interface used by the <code>CxWindow</code> object.

Operators

operator =	Assigns an HWND to an existing <code>CxWindow</code> object.

Remarks

This class provides methods for manipulating a window that hosts an ActiveX control. The hosting is provided by " **AtlAxWin80**", which is wrapped by `CAxWindow` .

Class `CAxWindow` is implemented as a specialization of the `CAxWindowT` class. This specialization is declared as:

```
typedef CAxWindowT<CWindow> CAxWindow;
```

If you need to change the base class, you can use `CAxWindowT` and specify the new base class as a template argument.

Requirements

Header: atlwin.h

CAxWindow::AttachControl

Creates a new host object if one isn't already present and attaches the specified control to the host.

```
HRESULT AttachControl(  
    IUnknown* pControl,  
    IUnknown** ppUnkContainer);
```

Parameters

pControl

[in] A pointer to the `IUnknown` of the control.

ppUnkContainer

[out] A pointer to the `IUnknown` of the host (the `AxWin` object).

Return Value

A standard HRESULT value.

Remarks

The control object being attached must be correctly initialized before calling `AttachControl` .

CAxWindow::CAxWindow

Constructs a `CAxWindow` object using an existing window object handle.

```
CAxWindow(HWND hWnd = NULL);
```

Parameters

hWnd

A handle to an existing window object.

CAxWindow::CreateControl

Creates an ActiveX control, initializes it, and hosts it in the specified window.

```

HRESULT CreateControl(
    LPCOLESTR lpszName,
    IStream* pStream = NULL,
    IUnknown** ppUnkContainer = NULL);

HRESULT CreateControl(
    DWORD dwResID,
    IStream* pStream = NULL,
    IUnknown** ppUnkContainer = NULL);

```

Parameters

lpszName

A pointer to a string to create the control. Must be formatted in one of the following ways:

- A ProgID such as "MSCAL.Calendar.7"
- A CLSID such as "{8E27C92B-1264-101C-8A2F-040224009C02}"
- A URL such as "<http://www.microsoft.com>"
- A reference to an Active document such as "file://\\Documents\\MyDoc.doc"
- A fragment of HTML such as "MSHTML:<HTML><BODY>This is a line of text</BODY></HTML>"

NOTE

"MSHTML:" must precede the HTML fragment so that it is designated as being an MSHTML stream. Only the ProgID and CLSID are supported in Windows Mobile platforms. Windows CE embedded platforms, other than Windows Mobile with support for CE IE support all types including ProgID, CLSID, URL, reference to active document, and fragment of HTML.

pStream

[in] A pointer to a stream that is used to initialize the properties of the control. Can be NULL.

ppUnkContainer

[out] The address of a pointer that will receive the `IUnknown` of the container. Can be NULL.

dwResID

The resource ID of an HTML resource. The WebBrowser control will be created and loaded with the specified resource.

Return Value

A standard HRESULT value.

Remarks

If the second version of this method is used, an HTML control is created and bound to the resource identified by *dwResID*.

This method gives you the same result as calling:

```
AtlAxCreateControlEx(lpszName, hWnd, pStream, NULL, NULL, GUID_NULL, NULL);
```

See [CAxWindow2T::CreateControlLic](#) to create, initialize, and host a licensed ActiveX control.

Example

See [Hosting ActiveX Controls Using ATL AXHost](#) for a sample that uses `CreateControl`.

CxWindow::CreateControlEx

Creates an ActiveX control, initializes it, and hosts it in the specified window.

```
HRESULT CreateControlEx(  
    LPCOLESTR lpszName,  
    IStream* pStream = NULL,  
    IUnknown** ppUnkContainer = NULL,  
    IUnknown** ppUnkControl = NULL,  
    REFIID iidSink = IID_NULL,  
    IUnknown* punkSink = NULL);  
  
HRESULT CreateControlEx(  
    DWORD dwResID,  
    IStream* pStream = NULL,  
    IUnknown** ppUnkContainer = NULL,  
    IUnknown** ppUnkControl = NULL,  
    REFIID iidSink = IID_NULL,  
    IUnknown* punkSink = NULL);
```

Parameters

lpszName

A pointer to a string to create the control. Must be formatted in one of the following ways:

- A ProgID such as "MSCAL.Calendar.7"
- A CLSID such as "{8E27C92B-1264-101C-8A2F-040224009C02}"
- A URL such as "<http://www.microsoft.com>"
- A reference to an Active document such as "file://\\Documents\\MyDoc.doc"
- A fragment of HTML such as "MSHTML:<HTML><BODY>This is a line of text</BODY></HTML>"

NOTE

"MSHTML:" must precede the HTML fragment so that it is designated as being an MSHTML stream. Only the ProgID and CLSID are supported in Windows Mobile platforms. Windows CE embedded platforms, other than Windows Mobile with support for CE IE support all types including ProgID, CLSID, URL, reference to active document, and fragment of HTML.

pStream

[in] A pointer to a stream that is used to initialize the properties of the control. Can be NULL.

ppUnkContainer

[out] The address of a pointer that will receive the `IUnknown` of the container. Can be NULL.

ppUnkControl

[out] The address of a pointer that will receive the `IUnknown` of the control. Can be NULL.

iidSink

[in] The interface identifier of an outgoing interface on the contained object. Can be IID_NULL.

punkSink

[in] A pointer to the `IUnknown` interface of the sink object to be connected to the connection point on the contained object specified by *iidSink*.

dwResID

[in] The resource ID of an HTML resource. The WebBrowser control will be created and loaded with the specified

resource.

Return Value

A standard HRESULT value.

Remarks

This method is similar to [CAxWindow::CreateControl](#), but unlike that method, `CreateControlEx` also allows you to receive an interface pointer to the newly created control and set up an event sink to receive events fired by the control.

See [CAxWindow2T::CreateControlLicEx](#) to create, initialize, and host a licensed ActiveX control.

Example

See [Hosting ActiveX Controls Using ATL AXHost](#) for a sample that uses `CreateControlEx`.

CAxWindow::GetWndClassName

Retrieves the name of the window class.

```
static LPCTSTR GetWndClassName();
```

Return Value

A pointer to a string containing the name of the window class that can host nonlicensed ActiveX controls.

CAxWindow::operator =

Assigns an HWND to an existing `CAxWindow` object.

```
CAxWindow<TBase>& operator=(HWND hWnd);
```

Parameters

hWnd

A handle to an existing window.

Return Value

Returns a reference to the current `CAxWindow` object.

CAxWindow::QueryControl

Retrieves the specified interface of the hosted control.

```
HRESULT QueryControl(REFIID iid, void** ppUnk);  
template <class Q>  
HRESULT QueryControl(Q** ppUnk);
```

Parameters

iid

[in] Specifies the IID of the control's interface.

ppUnk

[out] A pointer to the interface of the control. In the template version of this method, there is no need for a reference ID as long as a typed interface with an associated UUID is passed.

Q

[in] The interface that is being queried for.

Return Value

A standard HRESULT value.

CxWindow::QueryHost

Returns the specified interface of the host.

```
HRESULT QueryHost(REFIID iid, void** ppUnk);  
template <class Q>  
HRESULT QueryHost(Q** ppUnk);
```

Parameters

iid

[in] Specifies the IID of the control's interface.

ppUnk

[out] A pointer to the interface on the host. In the template version of this method, there is no need for a reference ID as long as a typed interface with an associated UUID is passed.

Q

[in] The interface that is being queried for.

Return Value

A standard HRESULT value.

Remarks

The interface of the host allows access to the underlying functionality of the window-hosting code, implemented by `AxWin`.

CxWindow::SetExternalDispatch

Sets the external dispatch interface for the `CxWindow` object.

```
HRESULT SetExternalDispatch(IDispatch* pDisp);
```

Parameters

pDisp

[in] A pointer to an `IDispatch` interface.

Return Value

A standard HRESULT value.

CxWindow::SetExternalUIHandler

Sets the external [IDocHostUIHandlerDispatch](#) interface for the `CxWindow` object.

```
HRESULT SetExternalUIHandler(IDocHostUIHandlerDispatch* pUIHandler);
```

Parameters

pUIHandler

[in] A pointer to an `IDocHostUIHandlerDispatch` interface.

Return Value

A standard HRESULT value.

Remarks

The external `IDocHostUIHandlerDispatch` interface is used by controls that query the host's site for the `IDocHostUIHandlerDispatch` interface. The WebBrowser control is one control that does this.

See also

[ATLCON Sample](#)

[CWindow Class](#)

[Composite Control Fundamentals](#)

[Class Overview](#)

[Control Containment FAQ](#)

CAxWindow2T Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

This class provides methods for manipulating a window that hosts an ActiveX control, and also has support for hosting licensed ActiveX controls.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class TBase = CWindow>
class CAxWindow2T :
public CAxWindowT<TBase>
```

Parameters

TBase

The class from which `CAxWindowT` derives.

Members

Public Constructors

NAME	DESCRIPTION
CAxWindow2T::CAxWindow2T	Constructs a <code>CAxWindow2T</code> object.

Public Methods

NAME	DESCRIPTION
CAxWindow2T::Create	Creates a host window.
CAxWindow2T::CreateControlLic	Creates a licensed ActiveX control, initializes it, and hosts it in the specified window.
CAxWindow2T::CreateControlLicEx	Creates a licensed ActiveX control, initializes it, hosts it in the specified window, and retrieves an interface pointer (or pointers) from the control.
CAxWindow2T::GetWndClassName	Static method that retrieves the name of the window class.

Public Operators

NAME	DESCRIPTION
CAxWindow2T::operator =	Assigns an HWND to an existing <code>CAxWindow2T</code> object.

Remarks

`CxWindow2T` provides methods for manipulating a window that hosts an ActiveX control. `CxWindow2T` also has support for hosting licensed ActiveX controls. The hosting is provided by "**AtlAxWinLic80**", which is wrapped by `CxWindow2T`.

Class `CxWindow2` is implemented as a specialization of the `CxWindow2T` class. This specialization is declared as:

```
typedef CxWindow2T <CWindow> CxWindow2;
```

NOTE

`CxWindowT` members are documented under [CAxWindow](#).

See [Hosting ActiveX Controls Using ATL AXHost](#) for a sample that uses the members of this class.

Inheritance Hierarchy

`TBase`

`CxWindowT`

`CxWindow2T`

Requirements

Header: atlwin.h

CAxWindow2T::CAxWindow2T

Constructs a `CxWindow2T` object.

```
CAxWindow2T(HWND hWnd = NULL) : CxWindowT<TBase>(hWnd)
```

Parameters

hWnd

A handle of an existing window.

CAxWindow2T::Create

Creates a host window.

```
HWND Create(  
    HWND hWndParent,  
    _U_RECT rect = NULL,  
    LPCTSTR szWindowName = NULL,  
    DWORD dwStyle = 0,  
    DWORD dwExStyle = 0,  
    _U_MENUorID MenuOrID = 0U,  
    LPVOID lpCreateParam = NULL);
```

Remarks

`CAxWindow2T::Create` calls `CWindow::Create` with the LPCTSTR *lpstrWndClass* parameter set to the window class that provides control hosting (`AtlAxWinLic80`).

See `CWindow::Create` for a description of the parameters and return value.

Note If 0 is used as the value for the *MenuOrID* parameter, it must be specified as 0U (the default value) to avoid a compiler error.

Example

See [Hosting ActiveX Controls Using ATL AXHost](#) for a sample that uses `CAXWindow2T::Create`.

CAXWindow2T::CreateControlLic

Creates a licensed ActiveX control, initializes it, and hosts it in the specified window.

```
HRESULT CreateControlLic(
    DWORD dwResID,
    IStream* pStream = NULL,
    IUnknown** ppUnkContainer = NULL,
    BSTR bstrLicKey = NULL);

HRESULT CreateControlLic(
    LPCOLESTR lpszName,
    IStream* pStream = NULL,
    IUnknown** ppUnkContainer = NULL,
    BSTR bstrLicKey = NULL);
```

Parameters

bstrLicKey

The license key for the control; NULL if creating a nonlicensed control.

Remarks

See [CAXWindow::CreateControl](#) for a description of the remaining parameters and return value.

Example

See [Hosting ActiveX Controls Using ATL AXHost](#) for a sample that uses `CAXWindow2T::CreateControlLic`.

CAXWindow2T::CreateControlLicEx

Creates a licensed ActiveX control, initializes it, hosts it in the specified window, and retrieves an interface pointer (or pointers) from the control.

```
HRESULT CreateControlLicEx(
    LPCOLESTR lpszName,
    IStream* pStream = NULL,
    IUnknown** ppUnkContainer = NULL,
    IUnknown** ppUnkControl = NULL,
    REFIID iidSink = IID_NULL,
    IUnknown* punkSink = NULL,
    BSTR bstrLicKey = NULL);

HRESULT CreateControlLicEx(
    DWORD dwResID,
    IStream* pStream = NULL,
    IUnknown** ppUnkContainer = NULL,
    IUnknown** ppUnkControl = NULL,
    REFIID iidSink = IID_NULL,
    IUnknown* punkSink = NULL,
    BSTR bstrLicKey = NULL);
```

Parameters

bstrLicKey

The license key for the control; NULL if creating a nonlicensed control.

Remarks

See [CAxWindow::CreateControlEx](#) for a description of the remaining parameters and return value.

Example

See [Hosting ActiveX Controls Using ATL AXHost](#) for a sample that uses `CAxWindow2T::CreateControlLicEx`.

CAxWindow2T::GetWndClassName

Retrieves the name of the window class.

```
static LPCTSTR GetWndClassName();
```

Return Value

A pointer to a string containing the name of the window class (`AtlAxWinLic80`) that can host licensed and nonlicensed ActiveX controls.

CAxWindow2T::operator =

Assigns an HWND to an existing `CAxWindow2T` object.

```
CAxWindow2T<TBase>& operator= (HWND hWnd);
```

Parameters

hWnd

A handle of an existing window.

See also

[Class Overview](#)

[Control Containment FAQ](#)

CBindStatusCallback Class

4/1/2019 • 9 minutes to read • [Edit Online](#)

This class implements the `IBindStatusCallback` interface.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class T,
    int nBindFlags = BINDF_ASYNCHRONOUS | BINDF_ASYNCSTORAGE | BINDF_GETNEWESTVERSION | BINDF_NOWRITECACHE>
class ATL_NO_VTABLE CBindStatusCallback : public CComObjectRootEx <T ::_ThreadModel::_ThreadModelNoCS>,
    public IBindStatusCallbackImpl<T>
```

Parameters

T

Your class containing the function that will be called as the data is received.

nBindFlags

Specifies the bind flags that are returned by [GetBindInfo](#). The default implementation sets the binding to be asynchronous, retrieves the newest version of the data/object, and does not store retrieved data in the disk cache.

Members

Public Constructors

NAME	DESCRIPTION
CBindStatusCallback::CBindStatusCallback	The constructor.
CBindStatusCallback::~~CBindStatusCallback	The destructor.

Public Methods

NAME	DESCRIPTION
CBindStatusCallback::Download	Static method that starts the download process, creates a <code>CBindStatusCallback</code> object, and calls <code>StartAsyncDownload</code> .
CBindStatusCallback::GetBindInfo	Called by the asynchronous moniker to request information on the type of bind to be created.
CBindStatusCallback::GetPriority	Called by the asynchronous moniker to get the priority of the bind operation. The ATL implementation returns <code>E_NOTIMPL</code> .

NAME	DESCRIPTION
CBindStatusCallback::OnDataAvailable	Called to provide data to your application as it becomes available. Reads the data, then calls the function passed to it to use the data.
CBindStatusCallback::OnLowResource	Called when resources are low. The ATL implementation returns S_OK.
CBindStatusCallback::OnObjectAvailable	Called by the asynchronous moniker to pass an object interface pointer to your application. The ATL implementation returns S_OK.
CBindStatusCallback::OnProgress	Called to indicate the progress of a data downloading process. The ATL implementation returns S_OK.
CBindStatusCallback::OnStartBinding	Called when binding is started.
CBindStatusCallback::OnStopBinding	Called when the asynchronous data transfer is stopped.
CBindStatusCallback::StartAsyncDownload	Initializes the bytes available and bytes read to zero, creates a push-type stream object from a URL, and calls OnDataAvailable every time data is available.

Public Data Members

NAME	DESCRIPTION
CBindStatusCallback::m_dwAvailableToRead	Number of bytes available to read.
CBindStatusCallback::m_dwTotalRead	Total number of bytes read.
CBindStatusCallback::m_pFunc	Pointer to the function called when data is available.
CBindStatusCallback::m_pT	Pointer to the object requesting the asynchronous data transfer.
CBindStatusCallback::m_spBindCtx	Pointer to the IBindCtx interface for the current bind operation.
CBindStatusCallback::m_spBinding	Pointer to the IBinding interface for the current bind operation.
CBindStatusCallback::m_spMoniker	Pointer to the IMoniker interface for the URL to use.
CBindStatusCallback::m_spStream	Pointer to the IStream interface for the data transfer.

Remarks

The [CBindStatusCallback](#) class implements the [IBindStatusCallback](#) interface. [IBindStatusCallback](#) must be implemented by your application so it can receive notifications from an asynchronous data transfer. The asynchronous moniker provided by the system uses [IBindStatusCallback](#) methods to send and receive information about the asynchronous data transfer to and from your object.

Typically, the [CBindStatusCallback](#) object is associated with a specific bind operation. For example, in the [ASYNC](#)

sample, when you set the URL property, it creates a `CBindStatusCallback` object in the call to `Download`:

```
STDMETHOD(put_URL)(BSTR newVal)
{
    HRESULT hResult = E_UNEXPECTED;

    ATLTRACE(_T("IATLAsync::put_URL\n"));
    m_bstrURL = newVal;

    if (::IsWindow(m_EditCtrl.m_hWnd))
    {
        ::SendMessage(m_EditCtrl.m_hWnd, WM_SETTEXT, 0, (LPARAM)_T(""));
        hResult = CBindStatusCallback<CATLAsync>::Download(this, &CATLAsync::OnData,
            m_bstrURL, m_spClientSite, FALSE);
    }

    return hResult;
}
```

The asynchronous moniker uses the callback function `OnData` to call your application when it has data. The asynchronous moniker is provided by the system.

Inheritance Hierarchy

`COMObjectRootBase`

`IBindStatusCallback`

[COMObjectRootEx](#)

`CBindStatusCallback`

Requirements

Header: atlctl.h

CBindStatusCallback::CBindStatusCallback

The constructor.

```
CBindStatusCallback();
```

Remarks

Creates an object to receive notifications concerning the asynchronous data transfer. Typically, one object is created for each bind operation.

The constructor also initializes `m_pT` and `m_pFunc` to NULL.

CBindStatusCallback::~CBindStatusCallback

The destructor.

```
~CBindStatusCallback();
```

Remarks

Frees all allocated resources.

CBindStatusCallback::Download

Creates a `CBindStatusCallback` object and calls `StartAsyncDownload` to start downloading data asynchronously from the specified URL.

```
static HRESULT Download(  
    T* pT,  
    ATL_PDATAAVAILABLE pFunc,  
    BSTR bstrURL,  
    IUnknown* pUnkContainer = NULL,  
    BOOL bRelative = FALSE);
```

Parameters

pT

[in] A pointer to the object requesting the asynchronous data transfer. The `CBindStatusCallback` object is templated on this object's class.

pFunc

[in] A pointer to the function that receives the data that is read. The function is a member of your object's class of type `T`. See [StartAsyncDownload](#) for syntax and an example.

bstrURL

[in] The URL to obtain data from. Can be any valid URL or file name. Cannot be NULL. For example:

```
CComBSTR mybstr = _T("http://somesite/data.htm")
```

pUnkContainer

[in] The `IUnknown` of the container. NULL by default.

bRelative

[in] A flag indicating whether the URL is relative or absolute. FALSE by default, meaning the URL is absolute.

Return Value

One of the standard HRESULT values.

Remarks

Every time data is available it is sent to the object through `OnDataAvailable`. `OnDataAvailable` reads the data and calls the function pointed to by *pFunc* (for example, to store the data or print it to the screen).

CBindStatusCallback::GetBindInfo

Called to tell the moniker how to bind.

```
STDMETHOD(GetBindInfo)(  
    DWORD* pgrfBSCF,  
    BINDINFO* pbindinfo);
```

Parameters

pgrfBSCF

[out] A pointer to BINDF enumeration values indicating how the bind operation should occur. By default, set with the following enumeration values:

BINDF_ASYNCHRONOUS Asynchronous download.

BINDF_ASYNCSTORAGE `OnDataAvailable` returns E_PENDING when data is not yet available rather than blocking until data is available.

BINDF_GETNEWESTVERSION The bind operation should retrieve the newest version of the data.

BINDF_NOWRITECACHE The bind operation should not store retrieved data in the disk cache.

pbindinfo

[in, out] A pointer to the `BINDINFO` structure giving more information about how the object wants binding to occur.

Return Value

One of the standard HRESULT values.

Remarks

The default implementation sets the binding to be asynchronous and to use the data-push model. In the data-push model, the moniker drives the asynchronous bind operation and continuously notifies the client whenever new data is available.

CBindStatusCallback::GetPriority

Called by the asynchronous moniker to get the priority of the bind operation.

```
STDMETHOD(GetPriority)(LONG* pnPriority);
```

Parameters

pnPriority

[out] Address of the **LONG** variable that, on success, receives the priority.

Return Value

Returns E_NOTIMPL.

CBindStatusCallback::m_dwAvailableToRead

Can be used to store the number of bytes available to be read.

```
DWORD m_dwAvailableToRead;
```

Remarks

Initialized to zero in `StartAsyncDownload`.

CBindStatusCallback::m_dwTotalRead

The cumulative total of bytes read in the asynchronous data transfer.

```
DWORD m_dwTotalRead;
```

Remarks

Incremented every time `OnDataAvailable` is called by the number of bytes actually read. Initialized to zero in `StartAsyncDownload`.

CBindStatusCallback::m_pFunc

The function pointed to by `m_pFunc` is called by `OnDataAvailable` after it reads the available data (for example, to store the data or print it to the screen).

```
ATL_PDATABAAVAILABLE m_pFunc;
```

Remarks

The function pointed to by `m_pFunc` is a member of your object's class and has the following syntax:

```
void Function_Name(  
    CBindStatusCallback<T>* pbsc,  
    BYTE* pBytes,  
    DWORD dwSize  
);
```

CBindStatusCallback::m_pT

A pointer to the object requesting the asynchronous data transfer.

```
T* m_pT;
```

Remarks

The `CBindStatusCallback` object is templated on this object's class.

CBindStatusCallback::m_spBindCtx

A pointer to an [IBindCtx](#) interface that provides access to the bind context (an object that stores information about a particular moniker binding operation).

```
CComPtr<IBindCtx> m_spBindCtx;
```

Remarks

Initialized in `StartAsyncDownload`.

CBindStatusCallback::m_spBinding

A pointer to the `IBinding` interface of the current bind operation.

```
CComPtr<IBinding> m_spBinding;
```

Remarks

Initialized in `OnStartBinding` and released in `OnStopBinding`.

CBindStatusCallback::m_spMoniker

A pointer to the [IMoniker](#) interface for the URL to use.

```
CComPtr<IMoniker> m_spMoniker;
```

Remarks

Initialized in `StartAsyncDownload`.

CBindStatusCallback::m_spStream

A pointer to the [IStream](#) interface of the current bind operation.

```
CComPtr<IStream> m_spStream;
```

Remarks

Initialized in `OnDataAvailable` from the `STGMEDIUM` structure when the BCSF flag is BCSF_FIRSTDATANOTIFICATION and released when the BCSF flag is BCSF_LASTDATANOTIFICATION.

CBindStatusCallback::OnDataAvailable

The system-supplied asynchronous moniker calls `OnDataAvailable` to provide data to the object as it becomes available.

```
STDMETHOD(  
    OnDataAvailable)(DWORD grfBSCF,  
    DWORD dwSize,  
    FORMATETC* /* pformatetc */,  
    STGMEDIUM* pstgmed);
```

Parameters

grfBSCF

[in] A BSCF enumeration value. One or more of the following: BSCF_FIRSTDATANOTIFICATION, BSCF_INTERMEDIARYDATANOTIFICATION, or BSCF_LASTDATANOTIFICATION.

dwSize

[in] The cumulative amount (in bytes) of data available since the beginning of the binding. Can be zero, indicating that the amount of data is not relevant or that no specific amount became available.

pformatetc

[in] Pointer to the [FORMATETC](#) structure that contains the format of the available data. If there is no format, can be CF_NULL.

pstgmed

[in] Pointer to the [STGMEDIUM](#) structure that holds the actual data now available.

Return Value

One of the standard HRESULT values.

Remarks

`OnDataAvailable` reads the data, then calls a method of your object's class (for example, to store the data or print it to the screen). See [CBindStatusCallback::StartAsyncDownload](#) for details.

CBindStatusCallback::OnLowResource

Called when resources are low.

```
STDMETHOD(OnLowResource)(DWORD /* dwReserved */);
```

Parameters

dwReserved

Reserved.

Return Value

Returns S_OK.

CBindStatusCallback::OnObjectAvailable

Called by the asynchronous moniker to pass an object interface pointer to your application.

```
STDMETHOD(OnObjectAvailable)(REFID /* riid */, IUnknown* /* punk */);
```

Parameters

riid

Interface identifier of the requested interface. Unused.

punk

Address of the IUnknown interface. Unused.

Return Value

Returns S_OK.

CBindStatusCallback::OnProgress

Called to indicate the progress of a data downloading process.

```
STDMETHOD(OnProgress)(  
    ULONG /* ulProgress */,  
    ULONG /* ulProgressMax */,  
    ULONG /* ulStatusCode */,  
    LPCWSTR /* szStatusText */);
```

Parameters

ulProgress

Unsigned long integer. Unused.

ulProgressMax

Unsigned long integer Unused.

ulStatusCode

Unsigned long integer. Unused.

szStatusText

Address of a string value. Unused.

Return Value

Returns S_OK.

CBindStatusCallback::OnStartBinding

Sets the data member [m_spBinding](#) to the `IBinding` pointer in *pBinding*.

```
STDMETHOD(OnStartBinding)(DWORD /* dwReserved */, IBinding* pBinding);
```

Parameters

dwReserved

Reserved for future use.

pBinding

[in] Address of the IBinding interface of the current bind operation. This cannot be NULL. The client should call AddRef on this pointer to keep a reference to the binding object.

CBindStatusCallback::OnStopBinding

Releases the `IBinding` pointer in the data member `m_spBinding`.

```
STDMETHOD(OnStopBinding)(HRESULT hresult, LPCWSTR /* szError */);
```

Parameters

hresult

Status code returned from the bind operation.

szError

Address of a string value. Unused.

Remarks

Called by the system-supplied asynchronous moniker to indicate the end of the bind operation.

CBindStatusCallback::StartAsyncDownload

Starts downloading data asynchronously from the specified URL.

```
HRESULT StartAsyncDownload(  
    T* pT,  
    ATL_PDATAAVAILABLE pFunc,  
    BSTR bstrURL,  
    IUnknown* pUnkContainer = NULL,  
    BOOL bRelative = FALSE);
```

Parameters

pT

[in] A pointer to the object requesting the asynchronous data transfer. The `CBindStatusCallback` object is templated on this object's class.

pFunc

[in] A pointer to the function that receives the data being read. The function is a member of your object's class of type `T`. See **Remarks** for syntax and an example.

bstrURL

[in] The URL to obtain data from. Can be any valid URL or file name. Cannot be NULL. For example:

```
CComBSTR mybstr = _T("http://somesite/data.htm")
```

pUnkContainer

[in] The `IUnknown` of the container. NULL by default.

bRelative

[in] A flag indicating whether the URL is relative or absolute. FALSE by default, meaning the URL is absolute.

Return Value

One of the standard HRESULT values.

Remarks

Every time data is available it is sent to the object through `OnDataAvailable`. `OnDataAvailable` reads the data and calls the function pointed to by *pFunc* (for example, to store the data or print it to the screen).

The function pointed to by *pFunc* is a member of your object's class and has the following syntax:

```
void Function_Name(  
    CBindStatusCallback<T>* pbsc,  
    BYTE* pBytes,  
    DWORD dwSize);
```

In the following example (taken from the [ASYNC](#) sample), the function `onData` writes the received data into a text box.

Example

```
void OnData(CBindStatusCallback<CATLAsync>* , BYTE* pBytes, DWORD /*cBytes*/)
{
    ATLTRACE(_T("OnData called\n"));

    m_bstrText.Append((LPCSTR)pBytes);
    if (::IsWindow(m_EditCtrl.m_hWnd))
    {
        USES_CONVERSION;
        _ATLTRY {
            ::SendMessage(m_EditCtrl.m_hWnd, WM_SETTEXT, 0,
                (LPARAM)(LPCTSTR)COLE2CT((BSTR)m_bstrText));
        }
        _ATLCATCH( e ) {
            e; // unused
            // COLE2CT threw an exception!
            ::SendMessage(m_EditCtrl.m_hWnd, WM_SETTEXT, 0,
                (LPARAM)_T("Could not allocate enough memory!!!"));
        }
    }
}
```

See also

[Class Overview](#)

CComAggObject Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

This class implements the [IUnknown](#) interface for an aggregated object. By definition, an aggregated object is contained within an outer object. The `CComAggObject` class is similar to the [CComObject Class](#), except that it exposes an interface that is directly accessible to external clients.

Syntax

```
template<class contained>
class CComAggObject : public IUnknown,
    public CComObjectRootEx<contained::_ThreadModel::_ThreadModelNoCS>
```

Parameters

contained

Your class, derived from [CComObjectRoot](#) or [CComObjectRootEx](#), as well as from any other interfaces you want to support on the object.

Members

Public Constructors

NAME	DESCRIPTION
CComAggObject::CComAggObject	The constructor.
CComAggObject::~~CComAggObject	The destructor.

Public Methods

NAME	DESCRIPTION
CComAggObject::AddRef	Increments the reference count on the aggregated object.
CComAggObject::CreateInstance	This static function allows you to create a new CComAggObject < <code>contained</code> > object without the overhead of CoCreateInstance .
CComAggObject::FinalConstruct	Performs final initialization of <code>m_contained</code> .
CComAggObject::FinalRelease	Performs final destruction of <code>m_contained</code> .
CComAggObject::QueryInterface	Retrieves a pointer to the requested interface.
CComAggObject::Release	Decrements the reference count on the aggregated object.

Public Data Members

NAME	DESCRIPTION
CComAggObject::m_contained	Delegates <code>IUnknown</code> calls to the outer unknown.

Remarks

`CComAggObject` implements [IUnknown](#) for an aggregated object. `CComAggObject` has its own `IUnknown` interface, separate from the outer object's `IUnknown` interface, and maintains its own reference count.

For more information about aggregation, see the article [Fundamentals of ATL COM Objects](#).

Inheritance Hierarchy

`CComObjectRootBase`

[CComObjectRootEx](#)

`IUnknown`

`CComAggObject`

Requirements

Header: atlcom.h

CComAggObject::AddRef

Increments the reference count on the aggregated object.

```
STDMETHOD_(ULONG, AddRef)();
```

Return Value

A value that may be useful for diagnostics or testing.

CComAggObject::CComAggObject

The constructor.

```
CComAggObject(void* pv);
```

Parameters

pv

[in] The outer unknown.

Remarks

Initializes the `CComContainedObject` member, [m_contained](#), and increments the module lock count.

The destructor decrements the module lock count.

CComAggObject::~~CComAggObject

The destructor.


```
~CComAggObject();
```

Remarks

Frees all allocated resources, calls [FinalRelease](#), and decrements the module lock count.

CComAggObject::CreateInstance

This static function allows you to create a new **CComAggObject**<[contained](#)> object without the overhead of [CoCreateInstance](#).

```
static HRESULT WINAPI CreateInstance(  
    LPUNKNOWN pUnkOuter,  
    CComAggObject<contained>** pp);
```

Parameters

pp

[out] A pointer to a **CComAggObject**<*contained*> pointer. If [CreateInstance](#) is unsuccessful, *pp* is set to NULL.

Return Value

A standard HRESULT value.

Remarks

The object returned has a reference count of zero, so call [AddRef](#) immediately, then use [Release](#) to free the reference on the object pointer when you're done.

If you do not need direct access to the object, but still want to create a new object without the overhead of [CoCreateInstance](#), use [CComCoClass::CreateInstance](#) instead.

CComAggObject::FinalConstruct

Called during the final stages of object construction, this method performs any final initialization on the [m_contained](#) member.

```
HRESULT FinalConstruct();
```

Return Value

A standard HRESULT value.

CComAggObject::FinalRelease

Called during object destruction, this method frees the [m_contained](#) member.

```
void FinalRelease();
```

CComAggObject::m_contained

A [CComContainedObject](#) object derived from your class.

```
CComContainedObject<contained> m_contained;
```

Parameters

contained

[in] Your class, derived from [CComObjectRoot](#) or [CComObjectRootEx](#), as well as from any other interfaces you want to support on the object.

Remarks

All [IUnknown](#) calls through [m_contained](#) are delegated to the outer unknown.

CComAggObject::QueryInterface

Retrieves a pointer to the requested interface.

```
STDMETHOD(QueryInterface)(REFIID iid, void** ppvObject);  
template <class Q>  
HRESULT STDMETHODCALLTYPE QueryInterface(Q** pp);
```

Parameters

iid

[in] The identifier of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by *iid*. If the object does not support this interface, *ppvObject* is set to NULL.

pp

[out] A pointer to the interface pointer identified by type [Q](#). If the object does not support this interface, *pp* is set to NULL.

Return Value

A standard HRESULT value.

Remarks

If the requested interface is [IUnknown](#), [QueryInterface](#) returns a pointer to the aggregated object's own [IUnknown](#) and increments the reference count. Otherwise, this method queries for the interface through the [CComContainedObject](#) member, [m_contained](#).

CComAggObject::Release

Decrements the reference count on the aggregated object.

```
STDMETHOD_(ULONG, Release)();
```

Return Value

In debug builds, [Release](#) returns a value that may be useful for diagnostics or testing. In non-debug builds, [Release](#) always returns 0.

See also

[CComObject Class](#)

[CComPolyObject Class](#)

[DECLARE_AGGREGATABLE](#)

[DECLARE_ONLY_AGGREGATABLE](#)

[DECLARE_NOT_AGGREGATABLE](#)

CComAllocator Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods for managing memory using COM memory routines.

Syntax

```
class CComAllocator
```

Members

Public Methods

NAME	DESCRIPTION
CComAllocator::Allocate	Call this static method to allocate memory.
CComAllocator::Free	Call this static method to free allocated memory.
CComAllocator::Reallocate	Call this static method to reallocate memory.

Remarks

This class is used by [CComHeapPtr](#) to provide the COM memory allocation routines. The counterpart class, [CCRTAllocator](#), provides the same methods using CRT routines.

Requirements

Header: atlbase.h

CComAllocator::Allocate

Call this static function to allocate memory.

```
static void* Allocate(size_t nBytes) throw();
```

Parameters

nBytes

The number of bytes to allocate.

Return Value

Returns a void pointer to the allocated space, or NULL if there is insufficient memory available.

Remarks

Allocates memory. See [CoTaskMemAlloc](#) for more details.

CComAllocator::Free

Call this static function to free allocated memory.

```
static void Free(void* p) throw();
```

Parameters

p

Pointer to the allocated memory.

Remarks

Frees the allocated memory. See [CoTaskMemFree](#) for more details.

CComAllocator::Reallocate

Call this static function to reallocate memory.

```
static void* Reallocate(void* p, size_t nBytes) throw();
```

Parameters

p

Pointer to the allocated memory.

nBytes

The number of bytes to reallocate.

Return Value

Returns a void pointer to the allocated space, or NULL if there is insufficient memory

Remarks

Resizes the amount of allocated memory. See [CoTaskMemRealloc](#) for more details.

See also

[CComHeapPtr Class](#)

[CCRTAllocator Class](#)

[Class Overview](#)

CComApartment Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides support for managing an apartment in a thread-pooled EXE module.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CComApartment
```

Members

Public Constructors

NAME	DESCRIPTION
CComApartment::CComApartment	The constructor.

Public Methods

NAME	DESCRIPTION
CComApartment::Apartment	Marks the thread's starting address.
CComApartment::GetLockCount	Returns the thread's current lock count.
CComApartment::Lock	Increments the thread's lock count.
CComApartment::Unlock	Decrements the thread's lock count.

Public Data Members

NAME	DESCRIPTION
CComApartment::m_dwThreadId	Contains the thread's identifier.
CComApartment::m_hThread	Contains the thread's handle.
CComApartment::m_nLockCnt	Contains the thread's current lock count.

Remarks

[CComApartment](#) is used by [CComAutoThreadModule](#) to manage an apartment in a thread-pooled EXE module.
[CComApartment](#) provides methods for incrementing and decrementing the lock count on a thread.

Requirements

Header: atlbase.h

CComartment::Apartment

Marks the thread's starting address.

```
DWORD Apartment();
```

Return Value

Always 0.

Remarks

Automatically set during [CComAutoThreadModule::Init](#).

CComartment::CComartment

The constructor.

```
CComartment();
```

Remarks

Initializes the `CComartment` data members [m_nLockCnt](#) and [m_hThread](#).

CComartment::GetLockCount

Returns the thread's current lock count.

```
LONG GetLockCount();
```

Return Value

The lock count on the thread.

CComartment::Lock

Increments the thread's lock count.

```
LONG Lock();
```

Return Value

A value that may be useful for diagnostics or testing.

Remarks

Called by [CComAutoThreadModule::Lock](#).

The lock count on the thread is used for statistical purposes.

CComartment::m_dwThreadId

Contains the thread's identifier.

```
DWORD m_dwThreadID;
```

CComApartment::m_hThread

Contains the thread's handle.

```
HANDLE m_hThread;
```

CComApartment::m_nLockCnt

Contains the thread's current lock count.

```
LONG m_nLockCnt;
```

CComApartment::Unlock

Decrements the thread's lock count.

```
LONG Unlock();
```

Return Value

A value that may be useful for diagnostics or testing.

Remarks

Called by [CComAutoThreadModule::Unlock](#).

The lock count on the thread is used for statistical purposes.

See also

[Class Overview](#)

CComAutoCriticalSection Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

`CComAutoCriticalSection` provides methods for obtaining and releasing ownership of a critical section object.

Syntax

```
class CComAutoCriticalSection : public CComCriticalSection
```

Members

Public Constructors

NAME	DESCRIPTION
CComAutoCriticalSection::CComAutoCriticalSection	The constructor.
CComAutoCriticalSection::~~CComAutoCriticalSection	The destructor.

Remarks

`CComAutoCriticalSection` is similar to class [CComCriticalSection](#), except `CComAutoCriticalSection` automatically initializes the critical section object in the constructor.

Typically, you use `CComAutoCriticalSection` through the `typedef` name [AutoCriticalSection](#). This name references `CComAutoCriticalSection` when [CComMultiThreadModel](#) is being used.

The `Init` and `Term` methods from [CComCriticalSection](#) are not available when using this class.

Inheritance Hierarchy

[CComCriticalSection](#)

`CComAutoCriticalSection`

Requirements

Header: `atlcore.h`

CComAutoCriticalSection::CComAutoCriticalSection

The constructor.

```
CComAutoCriticalSection();
```

Remarks

Calls the Win32 function [InitializeCriticalSection](#), which initializes the critical section object.

CComAutoCriticalSection::~~CComAutoCriticalSection

The destructor.

```
~CComAutoCriticalSection() throw();
```

Remarks

The destructor calls [DeleteCriticalSection](#), which releases all system resources used by the critical section object.

See also

[CComFakeCriticalSection Class](#)

[Class Overview](#)

[CComCriticalSection Class](#)

CComAutoDeleteCriticalSection Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods for obtaining and releasing ownership of a critical section object.

Syntax

```
class CComAutoDeleteCriticalSection : public CComSafeDeleteCriticalSection
```

Remarks

`CComAutoDeleteCriticalSection` derives from the class [CComSafeDeleteCriticalSection](#). However, `CComAutoDeleteCriticalSection` overrides the [Term](#) method to **private** access, which forces internal memory cleanup to occur only when instances of this class go out of scope or are explicitly deleted from memory.

This class introduces no additional methods over its base class. See [CComSafeDeleteCriticalSection](#) and [CComCriticalSection](#) for more information on critical section helper classes.

Inheritance Hierarchy

[CComCriticalSection](#)

[CComSafeDeleteCriticalSection](#)

`CComAutoDeleteCriticalSection`

Requirements

Header: atlc core.h

See also

[CComSafeDeleteCriticalSection Class](#)

[CComCriticalSection Class](#)

[Class Overview](#)

CComAutoThreadModule Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

As of ATL 7.0, `CComAutoThreadModule` is obsolete; see [ATL Module Classes](#) for more details.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class ThreadAllocator = CComSimpleThreadAllocator>  
class CComAutoThreadModule : public CComModule
```

Parameters

ThreadAllocator

[in] The class managing thread selection. The default value is [CComSimpleThreadAllocator](#).

Members

Methods

CreateInstance	Selects a thread and then creates an object in the associated apartment.
GetDefaultThreads	(Static) Dynamically calculates the number of threads for the module based on the number of processors.
Init	Creates the module's threads.
Lock	Increments the lock count on the module and on the current thread.
Unlock	Decrements the lock count on the module and on the current thread.

Data Members

Data Members

dwThreadID	Contains the identifier of the current thread.
m_Allocator	Manages thread selection.
m_nThreads	Contains the number of threads in the module.
m_pApartments	Manages the module's apartments.

Remarks

NOTE

This class is obsolete, having been replaced by the [CAtlAutoThreadModule](#) and [CAtlModule](#) derived classes. The information that follows is for use with older releases of ATL.

[CComAutoThreadModule](#) derives from [CComModule](#) to implement a thread-pooled, apartment-model COM server for EXEs and Windows services. [CComAutoThreadModule](#) uses [CComApartment](#) to manage an apartment for each thread in the module.

Derive your module from [CComAutoThreadModule](#) when you want to create objects in multiple apartments. You must also include the [DECLARE_CLASSFACTORY_AUTO_THREAD](#) macro in your object's class definition to specify [CComClassFactoryAutoThread](#) as the class factory.

By default, the ATL COM AppWizard (the ATL Project Wizard in Visual Studio .NET) will derive your module from [CComModule](#). To use [CComAutoThreadModule](#), modify the class definition. For example:

```
class CMyModule :
public CComAutoThreadModule<CComSimpleThreadAllocator>
{
public:
    LONG Unlock()
    {
        LONG l = CComAutoThreadModule<CComSimpleThreadAllocator>::Unlock();
        if (l == 0)
            PostThreadMessage(dwThreadId, WM_QUIT, 0, 0);
        return l;
    }

    DWORD dwThreadId;
};
```

Inheritance Hierarchy

[_ATL_MODULE](#)

[CAtlModule](#)

[IAtlAutoThreadModule](#)

[CAtlModuleT](#)

[CAtlAutoThreadModuleT](#)

[CComModule](#)

[CComAutoThreadModule](#)

Requirements

Header: atlbase.h

CComAutoThreadModule::CreateInstance

As of ATL 7.0, [CComAutoThreadModule](#) is obsolete: see [ATL Module Classes](#) for more details.

```
HRESULT CreateInstance(  
    void* pfnCreateInstance,  
    REFIID riid,  
    void** ppvObj);
```

Parameters

pfnCreateInstance

[in] A pointer to a creator function.

riid

[in] The IID of the requested interface.

ppvObj

[out] A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppvObj* is set to NULL.

Return Value

A standard HRESULT value.

Remarks

Selects a thread and then creates an object in the associated apartment.

CComAutoThreadModule::dwThreadID

As of ATL 7.0, `CComAutoThreadModule` is obsolete: see [ATL Module Classes](#) for more details.

```
DWORD dwThreadID;
```

Remarks

Contains the identifier of the current thread.

CComAutoThreadModule::GetDefaultThreads

As of ATL 7.0, `CComAutoThreadModule` is obsolete: see [ATL Module Classes](#) for more details.

```
static int GetDefaultThreads();
```

Return Value

The number of threads to be created in the EXE module.

Remarks

This static function dynamically calculates the maximum number of threads for the EXE module, based on the number of processors. By default, this return value is passed to the [Init](#) method to create the threads.

CComAutoThreadModule::Init

As of ATL 7.0, `CComAutoThreadModule` is obsolete: see [ATL Module Classes](#) for more details.

```
HRESULT Init(  
    _ATL_OBJMAP_ENTRY* p,  
    HINSTANCE h,  
    const GUID* plibid = NULL,  
    int nThreads = GetDefaultThreads());
```

Parameters

p

[in] A pointer to an array of object map entries.

h

[in] The HINSTANCE passed to `DLLMain` or `WinMain`.

plibid

[in] A pointer to the LIBID of the type library associated with the project.

nThreads

[in] The number of threads to be created. By default, *nThreads* is the value returned by [GetDefaultThreads](#).

Remarks

Initializes data members and creates the number of threads specified by *nThreads*.

CComAutoThreadModule::Lock

As of ATL 7.0, `CComAutoThreadModule` is obsolete: see [ATL Module Classes](#) for more details.

```
LONG Lock();
```

Return Value

A value that may be useful for diagnostics or testing.

Remarks

Performs an atomic increment on the lock count for the module and for the current thread.

`CComAutoThreadModule` uses the module lock count to determine whether any clients are accessing the module. The lock count on the current thread is used for statistical purposes.

CComAutoThreadModule::m_Allocator

As of ATL 7.0, `CComAutoThreadModule` is obsolete: see [ATL Module Classes](#) for more details.

```
ThreadAllocator m_Allocator;
```

Remarks

The object managing thread selection. By default, the `ThreadAllocator` class template parameter is [CComSimpleThreadAllocator](#).

CComAutoThreadModule::m_nThreads

As of ATL 7.0, `CComAutoThreadModule` is obsolete: see [ATL Module Classes](#) for more details.

```
int m_nThreads;
```

Remarks

Contains the number of threads in the EXE module. When [Init](#) is called, `m_nThreads` is set to the *nThreads* parameter value. Each thread's associated apartment is managed by a [CComApartment](#) object.

CComAutoThreadModule::m_pApartments

As of ATL 7.0, `CComAutoThreadModule` is obsolete: see [ATL Module Classes](#) for more details.

```
CComApartment* m_pApartments;
```

Remarks

Points to an array of [CComApartment](#) objects, each of which manages an apartment in the module. The number of elements in the array is based on the [m_nThreads](#) member.

CComAutoThreadModule::Unlock

As of ATL 7.0, `CComAutoThreadModule` is obsolete: see [ATL Module Classes](#) for more details.

```
LONG Unlock();
```

Return Value

A value that may be useful for diagnostics or testing.

Remarks

Performs an atomic decrement on the lock count for the module and for the current thread.

`CComAutoThreadModule` uses the module lock count to determine whether any clients are accessing the module. The lock count on the current thread is used for statistical purposes.

When the module lock count reaches zero, the module can be unloaded.

See also

[Class Overview](#)

[Module Classes](#)

CComBSTR Class

3/4/2019 • 15 minutes to read • [Edit Online](#)

This class is a wrapper for [BSTRs](#).

Syntax

```
class CComBSTR
```

Members

Public Constructors

NAME	DESCRIPTION
CComBSTR::CComBSTR	The constructor.
CComBSTR::~~CComBSTR	The destructor.

Public Methods

NAME	DESCRIPTION
CComBSTR::Append	Appends a string to <code>m_str</code> .
CComBSTR::AppendBSTR	Appends a BSTR to <code>m_str</code> .
CComBSTR::AppendBytes	Appends a specified number of bytes to <code>m_str</code> .
CComBSTR::ArrayToBSTR	Creates a BSTR from the first character of each element in the safearray and attaches it to the <code>CComBSTR</code> object.
CComBSTR::AssignBSTR	Assigns a BSTR to <code>m_str</code> .
CComBSTR::Attach	Attaches a BSTR to the <code>CComBSTR</code> object.
CComBSTR::BSTRToArray	Creates a zero-based one-dimensional safearray, where each element of the array is a character from the <code>CComBSTR</code> object.
CComBSTR::ByteLength	Returns the length of <code>m_str</code> in bytes.
CComBSTR::Copy	Returns a copy of <code>m_str</code> .
CComBSTR::CopyTo	Returns a copy of <code>m_str</code> via an [out] parameter
CComBSTR::Detach	Detaches <code>m_str</code> from the <code>CComBSTR</code> object.

NAME	DESCRIPTION
CComBSTR::Empty	Frees <code>m_str</code> .
CComBSTR::Length	Returns the length of <code>m_str</code> .
CComBSTR::LoadString	Loads a string resource.
CComBSTR::ReadFromStream	Loads a BSTR object from a stream.
CComBSTR::ToLower	Converts the string to lowercase.
CComBSTR::ToUpper	Converts the string to uppercase.
CComBSTR::WriteToStream	Saves <code>m_str</code> to a stream.

Public Operators

NAME	DESCRIPTION
CComBSTR::operator BSTR	Casts a <code>CComBSTR</code> object to a BSTR.
CComBSTR::operator !	Returns TRUE or FALSE, depending on whether <code>m_str</code> is NULL.
CComBSTR::operator !=	Compares a <code>CComBSTR</code> with a string.
CComBSTR::operator &	Returns the address of <code>m_str</code> .
CComBSTR::operator +=	Appends a <code>CComBSTR</code> to the object.
CComBSTR::operator <	Compares a <code>CComBSTR</code> with a string.
CComBSTR::operator =	Assigns a value to <code>m_str</code> .
CComBSTR::operator ==	Compares a <code>CComBSTR</code> with a string.
CComBSTR::operator >	Compares a <code>CComBSTR</code> with a string.

Public Data Members

NAME	DESCRIPTION
CComBSTR::m_str	Contains the BSTR associated with the <code>CComBSTR</code> object.

Remarks

The `CComBSTR` class is a wrapper for BSTRs, which are length-prefixed strings. The length is stored as an integer at the memory location preceding the data in the string.

A `BSTR` is null-terminated after the last counted character but may also contain null characters embedded within the string. The string length is determined by the character count, not the first null character.

NOTE

The `CCoMBSR` class provides a number of members (constructors, assignment operators, and comparison operators) that take either ANSI or Unicode strings as arguments. The ANSI versions of these functions are less efficient than their Unicode counterparts because temporary Unicode strings are often created internally. For efficiency, use the Unicode versions where possible.

NOTE

Because of the improved lookup behavior implemented in Visual Studio .NET, code such as `bstr = L"String2" + bstr;`, which may have compiled in previous releases, should instead be implemented as `bstr = CStringW(L"String2") + bstr`.

For a list of cautions when using `CCoMBSR`, see [Programming with CCoMBSR](#).

Requirements

Header: `atlbase.h`

CCoMBSR::Append

Appends either *lpz* or the BSTR member of *bstrSrc* to *m_str*.

```
HRESULT Append(const CCoMBSR& bstrSrc) throw();
HRESULT Append(wchar_t ch) throw();
HRESULT Append(char ch) throw();
HRESULT Append(LPCOLESTR lpz) throw();
HRESULT Append(LPCSTR lpz) throw();
HRESULT Append(LPCOLESTR lpz, int nLen) throw();
```

Parameters

bstrSrc

[in] A `CCoMBSR` object to append.

ch

[in] A character to append.

lpz

[in] A zero-terminated character string to append. You can pass a Unicode string via the LPCOLESTR overload or an ANSI string via the LPCSTR version.

nLen

[in] The number of characters from *lpz* to append.

Return Value

S_OK on success, or any standard HRESULT error value.

Remarks

An ANSI string will be converted to Unicode before being appended.

Example

```

enum { urlASP, urlHTM, urlISAPI } urlType;
urlType = urlASP;

CComBSTR bstrURL = OLESTR("http://SomeSite/");
CComBSTR bstrDEF = OLESTR("/OtherSite");
CComBSTR bstrASP = OLESTR("default.asp");

CComBSTR bstrTemp;
HRESULT hr;

switch (urlType)
{
case urlASP:
    // bstrURL is 'http://SomeSite/default.asp'
    hr = bstrURL.Append(bstrASP);
    break;

case urlHTM:
    // bstrURL is 'http://SomeSite/default.htm'
    hr = bstrURL.Append(OLESTR("default.htm"));
    break;

case urlISAPI:
    // bstrURL is 'http://SomeSite/default.dll?func'
    hr = bstrURL.Append(OLESTR("default.dll?func"));
    break;

default:
    // bstrTemp is 'http://'
    hr = bstrTemp.Append(bstrURL, 7);
    // bstrURL is 'http://OtherSite'
    if (hr == S_OK)
        hr = bstrTemp.Append(bstrDEF);
    bstrURL = bstrTemp;

    break;
}

```

CComBSTR::AppendBSTR

Appends the specified BSTR to [m_str](#).

```
HRESULT AppendBSTR(BSTR p) throw();
```

Parameters

p

[in] A BSTR to append.

Return Value

S_OK on success, or any standard HRESULT error value.

Remarks

Do not pass an ordinary wide-character string to this method. The compiler cannot catch the error and run time errors will occur.

Example

```

CComBSTR bstrPre(OLESTR("Hello "));
CComBSTR bstrSuf(OLESTR("World!"));
HRESULT hr;

// Appends "World!" to "Hello "
hr = bstrPre.AppendBSTR(bstrSuf);

// Displays a message box with text "Hello World!"
::MessageBox(NULL, CW2CT(bstrPre), NULL, MB_OK);

```

CComBSTR::AppendBytes

Appends the specified number of bytes to [m_str](#) without conversion.

```

HRESULT AppendBytes(const char* lpsz, int nLen) throw();

```

Parameters

lpsz

[in] A pointer to an array of bytes to append.

p

[in] The number of bytes to append.

Return Value

S_OK on success, or any standard HRESULT error value.

Example

```

CComBSTR bstrPre(OLESTR("Hello "));
HRESULT hr;

// Appends "Wo" to "Hello " (4 bytes == 2 characters)
hr = bstrPre.AppendBytes(reinterpret_cast<char*>(OLESTR("World!")), 4);

// Displays a message box with text "Hello Wo"
::MessageBox(NULL, CW2CT(bstrPre), NULL, MB_OK);

```

CComBSTR::ArrayToBSTR

Frees any existing string held in the `CComBSTR` object, then creates a BSTR from the first character of each element in the safearray and attaches it to the `CComBSTR` object.

```

HRESULT ArrayToBSTR(const SAFEARRAY* pSrc) throw();

```

Parameters

pSrc

[in] The safearray containing the elements used to create the string.

Return Value

S_OK on success, or any standard HRESULT error value.

CComBSTR::AssignBSTR

Assigns a BSTR to [m_str](#).

```
HRESULT AssignBSTR(const BSTR bstrSrc) throw();
```

Parameters

bstrSrc

[in] A BSTR to assign to the current `CComBSTR` object.

Return Value

S_OK on success, or any standard HRESULT error value.

CComBSTR::Attach

Attaches a BSTR to the `CComBSTR` object by setting the `m_str` member to *src*.

```
void Attach(BSTR src) throw();
```

Parameters

src

[in] The BSTR to attach to the object.

Remarks

Do not pass an ordinary wide-character string to this method. The compiler cannot catch the error and run time errors will occur.

NOTE

This method will assert if `m_str` is non- NULL.

Example

```
// STDMETHOD(BSTRToUpper)(/*[in, out]*/ BSTR bstrConv);
STDMETHODIMP InplaceBSTRToUpper(BSTR bstrConv)
{
    // Assign bstrConv to m_str member of CComBSTR
    CComBSTR bstrTemp;
    bstrTemp.Attach(bstrConv);

    // Make sure BSTR is not NULL string
    if (!bstrTemp)
        return E_POINTER;

    // Make string uppercase
    HRESULT hr;
    hr = bstrTemp.ToUpper();
    if (hr != S_OK)
        return hr;

    // Set m_str to NULL, so the BSTR is not freed
    bstrTemp.Detach();

    return S_OK;
}
```

CComBSTR::BSTRToArray

Creates a zero-based one-dimensional safearray, where each element of the array is a character from the

CComBSTR object.

```
HRESULT BSTRToArray(LPSAFEARRAY* ppArray) throw();
```

Parameters

ppArray

[out] The pointer to the safearray used to hold the results of the function.

Return Value

S_OK on success, or any standard HRESULT error value.

CComBSTR::ByteLength

Returns the number of bytes in `m_str`, excluding the terminating null character.

```
unsigned int ByteLength() const throw();
```

Return Value

The length of the `m_str` member in bytes.

Remarks

Returns 0 if `m_str` is NULL.

Example

```
// string with 11 chars (22 bytes)
CComBSTR bstrTemp(OLESTR("Hello World"));

unsigned int len = bstrTemp.ByteLength();

ATLASSERT(len == 22);
```

CComBSTR::CComBSTR

The constructor. The default constructor sets the `m_str` member to NULL.

```
CComBSTR() throw();
CComBSTR(const CComBSTR& src);
CComBSTR(REFGUID guid);
CComBSTR(int nSize);
CComBSTR(int nSize, LPCOLESTR sz);
CComBSTR(int nSize, LPCSTR sz);
CComBSTR(LPCOLESTR pSrc);
CComBSTR(LPCSTR pSrc);
CComBSTR(CComBSTR&& src) throw(); // (Visual Studio 2017)
```

Parameters

nSize

[in] The number of characters to copy from *sz* or the initial size in characters for the `CComBSTR`.

sz

[in] A string to copy. The Unicode version specifies an LPCOLESTR; the ANSI version specifies an LPCSTR.

pSrc

[in] A string to copy. The Unicode version specifies an LPCOLESTR; the ANSI version specifies an LPCSTR.

src

[in] A `CComBSTR` object.

guid

[in] A reference to a `GUID` structure.

Remarks

The copy constructor sets `m_str` to a copy of the BSTR member of *src*. The `REFGUID` constructor converts the GUID to a string using `StringFromGUID2` and stores the result.

The other constructors set `m_str` to a copy of the specified string. If you pass a value for *nSize*, then only *nSize* characters will be copied, followed by a terminating null character.

`CComBSTR` supports move semantics. You can use the move constructor (the constructor that takes an rvalue reference (`&&`) to create a new object that uses the same underlying data as the old object you pass in as an argument, without the overhead of copying the object.

The destructor frees the string pointed to by `m_str`.

Example

```
CComBSTR bstr1;    // BSTR points to NULL
bstr1 = "Bye";     // initialize with assignment operator
                  // ANSI string is converted to wide char

OLECHAR* str = OLESTR("Bye bye!"); // wide char string of length 5
int len = (int)wcslen(str);
CComBSTR bstr2(len + 1); // uninitialized BSTR of length 6
wcsncpy_s(bstr2.m_str, bstr2.Length(), str, len); // copy wide char string to BSTR

CComBSTR bstr3(5, OLESTR("Hello World")); // BSTR containing 'Hello',
                                          // input string is wide char
CComBSTR bstr4(5, "Hello World");         // same as above, input string
                                          // is ANSI

CComBSTR bstr5(OLESTR("Hey there")); // BSTR containing 'Hey there',
                                     // input string is wide char
CComBSTR bstr6("Hey there");          // same as above, input string
                                     // is ANSI

CComBSTR bstr7(bstr6);                // copy constructor, bstr7 contains 'Hey there'
```

CComBSTR::~CComBSTR

The destructor.

```
~CComBSTR();
```

Remarks

The destructor frees the string pointed to by `m_str`.

CComBSTR::Copy

Allocates and returns a copy of `m_str`.


```
BSTR Copy() const throw();
```

Return Value

A copy of the [m_str](#) member. If `m_str` is NULL, returns NULL.

Example

```
CComBSTR m_bstrURL;    // BSTR representing a URL

// put_URL is the put method for the URL property.
STDMETHOD(put_URL)(BSTR strURL)
{
    ATLTRACE(_T("put_URL\n"));

    // free existing string in m_bstrURL & make a copy
    // of strURL pointed to by m_bstrURL
    m_bstrURL = strURL;
    return S_OK;
}

// get_URL is the get method for the URL property.
STDMETHOD(get_URL)(BSTR* pstrURL)
{
    ATLTRACE(_T("get_URL\n"));

    // make a copy of m_bstrURL pointed to by pstrURL
    *pstrURL = m_bstrURL.Copy(); // See CComBSTR::CopyTo
    return S_OK;
}
```

CComBSTR::CopyTo

Allocates and returns a copy of [m_str](#) via the parameter.

```
HRESULT CopyTo(BSTR* pbstr) throw();

HRESULT CopyTo(VARIANT* pvarDest) throw();
```

Parameters

pbstr

[out] The address of a BSTR in which to return the string allocated by this method.

pvarDest

[out] The address of a VARIANT in which to return the string allocated by this method.

Return Value

A standard HRESULT value indicating the success or failure of the copy.

Remarks

After calling this method, the VARIANT pointed to by *pvarDest* will be of type VT_BSTR.

Example

```

CComBSTR m_bstrURL; // BSTR representing a URL

// get_URL is the get method for the URL property.
STDMETHOD(get_URL)(BSTR* pstrURL)
{
    // Make a copy of m_bstrURL and return it via pstrURL
    return m_bstrURL.CopyTo(pstrURL);
}

```

CComBSTR::Detach

Detaches [m_str](#) from the `CComBSTR` object and sets `m_str` to NULL.

```
BSTR Detach() throw();
```

Return Value

The BSTR associated with the `CComBSTR` object.

Example

```

// Method which converts bstrIn to uppercase
STDMETHODIMP BSTRToUpper(BSTR bstrIn, BSTR* pbstrOut)
{
    if (bstrIn == NULL || pbstrOut == NULL)
        return E_POINTER;

    // Create a temporary copy of bstrIn
    CComBSTR bstrTemp(bstrIn);

    if (!bstrTemp)
        return E_OUTOFMEMORY;

    // Make string uppercase
    HRESULT hr;
    hr = bstrTemp.ToUpper();
    if (hr != S_OK)
        return hr;

    // Return m_str member of bstrTemp
    *pbstrOut = bstrTemp.Detach();

    return S_OK;
}

```

CComBSTR::Empty

Frees the [m_str](#) member.

```
void Empty() throw();
```

Example

```

CComBSTR bstr(OLESTR("abc"));

// Calls SysFreeString to free the BSTR
bstr.Empty();
ATLASSERT(bstr.Length() == 0);

```

CComBSTR::Length

Returns the number of characters in `m_str`, excluding the terminating null character.

```
unsigned int Length() const throw();
```

Return Value

The length of the `m_str` member.

Example

```
// string with 11 chars
CComBSTR bstrTemp(OLESTR("Hello World"));

unsigned int len = bstrTemp.Length();

ATLASSERT(len == 11);
```

CComBSTR::LoadString

Loads a string resource specified by *nID* and stores it in this object.

```
bool LoadString(HINSTANCE hInst, UINT nID) throw();
bool LoadString(UINT nID) throw();
```

Parameters

See [LoadString](#) in the Windows SDK.

Return Value

Returns TRUE if the string is successfully loaded; otherwise, returns FALSE.

Remarks

The first function loads the resource from the module identified by you via the *hInst* parameter. The second function loads the resource from the resource module associated with the [CComModule](#)-derived object used in this project.

Example

```
CComBSTR bstrTemp;

// IDS_PROJNAME proj name stored as resource in string table
bstrTemp.LoadString(IDS_PROJNAME);

// the above is equivalent to:
// bstrTemp.LoadString(_Module.m_hInstResource, IDS_PROJNAME);

// display message box w/ proj name as title & text
::MessageBox(NULL, CW2CT(bstrTemp), CW2CT(bstrTemp), MB_OK);
```

CComBSTR::m_str

Contains the BSTR associated with the `CComBSTR` object.

```
BSTR m_str;
```

Example

```
CComBSTR GuidToBSTR(REFGUID guid)
{
    // 39 - length of string representation of GUID + 1
    CComBSTR b(39);

    // Convert GUID to BSTR
    // m_str member of CComBSTR is of type BSTR. When BSTR param
    // is required, pass the m_str member explicitly or use implicit
    // BSTR cast operator.
    int nRet = StringFromGUID2(guid, b.m_str, 39);

    // Above equivalent to:
    // int nRet = StringFromGUID2(guid, b, 39);
    // implicit BSTR cast operator used for 2nd param

    // Both lines are equivalent to:
    // CComBSTR b(guid);
    // CComBSTR constructor can convert GUIDs

    ATLASSERT(nRet);
    return b;
}
```

CComBSTR::operator BSTR

Casts a `CComBSTR` object to a BSTR.

```
operator BSTR() const throw();
```

Remarks

Allows you to pass `CComBSTR` objects to functions that have **[in] BSTR** parameters.

Example

See the example for [CComBSTR::m_str](#).

CComBSTR::operator !

Checks whether BSTR string is NULL.

```
bool operator!() const throw();
```

Return Value

Returns TRUE if the `m_str` member is NULL; otherwise, FALSE.

Remarks

This operator only checks for a NULL value, not for an empty string.

Example

```
// STDMETHODCALLTYPE(BSTRToUpper)(/*[in, out]*/ BSTR bstrConv);
STDMETHODIMP InplaceBSTRToUpper(BSTR bstrConv)
{
    // Assign bstrConv to m_str member of CComBSTR
    CComBSTR bstrTemp;
    bstrTemp.Attach(bstrConv);

    // Make sure BSTR is not NULL string
    if (!bstrTemp)
        return E_POINTER;

    // Make string uppercase
    HRESULT hr;
    hr = bstrTemp.ToUpper();
    if (hr != S_OK)
        return hr;

    // Set m_str to NULL, so the BSTR is not freed
    bstrTemp.Detach();

    return S_OK;
}
```

CComBSTR::operator !=

Returns the logical opposite of [operator ==](#).

```
bool operator!= (const CComBSTR& bstrSrc) const throw();
bool operator!= (LPCOLESTR pszSrc) const;
bool operator!= (LPCSTR pszSrc) const;
bool operator!= (int nNull) const throw();
```

Parameters

bstrSrc

[in] A `CComBSTR` object.

pszSrc

[in] A zero-terminated string.

nNull

[in] Must be NULL.

Return Value

Returns TRUE if the item being compared is not equal to the `CComBSTR` object; otherwise, returns FALSE.

Remarks

`CComBSTR`'s are compared textually in the context of the user's default locale. The final comparison operator just compares the contained string against NULL.

CComBSTR::operator &

Returns the address of the BSTR stored in the `m_str` member.

```
BSTR* operator&() throw();
```

Remarks

`CComBstr` operator `&` has a special assertion associated with it to help identify memory leaks. The program will assert when the `m_str` member is initialized. This assertion was created to identify situations where a programmer uses the `&` operator to assign a new value to `m_str` member without freeing the first allocation of `m_str`. If `m_str` equals `NULL`, the program assumes that `m_str` wasn't allocated yet. In this case, the program will not assert.

This assertion is not enabled by default. Define `ATL_CCOMBSTR_ADDRESS_OF_ASSERT` to enable this assertion.

Example

```
#define ATL_NO_CCOMBSTR_ADDRESS_OF_ASSERT

void MyInitFunction(BSTR* pbstr)
{
    ::SysReAllocString(pbstr, OLESTR("Hello World"));
    return;
}
```

```
CComBSTR bstrStr ;
// bstrStr is not initialized so this call will not assert.
MyInitFunction(&bstrStr);

CComBSTR bstrStr2(OLESTR("Hello World"));
// bstrStr2 is initialized so this call will assert.
::SysReAllocString(&bstrStr2, OLESTR("Bye"));
```

CComBSTR::operator +=

Appends a string to the `CComBSTR` object.

```
CComBSTR& operator+= (const CComBSTR& bstrSrc);
CComBSTR& operator+= (const LPCOLESTR pszSrc);
```

Parameters

bstrSrc

[in] A `CComBSTR` object to append.

pszSrc

[in] A zero-terminated string to append.

Remarks

`CComBSTR`'s are compared textually in the context of the user's default locale. The `LPCOLESTR` comparison is done using `memcmp` on the raw data in each string. The `LPCSTR` comparison is carried out in the same way once a temporary Unicode copy of *pszSrc* has been created. The final comparison operator just compares the contained string against `NULL`.

Example

```

CComBSTR bstrPre(OLESTR("Hello "));
CComBSTR bstrSuf(OLESTR("World!"));

// Appends "World!" to "Hello "
bstrPre += bstrSuf;

// Displays a message box with text "Hello World!"
::MessageBox(NULL, CW2CT(bstrPre), NULL, MB_OK);

```

CComBSTR::operator <

Compares a `CComBSTR` with a string.

```

bool operator<(const CComBSTR& bstrSrc) const throw();
bool operator<(LPCOLESTR pszSrc) const throw();
bool operator<(LPCSTR pszSrc) const throw();

```

Return Value

Returns TRUE if the item being compared is less than the `CComBSTR` object; otherwise, returns FALSE.

Remarks

The comparison is performed using the user's default locale.

CComBSTR::operator =

Sets the `m_str` member to a copy of *pSrc* or to a copy of the BSTR member of *src*. The move assignment operator moves `src` without copying it.

```

CComBSTR& operator= (const CComBSTR& src);
CComBSTR& operator= (LPCOLESTR pSrc);
CComBSTR& operator= (LPCSTR pSrc);
CComBSTR& operator= (CComBSTR&& src) throw(); // (Visual Studio 2017)

```

Remarks

The *pSrc* parameter specifies either an LPCOLESTR for Unicode versions or LPCSTR for ANSI versions.

Example

See the example for [CComBSTR::Copy](#).

CComBSTR::operator ==

Compares a `CComBSTR` with a string. `CComBSTR`s are compared textually in the context of the user's default locale.

```

bool operator==(const CComBSTR& bstrSrc) const throw();
bool operator==(LPCOLESTR pszSrc) const;
bool operator==(LPCSTR pszSrc) const;
bool operator==(int nNull) const throw();

```

Parameters

bstrSrc

[in] A `CComBSTR` object.

pszSrc

[in] A zero-terminated string.

nNull

[in] Must be NULL.

Return Value

Returns TRUE if the item being compared is equal to the `CComBSTR` object; otherwise, returns FALSE.

Remarks

The final comparison operator just compares the contained string against NULL.

CComBSTR::operator >

Compares a `CComBSTR` with a string.

```
bool operator>(const CComBSTR& bstrSrc) const throw();
```

Return Value

Returns TRUE if the item being compared is greater than the `CComBSTR` object; otherwise, returns FALSE.

Remarks

The comparison is performed using the user's default locale.

CComBSTR::ReadFromStream

Sets the `m_str` member to the BSTR contained in the specified stream.

```
HRESULT ReadFromStream(IStream* pStream) throw();
```

Parameters

pStream

[in] A pointer to the [IStream](#) interface on the stream containing the data.

Return Value

A standard HRESULT value.

Remarks

`ReadToStream` requires the contents of the stream at the current position to be compatible with the data format written out by a call to [WriteToStream](#).

Example


```

IDataObject* pDataObj;

// Fill in the FORMATETC struct to retrieve desired format
// from clipboard
FORMATETC formatetcIn = {CF_TEXT, NULL, DVASPECT_CONTENT, -1, TYMED_ISTREAM};
STGMEDIUM medium;
ZeroMemory(&medium, sizeof(STGMEDIUM));

// Get IDataObject from clipboard
HRESULT hr = ::OleGetClipboard(&pDataObj);

// Retrieve data from clipboard
hr = pDataObj->GetData(&formatetcIn, &medium);

if (SUCCEEDED(hr) && medium.tymed == TYMED_ISTREAM)
{
    CComBSTR bstrStr;
    // Get BSTR out of the stream
    hr = bstrStr.ReadFromStream(medium.pstm);

    //release the stream
    ::ReleaseStgMedium(&medium);
}

```

CComBSTR::ToLower

Converts the contained string to lowercase.

```
HRESULT ToLower() throw();
```

Return Value

A standard HRESULT value.

Remarks

See `CharLowerBuff` for more information on how the conversion is performed.

CComBSTR::ToUpper

Converts the contained string to uppercase.

```
HRESULT ToUpper() throw();
```

Return Value

A standard HRESULT value.

Remarks

See `CharUpperBuff` for more information on how the conversion is performed.

CComBSTR::WriteToStream

Saves the `m_str` member to a stream.

```
HRESULT WriteToStream(IStream* pStream) throw();
```

Parameters

pStream

[in] A pointer to the [IStream](#) interface on a stream.

Return Value

A standard HRESULT value.

Remarks

You can recreate a BSTR from the contents of the stream using the [ReadFromStream](#) function.

Example

```
//implementation of IDataObject::GetData()
STDMETHODIMP CMyDataObj::GetData(FORMATETC *pformatetcIn, STGMEDIUM *pmedium)
{
    HRESULT hr = S_OK;
    if (pformatetcIn->cfFormat == CF_TEXT && pformatetcIn->tymed == TYMED_ISTREAM)
    {
        IStream *pStm;
        // Create an IStream from global memory
        hr = CreateStreamOnHGlobal(NULL, TRUE, &pStm);
        if (FAILED(hr))
            return hr;

        // Initialize CComBSTR
        CComBSTR bstrStr = OLESTR("Hello World");

        // Serialize string into stream
        // the length followed by actual string is serialized into stream
        hr = bstrStr.WriteToStream(pStm);

        // Pass the IStream pointer back through STGMEDIUM struct
        pmedium->tymed = TYMED_ISTREAM;
        pmedium->pstm = pStm;
        pmedium->pUnkForRelease = NULL;
    }

    return hr;
}
```

See also

[Class Overview](#)

[ATL and MFC String Conversion Macros](#)

CComCachedTearOffObject Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements [IUnknown](#) for a tear-off interface.

Syntax

```
template
<class contained>
class CComCachedTearOffObject : public
    IUnknown,
public CComObjectRootEx<contained
    ::_ThreadModel::ThreadModelNoCS>
```

Parameters

contained

Your tear-off class, derived from `CComTearOffObjectBase` and the interfaces you want your tear-off object to support.

Members

Public Constructors

NAME	DESCRIPTION
CComCachedTearOffObject::CComCachedTearOffObject	The constructor.
CComCachedTearOffObject::~~CComCachedTearOffObject	The destructor.

Public Methods

NAME	DESCRIPTION
CComCachedTearOffObject::AddRef	Increments the reference count for a <code>CComCachedTearOffObject</code> object.
CComCachedTearOffObject::FinalConstruct	Calls the <code>m_contained::FinalConstruct</code> (the tear-off class' method).
CComCachedTearOffObject::FinalRelease	Calls the <code>m_contained::FinalRelease</code> (the tear-off class' method).
CComCachedTearOffObject::QueryInterface	Returns a pointer to the <code>IUnknown</code> of the <code>CComCachedTearOffObject</code> object, or to the requested interface on your tear-off class (the class <code>contained</code>).
CComCachedTearOffObject::Release	Decrements the reference count for a <code>CComCachedTearOffObject</code> object and destroys it if the reference count is 0.

Public Data Members

NAME	DESCRIPTION
<code>CComCachedTearOffObject::m_contained</code>	A <code>CComContainedObject</code> object derived from your tear-off class (the class <code>contained</code>).

Remarks

`CComCachedTearOffObject` implements `IUnknown` for a tear-off interface. This class differs from `CComTearOffObject` in that `CComCachedTearOffObject` has its own `IUnknown` , separate from the owner object's `IUnknown` (the owner is the object for which the tear-off is being created). `CComCachedTearOffObject` maintains its own reference count on its `IUnknown` and deletes itself once its reference count is zero. However, if you query for any of its tear-off interfaces, the reference count of the owner object's `IUnknown` will be incremented.

If the `CComCachedTearOffObject` object implementing the tear-off is already instantiated, and the tear-off interface is queried for again, the same `CComCachedTearOffObject` object is reused. In contrast, if a tear-off interface implemented by a `CComTearOffObject` is again queried for through the owner object, another `CComTearOffObject` will be instantiated.

The owner class must implement `FinalRelease` and call `Release` on the cached `IUnknown` for the `CComCachedTearOffObject` , which will decrement its reference count. This will cause `CComCachedTearOffObject` 's `FinalRelease` to be called and delete the tear-off.

Inheritance Hierarchy

`CComObjectRootBase`

`CComObjectRootEx`

`IUnknown`

`CComCachedTearOffObject`

Requirements

Header: atlcom.h

CComCachedTearOffObject::AddRef

Increments the reference count of the `CComCachedTearOffObject` object by 1.

```
STDMETHOD_(ULONG, AddRef)();
```

Return Value

A value that may be useful for diagnostics and testing.

CComCachedTearOffObject::CComCachedTearOffObject

The constructor.

```
CComCachedTearOffObject(void* pv);
```

Parameters

pv

[in] Pointer to the `IUnknown` of the `CComCachedTearOffObject`.

Remarks

Initializes the `CComContainedObject` member, [m_contained](#).

CComCachedTearOffObject::~~CComCachedTearOffObject

The destructor.

```
~CComCachedTearOffObject();
```

Remarks

Frees all allocated resources and calls [FinalRelease](#).

CComCachedTearOffObject::FinalConstruct

Calls `m_contained::FinalConstruct` to create `m_contained`, the `CComContainedObject` < `contained` > object used to access the interface implemented by your tear-off class.

```
HRESULT FinalConstruct();
```

Return Value

A standard HRESULT value.

CComCachedTearOffObject::FinalRelease

Calls `m_contained::FinalRelease` to free `m_contained`, the `CComContainedObject` < `contained` > object.

```
void FinalRelease();
```

CComCachedTearOffObject::m_contained

A [CComContainedObject](#) object derived from your tear-off class.

```
CComContainedObject <contained> m_contained;
```

Parameters

contained

[in] Your tear-off class, derived from `CComTearOffObjectBase` and the interfaces you want your tear-off object to support.

Remarks

The methods `m_contained` inherits are used to access the tear-off interface in your tear-off class through the cached tear-off object's `QueryInterface`, `FinalConstruct`, and `FinalRelease`.

CComCachedTearOffObject::QueryInterface

Retrieves a pointer to the requested interface.

```
STDMETHOD(QueryInterface)(REFIID iid, void** ppvObject);
```

Parameters

iid

[in] The GUID of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by *iid*, or NULL if the interface is not found.

Return Value

A standard HRESULT value.

Remarks

If the requested interface is `IUnknown`, returns a pointer to the `CComCachedTearOffObject`'s own `IUnknown` and increments the reference count. Otherwise, queries for the interface on your tear-off class using the [InternalQueryInterface](#) method inherited from `CComObjectRootEx`.

CComCachedTearOffObject::Release

Decrements the reference count by 1 and, if the reference count is 0, deletes the `CComCachedTearOffObject` object.

```
STDMETHOD_(ULONG, Release)();
```

Return Value

In non-debug builds, always returns 0. In debug builds, returns a value that may be useful for diagnostics or testing.

See also

[CComTearOffObject Class](#)

[CComObjectRootEx Class](#)

[Class Overview](#)

CComClassFactory Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements the [IClassFactory](#) interface.

Syntax

```
class CComClassFactory
: public IClassFactory,
  public CComObjectRootEx<CComGlobalsThreadModel>
```

Members

Public Methods

NAME	DESCRIPTION
CComClassFactory::CreateInstance	Creates an object of the specified CLSID.
CComClassFactory::LockServer	Locks the class factory in memory.

Remarks

[CComClassFactory](#) implements the [IClassFactory](#) interface, which contains methods for creating an object of a particular CLSID, as well as locking the class factory in memory to allow new objects to be created more quickly. [IClassFactory](#) must be implemented for every class that you register in the system registry and to which you assign a CLSID.

ATL objects normally acquire a class factory by deriving from [CComCoClass](#). This class includes the macro [DECLARE_CLASSFACTORY](#), which declares [CComClassFactory](#) as the default class factory. To override this default, specify one of the [DECLARE_CLASSFACTORY](#) XXX macros in your class definition. For example, the [DECLARE_CLASSFACTORY_EX](#) macro uses the specified class for the class factory:

```
class ATL_NO_VTABLE CMyCustomClass :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CMyCustomClass, &CLSID_MyCustomClass>,
  public IDispatchImpl<IMyCustomClass, &IID_IMyCustomClass, &LIBID_NVC_ATL_COMLib, /*wMajor =*/ 1,
/*wMinor =*/ 0>
{
public:
  DECLARE_CLASSFACTORY_EX(CMyClassFactory)

  // Remainder of class declaration omitted.
```

The above class definition specifies that [CMyClassFactory](#) will be used as the object's default class factory. [CMyClassFactory](#) must derive from [CComClassFactory](#) and override [CreateInstance](#).

ATL provides three other macros that declare a class factory:

- [DECLARE_CLASSFACTORY2](#) Uses [CComClassFactory2](#), which controls creation through a license.

- [DECLARE_CLASSFACTORY_AUTO_THREAD](#) Uses [CComClassFactoryAutoThread](#), which creates objects in multiple apartments.
- [DECLARE_CLASSFACTORY_SINGLETON](#) Uses [CComClassFactorySingleton](#), which constructs a single [CComObjectGlobal](#) object.

Requirements

Header: atlcom.h

CComClassFactory::CreateInstance

Creates an object of the specified CLSID and retrieves an interface pointer to this object.

```
STDMETHOD(CreateInstance)(LPUNKNOWN pUnkOuter, REFIID riid, void** ppvObj);
```

Parameters

pUnkOuter

[in] If the object is being created as part of an aggregate, then *pUnkOuter* must be the outer unknown. Otherwise, *pUnkOuter* must be NULL.

riid

[in] The IID of the requested interface. If *pUnkOuter* is non- NULL, *riid* must be `IID_IUnknown`.

ppvObj

[out] A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppvObj* is set to NULL.

Return Value

A standard HRESULT value.

CComClassFactory::LockServer

Increments and decrements the module lock count by calling `_Module::Lock` and `_Module::Unlock`, respectively.

```
STDMETHOD(LockServer)(BOOL fLock);
```

Parameters

fLock

[in] If TRUE, the lock count is incremented; otherwise, the lock count is decremented.

Return Value

A standard HRESULT value.

Remarks

`_Module` refers to the global instance of [CComModule](#) or a class derived from it.

Calling `LockServer` allows a client to hold onto a class factory so that multiple objects can be created quickly.

See also

[CComObjectRootEx Class](#)

[CComGlobalsThreadModel](#)

[Class Overview](#)

CComClassFactory2 Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

This class implements the [IClassFactory2](#) interface.

Syntax

```
template <class license>
class CComClassFactory2 : public IClassFactory2,
    public CComObjectRootEx<CComGlobalsThreadModel>,
    public license
```

Parameters

license

A class that implements the following static functions:

- `static BOOL VerifyLicenseKey( BSTR bstr );`
- `static BOOL GetLicenseKey( DWORD dwReserved, BSTR * pBstr );`
- `static BOOL IsLicenseValid( );`

Members

Public Methods

NAME	DESCRIPTION
CComClassFactory2::CreateInstance	Creates an object of the specified CLSID.
CComClassFactory2::CreateInstanceLic	Given a license key, creates an object of the specified CLSID.
CComClassFactory2::GetLicInfo	Retrieves information describing the licensing capabilities of the class factory.
CComClassFactory2::LockServer	Locks the class factory in memory.
CComClassFactory2::RequestLicKey	Creates and returns a license key.

Remarks

`CComClassFactory2` implements the [IClassFactory2](#) interface, which is an extension of [IClassFactory](#).

`IClassFactory2` controls object creation through a license. A class factory executing on a licensed machine can provide a run-time license key. This license key allows an application to instantiate objects when a full machine license does not exist.

ATL objects normally acquire a class factory by deriving from [CComCoClass](#). This class includes the macro [DECLARE_CLASSFACTORY](#), which declares [CComClassFactory](#) as the default class factory. To use `CComClassFactory2`, specify the [DECLARE_CLASSFACTORY2](#) macro in your object's class definition. For example:

```

class ATL_NO_VTABLE CMyClass2 :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CMyClass2, &CLSID_MyClass>,
    public IDispatchImpl<IMyClass, &IID_IMyClass, &LIBID_NVC_ATL_COMLib, /*wMajor =*/ 1, /*wMinor =*/ 0>,
    public IDispatchImpl<IMyDualInterface, &__uuidof(IMyDualInterface), &LIBID_NVC_ATL_COMLib, /* wMajor = */
1, /* wMinor = */ 0>
{
public:
    DECLARE_CLASSFACTORY2(CMyLicense)

    // Remainder of class declaration omitted

```

`CMyLicense`, the template parameter to `CComClassFactory2`, must implement the static functions `VerifyLicenseKey`, `GetLicenseKey`, and `IsLicenseValid`. The following is an example of a simple license class:

```

class CMyLicense
{
protected:
    static BOOL VerifyLicenseKey(BSTR bstr)
    {
        USES_CONVERSION;
        return !lstrcmp(OLE2T(bstr), _T("My run-time license key"));
    }

    static BOOL GetLicenseKey(DWORD /*dwReserved*/, BSTR* pBstr)
    {
        USES_CONVERSION;
        *pBstr = SysAllocString( T2OLE(_T("My run-time license key")));
        return TRUE;
    }

    static BOOL IsLicenseValid() { return TRUE; }
};

```

`CComClassFactory2` derives from both `CComClassFactory2Base` and *license*. `CComClassFactory2Base`, in turn, derives from `IClassFactory2` and `CComObjectRootEx< CComGlobalsThreadModel >`.

Inheritance Hierarchy

`CComObjectRootBase`

`license`

`CComObjectRootEx`

`IClassFactory2`

`CComClassFactory2`

Requirements

Header: atlcom.h

CComClassFactory2::CreateInstance

Creates an object of the specified CLSID and retrieves an interface pointer to this object.

```

STDMETHOD(CreateInstance)(LPUNKNOWN pUnkOuter, REFIID riid, void** ppvObj);

```

Parameters

pUnkOuter

[in] If the object is being created as part of an aggregate, then *pUnkOuter* must be the outer unknown. Otherwise, *pUnkOuter* must be NULL.

riid

[in] The IID of the requested interface. If *pUnkOuter* is non- NULL, *riid* must be `IID_IUnknown`.

ppvObj

[out] A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppvObj* is set to NULL.

Return Value

A standard HRESULT value.

Remarks

Requires the machine to be fully licensed. If a full machine license does not exist, call [CreateInstanceLic](#).

CComClassFactory2::CreateInstanceLic

Similar to [CreateInstance](#), except that `CreateInstanceLic` requires a license key.

```
STDMETHOD(CreateInstanceLic)(
    IUnknown* pUnkOuter,
    IUnknown* /* pUnkReserved
*/,
    REFIID riid,
    BSTR bstrKey,
    void** ppvObject);
```

Parameters

pUnkOuter

[in] If the object is being created as part of an aggregate, then *pUnkOuter* must be the outer unknown. Otherwise, *pUnkOuter* must be NULL.

pUnkReserved

[in] Not used. Must be NULL.

riid

[in] The IID of the requested interface. If *pUnkOuter* is non- NULL, *riid* must be `IID_IUnknown`.

bstrKey

[in] The run-time license key previously obtained from a call to `RequestLicKey`. This key is required to create the object.

ppvObject

[out] A pointer to the interface pointer specified by *riid*. If the object does not support this interface, *ppvObject* is set to NULL.

Return Value

A standard HRESULT value.

Remarks

You can obtain a license key using [RequestLicKey](#). In order to create an object on an unlicensed machine, you must call `CreateInstanceLic`.

CComClassFactory2::GetLicInfo

Fills a [LICINFO](#) structure with information that describes the class factory's licensing capabilities.

```
STDMETHOD(GetLicInfo)(LICINFO* pLicInfo);
```

Parameters

pLicInfo

[out] Pointer to a `LICINFO` structure.

Return Value

A standard HRESULT value.

Remarks

The `fRuntimeKeyAvail` member of this structure indicates whether, given a license key, the class factory allows objects to be created on an unlicensed machine. The *fLicVerified* member indicates whether a full machine license exists.

CComClassFactory2::LockServer

Increments and decrements the module lock count by calling `_Module::Lock` and `_Module::Unlock`, respectively.

```
STDMETHOD(LockServer)(BOOL fLock);
```

Parameters

fLock

[in] If TRUE, the lock count is incremented; otherwise, the lock count is decremented.

Return Value

A standard HRESULT value.

Remarks

`_Module` refers to the global instance of [CComModule](#) or a class derived from it.

Calling `LockServer` allows a client to hold onto a class factory so that multiple objects can be quickly created.

CComClassFactory2::RequestLicKey

Creates and returns a license key, provided that the `fRuntimeKeyAvail` member of the [LICINFO](#) structure is TRUE.

```
STDMETHOD(RequestLicKey)(DWORD dwReserved, BSTR* pbstrKey);
```

Parameters

dwReserved

[in] Not used. Must be zero.

pbstrKey

[out] Pointer to the license key.

Return Value

A standard HRESULT value.

Remarks

A license key is required for calling [CreateInstanceLic](#) to create an object on an unlicensed machine. If

`fRuntimeKeyAvail` is FALSE, then objects can only be created on a fully licensed machine.

Call [GetLicInfo](#) to retrieve the value of `fRuntimeKeyAvail`.

See also

[CComClassFactoryAutoThread Class](#)

[CComClassFactorySingleton Class](#)

[CComObjectRootEx Class](#)

[CComGlobalsThreadModel](#)

[Class Overview](#)

CComClassFactoryAutoThread Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements the [IClassFactory](#) interface, and allows objects to be created in multiple apartments.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CComClassFactoryAutoThread
: public IClassFactory,
  public CComObjectRootEx<CComGlobalsThreadModel>
```

Members

Public Methods

NAME	DESCRIPTION
CComClassFactoryAutoThread::CreateInstance	Creates an object of the specified CLSID.
CComClassFactoryAutoThread::LockServer	Locks the class factory in memory.

Remarks

[CComClassFactoryAutoThread](#) is similar to [CComClassFactory](#), but allows objects to be created in multiple apartments. To take advantage of this support, derive your EXE module from [CComAutoThreadModule](#).

ATL objects normally acquire a class factory by deriving from [CComCoClass](#). This class includes the macro [DECLARE_CLASSFACTORY](#), which declares [CComClassFactory](#) as the default class factory. To use [CComClassFactoryAutoThread](#), specify the [DECLARE_CLASSFACTORY_AUTO_THREAD](#) macro in your object's class definition. For example:

```
class ATL_NO_VTABLE CMyAutoClass :
  public CComObjectRootEx<CComMultiThreadModel>,
  public CComCoClass<CMyAutoClass, &CLSID_MyAutoClass>,
  public IMyAutoClass
{
public:
  DECLARE_CLASSFACTORY_AUTO_THREAD()

  // Remainder of class declaration omitted.
```

Inheritance Hierarchy

[CComObjectRootBase](#)

Requirements

Header: atlcom.h

CComClassFactoryAutoThread::CreateInstance

Creates an object of the specified CLSID and retrieves an interface pointer to this object.

```
STDMETHODIMP CreateInstance(  
    LPUNKNOWN pUnkOuter,  
    REFIID riid,  
    void** ppvObj);
```

Parameters

pUnkOuter

[in] If the object is being created as part of an aggregate, then *pUnkOuter* must be the outer unknown. Otherwise, *pUnkOuter* must be NULL.

riid

[in] The IID of the requested interface. If *pUnkOuter* is non- NULL, *riid* must be `IID_IUnknown`.

ppvObj

[out] A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppvObj* is set to NULL.

Return Value

A standard HRESULT value.

Remarks

If your module derives from [CComAutoThreadModule](#), `CreateInstance` first selects a thread to create the object in the associated apartment.

CComClassFactoryAutoThread::LockServer

Increments and decrements the module lock count by calling `_Module::Lock` and `_Module::Unlock`, respectively.

```
STDMETHODIMP LockServer(BOOL fLock);
```

Parameters

fLock

[in] If TRUE, the lock count is incremented; otherwise, the lock count is decremented.

Return Value

A standard HRESULT value.

Remarks

When using `CComClassFactoryAutoThread`, `_Module` typically refers to the global instance of [CComAutoThreadModule](#).

Calling `LockServer` allows a client to hold onto a class factory so that multiple objects can be quickly created.

See also

[IClassFactory](#)

[CComClassFactory2 Class](#)

[CComClassFactorySingleton Class](#)

[CComObjectRootEx Class](#)

[CComGlobalsThreadModel](#)

[Class Overview](#)

CComClassFactorySingleton Class

5/7/2019 • 2 minutes to read • [Edit Online](#)

This class derives from [CComClassFactory](#) and uses [CComObjectGlobal](#) to construct a single object.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class T>
class CComClassFactorySingleton : public CComClassFactory
```

Parameters

T

Your class.

[CComClassFactorySingleton](#) derives from [CComClassFactory](#) and uses [CComObjectGlobal](#) to construct a single object. Each call to the [CreateInstance](#) method simply queries this object for an interface pointer.

Members

Public Methods

NAME	DESCRIPTION
CComClassFactorySingleton::CreateInstance	Queries m_spObj for an interface pointer.

Public Data Members

NAME	DESCRIPTION
CComClassFactorySingleton::m_spObj	The CComObjectGlobal object constructed by CComClassFactorySingleton .

Remarks

ATL objects normally acquire a class factory by deriving from [CComCoClass](#). This class includes the macro [DECLARE_CLASSFACTORY](#), which declares [CComClassFactory](#) as the default class factory. To use [CComClassFactorySingleton](#), specify the [DECLARE_CLASSFACTORY_SINGLETON](#) macro in your object's class definition. For example:

```
class ATL_NO_VTABLE CMySingletonClass :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CMySingletonClass, &CLSID_MySingletonClass>,
    public IMySingletonClass
{
public:
    DECLARE_CLASSFACTORY_SINGLETON(CMySingletonClass)

    // Remainder of class declaration omitted.
```

Inheritance Hierarchy

CComObjectRootBase

CComObjectRootEx

IClassFactory

CComClassFactory

CComClassFactorySingleton

Requirements

Header: atlcom.h

CComClassFactorySingleton::CreateInstance

Calls [QueryInterface](#) through [m_spObj](#) to retrieve an interface pointer.

```
STDMETHOD(CreateInstance)(LPUNKNOWN pUnkOuter, REFIID riid, void** ppvObj);
```

Parameters

pUnkOuter

[in] If the object is being created as part of an aggregate, then *pUnkOuter* must be the outer unknown. Otherwise, *pUnkOuter* must be NULL.

riid

[in] The IID of the requested interface. If *pUnkOuter* is non- NULL, *riid* must be [IID_IUnknown](#) .

ppvObj

[out] A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppvObj* is set to NULL.

Return Value

A standard HRESULT value.

CComClassFactorySingleton::m_spObj

The [CComObjectGlobal](#) object constructed by [CComClassFactorySingleton](#) .

```
CComPtr<IUnknown> m_spObj;
```

Remarks

Each call to the [CreateInstance](#) method simply queries this object for an interface pointer.

Note that the current form of `m_spObj` presents a breaking change from the way that `CComClassFactorySingleton` worked in previous versions of ATL. In previous versions the `CComClassFactorySingleton` object was created at the same time as the class factory, during server initialization. In Visual C++ .NET 2003 and later, the object is created lazily, on the first request. This change could cause errors in programs that rely on early initialization.

See also

[IClassFactory](#)

[CComClassFactory2 Class](#)

[CComClassFactoryAutoThread Class](#)

[CComObjectRootEx Class](#)

[CComGlobalsThreadModel](#)

[Class Overview](#)

CComCoClass Class

3/4/2019 • 5 minutes to read • [Edit Online](#)

This class provides methods for creating instances of a class, and obtaining its properties.

Syntax

```
template <class T, const CLSID* pclsid = &CLSID_NULL>
class CComCoClass
```

Parameters

T

Your class, derived from `CComCoClass`.

pclsid

A pointer to the CLSID of the object.

Members

Public Methods

NAME	DESCRIPTION
CComCoClass::CreateInstance	(Static) Creates an instance of the class and queries for an interface.
CComCoClass::Error	(Static) Returns rich error information to the client.
CComCoClass::GetObjectCLSID	(Static) Returns the object's class identifier.
CComCoClass::GetObjectDescription	(Static) Override to return the object's description.

Remarks

`CComCoClass` provides methods for retrieving an object's CLSID, setting error information, and creating instances of the class. Any class registered in the object map should be derived from `CComCoClass`.

`CComCoClass` also defines the default class factory and aggregation model for your object. `CComCoClass` uses the following two macros:

- [DECLARE_CLASSFACTORY](#) Declares the class factory to be [CComClassFactory](#).
- [DECLARE_AGGREGATABLE](#) Declares that your object can be aggregated.

You can override either of these defaults by specifying another macro in your class definition. For example, to use [CComClassFactory2](#) instead of `CComClassFactory`, specify the [DECLARE_CLASSFACTORY2](#) macro:

```
class ATL_NO_VTABLE CMyClass2 :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CMyClass2, &CLSID_MyClass>,
    public IDispatchImpl<IMyClass, &IID_IMyClass, &LIBID_NVC_ATL_COMLib, /*wMajor =*/ 1, /*wMinor =*/ 0>,
    public IDispatchImpl<IMyDualInterface, &__uuidof(IMyDualInterface), &LIBID_NVC_ATL_COMLib, /* wMajor =
*/ 1, /* wMinor = */ 0>
{
public:
    DECLARE_CLASSFACTORY2(CMyLicense)

    // Remainder of class declaration omitted
```

Requirements

Header: atlcom.h

CComCoClass::CreateInstance

Use these `CreateInstance` functions to create an instance of a COM object and retrieve an interface pointer without using the COM API.

```
template <class Q>
static HRESULT CreateInstance( Q** pp);

template <class Q>
static HRESULT CreateInstance(IUnknown* punkOuter, Q** pp);
```

Parameters

Q

The COM interface that should be returned via *pp*.

punkOuter

[in] The outer unknown or controlling unknown of the aggregate.

pp

[out] The address of a pointer variable that receives the requested interface pointer if creation succeeds.

Return Value

A standard HRESULT value. See [CoCreateInstance](#) in the Windows SDK for a description of possible return values.

Remarks

Use the first overload of this function for typical object creation; use the second overload when you need to aggregate the object being created.

The ATL class implementing the required COM object (that is, the class used as the first template parameter to [CComCoClass](#)) must be in the same project as the calling code. The creation of the COM object is carried out by the class factory registered for this ATL class.

These functions are useful for creating objects that you have prevented from being externally creatable by using the [OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO](#) macro. They are also useful in situations where you want to avoid the COM API for reasons of efficiency.

Note that the interface *Q* must have an IID associated with it that can be retrieved using the [__uuidof](#) operator.

Example

In the following example, `CDocument` is a wizard-generated ATL class derived from `CComCoClass` that implements the `IDocument` interface. The class is registered in the object map with the `OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO` macro so clients can't create instances of the document using [CoCreateInstance](#). `CApplication` is a CoClass that provides a method on one of its own COM interfaces to create instances of the document class. The code below shows how easy it to create instances of the document class using the `CreateInstance` member inherited from the `CComCoClass` base class.

```
STDMETHODIMP CMyApp::CreateDocument( /* [out, retval] */ IDocument** ppDoc)
{
    *ppDoc = NULL;
    return CMyDoc::CreateInstance(ppDoc);
}
```

CComCoClass::Error

This static function sets up the `IErrorInfo` interface to provide error information to the client.

```
static HRESULT WINAPI Error(
    LPCOLESTR lpszDesc,
    const IID& iid = GUID_NULL,
    HRESULT hRes = 0);

static HRESULT WINAPI Error(
    LPCOLESTR lpszDesc,
    DWORD dwHelpID,
    LPCOLESTR lpszHelpFile,
    const IID& iid = GUID_NULL,
    HRESULT hRes = 0);

static HRESULT WINAPI Error(
    LPCSTR lpszDesc,
    const IID& iid = GUID_NULL,
    HRESULT hRes = 0);

static HRESULT WINAPI Error(
    LPCSTR lpszDesc,
    DWORD dwHelpID,
    LPCSTR lpszHelpFile,
    const IID& iid = GUID_NULL,
    HRESULT hRes = 0);

static HRESULT WINAPI Error(
    UINT nID,
    const IID& iid = GUID_NULL,
    HRESULT hRes = 0,
    HINSTANCE hInst = _AtlBaseModule.GetResourceInstance ());

static HRESULT Error(
    UINT nID,
    DWORD dwHelpID,
    LPCOLESTR lpszHelpFile,
    const IID& iid = GUID_NULL,
    HRESULT hRes = 0,
    HINSTANCE hInst = _AtlBaseModule.GetResourceInstance());
```

Parameters

lpszDesc

[in] The string describing the error. The Unicode version of `Error` specifies that *lpszDesc* is of type `LPCOLESTR`; the ANSI version specifies a type of `LPCSTR`.

iid

[in] The IID of the interface defining the error or GUID_NULL (the default value) if the error is defined by the operating system.

hRes

[in] The HRESULT you want returned to the caller. The default value is 0. For more details about *hRes*, see Remarks.

nID

[in] The resource identifier where the error description string is stored. This value should lie between 0x0200 and 0xFFFF, inclusively. In debug builds, an **ASSERT** will result if *nID* does not index a valid string. In release builds, the error description string will be set to "Unknown Error."

dwHelpID

[in] The help context identifier for the error.

lpSzHelpFile

[in] The path and name of the help file describing the error.

hInst

[in] The handle to the resource. By default, this parameter is `_AtlModule::GetResourceInstance`, where `_AtlModule` is the global instance of [CAtlModule](#).

Return Value

A standard HRESULT value. For details, see Remarks.

Remarks

To call `Error`, your object must implement the `ISupportErrorInfo` interface.

If the *hRes* parameter is nonzero, then `Error` returns the value of *hRes*. If *hRes* is zero, then the first four versions of `Error` return DISP_E_EXCEPTION. The last two versions return the result of the macro **MAKE_HRESULT(1, FACILITY_ITF, *nID*)**.

CComCoClass::GetObjectCLSID

Provides a consistent way of retrieving the object's CLSID.

```
static const CLSID& WINAPI GetObjectCLSID();
```

Return Value

The object's class identifier.

CComCoClass::GetObjectDescription

This static function retrieves the text description for your class object.

```
static LPCTSTR WINAPI GetObjectDescription();
```

Return Value

The class object's description.

Remarks

The default implementation returns NULL. You can override this method with the [DECLARE_OBJECT_DESCRIPTION](#) macro. For example:


```
class ATL_NO_VTABLE CMyDoc :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CMyDoc, &CLSID_MyDoc>,
    public IDocument
{
public:
    DECLARE_OBJECT_DESCRIPTION("My Document Object 1.0")

    // Remainder of class declaration omitted.
```

`GetObjectDescription` is called by `IComponentRegistrar::GetComponents`. `IComponentRegistrar` is an Automation interface that allows you to register and unregister individual components in a DLL. When you create a Component Registrar object with the ATL Project Wizard, the wizard will automatically implement the `IComponentRegistrar` interface. `IComponentRegistrar` is typically used by Microsoft Transaction Server.

For more information about the ATL Project Wizard, see the article [Creating an ATL Project](#).

See also

[Class Overview](#)

CComCompositeControl Class

3/4/2019 • 5 minutes to read • [Edit Online](#)

This class provides the methods required to implement a composite control.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class T>
class CComCompositeControl : public CComControl<T,CAxDialogImpl<T>>
```

Parameters

T

Your class, derived from [CComObjectRoot](#) or [CComObjectRootEx](#), as well as from any other interfaces you want to support for your composite control.

Members

Public Constructors

NAME	DESCRIPTION
CComCompositeControl::CComCompositeControl	The constructor.
CComCompositeControl::~~CComCompositeControl	The destructor.

Public Methods

NAME	DESCRIPTION
CComCompositeControl::AdviseSinkMap	Call this method to advise or unadvise all controls hosted by the composite control.
CComCompositeControl::CalcExtent	Call this method to calculate the size in HIMETRIC units of the dialog resource used to host the composite control.
CComCompositeControl::Create	This method is called to create the control window for the composite control.
CComCompositeControl::CreateControlWindow	Call this method to create the control window and advise any hosted control.
CComCompositeControl::SetBackgroundColorFromAmbient	Call this method to set the background color of the composite control using the container's background color.

Public Data Members

NAME	DESCRIPTION
CComCompositeControl::m_hbrBackground	The background brush.
CComCompositeControl::m_hWndFocus	The handle of the window that currently has focus.

Remarks

Classes derived from class `CComCompositeControl` inherit the functionality of an ActiveX composite control. ActiveX controls derived from `CComCompositeControl` are hosted by a standard dialog box. These types of controls are called composite controls because they are able to host other controls (native Windows controls and ActiveX controls).

`CComCompositeControl` identifies the dialog resource to use in creating the composite control by looking for an enumerated data member in the child class. The member `IDD` of this child class is set to the resource ID of the dialog resource that will be used as the control's window. The following is an example of the data member that the class derived from `CComCompositeControl` should contain to identify the dialog resource to be used for the control's window:

```
enum { IDD = IDD_MYCOMPOSITE };
```

NOTE

Composite controls are always windowed controls, although they can contain windowless controls.

A control implemented by a `CComCompositeControl`-derived class has default tabbing behavior built in. When the control receives focus by being tabbed to in a containing application, successively pressing the TAB key will cause the focus to be cycled through all of the composite control's contained controls, then out of the composite control and on to the next item in the tab order of the container. The tab order of the hosted controls is determined by the dialog resource and determines the order in which tabbing will occur.

NOTE

In order for accelerators to work properly with a `CComCompositeControl`, it is necessary to load an accelerator table as the control is created, pass the handle and number of accelerators back into [IOleControlImpl::GetControlInfo](#), and finally destroy the table when the control is released.

Example

```
// Example for overriding IOleControlImpl::GetControlInfo()
// This example uses the accelerator table from the project resources
// with the identifier IDR_ACCELTABLE
// Define GetControlInfo() in the header of your composite
// control class as follows:

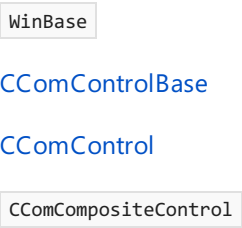
STDMETHOD(GetControlInfo)(CONTROLINFO* pCI)
{
    // Load the accelerator table from the resource
    pCI->hAccel = LoadAccelerators(_AtlBaseModule.GetResourceInstance(),
        MAKEINTRESOURCE(IDR_ACCELTABLE));

    if (pCI->hAccel == NULL)
        return E_FAIL;

    // Get the number of accelerators in the table
    pCI->cAccel = (USHORT)CopyAcceleratorTable(pCI->hAccel, NULL, 0);
    // The following is optional if you want your control
    // to process the return and/or escape keys
    // pCI->dwFlags = CTRLINFO_EATS_RETURN | CTRLINFO_EATS_ESCAPE;
    pCI->dwFlags = 0;

    return S_OK;
}
```

Inheritance Hierarchy



Requirements

Header: atlctl.h

CComCompositeControl::AdviseSinkMap

Call this method to advise or unadvise all controls hosted by the composite control.

```
HRESULT AdviseSinkMap(bool bAdvise);
```

Parameters

bAdvise

True if all controls are to be advised; otherwise false.

Return Value

S_OK	All controls in the event sink map were connected or disconnected from their event source successfully.
E_FAIL	Not all controls in the event sink map could be connected or disconnected from their event source successfully.

E_POINTER	This error usually indicates a problem with an entry in the control's event sink map or a problem with a template argument used in an <code>IDisEventImpl</code> or <code>IDisEventSimpleImpl</code> base class.
CONNECT_E_ADVISELIMIT	The connection point has already reached its limit of connections and cannot accept any more.
CONNECT_E_CANNOTCONNECT	The sink does not support the interface required by this connection point.
CONNECT_E_NOCONNECTION	The cookie value does not represent a valid connection. This error usually indicates a problem with an entry in the control's event sink map or a problem with a template argument used in an <code>IDisEventImpl</code> or <code>IDisEventSimpleImpl</code> base class.

Remarks

The base implementation of this method searches through the entries in the event sink map. It then advises or unadvisees the connection points to the COM objects described by the event sink map's sink entries. This member method also relies on the fact that the derived class inherits from one instance of `IDisEventImpl` for every control in the sink map that is to be advised or unadvised.

CComCompositeControl::CalcExtent

Call this method to calculate the size in HIMETRIC units of the dialog resource used to host the composite control.

```
BOOL CalcExtent(SIZE& size);
```

Parameters

size

A reference to a `SIZE` structure to be filled by this method.

Return Value

TRUE if the control is hosted by a dialog box; otherwise FALSE.

Remarks

The size is returned in the *size* parameter.

CComCompositeControl::Create

This method is called to create the control window for the composite control.

```
HWND Create(
    HWND hWndParent,
    RECT& /* rcPos */,
    LPARAM dwInitParam = NULL);
```

Parameters

hWndParent

A handle to the parent window of the control.

rcPos

Reserved.

dwInitParam

Data to be passed to the control during control creation. The data passed as *dwInitParam* will show up as the LPARAM parameter of the [WM_INITDIALOG](#) message, which will be sent to the composite control when it gets created.

Return Value

A handle to the newly created composite control dialog box.

Remarks

This method is usually called during in-place activation of the control.

CComCompositeControl::CComCompositeControl

The constructor.

```
CComCompositeControl();
```

Remarks

Initializes the [CComCompositeControl::m_hbrBackground](#) and [CComCompositeControl::m_hWndFocus](#) data members to NULL.

CComCompositeControl::~~CComCompositeControl

The destructor.

```
~CComCompositeControl();
```

Remarks

Deletes the background object, if it exists.

CComCompositeControl::CreateControlWindow

Call this method to create the control window and advise any hosted controls.

```
virtual HWND CreateControlWindow(  
    HWND hWndParent,  
    RECT& rcPos);
```

Parameters

hWndParent

A handle to the parent window of the control.

rcPos

The position rectangle of the composite control in client coordinates relative to *hWndParent*.

Return Value

Returns a handle to the newly created composite control dialog box.

Remarks

This method calls [CComCompositeControl::Create](#) and [CComCompositeControl::AdviseSinkMap](#).

CComCompositeControl::m_hbrBackground

The background brush.

```
HBRUSH m_hbrBackground;
```

CComCompositeControl::m_hWndFocus

The handle of the window that currently has focus.

```
HWND m_hWndFocus;
```

CComCompositeControl::SetBackgroundColorFromAmbient

Call this method to set the background color of the composite control using the container's background color.

```
HRESULT SetBackgroundColorFromAmbient();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See also

[CComControl Class](#)

[Composite Control Fundamentals](#)

[Class Overview](#)

CComContainedObject Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements [IUnknown](#) by delegating to the owner object's `IUnknown`.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class Base>
class CComContainedObject : public Base
```

Parameters

Base

Your class, derived from [CComObjectRoot](#) or [CComObjectRootEx](#).

Members

Public Constructors

NAME	DESCRIPTION
CComContainedObject::CComContainedObject	The constructor. Initializes the member pointer to the owner object's <code>IUnknown</code> .
CComContainedObject::~~CComContainedObject	The destructor.

Public Methods

NAME	DESCRIPTION
CComContainedObject::AddRef	Increments the reference count on the owner object.
CComContainedObject::GetControllingUnknown	Retrieves the owner object's <code>IUnknown</code> .
CComContainedObject::QueryInterface	Retrieves a pointer to the interface requested on the owner object.
CComContainedObject::Release	Decrements the reference count on the owner object.

Remarks

ATL uses `CComContainedObject` in classes [CComAggObject](#), [CComPolyObject](#), and [CComCachedTearOffObject](#).

`CComContainedObject` implements [IUnknown](#) by delegating to the owner object's `IUnknown`. (The owner is either the outer object of an aggregation, or the object for which a tear-off interface is being created.)

`CComContainedObject` calls `CComObjectRootEx`'s `OuterQueryInterface`, `OuterAddRef`, and `OuterRelease`, all inherited

through `Base` .

Inheritance Hierarchy

`Base`

`CComContainedObject`

Requirements

Header: atlcom.h

CComContainedObject::AddRef

Increments the reference count on the owner object.

```
STDMETHOD_(ULONG, AddRef)();
```

Return Value

A value that may be useful for diagnostics or testing.

CComContainedObject::CComContainedObject

The constructor.

```
CComContainedObject(void* pv);
```

Parameters

pv

[in] The owner object's `IUnknown` .

Remarks

Sets the `m_pOuterUnknown` member pointer (inherited through the `Base` class) to *pv*.

CComContainedObject::~~CComContainedObject

The destructor.

```
~CComContainedObject();
```

Remarks

Frees all allocated resources.

CComContainedObject::GetControllingUnknown

Returns the `m_pOuterUnknown` member pointer (inherited through the *Base* class) that holds the owner object's `IUnknown` .

```
IUnknown* GetControllingUnknown();
```

Return Value

The owner object's `IUnknown` .

Remarks

This method may be virtual if `Base` has declared the `DECLARE_GET_CONTROLLING_UNKNOWN` macro.

CComContainedObject::QueryInterface

Retrieves a pointer to the interface requested on the owner object.

```
STDMETHOD(QueryInterface)(REFIID iid, void** ppvObject);  
template <class Q>  
HRESULT STDMETHODCALLTYPE QueryInterface(Q** pp);
```

Parameters

iid

[in] The identifier of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by *iid*. If the object does not support this interface, *ppvObject* is set to NULL.

pp

[out] A pointer to the interface pointer identified by type `Q`. If the object does not support this interface, *pp* is set to NULL.

Return Value

A standard HRESULT value.

CComContainedObject::Release

Decrements the reference count on the owner object.

```
STDMETHOD_(ULONG, Release)();
```

Return Value

In debug builds, `Release` returns a value that may be useful for diagnostics or testing. In non-debug builds, `Release` always returns 0.

See also

[Class Overview](#)

CComControl Class

4/1/2019 • 4 minutes to read • [Edit Online](#)

This class provides methods for creating and managing ATL controls.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class T, class WinBase = CWindowImpl<T>>
class ATL_NO_VTABLE CComControl : public CComControlBase,
    public WinBase;
```

Parameters

T

The class implementing the control.

WinBase

The base class that implements windowing functions. Defaults to [CWindowImpl](#).

Members

Public Constructors

NAME	DESCRIPTION
CComControl::CComControl	Constructor.

Public Methods

NAME	DESCRIPTION
CComControl::ControlQueryInterface	Retrieves a pointer to the requested interface.
CComControl::CreateControlWindow	Creates a window for the control.
CComControl::FireOnChanged	Notifies the container's sink that a control property has changed.
CComControl::FireOnRequestEdit	Notifies the container's sink that a control property is about to change and that the object is asking the sink how to proceed.
CComControl::MessageBox	Call this method to create, display, and operate a message box.

Remarks

`CComControl` is a set of useful control helper functions and essential data members for ATL controls. When you create a standard control or a DHTML control using the ATL Control Wizard, the wizard will automatically derive your class from `CComControl`. `CComControl` derives most of its methods from [CComControlBase](#).

For more information about creating a control, see the [ATL Tutorial](#). For more information about the ATL Project Wizard, see the article [Creating an ATL Project](#).

For a demonstration of `CComControl` methods and data members, see the [CIRC](#) sample.

Inheritance Hierarchy

`WinBase`
[CComControlBase](#)

`CComControl`

Requirements

Header: atlctl.h

CComControl::CComControl

The constructor.

```
CComControl();
```

Remarks

Calls the [CComControlBase](#) constructor, passing the `m_hwnd` data member inherited through [CWindowImpl](#).

CComControl::ControlQueryInterface

Retrieves a pointer to the requested interface.

```
virtual HRESULT ControlQueryInterface(const IID& iid, void** ppv);
```

Parameters

iid

[in] The GUID of the interface being requested.

ppv

[out] A pointer to the interface pointer identified by *iid*, or NULL if the interface is not found.

Remarks

Only handles interfaces in the COM map table.

Example

```
// Retrieve the control's IOleObject interface. Note interface
// is automatically released when pOleObject goes out of scope

CComPtr<IOleObject> pOleObject;
ControlQueryInterface(IID_IOleObject, (void**)&pOleObject);
```

CComControl::CreateControlWindow

By default, creates a window for the control by calling `CWindowImpl::Create`.

```
virtual HWND CreateControlWindow(HWND hWndParent, RECT& rcPos);
```

Parameters

hWndParent

[in] Handle to the parent or owner window. A valid window handle must be supplied. The control window is confined to the area of its parent window.

rcPos

[in] The initial size and position of the window to be created.

Remarks

Override this method if you want to do something other than create a single window, for example, to create two windows, one of which becomes a toolbar for your control.

Example

```
RECT rc = {10,10,210,110};
HWND hWndParent, hWndControl;

// get HWND of control's parent window from IoleInPlaceSite interface
m_spInPlaceSite->GetWindow(&hWndParent);
hWndControl = CreateControlWindow(hWndParent, rc);
```

CComControl::FireOnChanged

Notifies the container's sink that a control property has changed.

```
HRESULT FireOnChanged(DISPID dispID);
```

Parameters

dispID

[in] Identifier of the property that has changed.

Return Value

One of the standard HRESULT values.

Remarks

If your control class derives from [IPropertyNotifySink](#), this method calls [CFirePropNotifyEvent::FireOnChanged](#) to notify all connected [IPropertyNotifySink](#) interfaces that the specified control property has changed. If your control class does not derive from [IPropertyNotifySink](#), this method returns S_OK.

This method is safe to call even if your control doesn't support connection points.

Example

```

STDMETHODIMP CMyControl::put_MyText(BSTR newVal)
{
    // store newVal in CComBstr member
    m_bstrMyText = newVal;

    // note the DISPID for the MyText property is 3 in this example
    FireOnChanged(3);

    return S_OK;
}

```

CComControl::FireOnRequestEdit

Notifies the container's sink that a control property is about to change and that the object is asking the sink how to proceed.

```

HRESULT FireOnRequestEdit(DISPID dispID);

```

Parameters

dispID

[in] Identifier of the property about to change.

Return Value

One of the standard HRESULT values.

Remarks

If your control class derives from [IPropertyNotifySink](#), this method calls [CFirePropNotifyEvent::FireOnRequestEdit](#) to notify all connected [IPropertyNotifySink](#) interfaces that the specified control property is about to change. If your control class does not derive from [IPropertyNotifySink](#), this method returns S_OK.

This method is safe to call even if your control doesn't support connection points.

Example

```

STDMETHODIMP CMyControl::put_MyTitle(BSTR newVal)
{
    // the DISPID for MyTitle in this example is 4
    DISPID dispID = 4;

    // make sure we can change the property
    if (FireOnRequestEdit(dispID) == S_FALSE)
        return S_FALSE;

    // store newVal in CComBstr member
    m_bstrMyTitle = newVal;

    // signal that the property has been changed
    FireOnChanged(dispID);

    return S_OK;
}

```

CComControl::MessageBox

Call this method to create, display, and operate a message box.

```
int MessageBox(  
    LPCTSTR lpszText,  
    LPCTSTR lpszCaption = _T(""),  
    UINT nType = MB_OK);
```

Parameters

lpszText

The text to be displayed in the message box.

lpszCaption

The dialog box title. If NULL (the default), the title "Error" is used.

nType

Specifies the contents and behavior of the dialog box. See the [MessageBox](#) entry in the Windows SDK documentation for a list of the different message boxes available. The default provides a simple **OK** button.

Return Value

Returns an integer value specifying one of the menu-item values listed under [MessageBox](#) in the Windows SDK documentation.

Remarks

`MessageBox` is useful both during development and as an easy way to display an error or warning message to the user.

See also

[CWindowImpl Class](#)

[Class Overview](#)

[CComControlBase Class](#)

[CComCompositeControl Class](#)

CComControlBase Class

3/4/2019 • 30 minutes to read • [Edit Online](#)

This class provides methods for creating and managing ATL controls.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class ATL_NO_VTABLE CComControlBase
```

Members

Public Typedefs

NAME	DESCRIPTION
CComControlBase::AppearanceType	Override if your <code>m_nAppearance</code> stock property isn't of type short .

Public Constructors

NAME	DESCRIPTION
CComControlBase::CComControlBase	The constructor.
CComControlBase::~CComControlBase	The destructor.

Public Methods

NAME	DESCRIPTION
CComControlBase::ControlQueryInterface	Retrieves a pointer to the requested interface.
CComControlBase::DoesVerbActivate	Checks that the <i>iVerb</i> parameter used by <code>IObjectImpl::DoVerb</code> either activates the control's user interface (<i>iVerb</i> equals OLEIVERB_UIACTIVATE), defines the action taken when the user double-clicks the control (<i>iVerb</i> equals OLEIVERB_PRIMARY), displays the control (<i>iVerb</i> equals OLEIVERB_SHOW), or activates the control (<i>iVerb</i> equals OLEIVERB_INPLACEACTIVATE).
CComControlBase::DoesVerbUIActivate	Checks that the <i>iVerb</i> parameter used by <code>IObjectImpl::DoVerb</code> causes the control's user interface to activate and returns TRUE.
CComControlBase::DoVerbProperties	Displays the control's property pages.

NAME	DESCRIPTION
CComControlBase::FireViewChange	Call this method to tell the container to redraw the control, or notify the registered advise sinks that the control's view has changed.
CComControlBase::GetAmbientAppearance	Retrieves DISPID_AMBIENT_APPEARANCE, the current appearance setting for the control: 0 for flat and 1 for 3D.
CComControlBase::GetAmbientAutoClip	Retrieves DISPID_AMBIENT_AUTOCLIP, a flag indicating whether the container supports automatic clipping of the control display area.
CComControlBase::GetAmbientBackColor	Retrieves DISPID_AMBIENT_BACKCOLOR, the ambient background color for all controls, defined by the container.
CComControlBase::GetAmbientCharSet	Retrieves DISPID_AMBIENT_CHARSET, the ambient character set for all controls, defined by the container.
CComControlBase::GetAmbientCodePage	Retrieves DISPID_AMBIENT_CODEPAGE, the ambient character set for all controls, defined by the container.
CComControlBase::GetAmbientDisplayAsDefault	Retrieves DISPID_AMBIENT_DISPLAYASDEFAULT, a flag that is TRUE if the container has marked the control in this site to be a default button, and therefore a button control should draw itself with a thicker frame.
CComControlBase::GetAmbientDisplayName	Retrieves DISPID_AMBIENT_DISPLAYNAME, the name the container has supplied to the control.
CComControlBase::GetAmbientFont	Retrieves a pointer to the container's ambient <code>IFont</code> interface.
CComControlBase::GetAmbientFontDisp	Retrieves a pointer to the container's ambient <code>IFontDisp</code> dispatch interface.
CComControlBase::GetAmbientForeColor	Retrieves DISPID_AMBIENT_FORECOLOR, the ambient foreground color for all controls, defined by the container.
CComControlBase::GetAmbientLocaleID	Retrieves DISPID_AMBIENT_LOCALEID, the identifier of the language used by the container.
CComControlBase::GetAmbientMessageReflect	Retrieves DISPID_AMBIENT_MESSAGEREFLECT, a flag indicating whether the container wants to receive window messages (such as WM_DRAWITEM) as events.
CComControlBase::GetAmbientPalette	Retrieves DISPID_AMBIENT_PALETTE, used to access the container's HPALETTE.
CComControlBase::GetAmbientProperty	Retrieves the container property specified by <i>id</i> .
CComControlBase::GetAmbientRightToLeft	Retrieves DISPID_AMBIENT_RIGHTTOLEFT, the direction in which content is displayed by the container.

NAME	DESCRIPTION
CComControlBase::GetAmbientScaleUnits	Retrieves DISPID_AMBIENT_SCALEUNITS, the container's ambient units (such as inches or centimeters) for labeling displays.
CComControlBase::GetAmbientShowGrabHandles	Retrieves DISPID_AMBIENT_SHOWGRABHANDLES, a flag indicating whether the container allows the control to display grab handles for itself when active.
CComControlBase::GetAmbientShowHatching	Retrieves DISPID_AMBIENT_SHOWHATCHING, a flag indicating whether the container allows the control to display itself with a hatched pattern when the UI is active.
CComControlBase::GetAmbientSupportsMnemonics	Retrieves DISPID_AMBIENT_SUPPORTSMNEMONICS, a flag indicating whether the container supports keyboard mnemonics.
CComControlBase::GetAmbientTextAlign	Retrieves DISPID_AMBIENT_TEXTALIGN, the text alignment preferred by the container: 0 for general alignment (numbers right, text left), 1 for left alignment, 2 for center alignment, and 3 for right alignment.
CComControlBase::GetAmbientTopToBottom	Retrieves DISPID_AMBIENT_TOPTOBOTTOM, the direction in which content is displayed by the container.
CComControlBase::GetAmbientUIDead	Retrieves DISPID_AMBIENT_UIDEAD, a flag indicating whether the container wants the control to respond to user-interface actions.
CComControlBase::GetAmbientUserMode	Retrieves DISPID_AMBIENT_USERMODE, a flag indicating whether the container is in run-mode (TRUE) or design-mode (FALSE).
CComControlBase::GetDirty	Returns the value of data member <code>m_bRequiresSave</code> .
CComControlBase::GetZoomInfo	Retrieves the x and y values of the numerator and denominator of the zoom factor for a control activated for in-place editing.
CComControlBase::InPlaceActivate	Causes the control to transition from the inactive state to whatever state the verb in <i>iVerb</i> indicates.
CComControlBase::InternalGetSite	Call this method to query the control site for a pointer to the identified interface.
CComControlBase::OnDraw	Override this method to draw your control.
CComControlBase::OnDrawAdvanced	The default <code>OnDrawAdvanced</code> prepares a normalized device context for drawing, then calls your control class's <code>OnDraw</code> method.
CComControlBase::OnKillFocus	Checks that the control is in-place active and has a valid control site, then informs the container that the control has lost focus.

NAME	DESCRIPTION
CComControlBase::OnMouseActivate	Checks that the UI is in user mode, then activates the control.
CComControlBase::OnPaint	Prepares the container for painting, gets the control's client area, then calls the control class's <code>OnDraw</code> method.
CComControlBase::OnSetFocus	Checks that the control is in-place active and has a valid control site, then informs the container the control has gained focus.
CComControlBase::PreTranslateAccelerator	Override this method to provide your own keyboard accelerator handlers.
CComControlBase::SendOnClose	Notifies all advisory sinks registered with the advise holder that the control has been closed.
CComControlBase::SendOnDataChange	Notifies all advisory sinks registered with the advise holder that the control data has changed.
CComControlBase::SendOnRename	Notifies all advisory sinks registered with the advise holder that the control has a new moniker.
CComControlBase::SendOnSave	Notifies all advisory sinks registered with the advise holder that the control has been saved.
CComControlBase::SendOnViewChange	Notifies all registered advisory sinks that the control's view has changed.
CComControlBase::SetControlFocus	Sets or removes the keyboard focus to or from the control.
CComControlBase::SetDirty	Sets the data member <code>m_bRequiresSave</code> to the value in <i>bDirty</i> .

Public Data Members

NAME	DESCRIPTION
CComControlBase::m_bAutoSize	Flag indicating the control cannot be any other size.
CComControlBase::m_bDrawFromNatural	Flag indicating that <code>IDataObjectImpl::GetData</code> and <code>CComControlBase::GetZoomInfo</code> should set the control size from <code>m_sizeNatural</code> rather than from <code>m_sizeExtent</code> .
CComControlBase::m_bDrawGetDataInHimetric	Flag indicating that <code>IDataObjectImpl::GetData</code> should use HIMETRIC units and not pixels when drawing.
CComControlBase::m_bInPlaceActive	Flag indicating the control is in-place active.
CComControlBase::m_bInPlaceSiteEx	Flag indicating the container supports the <code>IOleInPlaceSiteEx</code> interface and OCX96 control features, such as windowless and flicker-free controls.

NAME	DESCRIPTION
CComControlBase::m_bNegotiatedWnd	Flag indicating whether or not the control has negotiated with the container about support for OCX96 control features (such as flicker-free and windowless controls), and whether the control is windowed or windowless.
CComControlBase::m_bRecomposeOnResize	Flag indicating the control wants to recompose its presentation when the container changes the control's display size.
CComControlBase::m_bRequiresSave	Flag indicating the control has changed since it was last saved.
CComControlBase::m_bResizeNatural	Flag indicating the control wants to resize its natural extent (its unscaled physical size) when the container changes the control's display size.
CComControlBase::m_bUIActive	Flag indicating the control's user interface, such as menus and toolbars, is active.
CComControlBase::m_bUsingWindowRgn	Flag indicating the control is using the container-supplied window region.
CComControlBase::m_bWasOnceWindowless	Flag indicating the control has been windowless, but may or may not be windowless now.
CComControlBase::m_bWindowOnly	Flag indicating the control should be windowed, even if the container supports windowless controls.
CComControlBase::m_bWndLess	Flag indicating the control is windowless.
CComControlBase::m_hWndCD	Contains a reference to the window handle associated with the control.
CComControlBase::m_nFreezeEvents	A count of the number of times the container has frozen events (refused to accept events) without an intervening thaw of events (acceptance of events).
CComControlBase::m_rcPos	The position in pixels of the control, expressed in the coordinates of the container.
CComControlBase::m_sizeExtent	The extent of the control in HIMETRIC units (each unit is 0.01 millimeters) for a particular display.
CComControlBase::m_sizeNatural	The physical size of the control in HIMETRIC units (each unit is 0.01 millimeters).
CComControlBase::m_spAdviseSink	A direct pointer to the advisory connection on the container (the container's IAdviseSink).
CComControlBase::m_spAmbientDispatch	A <code>CComDispatchDriver</code> object that lets you retrieve and set the container's properties through an <code>IDispatch</code> pointer.
CComControlBase::m_spClientSite	A pointer to the control's client site within the container.

NAME	DESCRIPTION
CComControlBase::m_spDataAdviseHolder	Provides a standard means to hold advisory connections between data objects and advise sinks.
CComControlBase::m_spInPlaceSite	A pointer to the container's IOleInPlaceSite , IOleInPlaceSiteEx , or IOleInPlaceSiteWindowless interface pointer.
CComControlBase::m_spOleAdviseHolder	Provides a standard implementation of a way to hold advisory connections.

Remarks

This class provides methods for creating and managing ATL controls. [CComControl Class](#) derives from `CComControlBase`. When you create a Standard Control or DHTML control using the ATL Control Wizard, the wizard will automatically derive your class from `CComControlBase`.

For more information about creating a control, see the [ATL Tutorial](#). For more information about the ATL Project Wizard, see the article [Creating an ATL Project](#).

Requirements

Header: atlctl.h

CComControlBase::AppearanceType

Override if your `m_nAppearance` stock property isn't of type **short**.

```
typedef short AppearanceType;
```

Remarks

The ATL Control Wizard adds `m_nAppearance` stock property of type short. Override `AppearanceType` if you use a different data type.

CComControlBase::CComControlBase

The constructor.

```
CComControlBase(HWND& h);
```

Parameters

h

The handle to the window associated with the control.

Remarks

Initializes the control size to 5080X5080 HIMETRIC units (2"X2") and initializes the `CComControlBase` data member values to NULL or FALSE.

CComControlBase::~~CComControlBase

The destructor.

```
~CComControlBase();
```

Remarks

If the control is windowed, `CComControlBase` destroys it by calling [DestroyWindow](#).

CComControlBase::ControlQueryInterface

Retrieves a pointer to the requested interface.

```
virtual HRESULT ControlQueryInterface(const IID& iid,  
    void** ppv);
```

Parameters

iid

The GUID of the interface being requested.

ppv

A pointer to the interface pointer identified by *iid*, or NULL if the interface is not found.

Remarks

Only handles interfaces in the COM map table.

Example

```
// Retrieve the control's IOleObject interface. Note interface  
// is automatically released when pOleObject goes out of scope  
  
CComPtr<IOleObject> pOleObject;  
ControlQueryInterface(IID_IOleObject, (void*)&pOleObject);
```

CComControlBase::DoesVerbActivate

Checks that the *iVerb* parameter used by `IOleObjectImpl::DoVerb` either activates the control's user interface (*iVerb* equals `OLEIVERB_UIACTIVATE`), defines the action taken when the user double-clicks the control (*iVerb* equals `OLEIVERB_PRIMARY`), displays the control (*iVerb* equals `OLEIVERB_SHOW`), or activates the control (*iVerb* equals `OLEIVERB_INPLACEACTIVATE`).

```
BOOL DoesVerbActivate(LONG iVerb);
```

Parameters

iVerb

Value indicating the action to be performed by `DoVerb`.

Return Value

Returns TRUE if *iVerb* equals `OLEIVERB_UIACTIVATE`, `OLEIVERB_PRIMARY`, `OLEIVERB_SHOW`, or `OLEIVERB_INPLACEACTIVATE`; otherwise, returns FALSE.

Remarks

You can override this method to define your own activation verb.

CComControlBase::DoesVerbUIActivate

Checks that the *iVerb* parameter used by `IOleObjectImpl1::DoVerb` causes the control's user interface to activate and returns TRUE.

```
BOOL DoesVerbUIActivate(LONG iVerb);
```

Parameters

iVerb

Value indicating the action to be performed by `DoVerb`.

Return Value

Returns TRUE if *iVerb* equals OLEIVERB_UIACTIVATE, OLEIVERB_PRIMARY, OLEIVERB_SHOW, or OLEIVERB_INPLACEACTIVATE. Otherwise, the method returns FALSE.

CComControlBase::DoVerbProperties

Displays the control's property pages.

```
HRESULT DoVerbProperties(LPCRECT /* prcPosRect */, HWND hwndParent);
```

Parameters

prcPosRec

Reserved.

hwndParent

Handle of the window containing the control.

Return Value

One of the standard HRESULT values.

Example

```
// The following implementation of the WM_RBUTTONDOWN message handler
// will pop up the ActiveX Control's PropertyPages
LRESULT CMyComposite::OnRButtonDown(UINT /*uMsg*/, WPARAM /*wParam*/,
    LPARAM /*lParam*/, BOOL& /*bHandled*/)
{
    DoVerbProperties(NULL, ::GetActiveWindow());
    return 0L;
}
```

```
MESSAGE_HANDLER(WM_RBUTTONDOWN, OnRButtonDown)
```

CComControlBase::FireViewChange

Call this method to tell the container to redraw the control, or notify the registered advise sinks that the control's view has changed.

```
HRESULT FireViewChange();
```

Return Value

One of the standard HRESULT values.

Remarks

If the control is active (the control class data member [CComControlBase::m_bInPlaceActive](#) is TRUE), notifies the container that you want to redraw the entire control. If the control is inactive, notifies the control's registered advise sinks (through the control class data member [CComControlBase::m_spAdviseSink](#)) that the control's view has changed.

Example

```
STDMETHODIMP CMyControl::put_Shape(int newVal)
{
    // store newVal in m_nShape user-defined member
    m_nShape = newVal;

    // notify container to redraw control
    FireViewChange();
    return S_OK;
}
```

CComControlBase::GetAmbientAppearance

Retrieves DISPID_AMBIENT_APPEARANCE, the current appearance setting for the control: 0 for flat and 1 for 3D.

```
HRESULT GetAmbientAppearance(short& nAppearance);
```

Parameters

nAppearance

The property DISPID_AMBIENT_APPEARANCE.

Return Value

One of the standard HRESULT values.

Example


```

HRESULT OnDraw(ATL_DRAWINFO& di)
{
    short nAppearance;
    RECT& rc = *(RECT*)di.prcBounds;

    // draw 3D border if AmbientAppearance is not supported or is set to 1
    HRESULT hr = GetAmbientAppearance(nAppearance);
    if (hr != S_OK || nAppearance==1)
    {
        DrawEdge(di.hdcDraw, &rc, EDGE_SUNKEN, BF_RECT);
    }
    else
    {
        Rectangle(di.hdcDraw, rc.left, rc.top, rc.right, rc.bottom);
    }

    SetTextAlign(di.hdcDraw, TA_CENTER|TA_BASELINE);
    LPCTSTR pszText = _T("ATL 8.0 : MyControl");

    // For security reasons, we recommend that you use the lstrlen function
    // with caution. Here, we can guarantee that pszText is NULL terminated,
    // and therefore it is safe to use this function.
    TextOut(di.hdcDraw,
        (rc.left + rc.right) / 2,
        (rc.top + rc.bottom) / 2,
        pszText,
        lstrlen(pszText));

    return S_OK;
}

```

CComControlBase::GetAmbientAutoClip

Retrieves DISPID_AMBIENT_AUTOCLIP, a flag indicating whether the container supports automatic clipping of the control display area.

```

HRESULT GetAmbientAutoClip(BOOL& bAutoClip);

```

Parameters

bAutoClip

The property DISPID_AMBIENT_AUTOCLIP.

Return Value

One of the standard HRESULT values.

CComControlBase::GetAmbientBackColor

Retrieves DISPID_AMBIENT_BACKCOLOR, the ambient background color for all controls, defined by the container.

```

HRESULT GetAmbientBackColor(OLE_COLOR& BackColor);

```

Parameters

BackColor

The property DISPID_AMBIENT_BACKCOLOR.

Return Value

One of the standard HRESULT values.

CComControlBase::GetAmbientCharSet

Retrieves DISPID_AMBIENT_CHARSET, the ambient character set for all controls, defined by the container.

```
HRESULT GetAmbientCharSet(BSTR& bstrCharSet);
```

Parameters

bstrCharSet

The property DISPID_AMBIENT_CHARSET.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CComControlBase::GetAmbientCodePage

Retrieves DISPID_AMBIENT_CODEPAGE, the ambient code page for all controls, defined by the container.

```
HRESULT GetAmbientCodePage(ULONG& ulCodePage);
```

Parameters

ulCodePage

The property DISPID_AMBIENT_CODEPAGE.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CComControlBase::GetAmbientDisplayAsDefault

Retrieves DISPID_AMBIENT_DISPLAYASDEFAULT, a flag that is TRUE if the container has marked the control in this site to be a default button, and therefore a button control should draw itself with a thicker frame.

```
HRESULT GetAmbientDisplayAsDefault(BOOL& bDisplayAsDefault);
```

Parameters

bDisplayAsDefault

The property DISPID_AMBIENT_DISPLAYASDEFAULT.

Return Value

One of the standard HRESULT values.

CComControlBase::GetAmbientDisplayName

Retrieves DISPID_AMBIENT_DISPLAYNAME, the name the container has supplied to the control.

```
HRESULT GetAmbientDisplayName(BSTR& bstrDisplayName);
```

Parameters

bstrDisplayName

The property DISPID_AMBIENT_DISPLAYNAME.

Return Value

One of the standard HRESULT values.

CComControlBase::GetAmbientFont

Retrieves a pointer to the container's ambient `IFont` interface.

```
HRESULT GetAmbientFont(IFont** ppFont);
```

Parameters

ppFont

A pointer to the container's ambient `IFont` interface.

Return Value

One of the standard HRESULT values.

Remarks

If the property is NULL, the pointer is NULL. If the pointer is not NULL, the caller must release the pointer.

CComControlBase::GetAmbientFontDisp

Retrieves a pointer to the container's ambient `IFontDisp` dispatch interface.

```
HRESULT GetAmbientFontDisp(IFontDisp** ppFont);
```

Parameters

ppFont

A pointer to the container's ambient `IFontDisp` dispatch interface.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If the property is NULL, the pointer is NULL. If the pointer is not NULL, the caller must release the pointer.

CComControlBase::GetAmbientForeColor

Retrieves DISPID_AMBIENT_FORECOLOR, the ambient foreground color for all controls, defined by the container.

```
HRESULT GetAmbientForeColor(OLE_COLOR& ForeColor);
```

Parameters

ForeColor

The property DISPID_AMBIENT_FORECOLOR.

Return Value

One of the standard HRESULT values.

CComControlBase::GetAmbientLocaleID

Retrieves DISPID_AMBIENT_LOCALEID, the identifier of the language used by the container.

```
HRESULT GetAmbientLocaleID(LCID& lcid);
```

Parameters

lcid

The property DISPID_AMBIENT_LOCALEID.

Return Value

One of the standard HRESULT values.

Remarks

The control can use this identifier to adapt its user interface to different languages.

CComControlBase::GetAmbientMessageReflect

Retrieves DISPID_AMBIENT_MESSAGEREFLECT, a flag indicating whether the container wants to receive window messages (such as `WM_DRAWITEM`) as events.

```
HRESULT GetAmbientMessageReflect(BOOL& bMessageReflect);
```

Parameters

bMessageReflect

The property DISPID_AMBIENT_MESSAGEREFLECT.

Return Value

One of the standard HRESULT values.

CComControlBase::GetAmbientPalette

Retrieves DISPID_AMBIENT_PALETTE, used to access the container's HPALETTE.

```
HRESULT GetAmbientPalette(HPALETTE& hPalette);
```

Parameters

hPalette

The property DISPID_AMBIENT_PALETTE.

Return Value

One of the standard HRESULT values.

CComControlBase::GetAmbientProperty

Retrieves the container property specified by *dispid*.

```
HRESULT GetAmbientProperty(DISPID dispid, VARIANT& var);
```

Parameters

dispid

Identifier of the container property to be retrieved.

var

Variable to receive the property.

Return Value

One of the standard HRESULT values.

Remarks

ATL has provided a set of helper functions to retrieve specific properties, for example, [CComControlBase::GetAmbientBackColor](#). If there is no suitable method available, use `GetAmbientProperty`.

CComControlBase::GetAmbientRightToLeft

Retrieves DISPID_AMBIENT_RIGHTTOLEFT, the direction in which content is displayed by the container.

```
HRESULT GetAmbientRightToLeft(BOOL& bRightToLeft);
```

Parameters

bRightToLeft

The property DISPID_AMBIENT_RIGHTTOLEFT. Set to TRUE if content is displayed right to left, FALSE if it is displayed left to right.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CComControlBase::GetAmbientScaleUnits

Retrieves DISPID_AMBIENT_SCALEUNITS, the container's ambient units (such as inches or centimeters) for labeling displays.

```
HRESULT GetAmbientScaleUnits(BSTR& bstrScaleUnits);
```

Parameters

bstrScaleUnits

The property DISPID_AMBIENT_SCALEUNITS.

Return Value

One of the standard HRESULT values.

CComControlBase::GetAmbientShowGrabHandles

Retrieves DISPID_AMBIENT_SHOWGRABHANDLES, a flag indicating whether the container allows the control to display grab handles for itself when active.

```
HRESULT GetAmbientShowGrabHandles(BOOL& bShowGrabHandles);
```

Parameters

bShowGrabHandles

The property DISPID_AMBIENT_SHOWGRABHANDLES.

Return Value

One of the standard HRESULT values.

CComControlBase::GetAmbientShowHatching

Retrieves DISPID_AMBIENT_SHOWHATCHING, a flag indicating whether the container allows the control to display itself with a hatched pattern when the control's user interface is active.

```
HRESULT GetAmbientShowHatching(BOOL& bShowHatching);
```

Parameters

bShowHatching

The property DISPID_AMBIENT_SHOWHATCHING.

Return Value

One of the standard HRESULT values.

CComControlBase::GetAmbientSupportsMnemonics

Retrieves DISPID_AMBIENT_SUPPORTSMNEMONICS, a flag indicating whether the container supports keyboard mnemonics.

```
HRESULT GetAmbientSupportsMnemonics(BOOL& bSupportsMnemonics);
```

Parameters

bSupportsMnemonics

The property DISPID_AMBIENT_SUPPORTSMNEMONICS.

Return Value

One of the standard HRESULT values.

CComControlBase::GetAmbientTextAlign

Retrieves DISPID_AMBIENT_TEXTALIGN, the text alignment preferred by the container: 0 for general alignment (numbers right, text left), 1 for left alignment, 2 for center alignment, and 3 for right alignment.

```
HRESULT GetAmbientTextAlign(short& nTextAlign);
```

Parameters

nTextAlign

The property DISPID_AMBIENT_TEXTALIGN.

Return Value

One of the standard HRESULT values.

CComControlBase::GetAmbientTopToBottom

Retrieves DISPID_AMBIENT_TOPTOBOTTOM, the direction in which content is displayed by the container.

```
HRESULT GetAmbientTopToBottom(BOOL& bTopToBottom);
```

Parameters

bTopToBottom

The property DISPID_AMBIENT_TOPTOBOTTOM. Set to TRUE if text is displayed top to bottom, FALSE if it is displayed bottom to top.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CComControlBase::GetAmbientUIDead

Retrieves DISPID_AMBIENT_UIDEAD, a flag indicating whether the container wants the control to respond to user-interface actions.

```
HRESULT GetAmbientUIDead(BOOL& bUIDead);
```

Parameters

bUIDead

The property DISPID_AMBIENT_UIDEAD.

Return Value

One of the standard HRESULT values.

Remarks

If TRUE, the control should not respond. This flag applies regardless of the DISPID_AMBIENT_USERMODE flag. See [CComControlBase::GetAmbientUserMode](#).

CComControlBase::GetAmbientUserMode

Retrieves DISPID_AMBIENT_USERMODE, a flag indicating whether the container is in run-mode (TRUE) or design-mode (FALSE).

```
HRESULT GetAmbientUserMode(BOOL& bUserMode);
```

Parameters

bUserMode

The property DISPID_AMBIENT_USERMODE.

Return Value

One of the standard HRESULT values.

CComControlBase::GetDirty

Returns the value of data member `m_bRequiresSave`.

```
BOOL GetDirty();
```

Return Value

Returns the value of data member [m_bRequiresSave](#).

Remarks

This value is set using [CComControlBase::SetDirty](#).

CComControlBase::GetZoomInfo

Retrieves the x and y values of the numerator and denominator of the zoom factor for a control activated for in-place editing.

```
void GetZoomInfo(ATL_DRAWINFO& di);
```

Parameters

di

The structure that will hold the zoom factor's numerator and denominator. For more information, see [ATL_DRAWINFO](#).

Remarks

The zoom factor is the proportion of the control's natural size to its current extent.

CComControlBase::InPlaceActivate

Causes the control to transition from the inactive state to whatever state the verb in *iVerb* indicates.

```
HRESULT InPlaceActivate(LONG iVerb, const RECT* prcPosRect = NULL);
```

Parameters

iVerb

Value indicating the action to be performed by [IOleObjectImpl::DoVerb](#).

prcPosRect

Pointer to the position of the in-place control.

Return Value

One of the standard HRESULT values.

Remarks

Before activation, this method checks that the control has a client site, checks how much of the control is visible, and gets the control's location in the parent window. After the control is activated, this method activates the control's user interface and tells the container to make the control visible.

This method also retrieves an `IOleInPlaceSite`, `IOleInPlaceSiteEx`, or `IOleInPlaceSiteWindowless` interface pointer for the control and stores it in the control class's data member [CComControlBase::m_spInPlaceSite](#). The control class data members [CComControlBase::m_bInPlaceSiteEx](#), [CComControlBase::m_bWndLess](#), [CComControlBase::m_bWasOnceWindowless](#), and [CComControlBase::m_bNegotiatedWnd](#) are set to true as appropriate.

CComControlBase::InternalGetSite

Call this method to query the control site for a pointer to the identified interface.

```
HRESULT InternalGetSite(REFIID riid, void** ppUnkSite);
```

Parameters

riid

The IID of the interface pointer that should be returned in *ppUnkSite*.

ppUnkSite

Address of the pointer variable that receives the interface pointer requested in *riid*.

Return Value

Returns `S_OK` on success, or an error HRESULT on failure.

Remarks

If the site supports the interface requested in *riid*, the pointer is returned by means of *ppUnkSite*. Otherwise, *ppUnkSite* is set to NULL.

CComControlBase::m_bAutoSize

Flag indicating the control cannot be any other size.

```
unsigned m_bAutoSize:1;
```

Remarks

This flag is checked by `IOleObjectImpl::SetExtent` and, if TRUE, causes the function to return E_FAIL.

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

If you add the **Auto Size** option on the [Stock Properties](#) tab of the ATL Control Wizard, the wizard automatically creates this data member in your control class, creates put and get methods for the property, and supports [IPropertyNotifySink](#) to automatically notify the container when the property changes.

CComControlBase::m_bDrawFromNatural

Flag indicating that `IDataObjectImpl::GetData` and `CComControlBase::GetZoomInfo` should set the control size from `m_sizeNatural` rather than from `m_sizeExtent`.

```
unsigned m_bDrawFromNatural:1;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

CComControlBase::m_bDrawGetDataInHimetric

Flag indicating that `IDataObjectImpl::GetData` should use HIMETRIC units and not pixels when drawing.

```
unsigned m_bDrawGetDataInHimetric:1;
```

Remarks

Each logical HIMETRIC unit is 0.01 millimeter.

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

CComControlBase::m_bInPlaceActive

Flag indicating the control is in-place active.

```
unsigned m_bInPlaceActive:1;
```

Remarks

This means the control is visible and its window, if any, is visible, but its menus and toolbars may not be active. The `m_bUIActive` flag indicates the control's user interface, such as menus, is also active.

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

CComControlBase::m_bInPlaceSiteEx

Flag indicating the container supports the `IOleInPlaceSiteEx` interface and OCX96 control features, such as windowless and flicker-free controls.

```
unsigned m_bInPlaceSiteEx:1;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

The data member `m_spInPlaceSite` points to an [IOleInPlaceSite](#), [IOleInPlaceSiteEx](#), or [IOleInPlaceSiteWindowless](#) interface, depending on the value of the `m_bWndLess` and `m_bInPlaceSiteEx` flags. (The data member `m_bNegotiatedWnd` must be TRUE for the `m_spInPlaceSite` pointer to be valid.)

If `m_bWndLess` is FALSE and `m_bInPlaceSiteEx` is TRUE, `m_spInPlaceSite` is an `IOleInPlaceSiteEx` interface pointer. See [m_spInPlaceSite](#) for a table showing the relationship among these three data members.

CComControlBase::m_bNegotiatedWnd

Flag indicating whether or not the control has negotiated with the container about support for OCX96 control features (such as flicker-free and windowless controls), and whether the control is windowed or windowless.

```
unsigned m_bNegotiatedWnd:1;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

The `m_bNegotiatedWnd` flag must be TRUE for the `m_spInPlaceSite` pointer to be valid.

CComControlBase::m_bRecomposeOnResize

Flag indicating the control wants to recompose its presentation when the container changes the control's display size.

```
unsigned m_bRecomposeOnResize:1;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

This flag is checked by [IOleObjectImpl::SetExtent](#) and, if TRUE, `SetExtent` notifies the container of view changes. if this flag is set, the OLEMISC_RECOMPOSEONRESIZE bit in the [OLEMISC](#) enumeration should also be set.

CComControlBase::m_bRequiresSave

Flag indicating the control has changed since it was last saved.

```
unsigned m_bRequiresSave:1;
```

Remarks

The value of `m_bRequiresSave` can be set with [CComControlBase::SetDirty](#) and retrieved with [CComControlBase::GetDirty](#).

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

CComControlBase::m_bResizeNatural

Flag indicating the control wants to resize its natural extent (its unscaled physical size) when the container changes the control's display size.

```
unsigned m_bResizeNatural:1;
```

Remarks

This flag is checked by [IOleObjectImpl::SetExtent](#) and, if TRUE, the size passed into `SetExtent` is assigned to `m_sizeNatural`.

The size passed into `SetExtent` is always assigned to `m_sizeExtent`, regardless of the value of `m_bResizeNatural`.

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

CComControlBase::m_bUIActive

Flag indicating the control's user interface, such as menus and toolbars, is active.

```
unsigned m_bUIActive:1;
```

Remarks

The `m_bInPlaceActive` flag indicates that the control is active, but not that its user interface is active.

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

CComControlBase::m_bUsingWindowRgn

Flag indicating the control is using the container-supplied window region.

```
unsigned m_bUsingWindowRgn:1;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

CComControlBase::m_bWasOnceWindowless

Flag indicating the control has been windowless, but may or may not be windowless now.

```
unsigned m_bWasOnceWindowless:1;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

CComControlBase::m_bWindowOnly

Flag indicating the control should be windowed, even if the container supports windowless controls.

```
unsigned m_bWindowOnly:1;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

CComControlBase::m_bWndLess

Flag indicating the control is windowless.

```
unsigned m_bWndLess:1;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

The data member `m_spInPlaceSite` points to an [IOleInPlaceSite](#), [IOleInPlaceSiteEx](#), or [IOleInPlaceSiteWindowless](#) interface, depending on the value of the `m_bWndLess` and [CComControlBase::m_bInPlaceSiteEx](#) flags. (The data member [CComControlBase::m_bNegotiatedWnd](#) must be TRUE for the [CComControlBase::m_spInPlaceSite](#) pointer to be valid.)

If `m_bWndLess` is TRUE, `m_spInPlaceSite` is an `IOleInPlaceSiteWindowless` interface pointer. See [CComControlBase::m_spInPlaceSite](#) for a table showing the complete relationship between these data members.

CComControlBase::m_hWndCD

Contains a reference to the window handle associated with the control.

```
HWND& m_hWndCD;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

CComControlBase::m_nFreezeEvents

A count of the number of times the container has frozen events (refused to accept events) without an intervening thaw of events (acceptance of events).

```
short m_nFreezeEvents;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

CComControlBase::m_rcPos

The position in pixels of the control, expressed in the coordinates of the container.

```
RECT m_rcPos;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

CComControlBase::m_sizeExtent

The extent of the control in HIMETRIC units (each unit is 0.01 millimeters) for a particular display.

```
SIZE m_sizeExtent;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

This size is scaled by the display. The control's physical size is specified in the `m_sizeNatural` data member and is fixed.

You can convert the size to pixels with the global function [AtlHiMetricToPixel](#).

CComControlBase::m_sizeNatural

The physical size of the control in HIMETRIC units (each unit is 0.01 millimeters).

```
SIZE m_sizeNatural;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

This size is fixed, while the size in `m_sizeExtent` is scaled by the display.

You can convert the size to pixels with the global function [AtlHiMetricToPixel](#).

CComControlBase::m_spAdviseSink

A direct pointer to the advisory connection on the container (the container's [IAdviseSink](#)).

```
CComPtr<IAdviseSink>  
    m_spAdviseSink;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

CComControlBase::m_spAmbientDispatch

A [CComDispatchDriver](#) object that lets you retrieve and set an object's properties through an [IDispatch](#) pointer.

```
CComDispatchDriver m_spAmbientDispatch;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

CComControlBase::m_spClientSite

A pointer to the control's client site within the container.

```
CComPtr<IOleClientSite>  
    m_spClientSite;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

CComControlBase::m_spDataAdviseHolder

Provides a standard means to hold advisory connections between data objects and advise sinks.

```
CComPtr<IDataAdviseHolder>  
    m_spDataAdviseHolder;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

A data object is a control that can transfer data and that implements [IDataObject](#), whose methods specify the format and transfer medium of the data.

The interface `m_spDataAdviseHolder` implements the [IDataObject::DAdvise](#) and [IDataObject::DUnadvise](#) methods to establish and delete advisory connections to the container. The control's container must implement an advise sink by supporting the [IAdviseSink](#) interface.

CComControlBase::m_spInPlaceSite

A pointer to the container's [IOleInPlaceSite](#), [IOleInPlaceSiteEx](#), or [IOleInPlaceSiteWindowless](#) interface pointer.

```
CComPtr<IOleInPlaceSiteWindowless>  
m_spInPlaceSite;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

The `m_spInPlaceSite` pointer is valid only if the [m_bNegotiatedWnd](#) flag is TRUE.

The following table shows how the `m_spInPlaceSite` pointer type depends on the [m_bWndLess](#) and [m_binPlaceSiteEx](#) data member flags:

M_SPINPLACESITE TYPE	M_BWNDLESS VALUE	M_BINPLACESITEEX VALUE
<code>IOleInPlaceSiteWindowless</code>	TRUE	TRUE or FALSE
<code>IOleInPlaceSiteEx</code>	FALSE	TRUE
<code>IOleInPlaceSite</code>	FALSE	FALSE

CComControlBase::m_spOleAdviseHolder

Provides a standard implementation of a way to hold advisory connections.

```
CComPtr<IOleAdviseHolder>  
m_spOleAdviseHolder;
```

Remarks

NOTE

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

The interface `m_spOleAdviseHolder` implements the [IOleObject::Advise](#) and [IOleObject::Unadvise](#) methods to establish and delete advisory connections to the container. The control's container must implement an advise sink by supporting the [IAdviseSink](#) interface.

CComControlBase::OnDraw

Override this method to draw your control.

```
virtual HRESULT OnDraw(ATL_DRAWINFO& di);
```

Parameters

di

A reference to the [ATL_DRAWINFO](#) structure that contains drawing information such as the draw aspect, the control bounds, and whether the drawing is optimized or not.

Return Value

A standard HRESULT value.

Remarks

The default `OnDraw` deletes or restores the device context or does nothing, depending on flags set in [CComControlBase::OnDrawAdvanced](#).

An `OnDraw` method is automatically added to your control class when you create your control with the ATL Control Wizard. The wizard's default `OnDraw` draws a rectangle with the label "ATL 8.0".

Example

See the example for [CComControlBase::GetAmbientAppearance](#).

CComControlBase::OnDrawAdvanced

The default `OnDrawAdvanced` prepares a normalized device context for drawing, then calls your control class's `OnDraw` method.

```
virtual HRESULT OnDrawAdvanced(ATL_DRAWINFO& di);
```

Parameters

di

A reference to the [ATL_DRAWINFO](#) structure that contains drawing information such as the draw aspect, the control bounds, and whether the drawing is optimized or not.

Return Value

A standard HRESULT value.

Remarks

Override this method if you want to accept the device context passed by the container without normalizing it.

See [CComControlBase::OnDraw](#) for more details.

CComControlBase::OnKillFocus

Checks that the control is in-place active and has a valid control site, then informs the container that the control has lost focus.

```
LRESULT OnKillFocus(UINT /* nMsg */,
    WPARAM /* wParam */,
    LPARAM /* lParam */,
    BOOL& bHandled);
```

Parameters

nMsg

Reserved.

wParam

Reserved.

lParam

Reserved.

bHandled

Flag that indicates whether the window message was successfully handled. The default is FALSE.

Return Value

Always returns 1.

CComControlBase::OnMouseActivate

Checks that the UI is in user mode, then activates the control.

```
LRESULT OnMouseActivate(UINT /* nMsg */,
    WPARAM /* wParam */,
    LPARAM /* lParam */,
    BOOL& bHandled);
```

Parameters

nMsg

Reserved.

wParam

Reserved.

lParam

Reserved.

bHandled

Flag that indicates whether the window message was successfully handled. The default is FALSE.

Return Value

Always returns 1.

CComControlBase::OnPaint

Prepares the container for painting, gets the control's client area, then calls the control class's `OnDrawAdvanced` method.

```
LRESULT OnPaint(UINT /* nMsg */,
    WPARAM wParam,
    LPARAM /* lParam */,
    BOOL& /* lResult */);
```

Parameters

nMsg

Reserved.

wParam

An existing HDC.

lParam

Reserved.

lResult

Reserved.

Return Value

Always returns zero.

Remarks

If *wParam* is not NULL, `OnPaint` assumes it contains a valid HDC and uses it instead of [CComControlBase::m_hWndCD](#).

CComControlBase::OnSetFocus

Checks that the control is in-place active and has a valid control site, then informs the container the control has gained focus.

```
LRESULT OnSetFocus(UINT /* nMsg */,  
    WPARAM /* wParam */,  
    LPARAM /* lParam */,  
    BOOL& bHandled);
```

Parameters

nMsg

Reserved.

wParam

Reserved.

lParam

Reserved.

bHandled

Flag that indicates whether the window message was successfully handled. The default is FALSE.

Return Value

Always returns 1.

Remarks

Sends a notification to the container that the control has received focus.

CComControlBase::PreTranslateAccelerator

Override this method to provide your own keyboard accelerator handlers.

```
BOOL PreTranslateAccelerator(LPMSG /* pMsg */,  
    HRESULT& /* hRet */);
```

Parameters

pMsg

Reserved.

hRet

Reserved.

Return Value

By default returns FALSE.

CComControlBase::SendOnClose

Notifies all advisory sinks registered with the advise holder that the control has been closed.

```
HRESULT SendOnClose();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Sends a notification that the control has closed its advisory sinks.

CComControlBase::SendOnDataChange

Notifies all advisory sinks registered with the advise holder that the control data has changed.

```
HRESULT SendOnDataChange(DWORD advf = 0);
```

Parameters

advf

Advise flags that specify how the call to [IAdviseSink::OnDataChange](#) is made. Values are from the [ADV_F](#) enumeration.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CComControlBase::SendOnRename

Notifies all advisory sinks registered with the advise holder that the control has a new moniker.

```
HRESULT SendOnRename(IMoniker* pmk);
```

Parameters

pmk

Pointer to the new moniker of the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Sends a notification that the moniker for the control has changed.

CComControlBase::SendOnSave

Notifies all advisory sinks registered with the advise holder that the control has been saved.

```
HRESULT SendOnSave();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Sends a notification that the control has just saved its data.

CComControlBase::SendOnViewChange

Notifies all registered advisory sinks that the control's view has changed.

```
HRESULT SendOnViewChange(DWORD dwAspect, LONG lindex = -1);
```

Parameters

dwAspect

The aspect or view of the control.

lindex

The portion of the view that has changed. Only -1 is valid.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

`SendOnViewChange` calls [IAdviseSink::OnViewChange](#). The only value of *lindex* currently supported is -1, which indicates that the entire view is of interest.

CComControlBase::SetControlFocus

Sets or removes the keyboard focus to or from the control.

```
BOOL SetControlFocus(BOOL bGrab);
```

Parameters

bGrab

If TRUE, sets the keyboard focus to the calling control. If FALSE, removes the keyboard focus from the calling control, provided it has the focus.

Return Value

Returns TRUE if the control successfully receives focus; otherwise, FALSE.

Remarks

For a windowed control, the Windows API function [SetFocus](#) is called. For a windowless control, [IOleInPlaceSiteWindowless::SetFocus](#) is called. Through this call, a windowless control obtains the keyboard focus and can respond to window messages.

CComControlBase::SetDirty

Sets the data member `m_bRequiresSave` to the value in *bDirty*.

```
void SetDirty(BOOL bDirty);
```

Parameters

bDirty

Value of the data member [CComControlBase::m_bRequiresSave](#).

Remarks

`SetDirty(TRUE)` should be called to flag that the control has changed since it was last saved. The value of `m_bRequiresSave` is retrieved with [CComControlBase::GetDirty](#).

See also

[CComControl Class](#)

[Class Overview](#)

CComCriticalSection Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods for obtaining and releasing ownership of a critical section object.

Syntax

```
class CComCriticalSection
```

Members

Public Constructors

NAME	DESCRIPTION
CComCriticalSection::CComCriticalSection	The constructor.

Public Methods

NAME	DESCRIPTION
CComCriticalSection::Init	Creates and initializes a critical section object.
CComCriticalSection::Lock	Obtains ownership of the critical section object.
CComCriticalSection::Term	Releases system resources used by the critical section object.
CComCriticalSection::Unlock	Releases ownership of the critical section object.

Public Data Members

NAME	DESCRIPTION
CComCriticalSection::m_sec	A CRITICAL_SECTION object.

Remarks

`CComCriticalSection` is similar to class [CComAutoCriticalSection](#), except that you must explicitly initialize and release the critical section.

Typically, you use `CComCriticalSection` through the **typedef** name [CriticalSection](#). This name references `CComCriticalSection` when [CComMultiThreadModel](#) is being used.

See [CComCritSecLock Class](#) for a safer way to use this class than calling `Lock` and `Unlock` directly.

Requirements

Header: atlccore.h

CComCriticalSection::CComCriticalSection

The constructor.

```
CComCriticalSection() throw();
```

Remarks

Sets the [m_sec](#) data member to NULL.

CComCriticalSection::Init

Calls the Win32 function [InitializeCriticalSection](#), which initializes the critical section object contained in the [m_sec](#) data member.

```
HRESULT Init() throw();
```

Return Value

Returns S_OK on success, E_OUTOFMEMORY or E_FAIL on failure.

CComCriticalSection::Lock

Calls the Win32 function [EnterCriticalSection](#), which waits until the thread can take ownership of the critical section object contained in the [m_sec](#) data member.

```
HRESULT Lock() throw();
```

Return Value

Returns S_OK on success, E_OUTOFMEMORY or E_FAIL on failure.

Remarks

The critical section object must first be initialized with a call to the [Init](#) method. When the protected code has finished executing, the thread must call [Unlock](#) to release ownership of the critical section.

CComCriticalSection::m_sec

Contains a critical section object that is used by all `CComCriticalSection` methods.

```
CRITICAL_SECTION m_sec;
```

CComCriticalSection::Term

Calls the Win32 function [DeleteCriticalSection](#), which releases all resources used by the critical section object contained in the [m_sec](#) data member.

```
HRESULT Term() throw();
```

Return Value

Returns S_OK.

Remarks

Once `Term` has been called, the critical section can no longer be used for synchronization.

CComCriticalSection::Unlock

Calls the Win32 function [LeaveCriticalSection](#), which releases ownership of the critical section object contained in the `m_sec` data member.

```
HRESULT Unlock() throw();
```

Return Value

Returns `S_OK`.

Remarks

To first obtain ownership, the thread must call the [Lock](#) method. Each call to `Lock` requires a corresponding call to `Unlock` to release ownership of the critical section.

See also

[CComFakeCriticalSection Class](#)

[Class Overview](#)

[CComCritSecLock Class](#)

CComCritSecLock Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods for locking and unlocking a critical section object.

Syntax

```
template<class TLock> class CComCritSecLock
```

Parameters

TLock

The object to be locked and unlocked.

Members

Public Constructors

NAME	DESCRIPTION
CComCritSecLock::CComCritSecLock	The constructor.
CComCritSecLock::~~CComCritSecLock	The destructor.

Public Methods

NAME	DESCRIPTION
CComCritSecLock::Lock	Call this method to lock the critical section object.
CComCritSecLock::Unlock	Call this method to unlock the critical section object.

Remarks

Use this class to lock and unlock objects in a safer way than with the [CComCriticalSection Class](#) or [CComAutoCriticalSection Class](#).

Requirements

Header: atlbase.h

CComCritSecLock::CComCritSecLock

The constructor.

```
CComCritSecLock(TLock& cs, bool bInitialLock = true);
```

Parameters

cs

The critical section object.

bInitialLock

The initial lock state: **true** means locked.

Remarks

Initializes the critical section object.

CComCritSecLock::~CComCritSecLock

The destructor.

```
~CComCritSecLock() throw();
```

Remarks

Unlocks the critical section object.

CComCritSecLock::Lock

Call this method to lock the critical section object.

```
HRESULT Lock() throw();
```

Return Value

Returns S_OK if the object has successfully been locked, or an error HRESULT on failure.

Remarks

If the object is already locked, an ASSERT error will occur in debug builds.

CComCritSecLock::Unlock

Call this method to unlock the critical section object.

```
void Unlock() throw();
```

Remarks

If the object is already unlocked, an ASSERT error will occur in debug builds.

See also

[CComCriticalSection Class](#)

[CComAutoCriticalSection Class](#)

CComCurrency Class

3/4/2019 • 11 minutes to read • [Edit Online](#)

`CComCurrency` has methods and operators for creating and managing a CURRENCY object.

Syntax

```
class CComCurrency
```

Members

Public Constructors

NAME	DESCRIPTION
<code>CComCurrency::CComCurrency</code>	The constructor for a <code>CComCurrency</code> object.

Public Methods

NAME	DESCRIPTION
<code>CComCurrency::GetCurrencyPtr</code>	Returns the address of an <code>m_currency</code> data member.
<code>CComCurrency::GetFraction</code>	Call this method to return the fractional component of a <code>CComCurrency</code> object.
<code>CComCurrency::GetInteger</code>	Call this method to return the integer component of a <code>CComCurrency</code> object.
<code>CComCurrency::Round</code>	Call this method to round a <code>CComCurrency</code> object to the nearest integer value.
<code>CComCurrency::SetFraction</code>	Call this method to set the fractional component of a <code>CComCurrency</code> object.
<code>CComCurrency::SetInteger</code>	Call this method to set the integer component of a <code>CComCurrency</code> object.

Public Operators

NAME	DESCRIPTION
<code>CComCurrency::operator -</code>	This operator is used to perform subtraction on a <code>CComCurrency</code> object.
<code>CComCurrency::operator !=</code>	Compares two <code>CComCurrency</code> objects for inequality.
<code>CComCurrency::operator *</code>	This operator is used to perform multiplication on a <code>CComCurrency</code> object.

NAME	DESCRIPTION
<code>CComCurrency::operator *</code>	This operator is used to perform multiplication on a <code>CComCurrency</code> object and assign it the result.
<code>CComCurrency::operator /</code>	This operator is used to perform division on a <code>CComCurrency</code> object.
<code>CComCurrency::operator /=</code>	This operator is used to perform division on a <code>CComCurrency</code> object and assign it the result.
<code>CComCurrency::operator +</code>	This operator is used to perform addition on a <code>CComCurrency</code> object.
<code>CComCurrency::operator +=</code>	This operator is used to perform addition on a <code>CComCurrency</code> object and assign the result to the current object.
<code>CComCurrency::operator <</code>	This operator compares two <code>CComCurrency</code> objects to determine the lesser.
<code>CComCurrency::operator <=</code>	This operator compares two <code>CComCurrency</code> objects to determine equality or the lesser.
<code>CComCurrency::operator =</code>	This operator assigns the <code>CComCurrency</code> object to a new value.
<code>CComCurrency::operator -=</code>	This operator is used to perform subtraction on a <code>CComCurrency</code> object and assign it the result.
<code>CComCurrency::operator ==</code>	This operator compares two <code>CComCurrency</code> objects for equality.
<code>CComCurrency::operator ></code>	This operator compares two <code>CComCurrency</code> objects to determine the larger.
<code>CComCurrency::operator >=</code>	This operator compares two <code>CComCurrency</code> objects to determine equality or the larger.
<code>CComCurrency::operator CURRENCY</code>	Casts a CURRENCY object.

Public Data Members

NAME	DESCRIPTION
<code>CComCurrency::m_currency</code>	The CURRENCY variable created by your class instance.

Remarks

`CComCurrency` is a wrapper for the CURRENCY data type. CURRENCY is implemented as an 8-byte two's-complement integer value scaled by 10,000. This gives a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right. The CURRENCY data type is extremely useful for calculations involving money, or for any fixed-point calculations where accuracy is important.

The `CComCurrency` wrapper implements arithmetic, assignment, and comparison operations for this fixed-point type. The supported applications have been selected to control the rounding errors that can occur during fixed-point calculations.

The `CComCurrency` object provides access to the numbers on either side of the decimal point in the form of two components: an integer component which stores the value to the left of the decimal point, and a fractional component which stores the value to the right of the decimal point. The fractional component is stored internally as an integer value between -9999 (CY_MIN_FRACTION) and +9999 (CY_MAX_FRACTION). The method `CComCurrency::GetFraction` returns a value scaled by a factor of 10000 (CY_SCALE).

When specifying the integer and fractional components of a `CComCurrency` object, remember that the fractional component is a number in the range 0 to 9999. This is important when dealing with a currency such as the US dollar that expresses amounts using only two significant digits after the decimal point. Even though the last two digits are not displayed, they must be taken into account.

VALUE	POSSIBLE CCOMCURRENCY ASSIGNMENTS
\$10.50	<code>CComCurrency(10,5000)</code> or <code>CComCurrency(10.50)</code>
\$10.05	<code>CComCurrency(10,500)</code> or <code>CComCurrency(10.05)</code>

The values CY_MIN_FRACTION, CY_MAX_FRACTION, and CY_SCALE are defined in `atlcur.h`.

Requirements

Header: `atlcur.h`

CComCurrency::CComCurrency

The constructor.

```
CComCurrency() throw();
CComCurrency(const CComCurrency& curSrc) throw();
CComCurrency(CURRENCY cySrc) throw();
CComCurrency(DECIMAL dSrc);
CComCurrency(ULONG ulSrc);
CComCurrency(USHORT usSrc);
CComCurrency(CHAR cSrc);
CComCurrency(DOUBLE dSrc);
CComCurrency(FLOAT fSrc);
CComCurrency(LONG lSrc);
CComCurrency(SHORT sSrc);
CComCurrency(BYTE bSrc);
CComCurrency(LONGLONG nInteger, SHORT nFraction);
explicit CComCurrency(LPDISPATCH pDispSrc);
explicit CComCurrency(const VARIANT& varSrc);
explicit CComCurrency(LPCWSTR szSrc);
explicit CComCurrency(LPCSTR szSrc);
```

Parameters

curSrc

An existing `CComCurrency` object.

cySrc

A variable of type CURRENCY.

bSrc, dSrc, fSrc, lSrc, sSrc, ulSrc, usSrc

The initial value given to the member variable `m_currency`.

cSrc

A character containing the initial value given to the member variable `m_currency`.

nInteger, nFraction

The integer and fractional components of the initial monetary value. See the [CComCurrency](#) overview for more information.

pDispSrc

An `IDispatch` pointer.

varSrc

A variable of type VARIANT. The locale of the current thread is used to perform the conversion.

szSrc

A Unicode or ANSI string containing the initial value. The locale of the current thread is used to perform the conversion.

Remarks

The constructor sets the initial value of `CComCurrency::m_currency`, and accepts a wide range of data types, including integers, strings, floating-point numbers, CURRENCY variables, and other `CComCurrency` objects. If no value is provided, `m_currency` is set to 0.

In the event of an error, such as an overflow, the constructors lacking an empty exception specification (`throw()`) call `AtlThrow` with an HRESULT describing the error.

When using floating-point or double values to assign a value, note that `CComCurrency(10.50)` is equivalent to `CComCurrency(10,5000)` and not `CComCurrency(10,50)`.

CComCurrency::GetCurrencyPtr

Returns the address of an `m_currency` data member.

```
CURRENCY* GetCurrencyPtr() throw();
```

Return Value

Returns the address of an `m_currency` data member

CComCurrency::GetFraction

Call this method to return the fractional component of the `CComCurrency` object.

```
SHORT GetFraction() const;
```

Return Value

Returns the fractional component of the `m_currency` data member.

Remarks

The fractional component is a 4-digit integer value between -9999 (CY_MIN_FRACTION) and +9999 (CY_MAX_FRACTION). `GetFraction` returns this value scaled by 10000 (CY_SCALE). The values of CY_MIN_FRACTION, CY_MAX_FRACTION, and CY_SCALE are defined in `atlcure.h`.

Example

```
CComCurrency cur(10, 5000);
int nFract;
nFract = cur.GetFraction();
ATLASSERT(nFract == 5000);
```

CComCurrency::GetInteger

Call this method to get the integer component of a `CComCurrency` object.

```
LONGLONG GetInteger() const;
```

Return Value

Returns the integer component of the `m_currency` data member.

Example

```
CComCurrency cur(10, 5000);
LONGLONG nInteger;
nInteger = cur.GetInteger();
ATLASSERT(nInteger == 10);
```

CComCurrency::m_currency

The CURRENCY data member.

```
CURRENCY m_currency;
```

Remarks

This member holds the currency accessed and manipulated by the methods of this class.

CComCurrency::operator -

This operator is used to perform subtraction on a `CComCurrency` object.

```
CComCurrency operator-() const;
CComCurrency operator-(const CComCurrency& cur) const;
```

Parameters

cur

A `CComCurrency` object.

Return Value

Returns a `CComCurrency` object representing the result of the subtraction. In the event of an error, such as an overflow, this operator calls `AtlThrow` with an HRESULT describing the error.

Example

```
CComCurrency cur1(10, 5000), cur2;
cur2 = cur1 - CComCurrency(4, 5000);
ATLASSERT(cur2 == CComCurrency(6, 0));
```


CComCurrency::operator !=

This operator compares two objects for inequality.

```
bool operator!= (const CComCurrency& cur) const;
```

Parameters

cur

The `CComCurrency` object to be compared.

Return Value

Returns TRUE if the item being compared is not equal to the `CComCurrency` object; otherwise, FALSE.

Example

```
CComCurrency cur1(10, 5000), cur2(10, 5001);  
ATLASSERT(cur1 != cur2);
```

CComCurrency::operator *

This operator is used to perform multiplication on a `CComCurrency` object.

```
CComCurrency operator*(long nOperand) const;  
CComCurrency operator*(const CComCurrency& cur) const;
```

Parameters

nOperand

The multiplier.

cur

The `CComCurrency` object used as the multiplier.

Return Value

Returns a `CComCurrency` object representing the result of the multiplication. In the event of an error, such as an overflow, this operator calls `AtlThrow` with an HRESULT describing the error.

Example

```
CComCurrency cur1(10, 5000), cur2;  
cur2 = cur1 * 2;  
ATLASSERT(cur2 == CComCurrency(21, 0));
```

CComCurrency::operator *=

This operator is used to perform multiplication on a `CComCurrency` object and assign it the result.

```
const CComCurrency& operator*= (long nOperand);  
const CComCurrency& operator*= (const CComCurrency& cur);
```

Parameters

nOperand

The multiplier.

cur

The `CComCurrency` object used as the multiplier.

Return Value

Returns the updated `CComCurrency` object. In the event of an error, such as an overflow, this operator calls `AtlThrow` with an HRESULT describing the error.

Example

```
CComCurrency cur(10, 5000);  
cur *= 2;  
ATLASSERT(cur == CComCurrency(21, 0));
```

CComCurrency::operator /

This operator is used to perform division on a `CComCurrency` object.

```
CComCurrency operator/(long nOperand) const;
```

Parameters

nOperand

The divisor.

Return Value

Returns a `CComCurrency` object representing the result of the division. If the divisor is 0, an assert failure will occur.

Example

```
CComCurrency cur1(10, 5000), cur2;  
cur2 = cur1 / 10;  
ATLASSERT(cur2 == CComCurrency(1, 500));
```

CComCurrency::operator /=

This operator is used to perform division on a `CComCurrency` object and assign it the result.

```
const CComCurrency& operator/= (long nOperand);
```

Parameters

nOperand

The divisor.

Return Value

Returns the updated `CComCurrency` object. If the divisor is 0, an assert failure will occur.

Example

```
CComCurrency cur(10, 5000);  
cur /= 10;  
ATLASSERT(cur == CComCurrency(1, 500));
```

CComCurrency::operator +

This operator is used to perform addition on a `CComCurrency` object.

```
CComCurrency operator+(const CComCurrency& cur) const;
```

Parameters

cur

The `CComCurrency` object to be added to the original object.

Return Value

Returns a `CComCurrency` object representing the result of the addition. In the event of an error, such as an overflow, this operator calls `AtlThrow` with an HRESULT describing the error.

Example

```
CComCurrency cur1(10, 5000), cur2;  
cur2 = cur1 + CComCurrency(4, 5000);  
ATLASSERT(cur2 == CComCurrency(15, 0));
```

CComCurrency::operator +=

This operator is used to perform addition on a `CComCurrency` object and assign the result to the current object.

```
const CComCurrency& operator+= (const CComCurrency& cur);
```

Parameters

cur

The `CComCurrency` object.

Return Value

Returns the updated `CComCurrency` object. In the event of an error, such as an overflow, this operator calls `AtlThrow` with an HRESULT describing the error.

Example

```
CComCurrency cur(10, 2500);  
cur += CComCurrency(4, 2500);  
ATLASSERT(cur == CComCurrency(14, 5000));
```

CComCurrency::operator <

This operator compares two `CComCurrency` objects to determine the lesser.

```
bool operator<(const CComCurrency& cur) const;
```

Parameters

cur

A `CComCurrency` object.

Return Value

Returns TRUE if the first object is less than the second, FALSE otherwise.

Example

```
CComCurrency cur1(10, 4900);  
CComCurrency cur2(10, 5000);  
ATLASSERT(cur1 < cur2);
```

CComCurrency::operator <=

This operator compares two `CComCurrency` objects to determine equality or the lesser.

```
bool operator<= (const CComCurrency& cur) const;
```

Parameters

cur

A `CComCurrency` object.

Return Value

Returns TRUE if the first object is less than or equal to the second, FALSE otherwise.

Example

```
CComCurrency cur1(10, 4900);  
CComCurrency cur2(10, 5000);  
ATLASSERT(cur1 <= cur2);
```

CComCurrency::operator =

This operator assigns the `CComCurrency` object to a new value.

```
const CComCurrency& operator= (const CComCurrency& curSrc) throw();  
const CComCurrency& operator= (CURRENCY cySrc) throw();  
const CComCurrency& operator= (FLOAT fSrc);  
const CComCurrency& operator= (SHORT sSrc);  
const CComCurrency& operator= (LONG lSrc);  
const CComCurrency& operator= (BYTE bSrc);  
const CComCurrency& operator= (USHORT usSrc);  
const CComCurrency& operator= (DOUBLE dSrc);  
const CComCurrency& operator= (CHAR cSrc);  
const CComCurrency& operator= (ULONG ulSrc);  
const CComCurrency& operator= (DECIMAL dSrc);
```

Parameters

curSrc

A `CComCurrency` object.

cySrc

A variable of type CURRENCY.

sSrc, fSrc, lSrc, bSrc, usSrc, dSrc, cSrc, ulSrc, dSrc

The numeric value to assign to the `CComCurrency` object.

Return Value

Returns the updated `CComCurrency` object. In the event of an error, such as an overflow, this operator calls `AtlThrow` with an HRESULT describing the error.

Example

```
CComCurrency cur1, cur2(10, 5000);
CURRENCY cy;

// Copying one object to another
cur1 = cur2;

// Using the CURRENCY data type
cy.int64 = 105000;
cur1 = cy;

ATLASSERT(cur1 == cur2);
```

CComCurrency::operator -=

This operator is used to perform subtraction on a `CComCurrency` object and assign it the result.

```
const CComCurrency& operator-= (const CComCurrency& cur);
```

Parameters

cur

A `CComCurrency` object.

Return Value

Returns the updated `CComCurrency` object. In the event of an error, such as an overflow, this operator calls `AtlThrow` with an HRESULT describing the error.

Example

```
CComCurrency cur(10, 5000);
cur -= CComCurrency(4, 5000);
ATLASSERT(cur == CComCurrency(6, 0));
```

CComCurrency::operator ==

This operator compares two `CComCurrency` objects for equality.

```
bool operator== (const CComCurrency& cur) const;
```

Parameters

cur

The `CComCurrency` object to compare.

Return Value

Returns TRUE if the objects are equal (that is, the `m_currency` data members, both integer and fractional, in both objects have the same value), FALSE otherwise.

Example

```
CComCurrency cur1(10, 5000), cur2;
cur2 = cur1;
ATLASSERT(cur1 == cur2);
```

CComCurrency::operator >

This operator compares two `CComCurrency` objects to determine the larger.

```
bool operator>(const CComCurrency& cur) const;
```

Parameters

cur

A `CComCurrency` object.

Return Value

Returns TRUE if the first object is greater than the second, FALSE otherwise.

Example

```
CComCurrency cur1(10, 5100);  
CComCurrency cur2(10, 5000);  
ATLASSERT(cur1 > cur2);
```

CComCurrency::operator >=

This operator compares two `CComCurrency` objects to determine equality or the larger.

```
bool operator>= (const CComCurrency& cur) const;
```

Parameters

cur

A `CComCurrency` object.

Return Value

Returns TRUE if the first object is greater than or equal to the second, FALSE otherwise.

Example

```
CComCurrency cur1(10, 5100);  
CComCurrency cur2(10, 5000);  
ATLASSERT(cur1 >= cur2);
```

CComCurrency::operator CURRENCY

These operators are used to cast a `CComCurrency` object to a CURRENCY data type.

```
operator CURRENCY&() throw();  
operator const CURRENCY&() const throw();
```

Return Value

Returns a reference to a CURRENCY object.

Example

```
CComCurrency cur(10, 5000);  
CURRENCY cy = static_cast<CURRENCY>(cur); // Note that explicit cast is not necessary  
ATLASSERT(cy.int64 == 105000);
```

CComCurrency::Round

Call this method to round the currency to a specified number of decimal places.

```
HRESULT Round(int nDecimals);
```

Parameters

nDecimals

The number of digits to which `m_currency` will be rounded, in the range 0 to 4.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

```
CComCurrency cur(10, 1234);  
cur.Round(3);  
ATLASSERT(cur.GetFraction() == 1230);
```

CComCurrency::SetFraction

Call this method to set the fractional component of a `CComCurrency` object.

```
HRESULT SetFraction(SHORT nFraction);
```

Parameters

nFraction

The value to be assigned to the fractional component of the `m_currency` data member. The sign of the fractional component must be the same as the integer component, and the value must be in range -9999 (CY_MIN_FRACTION) to +9999 (CY_MAX_FRACTION).

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

```
CComCurrency cur(10, 0);  
cur.SetFraction(5000);  
ATLASSERT(CComCurrency(10, 5000) == cur);
```

CComCurrency::SetInteger

Call this method to set the integer component of a `CComCurrency` object.

```
HRESULT SetInteger(LONGLONG nInteger);
```

Parameters

nInteger

The value to be assigned to the integer component of the `m_currency` data member. The sign of the integer component must match the sign of the existing fractional component.

nInteger must be in the range `CY_MIN_INTEGER` to `CY_MAX_INTEGER` inclusive. These values are defined in `atlcyr.h`.

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure.

Example

```
CComCurrency cur(0, 5000);  
cur.SetInteger(10);  
ATLASSERT(CComCurrency(10, 5000) == cur);
```

See also

[COleCurrency Class](#)

[CURRENCY](#)

[Class Overview](#)

CComDynamicUnkArray Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

This class stores an array of `IUnknown` pointers.

Syntax

```
class CComDynamicUnkArray
```

Members

Public Constructors

NAME	DESCRIPTION
CComDynamicUnkArray::CComDynamicUnkArray	Constructor. Initializes the collection values to NULL and the collection size to zero.
CComDynamicUnkArray::~~CComDynamicUnkArray	The destructor.

Public Methods

NAME	DESCRIPTION
CComDynamicUnkArray::Add	Call this method to add an <code>IUnknown</code> pointer to the array.
CComDynamicUnkArray::begin	Returns a pointer to the first <code>IUnknown</code> pointer in the collection.
CComDynamicUnkArray::clear	Empties the array.
CComDynamicUnkArray::end	Returns a pointer to one past the last <code>IUnknown</code> pointer in the collection.
CComDynamicUnkArray::GetAt	Retrieves the element at the specified index.
CComDynamicUnkArray::GetCookie	Call this method to get the cookie associated with a given <code>IUnknown</code> pointer.
CComDynamicUnkArray::GetSize	Returns the length of an array.
CComDynamicUnkArray::GetUnknown	Call this method to get the <code>IUnknown</code> pointer associated with a given cookie.
CComDynamicUnkArray::Remove	Call this method to remove an <code>IUnknown</code> pointer from the array.

Remarks

`CComDynamicUnkArray` holds a dynamically allocated array of `IUnknown` pointers, each an interface on a connection point. `CComDynamicUnkArray` can be used as a parameter to the [IConnectionPointImpl](#) template class.

The `CComDynamicUnkArray` methods [begin](#) and [end](#) can be used to loop through all connection points (for example, when an event is fired).

See [Adding Connection Points to an Object](#) for details on automating creation of connection point proxies.

NOTE

Note The class `CComDynamicUnkArray` is used by the **Add Class** wizard when creating a control which has Connection Points. If you wish to specify the number of Connection Points manually, change the reference from `CComDynamicUnkArray` to `CComUnkArray< n >`, where *n* is the number of connection points required.

Requirements

Header: atlcom.h

CComDynamicUnkArray::Add

Call this method to add an `IUnknown` pointer to the array.

```
DWORD Add(IUnknown* pUnk);
```

Parameters

pUnk

The `IUnknown` pointer to add to the array.

Return Value

Returns the cookie associated with the newly added pointer.

CComDynamicUnkArray::begin

Returns a pointer to the beginning of the collection of `IUnknown` interface pointers.

```
IUnknown**  
begin();
```

Return Value

A pointer to an `IUnknown` interface pointer.

Remarks

The collection contains pointers to interfaces stored locally as `IUnknown`. You cast each `IUnknown` interface to the real interface type and then call through it. You do not need to query for the interface first.

Before using the `IUnknown` interface, you should check that it is not NULL.

CComDynamicUnkArray::clear

Empties the array.

```
void clear();
```

CComDynamicUnkArray::CComDynamicUnkArray

The constructor.

```
CComDynamicUnkArray();
```

Remarks

Sets the collection size to zero and initializes the values to NULL. The destructor frees the collection, if necessary.

CComDynamicUnkArray::~~CComDynamicUnkArray

The destructor.

```
~CComDynamicUnkArray();
```

Remarks

Frees resources allocated by the class constructor.

CComDynamicUnkArray::end

Returns a pointer to one past the last `IUnknown` pointer in the collection.

```
IUnknown**  
    end();
```

Return Value

A pointer to an `IUnknown` interface pointer.

CComDynamicUnkArray::GetAt

Retrieves the element at the specified index.

```
IUnknown* GetAt(int nIndex);
```

Parameters

nIndex

The index of the element to retrieve.

Return Value

A pointer to an `IUnknown` interface.

CComDynamicUnkArray::GetCookie

Call this method to get the cookie associated with a given `IUnknown` pointer.

```
DWORD WINAPI GetCookie(IUnknown** ppFind);
```

Parameters

ppFind

The `IUnknown` pointer for which the associated cookie is required.

Return Value

Returns the cookie associated with the `IUnknown` pointer, or zero if no matching `IUnknown` pointer is found.

Remarks

If there is more than one instance of the same `IUnknown` pointer, this function returns the cookie for the first one.

CComDynamicUnkArray::GetSize

Returns the length of an array.

```
int GetSize() const;
```

Return Value

The length of the array.

CComDynamicUnkArray::GetUnknown

Call this method to get the `IUnknown` pointer associated with a given cookie.

```
IUnknown* WINAPI GetUnknown(DWORD dwCookie);
```

Parameters

dwCookie

The cookie for which the associated `IUnknown` pointer is required.

Return Value

Returns the `IUnknown` pointer, or NULL if no matching cookie is found.

CComDynamicUnkArray::Remove

Call this method to remove an `IUnknown` pointer from the array.

```
BOOL Remove(DWORD dwCookie);
```

Parameters

dwCookie

The cookie referencing the `IUnknown` pointer to be removed from the array.

Return Value

Returns TRUE if the pointer is removed; otherwise FALSE.

See also

[CComUnkArray Class](#)

[Class Overview](#)

CComEnum Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class defines a COM enumerator object based on an array.

Syntax

```
template <class Base,  
    const IID* piid, class T, class Copy, class ThreadModel = CComObjectThreadModel>  
class ATL_NO_VTABLE CComEnum : public CComEnumImpl<Base, piid,  
    T,  
    Copy>,  
    public CComObjectRootEx<ThreadModel>
```

Parameters

Base

A COM enumerator interface. See [IEnumString](#) for an example.

piid

A pointer to the interface ID of the enumerator interface.

T

The type of item exposed by the enumerator interface.

Copy

A homogeneous [copy policy class](#).

ThreadModel

The threading model of the class. This parameter defaults to the global object thread model used in your project.

Remarks

`CComEnum` defines a COM enumerator object based on an array. This class is analogous to [CComEnumOnSTL](#) which implements an enumerator based on a C++ Standard Library container. Typical steps for using this class are outlined below. For more information, see [ATL Collections and Enumerators](#).

To use this class:

- **typedef** a specialization of this class.
- Use the **typedef** as the template argument in a specialization of `CComObject`.
- Create an instance of the `CComObject` specialization.
- Initialize the enumerator object by calling [CComEnumImpl::Init](#).
- Return the enumerator interface to the client.

Inheritance Hierarchy

`CComObjectRootBase`

`Base`

[CComObjectRootEx](#)

[CComEnumImpl](#)

[CComEnum](#)

Requirements

Header: atlcom.h

Example

The code shown below provides a reusable function for creating and initializing an enumerator object.

```
template <class EnumType, class ElementType>
HRESULT CreateEnumerator(IUnknown** ppUnk, ElementType* begin, ElementType* end,
    IUnknown* pUnk, CComEnumFlags flags)
{
    if (ppUnk == NULL)
        return E_POINTER;
    *ppUnk = NULL;

    CComObject<EnumType>* pEnum = NULL;
    HRESULT hr = CComObject<EnumType>::CreateInstance(&pEnum);

    if (FAILED(hr))
        return hr;

    hr = pEnum->Init(begin, end, pUnk, flags);

    if (SUCCEEDED(hr))
        hr = pEnum->QueryInterface(ppUnk);

    if (FAILED(hr))
        delete pEnum;

    return hr;
} // CreateEnumerator
```

This template function can be used to implement the [_NewEnum](#) property of a collection interface as shown below:

```
typedef CComEnum<IEnumVARIANT, &IID_IEnumVARIANT, VARIANT, _Copy<VARIANT> > VarArrEnum;

class ATL_NO_VTABLE CVariantArrayCollection :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CVariantArrayCollection, &CLSID_VariantArrayCollection>,
    public IDispatchImpl<IVariantArrayCollection, &IID_IVariantArrayCollection, &LIBID_NVC_ATL_COMLib, /*wMajor
= */ 1, /*wMinor = */ 0>
{
    VARIANT m_arr[3];
public:
    STDMETHOD(get__NewEnum)(IUnknown** ppUnk)
    {
        return CreateEnumerator<VarArrEnum>(ppUnk, &m_arr[0], &m_arr[3], this,
            AtlFlagNoCopy);
    }

    // Remainder of class declaration omitted.
```

This code creates a **typedef** for [CComEnum](#) that exposes a vector of `VARIANT`s through the [IEnumVariant](#) interface. The [CVariantArrayCollection](#) class simply specializes [CreateEnumerator](#) to work with enumerator objects of this type and passes the necessary arguments.

See also

[Class Overview](#)

[CComObjectThreadModel](#)

[CComEnumImpl Class](#)

[CComObjectRootEx Class](#)

CComEnumImpl Class

3/4/2019 • 6 minutes to read • [Edit Online](#)

This class provides the implementation for a COM enumerator interface where the items being enumerated are stored in an array.

Syntax

```
template <class Base,  
         const IID* piid, class T, class Copy>  
class ATL_NO_VTABLE CComEnumImpl : public Base
```

Parameters

Base

A COM enumerator interface. See [IEnumString](#) for an example.

piid

A pointer to the interface ID of the enumerator interface.

T

The type of item exposed by the enumerator interface.

Copy

A homogeneous [copy policy class](#).

Members

Public Constructors

NAME	DESCRIPTION
CComEnumImpl::CComEnumImpl	The constructor.
CComEnumImpl::~~CComEnumImpl	The destructor.

Public Methods

NAME	DESCRIPTION
CComEnumImpl::Clone	The implementation of the Clone enumeration interface method.
CComEnumImpl::Init	Initializes the enumerator.
CComEnumImpl::Next	The implementation of Next .
CComEnumImpl::Reset	The implementation of Reset .
CComEnumImpl::Skip	The implementation of Skip .

Public Data Members

NAME	DESCRIPTION
CComEnumImpl::m_begin	A pointer to the first item in the array.
CComEnumImpl::m_dwFlags	Copy flags passed through <code>Init</code> .
CComEnumImpl::m_end	A pointer to the location just beyond the last item in the array.
CComEnumImpl::m_iter	A pointer to the current item in the array.
CComEnumImpl::m_spUnk	The <code>IUnknown</code> pointer of the object supplying the collection being enumerated.

Remarks

See [IEnumString](#) for an example of method implementations. `CComEnumImpl` provides the implementation for a COM enumerator interface where the items being enumerated are stored in an array. This class is analogous to the `IEnumOnSTLImpl` class, which provides an implementation of an enumerator interface based on a C++ Standard Library container.

NOTE

For details on further differences between `CComEnumImpl` and `IEnumOnSTLImpl`, see [CComEnumImpl::Init](#).

Typically, you will *not* need to create your own enumerator class by deriving from this interface implementation. If you want to use an ATL-supplied enumerator based on an array, it is more common to create an instance of [CComEnum](#).

However, if you do need to provide a custom enumerator (for example, one that exposes interfaces in addition to the enumerator interface), you can derive from this class. In this situation, it is likely that you'll need to override the [CComEnumImpl::Clone](#) method to provide your own implementation.

For more information, see [ATL Collections and Enumerators](#).

Inheritance Hierarchy

Base

`CComEnumImpl`

Requirements

Header: atlcom.h

CComEnumImpl::CComEnumImpl

The constructor.

```
CComEnumImpl();
```

CComEnumImpl::~~CComEnumImpl

The destructor.

```
~CComEnumImpl();
```

CComEnumImpl::Init

You must call this method before passing a pointer to the enumerator interface back to any clients.

```
HRESULT Init(  
    T* begin,  
    T* end,  
    IUnknown* pUnk,  
    CComEnumFlags flags = AtlFlagNoCopy);
```

Parameters

begin

A pointer to the first element of the array containing the items to be enumerated.

end

A pointer to the location just beyond the last element of the array containing the items to be enumerated.

pUnk

[in] The `IUnknown` pointer of an object that must be kept alive during the lifetime of the enumerator. Pass NULL if no such object exists.

flags

Flags specifying whether or not the enumerator should take ownership of the array or make a copy of it. Possible values are described below.

Return Value

A standard HRESULT value.

Remarks

Only call this method once — initialize the enumerator, use it, then throw it away.

If you pass pointers to items in an array held in another object (and you don't ask the enumerator to copy the data), you can use the *pUnk* parameter to ensure that the object and the array it holds are available for as long as the enumerator needs them. The enumerator simply holds a COM reference on the object to keep it alive. The COM reference is automatically released when the enumerator is destroyed.

The *flags* parameter allows you to specify how the enumerator should treat the array elements passed to it. *flags* can take one of the values from the `CComEnumFlags` enumeration shown below:

```
enum CComEnumFlags  
{  
    AtlFlagNoCopy = 0,  
    AtlFlagTakeOwnership = 2, // BitOwn  
    AtlFlagCopy = 3          // BitOwn | BitCopy  
};
```

`AtlFlagNoCopy` means that the array's lifetime is not controlled by the enumerator. In this case, either the array will be static or the object identified by *pUnk* will be responsible for freeing the array when it's no longer needed.

`AtlFlagTakeOwnership` means that the destruction of the array is to be controlled by the enumerator. In this case, the array must have been dynamically allocated using **new**. The enumerator will delete the array in its destructor.

Typically, you would pass NULL for *pUnk*, although you can still pass a valid pointer if you need to be notified of the destruction of the enumerator for some reason.

`AtlFlagCopy` means that a new array is to be created by copying the array passed to `Init`. The new array's lifetime is to be controlled by the enumerator. The enumerator will delete the array in its destructor. Typically, you would pass NULL for *pUnk*, although you can still pass a valid pointer if you need to be notified of the destruction of the enumerator for some reason.

NOTE

The prototype of this method specifies the array elements as being of type `T`, where `T` was defined as a template parameter to the class. This is the same type that is exposed by means of the COM interface method [CComEnumImpl::Next](#). The implication of this is that, unlike [IEnumOnSTLImpl](#), this class does not support different storage and exposed data types. The data type of elements in the array must be the same as the data type exposed by means of the COM interface.

CComEnumImpl::Clone

This method provides the implementation of the **Clone** method by creating an object of type `CComEnum`, initializing it with the same array and iterator used by the current object, and returning the interface on the newly created object.

```
STDMETHOD(Clone)(Base** ppEnum);
```

Parameters

ppEnum

[out] The enumerator interface on a newly created object cloned from the current enumerator.

Return Value

A standard HRESULT value.

Remarks

Note that cloned enumerators never make their own copy (or take ownership) of the data used by the original enumerator. If necessary, cloned enumerators will keep the original enumerator alive (using a COM reference) to ensure that the data is available for as long as they need it.

CComEnumImpl::m_spUnk

This smart pointer maintains a reference on the object passed to [CComEnumImpl::Init](#), ensuring that it remains alive during the lifetime of the enumerator.

```
CComPtr<IUnknown> m_spUnk;
```

CComEnumImpl::m_begin

A pointer to the location just beyond the last element of the array containing the items to be enumerated.

```
T* m_begin;
```

CComEnumImpl::m_end

A pointer to the first element of the array containing the items to be enumerated.

```
T* m_end;
```

CComEnumImpl::m_iter

A pointer to the current element of the array containing the items to be enumerated.

```
T* m_iter;
```

CComEnumImpl::m_dwFlags

The flags passed to [CComEnumImpl::Init](#).

```
DWORD m_dwFlags;
```

CComEnumImpl::Next

This method provides the implementation of the **Next** method.

```
STDMETHOD(Next)(ULONG celt, T* rgelt, ULONG* pceltFetched);
```

Parameters

celt

[in] The number of elements requested.

rgelt

[out] The array to be filled with the elements.

pceltFetched

[out] The number of elements actually returned in *rgelt*. This can be less than *celt* if fewer than *celt* elements remained in the list.

Return Value

A standard HRESULT value.

CComEnumImpl::Reset

This method provides the implementation of the **Reset** method.

```
STDMETHOD(Reset)(void);
```

Return Value

A standard HRESULT value.

CComEnumImpl::Skip

This method provides the implementation of the **Skip** method.

```
STDMETHOD(Skip)(ULONG celt);
```

Parameters

celt

[in] The number of elements to skip.

Return Value

A standard HRESULT value.

Remarks

Returns E_INVALIDARG if *celt* is zero, returns S_FALSE if less than *celt* elements are returned, returns S_OK otherwise.

See also

[IEnumOnSTLImpl Class](#)

[CComEnum Class](#)

[Class Overview](#)

CComEnumOnSTL Class

4/1/2019 • 2 minutes to read • [Edit Online](#)

This class defines a COM enumerator object based on a C++ Standard Library collection.

Syntax

```
template <class Base,
    const IID* piid, class T, class Copy, class CollType, class ThreadModel = CComObjectThreadModel>
class ATL_NO_VTABLE CComEnumOnSTL : public IEnumOnSTLImpl<Base, piid,
    T,
    Copy,
    CollType>,
    public CComObjectRootEx<ThreadModel>
```

Parameters

Base

A COM enumerator. See [IEnumString](#) for an example.

piid

A pointer to the interface ID of the enumerator interface.

T

The type of item exposed by the enumerator interface.

Copy

A [copy policy](#) class.

CollType

A C++ Standard Library container class.

Remarks

`CComEnumOnSTL` defines a COM enumerator object based on a C++ Standard Library collection. This class can be used on its own or in conjunction with [ICollectionOnSTLImpl](#). Typical steps for using this class are outlined below. For more information, see [ATL Collections and Enumerators](#).

To use this class with ICollectionOnSTLImpl:

- **typedef** a specialization of this class.
- Use the **typedef** as the final template argument in a specialization of `ICollectionOnSTLImpl`.

See [ATL Collections and Enumerators](#) for an example.

To use this class independently of ICollectionOnSTLImpl:

- **typedef** a specialization of this class.
- Use the **typedef** as the template argument in a specialization of `CComObject`.
- Create an instance of the `CComObject` specialization.
- Initialize the enumerator object by calling [IEnumOnSTLImpl::Init](#).

- Return the enumerator interface to the client.

Inheritance Hierarchy

CComObjectRootBase

Base

CComObjectRootEx

IEnumOnSTLImpl

CComEnumOnSTL

Requirements

Header: atlcom.h

Example

The code shown below provides a generic function to handle the creation and initialization of an enumerator object:

```
template <class EnumType, class CollType>
HRESULT CreateSTLEnumerator(IUnknown** ppUnk, IUnknown* pUnkForRelease,
    CollType& collection)
{
    if (ppUnk == NULL)
        return E_POINTER;
    *ppUnk = NULL;

    CComObject<EnumType>* pEnum = NULL;
    HRESULT hr = CComObject<EnumType>::CreateInstance(&pEnum);

    if (FAILED(hr))
        return hr;

    hr = pEnum->Init(pUnkForRelease, collection);

    if (SUCCEEDED(hr))
        hr = pEnum->QueryInterface(ppUnk);

    if (FAILED(hr))
        delete pEnum;

    return hr;
} // CreateSTLEnumerator
```

This template function can be used to implement the `_NewEnum` property of a collection interface as shown below:

```

typedef CComEnumOnSTL<IEnumVARIANT, &IID_IEnumVARIANT, VARIANT, _Copy<VARIANT>,
    std::vector<CComVariant> > VarVarEnum;

class ATL_NO_VTABLE CVariantCollection :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CVariantCollection, &CLSID_VariantCollection>,
    public IDispatchImpl<IVariantCollection, &IID_IVariantCollection, &LIBID_NVC_ATL_COMLib, /*wMajor =*/ 1,
/*wMinor =*/ 0>
{
public:
    std::vector<CComVariant> m_vec;

    STDMETHOD(get__NewEnum)(IUnknown** ppUnk)
    {
        return CreateSTLEnumerator<VarVarEnum>(ppUnk, this, m_vec);
    }

    // Remainder of class declaration omitted.

```

This code creates a **typedef** for `CComEnumOnSTL` that exposes a vector of `CComVariant` s by means of the `IEnumVariant` interface. The `CVariantCollection` class simply specializes `CreateSTLEnumerator` to work with enumerator objects of this type.

See also

[IEnumOnSTLImpl](#)

[ATLCollections Sample: Demonstrates ICollectionOnSTLImpl, CComEnumOnSTL, and Custom Copy Policy Classes](#)

[Class Overview](#)

[CComObjectRootEx Class](#)

[CComObjectThreadModel](#)

[IEnumOnSTLImpl Class](#)

CComFakeCriticalSection Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides the same methods as [CComCriticalSection](#) but does not provide a critical section.

Syntax

```
class CComFakeCriticalSection
```

Members

Public Methods

NAME	DESCRIPTION
CComFakeCriticalSection::Init	Does nothing since there is no critical section.
CComFakeCriticalSection::Lock	Does nothing since there is no critical section.
CComFakeCriticalSection::Term	Does nothing since there is no critical section.
CComFakeCriticalSection::Unlock	Does nothing since there is no critical section.

Remarks

`CComFakeCriticalSection` mirrors the methods found in [CComCriticalSection](#). However, `CComFakeCriticalSection` does not provide a critical section; therefore, its methods do nothing.

Typically, you use `CComFakeCriticalSection` through a `typedef` name, either `AutoCriticalSection` or `CriticalSection`. When using [CComSingleThreadModel](#) or [CComMultiThreadModelNoCS](#), both of these `typedef` names reference `CComFakeCriticalSection`. When using [CComMultiThreadModel](#), they reference [CComAutoCriticalSection](#) and `CComCriticalSection`, respectively.

Requirements

Header: atlcore.h

CComFakeCriticalSection::Init

Does nothing since there is no critical section.

```
HRESULT Init() throw();
```

Return Value

Returns S_OK.

CComFakeCriticalSection::Lock

Does nothing since there is no critical section.

```
HRESULT Lock() throw();
```

Return Value

Returns S_OK.

CComFakeCriticalSection::Term

Does nothing since there is no critical section.

```
HRESULT Term() throw();
```

Return Value

Returns S_OK.

CComFakeCriticalSection::Unlock

Does nothing since there is no critical section.

```
HRESULT Unlock() throw();
```

Return Value

Returns S_OK.

See also

[Class Overview](#)

CComGITPtr Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

This class provides methods for dealing with interface pointers and the global interface table (GIT).

Syntax

```
template <class T>
class CComGITPtr
```

Parameters

T

The type of the interface pointer to be stored in the GIT.

Members

Public Constructors

NAME	DESCRIPTION
CComGITPtr::CComGITPtr	The constructor.
CComGITPtr::~~CComGITPtr	The destructor.

Public Methods

NAME	DESCRIPTION
CComGITPtr::Attach	Call this method to register the interface pointer in the global interface table (GIT).
CComGITPtr::CopyTo	Call this method to copy the interface from the global interface table (GIT) to the passed pointer.
CComGITPtr::Detach	Call this method to disassociate the interface from the <code>CComGITPtr</code> object.
CComGITPtr::GetCookie	Call this method to return the cookie from the <code>CComGITPtr</code> object.
CComGITPtr::Revoke	Call this method to remove the interface from the global interface table (GIT).

Public Operators

NAME	DESCRIPTION
CComGITPtr::operator DWORD	Returns the cookie from the <code>CComGITPtr</code> object.
CComGITPtr::operator =	Assignment operator.

Public Data Members

NAME	DESCRIPTION
CComGITPtr::m_dwCookie	The cookie.

Remarks

Objects that aggregate the free threaded marshaler and need to use interface pointers obtained from other objects must take extra steps to ensure that the interfaces are correctly marshaled. Typically this involves storing the interface pointers in the GIT and getting the pointer from the GIT each time it is used. The class `CComGITPtr` is provided to help you use interface pointers stored in the GIT.

NOTE

The global interface table facility is only available on Windows 95 with DCOM version 1.1 and later, Windows 98, Windows NT 4.0 with Service Pack 3 and later, and Windows 2000.

Requirements

Header: `atlbase.h`

CComGITPtr::Attach

Call this method to register the interface pointer in the global interface table (GIT).

```
HRESULT Attach(T* p) throw();

HRESULT Attach(DWORD dwCookie) throw();
```

Parameters

p

The interface pointer to be added to the GIT.

dwCookie

The cookie used to identify the interface pointer.

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure.

Remarks

In debug builds, an assertion error will occur if the GIT is not valid, or if the cookie is equal to `NULL`.

CComGITPtr::CComGITPtr

The constructor.

```
CComGITPtr() throw();
CComGITPtr(T* p);
CComGITPtr(const CComGITPtr& git);
explicit CComGITPtr(DWORD dwCookie) throw();
CComGITPtr(CComGITPtr&& rv);
```

Parameters

p

[in] An interface pointer to be stored in the global interface table (GIT).

git

[in] A reference to an existing `CComGITPtr` object.

dwCookie

[in] A cookie used to identify the interface pointer.

rv

[in] The source `CComGITPtr` object to move data from.

Remarks

Creates a new `CComGITPtr` object, optionally using an existing `CComGITPtr` object.

The constructor utilizing *rv* is a move constructor. The data is moved from the source, *rv*, and then *rv* is cleared.

CComGITPtr::~~CComGITPtr

The destructor.

```
~CComGITPtr() throw();
```

Remarks

Removes the interface from the global interface table (GIT), using [CComGITPtr::Revoke](#).

CComGITPtr::CopyTo

Call this method to copy the interface from the global interface table (GIT) to the passed pointer.

```
HRESULT CopyTo(T** pp) const throw();
```

Parameters

pp

The pointer which is to receive the interface.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The interface from the GIT is copied to the passed pointer. The pointer must be released by the caller when it is no longer required.

CComGITPtr::Detach

Call this method to disassociate the interface from the `CComGITPtr` object.

```
DWORD Detach() throw();
```

Return Value

Returns the cookie from the `CComGITPtr` object.

Remarks

It is up to the caller to remove the interface from the GIT, using [CComGITPtr::Revoke](#).

CComGITPtr::GetCookie

Call this method to return the cookie from the `CComGITPtr` object.

```
DWORD GetCookie() const;
```

Return Value

Returns the cookie.

Remarks

The cookie is a variable used to identify an interface and its location.

CComGITPtr::m_dwCookie

The cookie.

```
DWORD m_dwCookie;
```

Remarks

The cookie is a member variable used to identify an interface and its location.

CComGITPtr::operator =

The assignment operator.

```
CComGITPtr& operator= (T* p);  
CComGITPtr& operator= (const CComGITPtr& git);  
CComGITPtr& operator= (DWORD dwCookie);  
CComGITPtr& operator= (CComGITPtr&& rv);
```

Parameters

p

[in] A pointer to an interface.

git

[in] A reference to a `CComGITPtr` object.

dwCookie

[in] A cookie used to identify the interface pointer.

rv

[in] The `CComGITPtr` to move data from.

Return Value

Returns the updated `CComGITPtr` object.

Remarks

Assigns a new value to a `CComGITPtr` object, either from an existing object or from a reference to a global interface table.

CComGITPtr::operator DWORD

Returns the cookie associated with the `CComGITPtr` object.

```
operator DWORD() const;
```

Remarks

The cookie is a variable used to identify an interface and its location.

CComGITPtr::Revoke

Call this method to remove the current interface from the global interface table (GIT).

```
HRESULT Revoke() throw();
```

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure.

Remarks

Removes the interface from the GIT.

See also

[Free Threaded Marshaler](#)

[Accessing Interfaces Across Apartments](#)

[When to Use the Global Interface Table](#)

[Class Overview](#)

CComHeap Class

4/1/2019 • 2 minutes to read • [Edit Online](#)

This class implements [IAtlMemMgr](#) using the COM memory allocation functions.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CComHeap : public IAtlMemMgr
```

Members

Public Methods

NAME	DESCRIPTION
CComHeap::Allocate	Call this method to allocate a block of memory.
CComHeap::Free	Call this method to free a block of memory allocated by this memory manager.
CComHeap::GetSize	Call this method to get the allocated size of a memory block allocated by this memory manager.
CComHeap::Reallocate	Call this method to reallocate memory allocated by this memory manager.

Remarks

[CComHeap](#) implements memory allocation functions using the COM allocation functions, including [CoTaskMemAlloc](#), [CoTaskMemFree](#), [IMalloc::GetSize](#), and [CoTaskMemRealloc](#). The maximum amount of memory that can be allocated is equal to `INT_MAX` (2147483647) bytes.

Example

See the example for [IAtlMemMgr](#).

Inheritance Hierarchy

[IAtlMemMgr](#)

[CComHeap](#)

Requirements

Header: ATLComMem.h

CComHeap::Allocate

Call this method to allocate a block of memory.

```
virtual __declspec(allocator) void* Allocate(size_t nBytes) throw();
```

Parameters

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call [CComHeap::Free](#) or [CComHeap::Reallocate](#) to free the memory allocated by this method.

Implemented using [CoTaskMemAlloc](#).

CComHeap::Free

Call this method to free a block of memory allocated by this memory manager.

```
virtual void Free(void* p) throw();
```

Parameters

p

Pointer to memory previously allocated by this memory manager. NULL is a valid value and does nothing.

Remarks

Implemented using [CoTaskMemFree](#).

CComHeap::GetSize

Call this method to get the allocated size of a memory block allocated by this memory manager.

```
virtual size_t GetSize(void* p) throw();
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

Return Value

Returns the size of the allocated memory block in bytes.

Remarks

Implemented using [IMalloc::GetSize](#).

CComHeap::Reallocate

Call this method to reallocate memory allocated by this memory manager.

```
virtual __declspec(allocator) void* Reallocate(void* p, size_t nBytes) throw();
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call [CComHeap::Free](#) to free the memory allocated by this method.

Implemented using [CoTaskMemRealloc](#).

See also

[DynamicConsumer Sample](#)

[Class Overview](#)

[CWin32Heap Class](#)

[CLocalHeap Class](#)

[CGlobalHeap Class](#)

[CCRTHeap Class](#)

[IAtlMemMgr Class](#)

CComHeapPtr Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

A smart pointer class for managing heap pointers.

Syntax

```
template<typename T>
class CComHeapPtr : public CHeapPtr<T, CComAllocator>
```

Parameters

T

The object type to be stored on the heap.

Members

Public Constructors

NAME	DESCRIPTION
CComHeapPtr::CComHeapPtr	The constructor.

Remarks

`CComHeapPtr` derives from `CHeapPtr`, but uses [CComAllocator](#) to allocate memory using COM routines. See [CHeapPtr](#) and [CHeapPtrBase](#) for the methods available.

Inheritance Hierarchy

[CHeapPtrBase](#)

[CHeapPtr](#)

`CComHeapPtr`

Requirements

Header: `atlbase.h`

CComHeapPtr::CComHeapPtr

The constructor.

```
CComHeapPtr() throw();
explicit CComHeapPtr(T* pData) throw();
```

Parameters

pData

An existing `CComHeapPtr` object.

Remarks

The heap pointer can optionally be created using an existing `CComHeapPtr` object. If so, the new `CComHeapPtr` object assumes responsibility for managing the new pointer and resources.

See also

[CHeapPtr Class](#)

[CHeapPtrBase Class](#)

[CComAllocator Class](#)

[Class Overview](#)

CComModule Class

3/4/2019 • 11 minutes to read • [Edit Online](#)

As of ATL 7.0, `CComModule` is deprecated: see [ATL Module Classes](#) for more details.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CComModule : public _ATL_MODULE
```

Members

Public Methods

NAME	DESCRIPTION
CComModule::GetClassObject	Creates an object of a specified CLSID. For DLLs only.
CComModule::GetModuleInstance	Returns <code>m_hInst</code> .
CComModule::GetResourceInstance	Returns <code>m_hInstResource</code> .
CComModule::GetTypeLibInstance	Returns <code>m_hInstTypeLib</code> .
CComModule::Init	Initializes data members.
CComModule::RegisterClassHelper	Enters an object's standard class registration in the system registry.
CComModule::RegisterClassObjects	Registers the class object. For EXEs only.
CComModule::RegisterServer	Updates the system registry for each object in the object map.
CComModule::RegisterTypeLib	Registers a type library.
CComModule::RevokeClassObjects	Revokes the class object. For EXEs only.
CComModule::Term	Releases data members.
CComModule::UnregisterClassHelper	Removes an object's standard class registration from the system registry.
CComModule::UnregisterServer	Unregisters each object in the object map.

NAME	DESCRIPTION
CComModule::UpdateRegistryClass	Registers or unregisters an object's standard class registration.
CComModule::UpdateRegistryFromResourceD	Runs the script contained in a specified resource to register or unregister an object.
CComModule::UpdateRegistryFromResourceS	Statically links to the ATL Registry Component. Runs the script contained in a specified resource to register or unregister an object.

Public Data Members

NAME	DESCRIPTION
CComModule::m_csObjMap	Ensures synchronized access to the object map information.
CComModule::m_csTypeInfoHolder	Ensures synchronized access to the type library information.
CComModule::m_csWindowCreate	Ensures synchronized access to window class information and static data used during window creation.
CComModule::m_hInst	Contains the handle to the module instance.
CComModule::m_hInstResource	By default, contains the handle to the module instance.
CComModule::m_hInstTypeLib	By default, contains the handle to the module instance.
CComModule::m_pObjMap	Points to the object map maintained by the module instance.

Remarks

NOTE

This class is deprecated, and the ATL code generation wizards now use the [CAtlAutoThreadModule](#) and [CAtlModule](#) derived classes. See [ATL Module Classes](#) for more information. The information that follows is for use with applications created with older releases of ATL. `CComModule` is still part of ATL for backwards capability.

`CComModule` implements a COM server module, allowing a client to access the module's components.

`CComModule` supports both DLL (in-process) and EXE (local) modules.

A `CComModule` instance uses an object map to maintain a set of class object definitions. This object map is implemented as an array of `_ATL_OBJMAP_ENTRY` structures, and contains information for:

- Entering and removing object descriptions in the system registry.
- Instantiating objects through a class factory.
- Establishing communication between a client and the root object in the component.
- Performing lifetime management of class objects.

When you run the ATL COM AppWizard, the wizard automatically generates `_Module`, a global instance of

`CComModule` or a class derived from it. For more information about the ATL Project Wizard, see the article [Creating an ATL Project](#).

In addition to `CComModule`, ATL provides [CComAutoThreadModule](#), which implements an apartment-model module for EXEs and Windows services. Derive your module from `CComAutoThreadModule` when you want to create objects in multiple apartments.

Inheritance Hierarchy

[_ATL_MODULE](#)

[CAtlModule](#)

[CAtlModuleT](#)

`CComModule`

Requirements

Header: `atlbase.h`

CComModule::GetClassObject

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
HRESULT GetClassObject(  
    REFCLSID rclsid,  
    REFIID riid,  
    LPVOID* ppv) throw();
```

Parameters

rclsid

[in] The CLSID of the object to be created.

riid

[in] The IID of the requested interface.

ppv

[out] A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppv* is set to NULL.

Return Value

A standard HRESULT value.

Remarks

Creates an object of the specified CLSID and retrieves an interface pointer to this object.

`GetClassObject` is only available to DLLs.

CComModule::GetModuleInstance

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
HINSTANCE GetModuleInstance() throw();
```

Return Value

The HINSTANCE identifying this module.

Remarks

Returns the [m_hInst](#) data member.

CComModule::GetResourceInstance

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
HINSTANCE GetResourceInstance() throw();
```

Return Value

An HINSTANCE.

Remarks

Returns the [m_hInstResource](#) data member.

CComModule::GetTypeLibInstance

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
HINSTANCE GetTypeLibInstance() const throw();
```

Return Value

An HINSTANCE.

Remarks

Returns the [m_hInstTypeLib](#) data member.

CComModule::Init

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
HRESULT Init(  
    _ATL_OBJMAP_ENTRY* p,  
    HINSTANCE h,  
    const GUID* plibid = NULL) throw();
```

Parameters

p

[in] A pointer to an array of object map entries.

h

[in] The HINSTANCE passed to `DLLMain` or `WinMain`.

plibid

[in] A pointer to the LIBID of the type library associated with the project.

Return Value

A standard HRESULT value.

Remarks

Initializes all data members.

CComModule::m_csObjMap

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
CRITICAL_SECTION m_csObjMap;
```

Remarks

Ensures synchronized access to the object map.

CComModule::m_csTypeInfoHolder

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
CRITICAL_SECTION m_csTypeInfoHolder;
```

Remarks

Ensures synchronized access to the type library.

CComModule::m_csWindowCreate

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
CRITICAL_SECTION m_csWindowCreate;
```

Remarks

Ensures synchronized access to window class information and to static data used during window creation.

CComModule::m_hInst

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
HINSTANCE m_hInst;
```

Remarks

Contains the handle to the module instance.

The [Init](#) method sets `m_hInst` to the handle passed to `DLLMain` or `WinMain`.

CComModule::m_hInstResource

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
HINSTANCE m_hInstResource;
```

Remarks

By default, contains the handle to the module instance.

The [Init](#) method sets `m_hInstResource` to the handle passed to `DLLMain` or `WinMain`. You can explicitly set `m_hInstResource` to the handle to a resource.

The [GetResourceInstance](#) method returns the handle stored in `m_hInstResource`.

CComModule::m_hInstTypeLib

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
HINSTANCE m_hInstTypeLib;
```

Remarks

By default, contains the handle to the module instance.

The [Init](#) method sets `m_hInstTypeLib` to the handle passed to `DLLMain` or `WinMain`. You can explicitly set `m_hInstTypeLib` to the handle to a type library.

The [GetTypeLibInstance](#) method returns the handle stored in `m_hInstTypeLib`.

CComModule::m_pObjMap

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
_ATL_OBJMAP_ENTRY* m_pObjMap;
```

Remarks

Points to the object map maintained by the module instance.

CComModule::RegisterClassHelper

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
ATL_DEPRECATED HRESULT RegisterClassHelper(  
    const CLSID& clsid,  
    LPCTSTR lpszProgID,  
    LPCTSTR lpszVerIndProgID,  
    UINT nDescID,  
    DWORD dwFlags);
```

Parameters

clsid

[in] The CLSID of the object to be registered.

lpszProgID

[in] The ProgID associated with the object.

lpszVerIndProgID

[in] The version-independent ProgID associated with the object.

nDescID

[in] The identifier of a string resource for the object's description.

dwFlags

[in] Specifies the threading model to enter in the registry. Possible values are `THREADFLAGS_APARTMENT`, `THREADFLAGS_BOTH`, or `AUTPRXFLAG`.

Return Value

A standard HRESULT value.

Remarks

Enters an object's standard class registration in the system registry.

The [UpdateRegistryClass](#) method calls `RegisterClassHelper`.

CComModule::RegisterClassObjects

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
HRESULT RegisterClassObjects(DWORD dwClsContext, DWORD dwFlags) throw();
```

Parameters

dwClsContext

[in] Specifies the context in which the class object is to be run. Possible values are CLSCTX_INPROC_SERVER, CLSCTX_INPROC_HANDLER, or CLSCTX_LOCAL_SERVER. For a description of these values, see [CLSCTX](#) in the Windows SDK.

dwFlags

[in] Determines the connection types to the class object. Possible values are REGCLS_SINGLEUSE, REGCLS_MULTIPLEUSE, or REGCLS_MULTI_SEPARATE. For a description of these values, see [REGCLS](#) in the Windows SDK.

Return Value

A standard HRESULT value.

Remarks

Registers an EXE class object with OLE so other applications can connect to it. This method is only available to EXEs.

CComModule::RegisterServer

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
HRESULT RegisterServer(  
    BOOL bRegTypeLib = FALSE,  
    const CLSID* pCLSID = NULL) throw();
```

Parameters

bRegTypeLib

[in] Indicates whether the type library will be registered. The default value is FALSE.

pCLSID

[in] Points to the CLSID of the object to be registered. If NULL (the default value), all objects in the object map will be registered.

Return Value

A standard HRESULT value.

Remarks

Depending on the *pCLSID* parameter, updates the system registry for a single class object or for all objects in the object map.

If *bRegTypeLib* is TRUE, the type library information will also be updated.

See [OBJECT_ENTRY_AUTO](#) for information on how to add an entry to the object map.

`RegisterServer` will be called automatically by `DLLRegisterServer` for a DLL or by `WinMain` for an EXE run with the `/RegServer` command line option.

CComModule::RegisterTypeLib

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
HRESULT RegisterTypeLib() throw();
HRESULT RegisterTypeLib(LPCTSTR lpszIndex) throw();
```

Parameters

lpszIndex

[in] String in the format `"\\N"`, where `N` is the integer index of the TYPELIB resource.

Return Value

A standard HRESULT value.

Remarks

Adds information about a type library to the system registry.

If the module instance contains multiple type libraries, use the second version of this method to specify which type library should be used.

CComModule::RevokeClassObjects

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
HRESULT RevokeClassObjects() throw();
```

Return Value

A standard HRESULT value.

Remarks

Removes the class object. This method is only available to EXEs.

CComModule::Term

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
void Term() throw();
```

Remarks

Releases all data members.

CComModule::UnregisterClassHelper

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
ATL_DEPRECATED HRESULT UnregisterClassHelper(  
    const CLSID& clsid,  
    LPCTSTR lpszProgID,  
    LPCTSTR lpszVerIndProgID);
```

Parameters

clsid

[in] The CLSID of the object to be unregistered.

lpszProgID

[in] The ProgID associated with the object.

lpszVerIndProgID

[in] The version-independent ProgID associated with the object.

Return Value

A standard HRESULT value.

Remarks

Removes an object's standard class registration from the system registry.

The [UpdateRegistryClass](#) method calls `UnregisterClassHelper`.

CComModule::UnregisterServer

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
HRESULT UnregisterServer(const CLSID* pCLSID = NULL) throw ();  
inline HRESULT UnregisterServer(BOOL bUnRegTypeLib, const CLSID* pCLSID = NULL) throw ();
```

Parameters

bUnRegTypeLib

If TRUE, the type library is also unregistered.

pCLSID

Points to the CLSID of the object to be unregistered. If NULL (the default value), all objects in the object map will be unregistered.

Return Value

A standard HRESULT value.

Remarks

Depending on the *pCLSID* parameter, unregisters either a single class object or all objects in the object map.

`UnregisterServer` will be called automatically by `DLLUnregisterServer` for a DLL or by `WinMain` for an EXE run with the `/UnregServer` command line option.

See [OBJECT_ENTRY_AUTO](#) for information on how to add an entry to the object map.

CComModule::UpdateRegistryClass

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```

ATL_DEPRECATED HRESULT UpdateRegistryClass(
    const CLSID& clsid,
    LPCTSTR lpszProgID,
    LPCTSTR lpszVerIndProgID,
    UINT nDescID,
    DWORD dwFlags,
    BOOL bRegister);

ATL_DEPRECATED HRESULT UpdateRegistryClass(
    const CLSID& clsid,
    LPCTSTR lpszProgID,
    LPCTSTR lpszVerIndProgID,
    LPCTSTR szDesc,
    DWORD dwFlags,
    BOOL bRegister);

```

Parameters

clsid

The CLSID of the object to be registered or unregistered.

lpszProgID

The ProgID associated with the object.

lpszVerIndProgID

The version-independent ProgID associated with the object.

nDescID

The identifier of the string resource for the object's description.

szDesc

A string containing the object's description.

dwFlags

Specifies the threading model to enter in the registry. Possible values are `THREADFLAGS_APARTMENT`, `THREADFLAGS_BOTH`, or `AUTPRXFLAG`.

bRegister

Indicates whether the object should be registered.

Return Value

A standard HRESULT value.

Remarks

If *bRegister* is TRUE, this method enters the object's standard class registration in the system registry.

If *bRegister* is FALSE, it removes the object's registration.

Depending on the value of *bRegister*, `UpdateRegistryClass` calls either [RegisterClassHelper](#) or [UnregisterClassHelper](#).

By specifying the `DECLARE_REGISTRY` macro, `UpdateRegistryClass` will be invoked automatically when your object map is processed.

CComModule::UpdateRegistryFromResourceD

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
virtual HRESULT UpdateRegistryFromResourceD(
    LPCTSTR lpszRes,
    BOOL bRegister,
    struct _ATL_REGMAP_ENTRY* pMapEntries = NULL) throw();

virtual HRESULT UpdateRegistryFromResourceD(
    UINT nResID,
    BOOL bRegister,
    struct _ATL_REGMAP_ENTRY* pMapEntries = NULL) throw ();
```

Parameters

lpszRes

[in] A resource name.

nResID

[in] A resource ID.

bRegister

[in] Indicates whether the object should be registered.

pMapEntries

[in] A pointer to the replacement map storing values associated with the script's replaceable parameters. ATL automatically uses `%MODULE%`. To use additional replaceable parameters, see the Remarks for details. Otherwise, use the NULL default value.

Return Value

A standard HRESULT value.

Remarks

Runs the script contained in the resource specified by *lpszRes* or *nResID*.

If *bRegister* is TRUE, this method registers the object in the system registry; otherwise, it unregisters the object.

By specifying the [DECLARE_REGISTRY_RESOURCE](#) or [DECLARE_REGISTRY_RESOURCEID](#) macro,

`UpdateRegistryFromResourceD` will be invoked automatically when your object map is processed.

NOTE

To substitute replacement values at run time, do not specify the `DECLARE_REGISTRY_RESOURCE` or `DECLARE_REGISTRY_RESOURCEID` macro. Instead, create an array of `_ATL_REGMAP_ENTRIES` structures, where each entry contains a variable placeholder paired with a value to replace the placeholder at run time. Then call `UpdateRegistryFromResourceD`, passing the array for the *pMapEntries* parameter. This adds all the replacement values in the `_ATL_REGMAP_ENTRIES` structures to the Registrar's replacement map.

NOTE

To statically link to the ATL Registry Component (Registrar), see [UpdateRegistryFromResourceS](#).

For more information about replaceable parameters and scripting, see the article [The ATL Registry Component \(Registrar\)](#).

CComModule::UpdateRegistryFromResourceS

As of ATL 7.0, `CComModule` is obsolete: see [ATL Module Classes](#) for more details.

```
virtual HRESULT UpdateRegistryFromResourceS(
    LPCTSTR lpszRes,
    BOOL bRegister,
    struct _ATL_REGMAP_ENTRY* pMapEntries = NULL) throw();

virtual HRESULT UpdateRegistryFromResourceS(
    UINT nResID,
    BOOL bRegister,
    struct _ATL_REGMAP_ENTRY* pMapEntries = NULL) throw();
```

Parameters

lpszRes

[in] A resource name.

nResID

[in] A resource ID.

bRegister

[in] Indicates whether the resource script should be registered.

pMapEntries

[in] A pointer to the replacement map storing values associated with the script's replaceable parameters. ATL automatically uses `%MODULE%`. To use additional replaceable parameters, see the Remarks for details. Otherwise, use the NULL default value.

Return Value

A standard HRESULT value.

Remarks

Similar to [UpdateRegistryFromResourceD](#) except `UpdateRegistryFromResourceS` creates a static link to the ATL Registry Component (Registrar).

`UpdateRegistryFromResourceS` will be invoked automatically when your object map is processed, provided you add `#define _ATL_STATIC_REGISTRY` to your `stdafx.h`.

NOTE

To substitute replacement values at run time, do not specify the [DECLARE_REGISTRY_RESOURCE](#) or [DECLARE_REGISTRY_RESOURCEID](#) macro. Instead, create an array of `_ATL_REGMAP_ENTRIES` structures, where each entry contains a variable placeholder paired with a value to replace the placeholder at run time. Then call `UpdateRegistryFromResourceS`, passing the array for the *pMapEntries* parameter. This adds all the replacement values in the `_ATL_REGMAP_ENTRIES` structures to the Registrar's replacement map.

For more information about replaceable parameters and scripting, see the article [The ATL Registry Component \(Registrar\)](#).

See also

[Class Overview](#)

CComMultiThreadModel Class

3/21/2019 • 3 minutes to read • [Edit Online](#)

`CComMultiThreadModel` provides thread-safe methods for incrementing and decrementing the value of a variable.

Syntax

```
class CComMultiThreadModel
```

Members

Public Typedefs

NAME	DESCRIPTION
<code>CComMultiThreadModel::AutoCriticalSection</code>	References class <code>CComAutoCriticalSection</code> .
<code>CComMultiThreadModel::CriticalSection</code>	References class <code>CComCriticalSection</code> .
<code>CComMultiThreadModel::ThreadModelNoCS</code>	References class <code>CComMultiThreadModelNoCS</code> .

Public Methods

NAME	DESCRIPTION
<code>CComMultiThreadModel::Decrement</code>	(Static) Decrements the value of the specified variable in a thread-safe manner.
<code>CComMultiThreadModel::Increment</code>	(Static) Increments the value of the specified variable in a thread-safe manner.

Remarks

Typically, you use `CComMultiThreadModel` through one of two **typedef** names, either `[CComObjectThreadModel]` (atl-typedefs.md#ccomobjectthreadmodel or `[CComGlobalsThreadModel]` (atl-typedefs.md#ccomglobalsthreadmodel). The class referenced by each **typedef** depends on the threading model used, as shown in the following table:

TYPEDF	SINGLE THREADING	APARTMENT THREADING	FREE THREADING
<code>CComObjectThreadModel</code>	S	S	M
<code>CComGlobalsThreadModel</code>	S	M	M

S= `CComSingleThreadModel` ; M= `CComMultiThreadModel`

`CComMultiThreadModel` itself defines three **typedef** names. `AutoCriticalSection` and `CriticalSection` reference classes that provide methods for obtaining and releasing ownership of a critical section. `ThreadModelNoCS` references class `[CComMultiThreadModelNoCS(ccommultithreadmodelnoc-class.md)]`.

Requirements

Header: atlbase.h

CComMultiThreadModel::AutoCriticalSection

When using `CComMultiThreadModel`, the **typedef** name `AutoCriticalSection` references class [CComAutoCriticalSection](#), which provides methods for obtaining and releasing ownership of a critical section object.

```
typedef CComAutoCriticalSection AutoCriticalSection;
```

Remarks

[CComSingleThreadModel](#) and [CComMultiThreadModelNoCS](#) also contain definitions for `AutoCriticalSection`.

The following table shows the relationship between the threading model class and the critical section class referenced by `AutoCriticalSection`:

CLASS DEFINED IN	CLASS REFERENCED
<code>CComMultiThreadModel</code>	<code>CComCriticalSection</code>
<code>CComSingleThreadModel</code>	<code>CComFakeCriticalSection</code>
<code>CComMultiThreadModelNoCS</code>	<code>CComFakeCriticalSection</code>

In addition to `AutoCriticalSection`, you can use the **typedef** name [CriticalSection](#). You should not specify `AutoCriticalSection` in global objects or static class members if you want to eliminate the CRT startup code.

Example

The following code is modeled after [CComObjectRootEx](#), and demonstrates `AutoCriticalSection` being used in a threading environment.

```
template<class ThreadModel>
class CMyAutoCritClass
{
public:
    typedef ThreadModel _ThreadModel;
    typedef typename _ThreadModel::AutoCriticalSection _CritSec;

    CMyAutoCritClass() : m_dwRef(0) {}

    ULONG InternalAddRef()
    {
        return _ThreadModel::Increment(&m_dwRef);
    }
    ULONG InternalRelease()
    {
        return _ThreadModel::Decrement(&m_dwRef);
    }
    void Lock() { m_critsec.Lock( ); }
    void Unlock() { m_critsec.Unlock(); }

private:
    _CritSec m_critsec;
    LONG m_dwRef;
```

The following tables show the results of the `InternalAddRef` and `Lock` methods, depending on the `ThreadModel` template parameter and the threading model used by the application:

ThreadModel = CComObjectThreadModel

METHOD	SINGLE OR APARTMENT THREADING	FREE THREADING
<code>InternalAddRef</code>	The increment is not thread-safe.	The increment is thread-safe.
<code>Lock</code>	Does nothing; there is no critical section to lock.	The critical section is locked.

ThreadModel = CComObjectThreadModel::ThreadModelNoCS

METHOD	SINGLE OR APARTMENT THREADING	FREE THREADING
<code>InternalAddRef</code>	The increment is not thread-safe.	The increment is thread-safe.
<code>Lock</code>	Does nothing; there is no critical section to lock.	Does nothing; there is no critical section to lock.

CComMultiThreadModel::CriticalSection

When using `CComMultiThreadModel`, the **typedef** name `CriticalSection` references class [CComCriticalSection](#), which provides methods for obtaining and releasing ownership of a critical section object.

```
typedef CComCriticalSection CriticalSection;
```

Remarks

[CComSingleThreadModel](#) and [CComMultiThreadModelNoCS](#) also contain definitions for `CriticalSection`. The following table shows the relationship between the threading model class and the critical section class referenced by `CriticalSection`:

CLASS DEFINED IN	CLASS REFERENCED
<code>CComMultiThreadModel</code>	<code>CComCriticalSection</code>
<code>CComSingleThreadModel</code>	<code>CComFakeCriticalSection</code>
<code>CComMultiThreadModelNoCS</code>	<code>CComFakeCriticalSection</code>

In addition to `CriticalSection`, you can use the **typedef** name [AutoCriticalSection](#). You should not specify `AutoCriticalSection` in global objects or static class members if you want to eliminate the CRT startup code.

Example

See [CComMultiThreadModel::AutoCriticalSection](#).

CComMultiThreadModel::Decrement

This static function calls the Win32 function [InterlockedDecrement](#), which decrements the value of the variable pointed to by *p*.

```
static ULONG WINAPI Decrement(LPLONG p) throw ();
```

Parameters

p

[in] Pointer to the variable to be decremented.

Return Value

If the result of the decrement is 0, then `Decrement` returns 0. If the result of the decrement is nonzero, the return value is also nonzero but may not equal the result of the decrement.

Remarks

`InterlockedDecrement` prevents more than one thread from simultaneously using this variable.

CComMultiThreadModel::Increment

This static function calls the Win32 function [InterlockedIncrement](#), which increments the value of the variable pointed to by *p*.

```
static ULONG WINAPI Increment(LPLONG p) throw ();
```

Parameters

p

[in] Pointer to the variable to be incremented.

Return Value

If the result of the increment is 0, then `Increment` returns 0. If the result of the increment is nonzero, the return value is also nonzero but may not equal the result of the increment.

Remarks

`InterlockedIncrement` prevents more than one thread from simultaneously using this variable.

CComMultiThreadModel::ThreadModelNoCS

When using `CComMultiThreadModel`, the **typedef** name `ThreadModelNoCS` references class [CComMultiThreadModelNoCS](#).

```
typedef CComMultiThreadModelNoCS ThreadModelNoCS;
```

Remarks

`CComMultiThreadModelNoCS` provides thread-safe methods for incrementing and decrementing a variable; however, it does not provide a critical section.

[CComSingleThreadModel](#) and `CComMultiThreadModelNoCS` also contain definitions for `ThreadModelNoCS`. The following table shows the relationship between the threading model class and the class referenced by

`ThreadModelNoCS`:

CLASS DEFINED IN	CLASS REFERENCED
<code>CComMultiThreadModel</code>	<code>CComMultiThreadModelNoCS</code>

CLASS DEFINED IN	CLASS REFERENCED
<code>CComSingleThreadModel</code>	<code>CComSingleThreadModel</code>
<code>CComMultiThreadModelNoCS</code>	<code>CComMultiThreadModelNoCS</code>

Example

See [CComMultiThreadModel::AutoCriticalSection](#).

See also

[CComSingleThreadModel Class](#)

[CComAutoCriticalSection Class](#)

[CComCriticalSection Class](#)

[Class Overview](#)

CComMultiThreadModelNoCS Class

3/21/2019 • 3 minutes to read • [Edit Online](#)

`CComMultiThreadModelNoCS` provides thread-safe methods for incrementing and decrementing the value of a variable, without critical section locking or unlocking functionality.

Syntax

```
class CComMultiThreadModelNoCS
```

Members

Public Typedefs

NAME	DESCRIPTION
CComMultiThreadModelNoCS::AutoCriticalSection	References class CComFakeCriticalSection .
CComMultiThreadModelNoCS::CriticalSection	References class <code>CComFakeCriticalSection</code> .
CComMultiThreadModelNoCS::ThreadModelNoCS	References class <code>CComMultiThreadModelNoCS</code> .

Public Methods

NAME	DESCRIPTION
CComMultiThreadModelNoCS::Decrement	(Static) Decrements the value of the specified variable in a thread-safe manner.
CComMultiThreadModelNoCS::Increment	(Static) Increments the value of the specified variable in a thread-safe manner.

Remarks

`CComMultiThreadModelNoCS` is similar to [CComMultiThreadModel](#) in that it provides thread-safe methods for incrementing and decrementing a variable. However, when you reference a critical section class through `CComMultiThreadModelNoCS`, methods such as `Lock` and `Unlock` will do nothing.

Typically, you use `CComMultiThreadModelNoCS` through the `ThreadModelNoCS` **typedef** name. This **typedef** is defined in `CComMultiThreadModelNoCS`, `CComMultiThreadModel`, and [CComSingleThreadModel](#).

NOTE

The global **typedef** names [CComObjectThreadModel](#) and [CComGlobalsThreadModel](#) do not reference `CComMultiThreadModelNoCS`.

In addition to `ThreadModelNoCS`, `CComMultiThreadModelNoCS` defines `AutoCriticalSection` and `CriticalSection`. These latter two **typedef** names reference [CComFakeCriticalSection](#), which provides empty methods associated

with obtaining and releasing a critical section.

Requirements

Header: atlbase.h

CComMultiThreadModelNoCS::AutoCriticalSection

When using `CComMultiThreadModelNoCS`, the **typedef** name `AutoCriticalSection` references class [CComFakeCriticalSection](#).

```
typedef CComFakeCriticalSection AutoCriticalSection;
```

Remarks

Because `CComFakeCriticalSection` does not provide a critical section, its methods do nothing.

[CComMultiThreadModel](#) and [CComSingleThreadModel](#) also contain definitions for `AutoCriticalSection`. The following table shows the relationship between the threading model class and the critical section class referenced by `AutoCriticalSection`:

CLASS DEFINED IN	CLASS REFERENCED
<code>CComMultiThreadModelNoCS</code>	<code>CComFakeCriticalSection</code>
<code>CComMultiThreadModel</code>	<code>CComAutoCriticalSection</code>
<code>CComSingleThreadModel</code>	<code>CComFakeCriticalSection</code>

In addition to `AutoCriticalSection`, you can use the **typedef** name [CriticalSection](#). You should not specify `AutoCriticalSection` in global objects or static class members if you want to eliminate the CRT startup code.

Example

See [CComMultiThreadModel::AutoCriticalSection](#).

CComMultiThreadModelNoCS::CriticalSection

When using `CComMultiThreadModelNoCS`, the **typedef** name `CriticalSection` references class [CComFakeCriticalSection](#).

```
typedef CComFakeCriticalSection CriticalSection;
```

Remarks

Because `CComFakeCriticalSection` does not provide a critical section, its methods do nothing.

[CComMultiThreadModel](#) and [CComSingleThreadModel](#) also contain definitions for `CriticalSection`. The following table shows the relationship between the threading model class and the critical section class referenced by `CriticalSection`:

CLASS DEFINED IN	CLASS REFERENCED
<code>CComMultiThreadModelNoCS</code>	<code>CComFakeCriticalSection</code>

CLASS DEFINED IN	CLASS REFERENCED
<code>CComMultiThreadModel</code>	<code>CComCriticalSection</code>
<code>CComSingleThreadModel</code>	<code>CComFakeCriticalSection</code>

In addition to `CriticalSection`, you can use the **typedef** name `AutoCriticalSection`. You should not specify `AutoCriticalSection` in global objects or static class members if you want to eliminate the CRT startup code.

Example

See [CComMultiThreadModel::AutoCriticalSection](#).

CComMultiThreadModelNoCS::Decrement

This static function calls the Win32 function [InterlockedDecrement](#), which decrements the value of the variable pointed to by *p*.

```
static ULONG WINAPI Decrement(LPULONG p) throw();
```

Parameters

p

[in] Pointer to the variable to be decremented.

Return Value

If the result of the decrement is 0, then `Decrement` returns 0. If the result of the decrement is nonzero, the return value is also nonzero but may not equal the result of the decrement.

Remarks

InterlockedDecrement prevents more than one thread from simultaneously using this variable.

CComMultiThreadModelNoCS::Increment

This static function calls the Win32 function [InterlockedIncrement](#), which increments the value of the variable pointed to by *p*.

```
static ULONG WINAPI Increment(LPULONG p) throw();
```

Parameters

p

[in] Pointer to the variable to be incremented.

Return Value

If the result of the increment is 0, then **Increment** returns 0. If the result of the increment is nonzero, the return value is also nonzero but may not equal the result of the increment.

Remarks

InterlockedIncrement prevents more than one thread from simultaneously using this variable.

CComMultiThreadModelNoCS::ThreadModelNoCS

When using `CComMultiThreadModelNoCS`, the **typedef** name `ThreadModelNoCS` simply references `CComMultiThreadModelNoCS`.


```
typedef CComMultiThreadModelNoCS ThreadModelNoCS;
```

Remarks

[CComMultiThreadModel](#) and [CComSingleThreadModel](#) also contain definitions for `ThreadModelNoCS`. The following table shows the relationship between the threading model class and the class referenced by `ThreadModelNoCS`:

CLASS DEFINED IN	CLASS REFERENCED
<code>CComMultiThreadModelNoCS</code>	<code>CComMultiThreadModelNoCS</code>
<code>CComMultiThreadModel</code>	<code>CComMultiThreadModelNoCS</code>
<code>CComSingleThreadModel</code>	<code>CComSingleThreadModel</code>

Note that the definition of `ThreadModelNoCS` in `CComMultiThreadModelNoCS` provides symmetry with `CComMultiThreadModel` and `CComSingleThreadModel`. For example, suppose the sample code in `CComMultiThreadModel::AutoCriticalSection` declared the following **typedef**:

```
typedef typename ThreadModel::ThreadModelNoCS _ThreadModel;
```

Regardless of the class specified for `ThreadModel` (such as `CComMultiThreadModelNoCS`), `_ThreadModel` resolves accordingly.

Example

See [CComMultiThreadModel::AutoCriticalSection](#).

See also

[Class Overview](#)

CComObject Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements `IUnknown` for a nonaggregated object.

Syntax

```
template<class Base>
class CComObject : public Base
```

Parameters

Base

Your class, derived from [CComObjectRoot](#) or [CComObjectRootEx](#), as well as from any other interfaces you want to support on the object.

Members

Public Constructors

NAME	DESCRIPTION
CComObject::CComObject	The constructor.
CComObject::~~CComObject	The destructor.

Public Methods

NAME	DESCRIPTION
CComObject::AddRef	Increments the reference count on the object.
CComObject::CreateInstance	(Static) Creates a new <code>CComObject</code> object.
CComObject::QueryInterface	Retrieves a pointer to the requested interface.
CComObject::Release	Decrements the reference count on the object.

Remarks

`CComObject` implements [IUnknown](#) for a nonaggregated object. However, calls to `QueryInterface`, `AddRef`, and `Release` are delegated to `CComObjectRootEx`.

For more information about using `CComObject`, see the article [Fundamentals of ATL COM Objects](#).

Inheritance Hierarchy

`Base`

`CComObject`

Requirements

Header: atlcom.h

CComObject::AddRef

Increments the reference count on the object.

```
STDMETHOD_(ULONG, AddRef)();
```

Return Value

This function returns the new incremented reference count on the object. This value may be useful for diagnostics or testing.

CComObject::CComObject

The constructor increments the module lock count.

```
CComObject(void* = NULL);
```

Parameters

*void**

[in] This unnamed parameter is not used. It exists for symmetry with other `CComXXXObjectXXX` constructors.

Remarks

The destructor decrements it.

If a `CComObject`-derived object is successfully constructed using the **new** operator, the initial reference count is 0. To set the reference count to the proper value (1), make a call to the [AddRef](#) function.

CComObject::~~CComObject

The destructor.

```
CComObject();
```

Remarks

Frees all allocated resources, calls [FinalRelease](#), and decrements the module lock count.

CComObject::CreateInstance

This static function allows you to create a new **CComObject**<`Base`> object, without the overhead of [CoCreateInstance](#).

```
static HRESULT WINAPI CreateInstance(CComObject<Base>** pp);
```

Parameters

pp

[out] A pointer to a **CComObject**<`Base`> pointer. If `CreateInstance` is unsuccessful, *pp* is set to NULL.

Return Value

A standard HRESULT value.

Remarks

The object returned has a reference count of zero, so call `AddRef` immediately, then use `Release` to free the reference on the object pointer when you're done.

If you do not need direct access to the object, but still want to create a new object without the overhead of `CoCreateInstance`, use [CComCoClass::CreateInstance](#) instead.

Example

```
class ATL_NO_VTABLE CMyCircle :
{
public:
    CMyCircle()
    {
    }

    DECLARE_REGISTRY_RESOURCEID(IDR_MYCIRCLE)

    DECLARE_NOT_AGGREGATABLE(CMyCircle)

    BEGIN_COM_MAP(CMyCircle)
        COM_INTERFACE_ENTRY(IMyCircle)
        COM_INTERFACE_ENTRY(IDispatch)
    END_COM_MAP()

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    HRESULT FinalConstruct()
    {
        return S_OK;
    }

    void FinalRelease()
    {
    }

public:
public:
    STDMETHOD(get_XCenter)(double* pVal);
};
```

```
// Create a local instance of COM object CMyCircle.
double x;
CComObject<CMyCircle>* pCircle;
HRESULT hRes = CComObject<CMyCircle>::CreateInstance(&pCircle);
ATLASSERT(SUCCEEDED(hRes));

// Increment reference count immediately
pCircle->AddRef();

// Access method of COM object
hRes = pCircle->get_XCenter(&x);

// Decrement reference count when done
pCircle->Release();
pCircle = NULL;
```

CComObject::QueryInterface

Retrieves a pointer to the requested interface.

```
STDMETHOD(QueryInterface)(REFIID iid, void** ppvObject);
template <class Q>
HRESULT STDMETHODCALLTYPE QueryInterface(Q** pp);
```

Parameters

iid

[in] The identifier of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by *iid*. If the object does not support this interface, *ppvObject* is set to NULL.

pp

[out] A pointer to the interface pointer identified by type `Q`. If the object does not support this interface, *pp* is set to NULL.

Return Value

A standard HRESULT value.

CComObject::Release

Decrements the reference count on the object.

```
STDMETHOD_(ULONG, Release)();
```

Return Value

This function returns the new decremented reference count on the object. In debug builds, the return value may be useful for diagnostics or testing. In non-debug builds, `Release` always returns 0.

See also

[CComAggObject Class](#)

[CComPolyObject Class](#)

[DECLARE_AGGREGATABLE](#)

[DECLARE_NOT_AGGREGATABLE](#)

CComObjectGlobal Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class manages a reference count on the module containing your `Base` object.

Syntax

```
template<class Base>
class CComObjectGlobal : public Base
```

Parameters

Base

Your class, derived from [CComObjectRoot](#) or [CComObjectRootEx](#), as well as from any other interface you want to support on the object.

Members

Public Constructors

NAME	DESCRIPTION
CComObjectGlobal::CComObjectGlobal	The constructor.
CComObjectGlobal::~~CComObjectGlobal	The destructor.

Public Methods

NAME	DESCRIPTION
CComObjectGlobal::AddRef	Implements a global <code>AddRef</code> .
CComObjectGlobal::QueryInterface	Implements a global <code>QueryInterface</code> .
CComObjectGlobal::Release	Implements a global <code>Release</code> .

Public Data Members

NAME	DESCRIPTION
CComObjectGlobal::m_hResFinalConstruct	Contains the HRESULT returned during construction of the <code>CComObjectGlobal</code> object.

Remarks

`CComObjectGlobal` manages a reference count on the module containing your `Base` object. `CComObjectGlobal` ensures your object will not be deleted as long as the module is not released. Your object will only be removed when the reference count on the entire module goes to zero.

For example, using `CComObjectGlobal`, a class factory can hold a common global object that is shared by all its

clients.

Inheritance Hierarchy

Base

CComObjectGlobal

Requirements

Header: atlcom.h

CComObjectGlobal::AddRef

Increments the reference count of the object by 1.

```
STDMETHOD_(ULONG, AddRef)();
```

Return Value

A value that may be useful for diagnostics and testing.

Remarks

By default, `AddRef` calls `_Module::Lock`, where `_Module` is the global instance of [CComModule](#) or a class derived from it.

CComObjectGlobal::CComObjectGlobal

The constructor. Calls `FinalConstruct` and then sets `m_hResFinalConstruct` to the `HRESULT` returned by `FinalConstruct`.

```
CComObjectGlobal(void* = NULL);
```

Remarks

If you have not derived your base class from [CComObjectRoot](#), you must supply your own `FinalConstruct` method. The destructor calls `FinalRelease`.

CComObjectGlobal::~~CComObjectGlobal

The destructor.

```
CComObjectGlobal();
```

Remarks

Frees all allocated resources and calls [FinalRelease](#).

CComObjectGlobal::m_hResFinalConstruct

Contains the `HRESULT` from calling `FinalConstruct` during construction of the `CComObjectGlobal` object.

```
HRESULT m_hResFinalConstruct;
```


CComObjectGlobal::QueryInterface

Retrieves a pointer to the requested interface pointer.

```
STDMETHOD(QueryInterface)(REFIID iid, void** ppvObject);
```

Parameters

iid

[in] The GUID of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by iid, or NULL if the interface is not found.

Return Value

A standard HRESULT value.

Remarks

`QueryInterface` only handles interfaces in the COM map table.

CComObjectGlobal::Release

Decrements the reference count of the object by 1.

```
STDMETHOD_(ULONG, Release)();
```

Return Value

In debug builds, `Release` returns a value that may be useful for diagnostics and testing. In non-debug builds, `Release` always returns 0.

Remarks

By default, `Release` calls `_Module::Unlock`, where `_Module` is the global instance of [CComModule](#) or a class derived from it.

See also

[CComObjectStack Class](#)

[CComAggObject Class](#)

[CComObject Class](#)

[Class Overview](#)

CComObjectNoLock Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements `IUnknown` for a nonaggregated object, but does not increment the module lock count in the constructor.

Syntax

```
template<class Base>
class CComObjectNoLock : public Base
```

Parameters

Base

Your class, derived from [CComObjectRoot](#) or [CComObjectRootEx](#), as well as from any other interface you want to support on the object.

Members

Public Constructors

NAME	DESCRIPTION
CComObjectNoLock::CComObjectNoLock	Constructor.
CComObjectNoLock::~~CComObjectNoLock	The destructor.

Public Methods

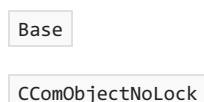
NAME	DESCRIPTION
CComObjectNoLock::AddRef	Increments the reference count on the object.
CComObjectNoLock::QueryInterface	Returns a pointer to the requested interface.
CComObjectNoLock::Release	Decrements the reference count on the object.

Remarks

`CComObjectNoLock` is similar to [CComObject](#) in that it implements [IUnknown](#) for a nonaggregated object; however, `CComObjectNoLock` does not increment the module lock count in the constructor.

ATL uses `CComObjectNoLock` internally for class factories. In general, you will not use this class directly.

Inheritance Hierarchy



Requirements

Header: atlcom.h

CComObjectNoLock::AddRef

Increments the reference count on the object.

```
STDMETHOD_(ULONG, AddRef)();
```

Return Value

A value that may be useful for diagnostics or testing.

CComObjectNoLock::CComObjectNoLock

The constructor. Unlike [CComObject](#), does not increment the module lock count.

```
CComObjectNoLock(void* = NULL);
```

Parameters

*void**

[in] This unnamed parameter is not used. It exists for symmetry with other `CComXXXObjectXXX` constructors.

CComObjectNoLock::~~CComObjectNoLock

The destructor.

```
~CComObjectNoLock();
```

Remarks

Frees all allocated resources and calls [FinalRelease](#).

CComObjectNoLock::QueryInterface

Retrieves a pointer to the requested interface.

```
STDMETHOD(QueryInterface)(REFIID iid, void** ppvObject);
```

Parameters

iid

[in] The identifier of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by *iid*. If the object does not support this interface, *ppvObject* is set to NULL.

Return Value

A standard HRESULT value.

CComObjectNoLock::Release

Decrements the reference count on the object.

```
STDMETHOD_(ULONG, Release)();
```

Return Value

In debug builds, `Release` returns a value that may be useful for diagnostics or testing. In non-debug builds, `Release` always returns 0.

See also

[Class Overview](#)

CComObjectRoot Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This typedef of [CComObjectRootEx](#) is templated on the default threading model of the server.

Syntax

```
typedef CComObjectRootEx<CComObjectThreadModel> CComObjectRoot;
```

Remarks

`CComObjectRoot` is a `typedef` of [CComObjectRootEx](#) templated on the default threading model of the server. Thus [CComObjectThreadModel](#) will reference either [CComSingleThreadModel](#) or [CComMultiThreadModel](#).

`CComObjectRootEx` handles object reference count management for both nonaggregated and aggregated objects. It holds the object reference count if your object is not being aggregated, and holds the pointer to the outer unknown if your object is being aggregated. For aggregated objects, `CComObjectRootEx` methods can be used to handle the failure of the inner object to construct, and to protect the outer object from deletion when inner interfaces are released or the inner object is deleted.

Requirements

Header: atlcom.h

See also

[CComObjectRootEx Class](#)

[CComAggObject Class](#)

[CComObject Class](#)

[CComPolyObject Class](#)

[Class Overview](#)

CComObjectRootEx Class

3/4/2019 • 9 minutes to read • [Edit Online](#)

This class provides methods to handle object reference count management for both nonaggregated and aggregated objects.

Syntax

```
template<class ThreadModel>
class CComObjectRootEx : public CComObjectRootBase
```

Parameters

ThreadModel

The class whose methods implement the desired threading model. You can explicitly choose the threading model by setting *ThreadModel* to [CComSingleThreadModel](#), [CComMultiThreadModel](#), or [CComMultiThreadModelNoCS](#). You can accept the server's default thread model by setting *ThreadModel* to [CComObjectThreadModel](#) or [CComGlobalsThreadModel](#).

Members

Methods

CComObjectRootEx	Constructor.
InternalAddRef	Increments the reference count for a nonaggregated object.
InternalRelease	Decrements the reference count for a nonaggregated object.
Lock	If the thread model is multithreaded, obtains ownership of a critical section object.
Unlock	If the thread model is multithreaded, releases ownership of a critical section object.

CComObjectRootBase Methods

FinalConstruct	Override in your class to perform any initialization required by your object.
FinalRelease	Override in your class to perform any cleanup required by your object.
OuterAddRef	Increments the reference count for an aggregated object.

OuterQueryInterface	Delegates to the outer <code>IUnknown</code> of an aggregated object.
OuterRelease	Decrements the reference count for an aggregated object.

Static Functions

InternalQueryInterface	Delegates to the <code>IUnknown</code> of a nonaggregated object.
ObjectMain	Called during module initialization and termination for derived classes listed in the object map.

Data Members

m_dwRef	With <code>m_pOuterUnknown</code> , part of a union. Used when the object is not aggregated to hold the reference count of <code>AddRef</code> and <code>Release</code> .
m_pOuterUnknown	With <code>m_dwRef</code> , part of a union. Used when the object is aggregated to hold a pointer to the outer unknown.

Remarks

`CComObjectRootEx` handles object reference count management for both nonaggregated and aggregated objects. It holds the object reference count if your object is not being aggregated, and holds the pointer to the outer unknown if your object is being aggregated. For aggregated objects, `CComObjectRootEx` methods can be used to handle the failure of the inner object to construct, and to protect the outer object from deletion when inner interfaces are released or the inner object is deleted.

A class that implements a COM server must inherit from `CComObjectRootEx` or [CComObjectRoot](#).

If your class definition specifies the [DECLARE_Poly_Aggregatable](#) macro, ATL creates an instance of `CComPolyObject<CYourClass>` when `IClassFactory::CreateInstance` is called. During creation, the value of the outer unknown is checked. If it is NULL, `IUnknown` is implemented for a nonaggregated object. If the outer unknown is not NULL, `IUnknown` is implemented for an aggregated object.

If your class does not specify the [DECLARE_Poly_Aggregatable](#) macro, ATL creates an instance of `CaggComObject<CYourClass>` for aggregated objects or an instance of `CComObject<CYourClass>` for nonaggregated objects.

The advantage of using `CComPolyObject` is that you avoid having both `CComAggObject` and `CComObject` in your module to handle the aggregated and nonaggregated cases. A single `CComPolyObject` object handles both cases. Therefore, only one copy of the vtable and one copy of the functions exist in your module. If your vtable is large, this can substantially decrease your module size. However, if your vtable is small, using `CComPolyObject` can result in a slightly larger module size because it is not optimized for an aggregated or nonaggregated object, as are `CComAggObject` and `CComObject`.

If your object is aggregated, [IUnknown](#) is implemented by `CComAggObject` or `CComPolyObject`. These classes delegate `QueryInterface`, `AddRef`, and `Release` calls to `CComObjectRootEx`'s `OuterQueryInterface`, `OuterAddRef`, and `OuterRelease` to forward to the outer unknown. Typically, you override

`CComObjectRootEx::FinalConstruct` in your class to create any aggregated objects, and override `CComObjectRootEx::FinalRelease` to free any aggregated objects.

If your object is not aggregated, `IUnknown` is implemented by `CComObject` or `CComPolyObject`. In this case, calls to `QueryInterface`, `AddRef`, and `Release` are delegated to `CComObjectRootEx`'s `InternalQueryInterface`, `InternalAddRef`, and `InternalRelease` to perform the actual operations.

Requirements

Header: `atlcom.h`

`CComObjectRootEx::CComObjectRootEx`

The constructor initializes the reference count to 0.

```
CComObjectRootEx();
```

`CComObjectRootEx::FinalConstruct`

You can override this method in your derived class to perform any initialization required for your object.

```
HRESULT FinalConstruct();
```

Return Value

Return `S_OK` on success or one of the standard error `HRESULT` values.

Remarks

By default, `CComObjectRootEx::FinalConstruct` simply returns `S_OK`.

There are advantages to performing initialization in `FinalConstruct` rather than the constructor of your class:

- You cannot return a status code from a constructor, but you can return an `HRESULT` by means of `FinalConstruct`'s return value. When objects of your class are being created using the standard class factory provided by ATL, this return value is propagated back to the COM client allowing you to provide them with detailed error information.
- You cannot call virtual functions through the virtual function mechanism from the constructor of a class. Calling a virtual function from the constructor of a class results in a statically resolved call to the function as it is defined at that point in the inheritance hierarchy. Calls to pure virtual functions result in linker errors.

Your class is not the most derived class in the inheritance hierarchy — it relies on a derived class supplied by ATL to provide some of its functionality. There is a good chance that your initialization will need to use the features provided by that class (this is certainly true when objects of your class need to aggregate other objects), but the constructor in your class has no way to access those features. The construction code for your class is executed before the most derived class is fully constructed.

However, `FinalConstruct` is called immediately after the most derived class is fully constructed allowing you to call virtual functions and use the reference-counting implementation provided by ATL.

Example

Typically, override this method in the class derived from `CComObjectRootEx` to create any aggregated objects. For example:

```
class ATL_NO_VTABLE CMyAggObject :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CMyAggObject, &CLSID_MyAggObject>,
    public IDispatchImpl<IMyAggObject, &IID_IMyAggObject, &LIBID_NVC_ATL_COMLib, /*wMajor =*/ 1,
    /*wMinor =*/ 0>
{
public:
    DECLARE_GET_CONTROLLING_UNKNOWN()
    HRESULT FinalConstruct()
    {
        return CoCreateInstance(CLSID_MyCustomClass, GetControllingUnknown(),
            CLSCTX_ALL, IID_IUnknown, (void*)&m_pMyCustomClass);
    }

    IMyCustomClass* m_pMyCustomClass;

    // Remainder of class declaration omitted.
```

If the construction fails, you can return an error. You can also use the macro [DECLARE_PROTECT_FINAL_CONSTRUCT](#) to protect your outer object from being deleted if, during creation, the internal aggregated object increments the reference count then decrements the count to 0.

Here is a typical way to create an aggregate:

- Add an `IUnknown` pointer to your class object and initialize it to NULL in the constructor.
- Override `FinalConstruct` to create the aggregate.
- Use the `IUnknown` pointer you defined as the parameter to the [COM_INTERFACE_ENTRY_AGGREGATE](#) macro.
- Override `FinalRelease` to release the `IUnknown` pointer.

CComObjectRootEx::FinalRelease

You can override this method in your derived class to perform any cleanup required for your object.

```
void FinalRelease();
```

Remarks

By default, `CComObjectRootEx::FinalRelease` does nothing.

Performing cleanup in `FinalRelease` is preferable to adding code to the destructor of your class since the object is still fully constructed at the point at which `FinalRelease` is called. This enables you to safely access the methods provided by the most derived class. This is particularly important for freeing any aggregated objects before deletion.

CComObjectRootEx::InternalAddRef

Increments the reference count of a nonaggregated object by 1.

```
ULONG InternalAddRef();
```

Return Value

A value that may be useful for diagnostics and testing.

Remarks

If the thread model is multithreaded, `InterlockedIncrement` is used to prevent more than one thread from changing the reference count at the same time.

CComObjectRootEx::InternalQueryInterface

Retrieves a pointer to the requested interface.

```
static HRESULT InternalQueryInterface(  
    void* pThis,  
    const _ATL_INTMAP_ENTRY* pEntries,  
    REFIID iid,  
    void** ppvObject);
```

Parameters

pThis

[in] A pointer to the object that contains the COM map of interfaces exposed to `QueryInterface`.

pEntries

[in] A pointer to the `_ATL_INTMAP_ENTRY` structure that accesses a map of available interfaces.

iid

[in] The GUID of the interface being requested.

ppvObject

[out] A pointer to the interface pointer specified in *iid*, or NULL if the interface is not found.

Return Value

One of the standard HRESULT values.

Remarks

`InternalQueryInterface` only handles interfaces in the COM map table. If your object is aggregated, `InternalQueryInterface` does not delegate to the outer unknown. You can enter interfaces into the COM map table with the macro `COM_INTERFACE_ENTRY` or one of its variants.

CComObjectRootEx::InternalRelease

Decrements the reference count of a nonaggregated object by 1.

```
ULONG InternalRelease();
```

Return Value

In both non-debug and debug builds, this function returns a value which may be useful for diagnostics or testing. The exact value returned depends on many factors such as the operating system used, and may, or may not, be the reference count.

Remarks

If the thread model is multithreaded, `InterlockedDecrement` is used to prevent more than one thread from changing the reference count at the same time.

CComObjectRootEx::Lock

If the thread model is multithreaded, this method calls the Win32 API function [EnterCriticalSection](#), which waits until the thread can take ownership of the critical section object obtained through a private data member.

```
void Lock();
```

Remarks

When the protected code finishes executing, the thread must call `Unlock` to release ownership of the critical section.

If the thread model is single-threaded, this method does nothing.

CComObjectRootEx::m_dwRef

Part of a union that accesses four bytes of memory.

```
long m_dwRef;
```

Remarks

With `m_pOuterUnknown`, part of a union:

```
union {
    long m_dwRef;
    IUnknown* m_pOuterUnknown;
};
```

If the object is not aggregated, the reference count accessed by `AddRef` and `Release` is stored in `m_dwRef`.

If the object is aggregated, the pointer to the outer unknown is stored in [m_pOuterUnknown](#).

CComObjectRootEx::m_pOuterUnknown

Part of a union that accesses four bytes of memory.

```
IUnknown*
    m_pOuterUnknown;
```

Remarks

With `m_dwRef`, part of a union:

```
union {
    long m_dwRef;
    IUnknown* m_pOuterUnknown;
};
```

If the object is aggregated, the pointer to the outer unknown is stored in `m_pOuterUnknown`. If the object is not aggregated, the reference count accessed by `AddRef` and `Release` is stored in [m_dwRef](#).

CComObjectRootEx::ObjectMain

For each class listed in the object map, this function is called once when the module is initialized, and again when it is terminated.

```
static void WINAPI ObjectMain(bool bStarting);
```

Parameters

bStarting

[out] The value is TRUE if the class is being initialized; otherwise FALSE.

Remarks

The value of the *bStarting* parameter indicates whether the module is being initialized or terminated. The default implementation of `ObjectMain` does nothing, but you can override this function in your class to initialize or clean up resources that you want to allocate for the class. Note that `ObjectMain` is called before any instances of the class are requested.

`ObjectMain` is called from the entry point of the DLL, so the type of operation that the entry-point function can perform is restricted. For more information on these restrictions, see [DLLs and Visual C++ run-time library behavior](#) and [DllMain](#).

Example

```
class ATL_NO_VTABLE CMyApp :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CMyApp, &CLSID_MyApp>,
    public IMyApp
{
public:
    CMyApp()
    {
    }

    static void WINAPI ObjectMain(bool bStarting)
    {
        if (bStarting)
            ;// Perform custom initialization routines
        else
            ;// Perform custom termination routines
    }

    // Remainder of class declaration omitted.
```

CComObjectRootEx::OuterAddRef

Increments the reference count of the outer unknown of an aggregation.

```
ULONG OuterAddRef();
```

Return Value

A value that may be useful for diagnostics and testing.

CComObjectRootEx::OuterQueryInterface

Retrieves an indirect pointer to the requested interface.

```
HRESULT OuterQueryInterface(REFIID iid, void** ppvObject);
```

Parameters

iid

[in] The GUID of the interface being requested.

ppvObject

[out] A pointer to the interface pointer specified in *iid*, or NULL if the aggregation does not support the interface.

Return Value

One of the standard HRESULT values.

CComObjectRootEx::OuterRelease

Decrements the reference count of the outer unknown of an aggregation.

```
ULONG OuterRelease();
```

Return Value

In non-debug builds, always returns 0. In debug builds, returns a value that may be useful for diagnostics or testing.

CComObjectRootEx::Unlock

If the thread model is multithreaded, this method calls the Win32 API function [LeaveCriticalSection](#), which releases ownership of the critical section object obtained through a private data member.

```
void Unlock();
```

Remarks

To obtain ownership, the thread must call `Lock`. Each call to `Lock` requires a corresponding call to `Unlock` to release ownership of the critical section.

If the thread model is single-threaded, this method does nothing.

See also

[CComAggObject Class](#)

[CComObject Class](#)

[CComPolyObject Class](#)

[Class Overview](#)

CComObjectStack Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class creates a temporary COM object and provides it with a skeletal implementation of `IUnknown`.

Syntax

```
template <class Base>
class CComObjectStack : public Base
```

Parameters

Base

Your class, derived from `CComObjectRoot` or `CComObjectRootEx`, as well as from any other interface you want to support on the object.

Members

Public Constructors

NAME	DESCRIPTION
<code>CComObjectStack::CComObjectStack</code>	The constructor.
<code>CComObjectStack::~~CComObjectStack</code>	The destructor.

Public Methods

NAME	DESCRIPTION
<code>CComObjectStack::AddRef</code>	Returns zero. In debug mode, calls <code>_ASSERTE</code> .
<code>CComObjectStack::QueryInterface</code>	Returns <code>E_NOINTERFACE</code> . In debug mode, calls <code>_ASSERTE</code> .
<code>CComObjectStack::Release</code>	Returns zero. In debug mode, calls <code>_ASSERTE</code> . ~

Public Data Members

NAME	DESCRIPTION
<code>CComObjectStack::m_hResFinalConstruct</code>	Contains the <code>HRESULT</code> returned during construction of the <code>CComObjectStack</code> object.

Remarks

`CComObjectStack` is used to create a temporary COM object and provide the object a skeletal implementation of `IUnknown`. Typically, the object is used as a local variable within one function (that is, pushed onto the stack). Since the object is destroyed when the function finishes, reference counting is not performed to increase efficiency.

The following example shows how to create a COM object used inside a function:

```
void MyFunc()
{
    CComObjectStack<CMyClass2> Tempobj;
    //...
}
```

The temporary object `Tempobj` is pushed onto the stack and automatically disappears when the function finishes.

Inheritance Hierarchy

Base

CComObjectStack

Requirements

Header: atlcom.h

CComObjectStack::AddRef

Returns zero.

```
STDMETHOD_(ULONG, AddRef)();
```

Return Value

Returns zero.

Remarks

In debug mode, calls `_ASSERTE`.

CComObjectStack::CComObjectStack

The constructor.

```
CComObjectStack(void* = NULL);
```

Remarks

Calls `FinalConstruct` and then sets `m_hResFinalConstruct` to the HRESULT returned by `FinalConstruct`. If you have not derived your base class from `CComObjectRoot`, you must supply your own `FinalConstruct` method. The destructor calls `FinalRelease`.

CComObjectStack::~~CComObjectStack

The destructor.

```
CComObjectStack();
```

Remarks

Frees all allocated resources and calls `FinalRelease`.

CComObjectStack::m_hResFinalConstruct

Contains the HRESULT returned from calling `FinalConstruct` during construction of the `CComObjectStack` object.

```
HRESULT    m_hResFinalConstruct;
```

CComObjectStack::QueryInterface

Returns E_NOINTERFACE.

```
HRESULT    QueryInterface(REFIID, void**);
```

Return Value

Returns E_NOINTERFACE.

Remarks

In debug mode, calls `_ASSERTE` .

CComObjectStack::Release

Returns zero.

```
STDMETHOD_(ULONG, Release)();
```

Return Value

Returns zero.

Remarks

In debug mode, calls `_ASSERTE` .

See also

[CComAggObject Class](#)

[CComObject Class](#)

[CComObjectGlobal Class](#)

[Class Overview](#)

CComPolyObject Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

This class implements `IUnknown` for an aggregated or nonaggregated object.

Syntax

```
template<class contained>
class CComPolyObject : public IUnknown,
    public CComObjectRootEx<contained::_ThreadModel::ThreadModelNoCS>
```

Parameters

contained

Your class, derived from [CComObjectRoot](#) or [CComObjectRootEx](#), as well as from any other interfaces you want to support on the object.

Members

Public Constructors

NAME	DESCRIPTION
CComPolyObject::CComPolyObject	The constructor.
CComPolyObject::~~CComPolyObject	The destructor.

Public Methods

NAME	DESCRIPTION
CComPolyObject::AddRef	Increments the object's reference count.
CComPolyObject::CreateInstance	(Static) Allows you to create a new CComPolyObject<contained> object without the overhead of CoCreateInstance .
CComPolyObject::FinalConstruct	Performs final initialization of <code>m_contained</code> .
CComPolyObject::FinalRelease	Performs final destruction of <code>m_contained</code> .
CComPolyObject::QueryInterface	Retrieves a pointer to the requested interface.
CComPolyObject::Release	Decrements the object's reference count.

Public Data Members

NAME	DESCRIPTION

NAME	DESCRIPTION
CComPolyObject::m_contained	Delegates IUnknown calls to the outer unknown if the object is aggregated or to the IUnknown of the object if the object is not aggregated.

Remarks

[CComPolyObject](#) implements [IUnknown](#) for an aggregated or nonaggregated object.

When an instance of [CComPolyObject](#) is created, the value of the outer unknown is checked. If it is NULL, [IUnknown](#) is implemented for a nonaggregated object. If the outer unknown is not NULL, [IUnknown](#) is implemented for an aggregated object.

The advantage of using [CComPolyObject](#) is that you avoid having both [CComAggObject](#) and [CComObject](#) in your module to handle the aggregated and nonaggregated cases. A single [CComPolyObject](#) object handles both cases. This means only one copy of the vtable and one copy of the functions exist in your module. If your vtable is large, this can substantially decrease your module size. However, if your vtable is small, using [CComPolyObject](#) can result in a slightly larger module size because it is not optimized for an aggregated or nonaggregated object, as are [CComAggObject](#) and [CComObject](#).

If the `DECLARE_POLY_AGGREGATABLE` macro is specified in your object's class definition, [CComPolyObject](#) will be used to create your object. `DECLARE_POLY_AGGREGATABLE` will automatically be declared if you use the ATL Project Wizard to create a full control or Internet Explorer control.

If aggregated, the [CComPolyObject](#) object has its own [IUnknown](#), separate from the outer object's [IUnknown](#), and maintains its own reference count. [CComPolyObject](#) uses [CComContainedObject](#) to delegate to the outer unknown.

For more information about aggregation, see the article [Fundamentals of ATL COM Objects](#).

Inheritance Hierarchy

[CComObjectRootBase](#)

[CComObjectRootEx](#)

[IUnknown](#)

[CComPolyObject](#)

Requirements

Header: atlcom.h

CComPolyObject::AddRef

Increments the reference count on the object.

```
STDMETHOD_(ULONG, AddRef)();
```

Return Value

A value that may be useful for diagnostics or testing.

CComPolyObject::CComPolyObject

The constructor.

```
CComPolyObject(void* pv);
```

Parameters

pv

[in] A pointer to the outer unknown if the object is to be aggregated, or NULL if the object is not aggregated.

Remarks

Initializes the `CComContainedObject` data member, `m_contained`, and increments the module lock count.

The destructor decrements the module lock count.

CComPolyObject::~~CComPolyObject

The destructor.

```
~CComPolyObject();
```

Remarks

Frees all allocated resources, calls `FinalRelease`, and decrements the module lock count.

CComPolyObject::CreateInstance

Allows you to create a new **CComPolyObject**< `contained` > object without the overhead of `CoCreateInstance`.

```
static HRESULT WINAPI CreateInstance(  
    LPUNKNOWN pUnkOuter,  
    CComPolyObject<contained>** pp);
```

Parameters

pp

[out] A pointer to a **CComPolyObject**< `contained` > pointer. If `CreateInstance` is unsuccessful, *pp* is set to NULL.

Return Value

A standard HRESULT value.

Remarks

The object returned has a reference count of zero, so call `AddRef` immediately, then use `Release` to free the reference on the object pointer when you're done.

If you don't need direct access to the object, but still want to create a new object without the overhead of `CoCreateInstance`, use `CComCoClass::CreateInstance` instead.

CComPolyObject::FinalConstruct

Called during the final stages of object construction, this method performs any final initialization on the `m_contained` data member.

```
HRESULT FinalConstruct();
```

Return Value

A standard HRESULT value.

CComPolyObject::FinalRelease

Called during object destruction, this method frees the [m_contained](#) data member.

```
void FinalRelease();
```

CComPolyObject::m_contained

A [CComContainedObject](#) object derived from your class.

```
CComContainedObject<contained> m_contained;
```

Parameters

contained

[in] Your class, derived from [CComObjectRoot](#) or [CComObjectRootEx](#), as well as from any other interfaces you want to support on the object.

Remarks

[IUnknown](#) calls through [m_contained](#) are delegated to the outer unknown if the object is aggregated, or to the [IUnknown](#) of this object if the object is not aggregated.

CComPolyObject::QueryInterface

Retrieves a pointer to the requested interface.

```
STDMETHOD(QueryInterface)(REFIID iid, void** ppvObject);  
template <class Q>  
HRESULT QueryInterface(Q** pp);
```

Parameters

Q

The COM interface.

iid

[in] The identifier of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by *iid*. If the object does not support this interface, *ppvObject* is set to NULL.

pp

[out] A pointer to the interface identified by `__uuidof(Q)`.

Return Value

A standard HRESULT value.

Remarks

For an aggregated object, if the requested interface is `IUnknown`, `QueryInterface` returns a pointer to the aggregated object's own `IUnknown` and increments the reference count. Otherwise, this method queries for the interface through the `CComContainedObject` data member, [m_contained](#).

CComPolyObject::Release

Decrements the reference count on the object.

```
STDMETHOD_(ULONG, Release)();
```

Return Value

In debug builds, `Release` returns a value that may be useful for diagnostics or testing. In nondebug builds, `Release` always returns 0.

See also

[CComObjectRootEx Class](#)

[DECLARE_POLY_AGGREGATABLE](#)

[Class Overview](#)

CComPtr Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

A smart pointer class for managing COM interface pointers.

Syntax

```
template<class T>  
class CComPtr
```

Parameters

T

A COM interface specifying the type of pointer to be stored.

Members

Public Constructors

NAME	DESCRIPTION
CComPtr::CComPtr	The constructor.

Public Operators

NAME	DESCRIPTION
CComPtr::operator =	Assigns a pointer to the member pointer.

Remarks

ATL uses `CComPtr` and `CComQIPtr` to manage COM interface pointers. Both are derived from `CComPtrBase`, and both perform automatic reference counting.

The `CComPtr` and `CComQIPtr` classes can help eliminate memory leaks by performing automatic reference counting. The following functions both perform the same logical operations; however, note how the second version may be less error-prone by using the `CComPtr` class:

```

// Error-checking routine that performs manual lifetime management
// of a COM IErrorInfo object
HRESULT CheckComError_Manual()
{
    HRESULT hr;
    CComBSTR bstrDescription;
    CComBSTR bstrSource;
    CComBSTR bstrHelpFile;

    IErrorInfo* pErrInfo = NULL; // naked COM interface pointer
    hr = ::GetErrorInfo(0, &pErrInfo);
    if(hr != S_OK)
        return hr;

    hr = pErrInfo->GetDescription(&bstrDescription);
    if(FAILED(hr))
    {
        pErrInfo->Release(); // must release interface pointer before returning
        return hr;
    }

    hr = pErrInfo->GetSource(&bstrSource);
    if(FAILED(hr))
    {
        pErrInfo->Release(); // must release interface pointer before returning
        return hr;
    }

    hr = pErrInfo->GetHelpFile(&bstrHelpFile);
    if(FAILED(hr))
    {
        pErrInfo->Release(); // must release interface pointer before returning
        return hr;
    }

    pErrInfo->Release(); // must release interface pointer before returning
    return S_OK;
}

```

```
// Error-checking routine that performs automatic lifetime management
// of a COM IErrorInfo object through a CComPtr smart pointer object
HRESULT CheckComError_SmartPtr()
{
    HRESULT hr;
    CComBSTR bstrDescription;
    CComBSTR bstrSource;
    CComBSTR bstrHelpFile;

    CComPtr<IErrorInfo> pErrInfo;
    hr = ::GetErrorInfo(0, &pErrInfo);
    if(hr != S_OK)
        return hr;

    hr = pErrInfo->GetDescription(&bstrDescription);
    if(FAILED(hr))
        return hr;

    hr = pErrInfo->GetSource(&bstrSource);
    if(FAILED(hr))
        return hr;

    hr = pErrInfo->GetHelpFile(&bstrHelpFile);
    if(FAILED(hr))
        return hr;

    return S_OK;
} // CComPtr will auto-release underlying IErrorInfo interface pointer as needed
```

In Debug builds, link `atlsd.lib` for code tracing.

Inheritance Hierarchy

[CComPtrBase](#)

CComPtr

Requirements

Header: `atlbase.h`

CComPtr::CComPtr

The constructor.

```
CComPtr() throw ();
CComPtr(T* lp) throw ();
CComPtr (const CComPtr<T>& lp) throw ();
```

Parameters

lp

Used to initialize the interface pointer.

T

A COM interface.

CComPtr::operator =

Assignment operator.


```
T* operator= (T* lp) throw ();  
T* operator= (const CComPtr<T>& lp) throw ();
```

Return Value

Returns a pointer to the updated `CComPtr` object

Remarks

This operation AddRefs the new object and releases the existing object, if one exists.

See also

[CComPtr::CComPtr](#)

[CComQIPtr::CComQIPtr](#)

[Class Overview](#)

CComPtrBase Class

3/4/2019 • 5 minutes to read • [Edit Online](#)

This class provides a basis for smart pointer classes using COM-based memory routines.

Syntax

```
template <class T>
class CComPtrBase
```

Parameters

T

The object type to be referenced by the smart pointer.

Members

Public Constructors

NAME	DESCRIPTION
CComPtrBase::~~CComPtrBase	The destructor.

Public Methods

NAME	DESCRIPTION
CComPtrBase::Advise	Call this method to create a connection between the <code>CComPtrBase</code> 's connection point and a client's sink.
CComPtrBase::Attach	Call this method to take ownership of an existing pointer.
CComPtrBase::CoCreateInstance	Call this method to create an object of the class associated with a specified Class ID or Program ID.
CComPtrBase::CopyTo	Call this method to copy the <code>CComPtrBase</code> pointer to another pointer variable.
CComPtrBase::Detach	Call this method to release ownership of a pointer.
CComPtrBase::IsEqualObject	Call this method to check if the specified <code>IUnknown</code> points to the same object associated with the <code>CComPtrBase</code> object.
CComPtrBase::QueryInterface	Call this method to return a pointer to a specified interface.
CComPtrBase::Release	Call this method to release the interface.
CComPtrBase::SetSite	Call this method to set the site of the <code>CComPtrBase</code> object to the <code>IUnknown</code> of the parent object.

Public Operators

NAME	DESCRIPTION
CComPtrBase::operator T*	The cast operator.
CComPtrBase::operator !	The NOT operator.
CComPtrBase::operator &	The & operator.
CComPtrBase::operator *	The * operator.
[CComPtrBase::operator <](#ccomptrbase__operator lt)	The less-than operator.
CComPtrBase::operator ==	The equality operator.
CComPtrBase::operator ->	The pointer-to-members operator.

Public Data Members

NAME	DESCRIPTION
CComPtrBase::p	The pointer data member variable.

Remarks

This class provides the basis for other smart pointers which use COM memory management routines, such as [CComQIPtr](#) and [CComPtr](#). The derived classes add their own constructors and operators, but rely on the methods provided by [CComPtrBase](#).

Requirements

Header: atlcomcli.h

CComPtrBase::Advise

Call this method to create a connection between the [CComPtrBase](#)'s connection point and a client's sink.

```
HRESULT Advise(
    IUnknown* pUnk,
    const IID& iid,
    LPDWORD pdw) throw();
```

Parameters

pUnk

A pointer to the client's [IUnknown](#).

iid

The GUID of the connection point. Typically, this is the same as the outgoing interface managed by the connection point.

pdw

A pointer to the cookie that uniquely identifies the connection.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

See [AtlAdvise](#) for more information.

CComPtrBase::Attach

Call this method to take ownership of an existing pointer.

```
void Attach(T* p2) throw();
```

Parameters

p2

The `CComPtrBase` object will take ownership of this pointer.

Remarks

`Attach` calls `CComPtrBase::Release` on the existing `CComPtrBase::p` member variable and then assigns *p2* to `CComPtrBase::p`. When a `CComPtrBase` object takes ownership of a pointer, it will automatically call `Release` on the pointer which will delete the pointer and any allocated data if the reference count on the object goes to 0.

CComPtrBase::~CComPtrBase

The destructor.

```
~CComPtrBase() throw();
```

Remarks

Releases the interface pointed to by `CComPtrBase`.

CComPtrBase::CoCreateInstance

Call this method to create an object of the class associated with a specified Class ID or Program ID.

```
HRESULT CoCreateInstance(  
    LPCOLESTR szProgID,  
    LPUNKNOWN pUnkOuter = NULL,  
    DWORD dwClsContext = CLSCTX_ALL) throw();  
  
HRESULT CoCreateInstance(  
    REFCLSID rclsid,  
    LPUNKNOWN pUnkOuter = NULL,  
    DWORD dwClsContext = CLSCTX_ALL) throw();
```

Parameters

szProgID

Pointer to a ProgID, used to recover the CLSID.

pUnkOuter

If NULL, indicates that the object is not being created as part of an aggregate. If non- NULL, is a pointer to the aggregate object's `IUnknown` interface (the controlling `IUnknown`).

dwClsContext

Context in which the code that manages the newly created object will run.

clsid

CLSID associated with the data and code that will be used to create the object.

Return Value

Returns S_OK on success, or REGDB_E_CLASSNOTREG, CLASS_E_NOAGGREGATION, CO_E_CLASSSTRING or E_NOINTERFACE on failure. See [CoCreateClassInstance](#) and [CLSIDFromProgID](#) for a description of these errors.

Remarks

If the first form of the method is called, [CLSIDFromProgID](#) is used to recover the CLSID. Both forms then call [CoCreateClassInstance](#).

In debug builds, an assertion error will occur if [CComPtrBase::p](#) is not equal to NULL.

CComPtrBase::CopyTo

Call this method to copy the [CComPtrBase](#) pointer to another pointer variable.

```
HRESULT CopyTo(T** ppT) throw();
```

Parameters

ppT

Address of the variable which will receive the [CComPtrBase](#) pointer.

Return Value

Returns S_OK on success, E_POINTER on failure.

Remarks

Copies the [CComPtrBase](#) pointer to *ppT*. The reference count on the [CComPtrBase::p](#) member variable is incremented.

An error HRESULT will be returned if *ppT* is equal to NULL. In debug builds, an assertion error will occur if *ppT* is equal to NULL.

CComPtrBase::Detach

Call this method to release ownership of a pointer.

```
T* Detach() throw();
```

Return Value

Returns a copy of the pointer.

Remarks

Releases ownership of a pointer, sets the [CComPtrBase::p](#) data member variable to NULL, and returns a copy of the pointer.

CComPtrBase::IsEqualObject

Call this method to check if the specified [IUnknown](#) points to the same object associated with the [CComPtrBase](#) object.

```
bool IsEqualObject(IUnknown* pOther) throw();
```

Parameters

pOther

The `IUnknown *` to compare.

Return Value

Returns true if the objects are identical, false otherwise.

CComPtrBase::operator !

The NOT operator.

```
bool operator!() const throw();
```

Return Value

Returns true if the `CComHeapPtr` pointer is equal to NULL, false otherwise.

CComPtrBase::operator &

The & operator.

```
T** operator&() throw();
```

Return Value

Returns the address of the object pointed to by the `CComPtrBase` object.

CComPtrBase::operator *

The * operator.

```
T& operator*() const throw();
```

Return Value

Returns the value of `CComPtrBase::p`; that is, a pointer to the object referenced by the `CComPtrBase` object.

If debug builds, an assertion error will occur if `CComPtrBase::p` is not equal to NULL.

CComPtrBase::operator ==

The equality operator.

```
bool operator== (T* pT) const throw();
```

Parameters

pT

A pointer to an object.

Return Value

Returns true if `CComPtrBase` and *pT* point to the same object, false otherwise.

CComPtrBase::operator ->

The pointer-to-member operator.

```
_NoAddRefReleaseOnCComPtr<T>* operator->() const throw();
```

Return Value

Returns the value of the [CComPtrBase::p](#) data member variable.

Remarks

Use this operator to call a method in a class pointed to by the `CComPtrBase` object. In debug builds, an assertion failure will occur if the `CComPtrBase` data member points to NULL.

CComPtrBase::operator <

The less-than operator.

```
bool operator<(T* pT) const throw();
```

Parameters

pT

A pointer to an object.

Return Value

Returns true if the pointer managed by current object is less than the pointer to which it is being compared.

CComPtrBase::operator T*

The cast operator.

```
operator T*() const throw();
```

Remarks

Returns a pointer to the object data type defined in the class template.

CComPtrBase::p

The pointer data member variable.

```
T* p;
```

Remarks

This member variable holds the pointer information.

CComPtrBase::QueryInterface

Call this method to return a pointer to a specified interface.

```
template <class Q> HRESULT QueryInterface(Q  
** pp) const throw();
```

Parameters

Q

The object type whose interface pointer is required.

pp

Address of output variable that receives the requested interface pointer.

Return Value

Returns S_OK on success, or E_NOINTERFACE on failure.

Remarks

This method calls [IUnknown::QueryInterface](#).

In debug builds, an assertion error will occur if *pp* is not equal to NULL.

CComPtrBase::Release

Call this method to release the interface.

```
void Release() throw();
```

Remarks

The interface is released, and [CComPtrBase::p](#) is set to NULL.

CComPtrBase::SetSite

Call this method to set the site of the [CComPtrBase](#) object to the [IUnknown](#) of the parent object.

```
HRESULT SetSite(IUnknown* punkParent) throw();
```

Parameters

punkParent

A pointer to the [IUnknown](#) interface of the parent.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method calls [AtlSetChildSite](#).

See also

[Class Overview](#)

CComQIPtr Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

A smart pointer class for managing COM interface pointers.

Syntax

```
template<class T, const IID* piid= &__uuidof(T)>
class CComQIPtr: public CComPtr<T>
```

Parameters

T

A COM interface specifying the type of pointer to be stored.

piid

A pointer to the IID of *T*.

Members

Public Constructors

NAME	DESCRIPTION
CComQIPtr::CComQIPtr	Constructor.

Public Operators

NAME	DESCRIPTION
CComQIPtr::operator =	Assigns a pointer to the member pointer.

Remarks

ATL uses [CComQIPtr](#) and [CComPtr](#) to manage COM interface pointers, both of which derive from [CComPtrBase](#). Both classes perform automatic reference counting through calls to [AddRef](#) and [Release](#). Overloaded operators handle pointer operations.

Inheritance Hierarchy

[CComPtrBase](#)

[CComPtr](#)

[CComQIPtr](#)

Requirements

Header: [atlcomcli.h](#)

[CComQIPtr::CComQIPtr](#)

The constructor.

```
CComQIPtr() throw();  
CComQIPtr(T* lp) throw();  
CComQIPtr(IUnknown* lp) throw();  
CComQIPtr(const CComQIPtr<T, piid>& lp) throw();
```

Parameters

lp

Used to initialize the interface pointer.

T

A COM interface.

piid

A pointer to the IID of *T*.

CComQIPtr::operator =

The assignment operator.

```
T* operator= (T* lp) throw();  
T* operator= (const CComQIPtr<T, piid>& lp) throw();  
T* operator= (IUnknown* lp) throw();
```

Parameters

lp

Used to initialize the interface pointer.

T

A COM interface.

piid

A pointer to the IID of *T*.

Return Value

Returns a pointer to the updated `CComQIPtr` object.

See also

[CComPtr::CComPtr](#)

[CComQIPtr::CComQIPtr](#)

[CComPtrBase Class](#)

[Class Overview](#)

[CComQIPtrElementTraits Class](#)

CComQIPtrElementTraits Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods, static functions, and typedefs useful when creating collections of COM interface pointers.

Syntax

```
template<typename I, const IID* piid=& __uuidof(I)>
class CComQIPtrElementTraits :
    public CDefaultElementTraits<ATL::CComQIPtr<I, piid>>
```

Parameters

I

A COM interface specifying the type of pointer to be stored.

piid

A pointer to the IID of *I*.

Members

Public Typedefs

NAME	DESCRIPTION
CComQIPtrElementTraits::INARGTYPE	The data type to use for adding elements to the collection class object.

Remarks

This class derives methods and provides a typedef useful when creating a collection class of [CComQIPtr](#) COM interface pointer objects. This class is utilized by both the [CInterfaceArray](#) and [CInterfaceList](#) classes.

For more information, see [ATL Collection Classes](#).

Inheritance Hierarchy

[CDefaultCompareTraits](#)

[CDefaultHashTraits](#)

[CElementTraitsBase](#)

[CDefaultElementTraits](#)

CComQIPtrElementTraits

Requirements

Header: atlcoll.h

CComQIPtrElementTraits::INARGTYPE

The data type to use for adding elements to the collection class object.

```
typedef I* INARGTYPE;
```

See also

[CDefaultElementTraits Class](#)

[Class Overview](#)

CComSafeArray Class

5/7/2019 • 10 minutes to read • [Edit Online](#)

This class is a wrapper for the `SAFEARRAY` structure.

Syntax

```
template <typename T, VARTYPE _vartype = _ATL_AutomationType<T>::type>
class CComSafeArray
```

Parameters

T

The type of data to be stored in the array.

Members

Public Constructors

NAME	DESCRIPTION
CComSafeArray::CComSafeArray	The constructor.
CComSafeArray::~~CComSafeArray	The destructor.

Public Methods

NAME	DESCRIPTION
CComSafeArray::Add	Adds one or more elements, or a <code>SAFEARRAY</code> structure, to a <code>CComSafeArray</code> .
CComSafeArray::Attach	Attaches a <code>SAFEARRAY</code> structure to a <code>CComSafeArray</code> object.
CComSafeArray::CopyFrom	Copies the contents of a <code>SAFEARRAY</code> structure into the <code>CComSafeArray</code> object.
CComSafeArray::CopyTo	Creates a copy of the <code>CComSafeArray</code> object.
CComSafeArray::Create	Creates a <code>CComSafeArray</code> object.
CComSafeArray::Destroy	Destroys a <code>CComSafeArray</code> object.
CComSafeArray::Detach	Detaches a <code>SAFEARRAY</code> from a <code>CComSafeArray</code> object.
CComSafeArray::GetAt	Retrieves a single element from a single-dimensional array.
CComSafeArray::GetCount	Returns the number of elements in the array.

NAME	DESCRIPTION
CComSafeArray::GetDimensions	Returns the number of dimensions in the array.
CComSafeArray::GetLowerBound	Returns the lower bound for a given dimension of the array.
CComSafeArray::GetSafeArrayPtr	Returns the address of the <code>m_psa</code> data member.
CComSafeArray::GetType	Returns the type of data stored in the array.
CComSafeArray::GetUpperBound	Returns the upper bound for any dimension of the array.
CComSafeArray::IsSizable	Tests if a <code>CComSafeArray</code> object can be resized.
CComSafeArray::MultiDimGetAt	Retrieves a single element from a multidimensional array.
CComSafeArray::MultiDimSetAt	Sets the value of an element in a multidimensional array.
CComSafeArray::Resize	Resizes a <code>CComSafeArray</code> object.
CComSafeArray::SetAt	Sets the value of an element in a single-dimensional array.

Public Operators

NAME	DESCRIPTION
CComSafeArray::operator LPSAFEARRAY	Casts a value to a <code>SAFEARRAY</code> pointer.
CComSafeArray::operator[]	Retrieves an element from the array.
CComSafeArray::operator =	Assignment operator.

Public Data Members

NAME	DESCRIPTION
CComSafeArray::m_psa	This data member holds the address of the <code>SAFEARRAY</code> structure.

Remarks

`CComSafeArray` provides a wrapper for the [SAFEARRAY Data Type](#) class, making it a simple matter to create and manage single- and multidimensional arrays of almost any of the VARIANT-supported types.

`CComSafeArray` simplifies passing arrays between processes, and in addition provides extra security by checking array index values against upper and lower bounds.

The lower bound of a `CComSafeArray` can start at any user-defined value; however, arrays that are accessed through C++ should use a lower bound of 0. Other languages such as Visual Basic may use other bounding values (for example, -10 to 10).

Use [CComSafeArray::Create](#) to create a `CComSafeArray` object, and [CComSafeArray::Destroy](#) to delete it.

A `CComSafeArray` can contain the following subset of VARIANT data types:

VARTYPE	DESCRIPTION
VT_I1	char
VT_I2	short
VT_I4	int
VT_I4	long
VT_I8	longlong
VT_UI1	byte
VT_UI2	ushort
VT_UI4	uint
VT_UI4	ulong
VT_UI8	ulonglong
VT_R4	float
VT_R8	double
VT_DECIMAL	decimal pointer
VT_VARIANT	variant pointer
VT_CY	Currency data type

Requirements

Header: atlsafe.h

Example

```

// Create a multidimensional array,
// then write and read elements

// Define an array of character pointers
CComSafeArray<char> *pSar;

char cElement;
char cTable[2][3] = {'A','B','C','D','E','F'};

// Declare the variable used to store the
// array indexes
LONG aIndex[2];

// Define the array bound structure
CComSafeArrayBound bound[2];
bound[0].SetCount(2);
bound[0].SetLowerBound(0);
bound[1].SetCount(3);
bound[1].SetLowerBound(0);

// Create a new 2 dimensional array
// each dimension size is 3
pSar = new CComSafeArray<char>(bound,2);

// Use MultiDimSetAt to store characters in the array
for (int x = 0; x < 2; x++)
{
    for (int y = 0; y < 3; y++)
    {
        aIndex[0] = x;
        aIndex[1] = y;
        HRESULT hr = pSar->MultiDimSetAt(aIndex,cTable[x][y]);
        ATLASSERT(hr == S_OK);
    }
}
// Use MultiDimGetAt to retrieve characters in the array
for (int x = 0; x < 2; x++)
{
    for (int y = 0; y < 3; y++)
    {
        aIndex[0]=x;
        aIndex[1]=y;
        HRESULT hr = pSar->MultiDimGetAt(aIndex,cElement);
        ATLASSERT(hr == S_OK);
        ATLASSERT(cElement == cTable[x][y]);
    }
}

```

CComSafeArray::Add

Adds one or more elements, or a `SAFEARRAY` structure, to a `CComSafeArray` .

```

HRESULT Add(const SAFEARRAY* psaSrc);
HRESULT Add(ULONG ulCount, const T* pT, BOOL bCopy = TRUE);
HRESULT Add(const T& t, BOOL bCopy = TRUE);

```

Parameters

psaSrc

A pointer to a `SAFEARRAY` object.

ulCount

The number of objects to add to the array.

pT

A pointer to one or more objects to be added to the array.

t

A reference to the object to be added to the array.

bCopy

Indicates whether a copy of the data should be created. The default value is TRUE.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The new objects are appended to the end of the existing `SAFEARRAY` object. Adding an object to a multidimensional `SAFEARRAY` object is not supported. When adding an existing array of objects, both arrays must contain elements of the same type.

The *bCopy* flag is taken into account when elements of type BSTR or VARIANT are added to an array. The default value of TRUE ensures that a new copy is made of the data when the element is added to the array.

CComSafeArray::Attach

Attaches a `SAFEARRAY` structure to a `CComSafeArray` object.

```
HRESULT Attach(const SAFEARRAY* psaSrc);
```

Parameters

psaSrc

A pointer to the `SAFEARRAY` structure.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Attaches a `SAFEARRAY` structure to a `CComSafeArray` object, making the existing `CComSafeArray` methods available.

CComSafeArray::CComSafeArray

The constructor.

```
CComSafeArray();  
CComSafeArray(const SAFEARRAYBOUND& bound);  
CComSafeArray(ULONG ulCount, LONG lBound = 0);  
CComSafeArray(const SAFEARRAYBOUND* pBound, UINT uDims = 1);  
CComSafeArray(const CComSafeArray& saSrc);  
CComSafeArray(const SAFEARRAY& saSrc);  
CComSafeArray(const SAFEARRAY* psaSrc);
```

Parameters

bound

A `SAFEARRAYBOUND` structure.

ulCount

The number of elements in the array.

lLBound

The lower bound value; that is, the index of the first element in the array.

pBound

A pointer to a `SAFEARRAYBOUND` structure.

uDims

The count of dimensions in the array.

saSrc

A reference to a `SAFEARRAY` structure or `CComSafeArray` object. In either case the constructor uses this reference to make a copy of the array, so the array is not referenced after construction.

psaSrc

A pointer to a `SAFEARRAY` structure. The constructor uses this address to make a copy of the array, so the array is not referenced after construction.

Remarks

Creates a `CComSafeArray` object.

CComSafeArray::~CComSafeArray

The destructor.

```
~CComSafeArray() throw()
```

Remarks

Frees all allocated resources.

CComSafeArray::CopyFrom

Copies the contents of a `SAFEARRAY` structure into the `CComSafeArray` object.

```
HRESULT CopyFrom(LPSAFEARRAY* ppArray);
```

Parameters

ppArray

Pointer to the `SAFEARRAY` to copy.

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure.

Remarks

This method copies the contents of a `SAFEARRAY` into the current `CComSafeArray` object. The existing contents of the array are replaced.

CComSafeArray::CopyTo

Creates a copy of the `CComSafeArray` object.

```
HRESULT CopyTo(LPSAFEARRAY* ppArray);
```

Parameters

ppArray

A pointer to a location in which to create the new `SAFEARRAY`.

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure.

Remarks

This method copies the contents of a `CComSafeArray` object into a `SAFEARRAY` structure.

CComSafeArray::Create

Creates a `CComSafeArray`.

```
HRESULT Create(const SAFEARRAYBOUND* pBound, UINT uDims = 1);  
HRESULT Create(ULONG ulCount = 0, LONG lLBound = 0);
```

Parameters

pBound

A pointer to a `SAFEARRAYBOUND` object.

uDims

The number of dimensions in the array.

ulCount

The number of elements in the array.

lLBound

The lower bound value; that is, the index of the first element in the array.

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure.

Remarks

A `CComSafeArray` object can be created from an existing `SAFEARRAYBOUND` structure and the number of dimensions, or by specifying the number of elements in the array and the lower bound. If the array is to be accessed from C++, the lower bound should be 0. Other languages may allow other values for the lower bound (for example, Visual Basic supports arrays with elements with a range such as -10 to 10).

CComSafeArray::Destroy

Destroys a `CComSafeArray` object.

```
HRESULT Destroy();
```

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure.

Remarks

Destroys an existing `CComSafeArray` object and all of the data it contains.

CComSafeArray::Detach

Detaches a `SAFEARRAY` from a `CComSafeArray` object.

```
LPSAFEARRAY Detach();
```

Return Value

Returns a pointer to a `SAFEARRAY` object.

Remarks

This method detaches the `SAFEARRAY` object from the `CComSafeArray` object.

CComSafeArray::GetAt

Retrieves a single element from a single-dimensional array.

```
T& GetAt(LONG lIndex) const;
```

Parameters

lIndex

The index number of the value in the array to return.

Return Value

Returns a reference to the required array element.

CComSafeArray::GetCount

Returns the number of elements in the array.

```
ULONG GetCount(UINT uDim = 0) const;
```

Parameters

uDim

The array dimension.

Return Value

Returns the number of elements in the array.

Remarks

When used with a multidimensional array, this method will return the number of elements in a specific dimension only.

CComSafeArray::GetDimensions

Returns the number of dimensions in the array.

```
UINT GetDimensions() const;
```

Return Value

Returns the number of dimensions in the array.

CComSafeArray::GetLowerBound

Returns the lower bound for a given dimension of the array.

```
LONG GetLowerBound(UINT uDim = 0) const;
```

Parameters

uDim

The array dimension for which to get the lower bound. If omitted, the default is 0.

Return Value

Returns the lower bound.

Remarks

If the lower bound is 0, this indicates a C-like array whose first element is element number 0. In the event of an error, for example, an invalid dimension argument, this method calls `AtlThrow` with an HRESULT describing the error.

CComSafeArray::GetSafeArrayPtr

Returns the address of the `m_psa` data member.

```
LPSAFEARRAY* GetSafeArrayPtr() throw();
```

Return Value

Returns a pointer to the `CComSafeArray::m_psa` data member.

CComSafeArray::GetType

Returns the type of data stored in the array.

```
VARTYPE GetType() const;
```

Return Value

Returns the type of data stored in the array, which could be any of the following types:

VARTYPE	DESCRIPTION
VT_I1	char
VT_I2	short
VT_I4	int
VT_I4	long
VT_I8	longlong
VT_UI1	byte
VT_UI2	ushort
VT_UI4	uint

VARTYPE	DESCRIPTION
VT_UI4	ulong
VT_UI8	ulonglong
VT_R4	float
VT_R8	double
VT_DECIMAL	decimal pointer
VT_VARIANT	variant pointer
VT_CY	Currency data type

CComSafeArray::GetUpperBound

Returns the upper bound for any dimension of the array.

```
LONG GetUpperBound(UINT uDim = 0) const;
```

Parameters

uDim

The array dimension for which to get the upper bound. If omitted, the default is 0.

Return Value

Returns the upper bound. This value is inclusive, the maximum valid index for this dimension.

Remarks

In the event of an error, for example, an invalid dimension argument, this method calls `AtlThrow` with an `HRESULT` describing the error.

CComSafeArray::IsSizable

Tests if a `CComSafeArray` object can be resized.

```
bool IsSizable() const;
```

Return Value

Returns TRUE if the `CComSafeArray` can be resized, FALSE if it cannot.

CComSafeArray::m_psa

Holds the address of the `SAFEARRAY` structure accessed.

```
LPSAFEARRAY m_psa;
```

CComSafeArray::MultiDimGetAt

Retrieves a single element from a multidimensional array.

```
HRESULT MultiDimGetAt(const LONG* aIndex, T& t);
```

Parameters

aIndex

Pointer to a vector of indexes for each dimension in the array. The leftmost (most significant) dimension is

`aIndex[0]`.

t

A reference to the data returned.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CComSafeArray::MultiDimSetAt

Sets the value of an element in a multidimensional array.

```
HRESULT MultiDimSetAt(const LONG* aIndex, const T& t);
```

Parameters

aIndex

Pointer to a vector of indexes for each dimension in the array. The rightmost (least significant) dimension is

`aIndex[0]`.

T

Specifies the value of the new element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This is a multidimensional version of [CComSafeArray::SetAt](#).

CComSafeArray::operator []

Retrieves an element from the array.

```
T& operator[](long lindex) const;  
T& operator[](int nindex) const;
```

Parameters

lIndex, nIndex

The index number of the required element in the array.

Return Value

Returns the appropriate array element.

Remarks

Performs a similar function to [CComSafeArray::GetAt](#), however this operator only works with single-dimensional arrays.

CComSafeArray::operator =

Assignment operator.

```
ATL::CComSafeArray<T>& operator=(const ATL::CComSafeArray& saSrc);  
ATL::CComSafeArray<T>& operator=(const SAFEARRAY* psaSrc);
```

Parameters

saSrc

A reference to a `CComSafeArray` object.

psaSrc

A pointer to a `SAFEARRAY` object.

Return Value

Returns the type of data stored in the array.

CComSafeArray::operator LPSAFEARRAY

Casts a value to a `SAFEARRAY` pointer.

```
operator LPSAFEARRAY() const;
```

Return Value

Casts a value to a `SAFEARRAY` pointer.

CComSafeArray::Resize

Resizes a `CComSafeArray` object.

```
HRESULT Resize(const SAFEARRAYBOUND* pBound);  
HRESULT Resize(ULONG ulCount, LONG lBound = 0);
```

Parameters

pBound

A pointer to a `SAFEARRAYBOUND` structure that contains information on the number of elements and the lower bound of an array.

ulCount

The requested number of objects in the resized array.

lBound

The lower bound.

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure.

Remarks

This method only resizes the rightmost dimension. It will not resize arrays that return `IsResizable` as `FALSE`.

CComSafeArray::SetAt

Sets the value of an element in a single-dimensional array.


```
HRESULT SetAt(LONG lIndex, const T& t, BOOL bCopy = TRUE);
```

Parameters

lIndex

The index number of the array element to set.

t

The new value of the specified element.

bCopy

Indicates whether a copy of the data should be created. The default value is TRUE.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The *bCopy* flag is taken into account when elements of type BSTR or VARIANT are added to an array. The default value of TRUE ensures that a new copy is made of the data when the element is added to the array.

See also

[SAFEARRAY Data Type](#)

[CComSafeArray::Create](#)

[CComSafeArray::Destroy](#)

[Class Overview](#)

CComSafeArrayBound Class

5/7/2019 • 2 minutes to read • [Edit Online](#)

This class is a wrapper for a [SAFEARRAYBOUND](#) structure.

Syntax

```
class CComSafeArrayBound : public SAFEARRAYBOUND
```

Members

Methods

CComSafeArrayBound	The constructor.
GetCount	Call this method to return the number of elements.
GetLowerBound	Call this method to return the lower bound.
GetUpperBound	Call this method to return the upper bound.
SetCount	Call this method to set the number of elements.
SetLowerBound	Call this method to set the lower bound.

Operators

operator =	Sets the CComSafeArrayBound to a new value.

Remarks

This class is a wrapper for the [SAFEARRAYBOUND](#) structure used by [CComSafeArray](#). It provides methods for querying and setting the upper and lower bounds of a single dimension of a [CComSafeArray](#) object and the number of elements it contains. A multidimensional [CComSafeArray](#) object uses an array of [CComSafeArrayBound](#) objects, one for each dimension. Therefore, when using methods such as [GetCount](#), be aware that this method will not return the total number of elements in a multidimensional array.

Header: [atlsafe.h](#)

Requirements

Header: [atlsafe.h](#)

CComSafeArrayBound::CComSafeArrayBound

The constructor.

```
CComSafeArrayBound(ULONG ulCount = 0, LONG lLowerBound = 0) throw();
```

Parameters

ulCount

The number of elements in the array.

lLowerBound

The lower bound from which the array is numbered.

Remarks

If the array is to be accessed from a C++ program, it is recommended that the lower bound be defined as 0. It may be preferable to use a different lower bound value if the array is to be used with other languages, such as Visual Basic.

CComSafeArrayBound::GetCount

Call this method to return the number of elements.

```
ULONG GetCount() const throw();
```

Return Value

Returns the number of elements.

Remarks

If the associated `CComSafeArray` object represents a multidimensional array, this method will only return the total number of elements in the rightmost dimension. Use [CComSafeArray::GetCount](#) to obtain the total number of elements.

CComSafeArrayBound::GetLowerBound

Call this method to return the lower bound.

```
LONG GetLowerBound() const throw();
```

Return Value

Returns the lower bound of the `CComSafeArrayBound` object.

CComSafeArrayBound::GetUpperBound

Call this method to return the upper bound.

```
LONG GetUpperBound() const throw();
```

Return Value

Returns the upper bound of the `CComSafeArrayBound` object.

Remarks

The upper bound depends on the number of elements and the lower bound value. For example, if the lower bound is 0 and the number of elements is 10, the upper bound will automatically be set to 9.

CComSafeArrayBound::operator =

Sets the `CComSafeArrayBound` to a new value.

```
CComSafeArrayBound& operator= (const CComSafeArrayBound& bound) throw();  
CComSafeArrayBound& operator= (ULONG ulCount) throw();
```

Parameters

bound

A `CComSafeArrayBound` object.

ulCount

The number of elements.

Return Value

Returns a pointer to the `CComSafeArrayBound` object.

Remarks

The `CComSafeArrayBound` object can be assigned using an existing `CComSafeArrayBound`, or by supplying the number of elements, in which case the lower bound is set to 0 by default.

CComSafeArrayBound::SetCount

Call this method to set the number of elements.

```
ULONG SetCount(ULONG ulCount) throw();
```

Parameters

ulCount

The number of elements.

Return Value

Returns the number of elements in the `CComSafeArrayBound` object.

CComSafeArrayBound::SetLowerBound

Call this method to set the lower bound.

```
LONG SetLowerBound(LONG lLowerBound) throw();
```

Parameters

lLowerBound

The lower bound.

Return Value

Returns the new lower bound of the `CComSafeArrayBound` object.

Remarks

If the array is to be accessed from a Visual C++ program, it is recommended that the lower bound be defined as 0. It may be preferable to use a different lower bound value if the array is to be used with other languages, such as Visual Basic.

The upper bound depends on the number of elements and the lower bound value. For example, if the lower bound

is 0 and the number of elements is 10, the upper bound will automatically be set to 9.

See also

[Class Overview](#)

CComSafeDeleteCriticalSection Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods for obtaining and releasing ownership of a critical section object.

Syntax

```
class CComSafeDeleteCriticalSection : public CComCriticalSection
```

Members

Public Constructors

NAME	DESCRIPTION
CComSafeDeleteCriticalSection::CComSafeDeleteCriticalSection	The constructor.
CComSafeDeleteCriticalSection::~~CComSafeDeleteCriticalSection	The destructor.

Public Methods

NAME	DESCRIPTION
CComSafeDeleteCriticalSection::Init	Creates and initializes a critical section object.
CComSafeDeleteCriticalSection::Lock	Obtains ownership of the critical section object.
CComSafeDeleteCriticalSection::Term	Releases system resources used by the critical section object.

Data Members

m_bInitialized	Flags whether the internal <code>CRITICAL_SECTION</code> object has been initialized.

Remarks

`CComSafeDeleteCriticalSection` derives from the class [CComCriticalSection](#). However, `CComSafeDeleteCriticalSection` provides additional safety mechanisms over [CComCriticalSection](#).

When an instance of `CComSafeDeleteCriticalSection` goes out of scope or is explicitly deleted from memory, the underlying critical section object will automatically be cleaned up if it is still valid. In addition, the [CComSafeDeleteCriticalSection::Term](#) method will exit gracefully if the underlying critical section object has not yet been allocated or has already been released from memory.

See [CComCriticalSection](#) for more information on critical section helper classes.

Inheritance Hierarchy

[CComCriticalSection](#)

CComSafeDeleteCriticalSection

Requirements

Header: atlcore.h

CComSafeDeleteCriticalSection::CComSafeDeleteCriticalSection

The constructor.

```
CComSafeDeleteCriticalSection();
```

Remarks

Sets the [m_bInitialized](#) data member to FALSE.

CComSafeDeleteCriticalSection::~~CComSafeDeleteCriticalSection

The destructor.

```
~CComSafeDeleteCriticalSection() throw();
```

Remarks

Releases the internal `CRITICAL_SECTION` object from memory if the [m_bInitialized](#) data member is set to TRUE.

CComSafeDeleteCriticalSection::Init

Calls the base class implementation of [Init](#) and sets [m_bInitialized](#) to TRUE if successful.

```
HRESULT Init() throw();
```

Return Value

Returns the result of [CComCriticalSection::Init](#).

CComSafeDeleteCriticalSection::Lock

Calls the base class implementation of [Lock](#).

```
HRESULT Lock();
```

Return Value

Returns the result of [CComCriticalSection::Lock](#).

Remarks

This method assumes the [m_bInitialized](#) data member is set to TRUE upon entry. An assertion is generated in Debug builds if this condition is not met.

For more information on the behavior of the function, refer to [CComCriticalSection::Lock](#).

CComSafeDeleteCriticalSection::m_bInitialized

Flags whether the internal `CRITICAL_SECTION` object has been initialized.

```
bool m_bInitialized;
```

Remarks

The `m_bInitialized` data member is used to track validity of the underlying `CRITICAL_SECTION` object associated with the [CComSafeDeleteCriticalSection](#) class. The underlying `CRITICAL_SECTION` object will not be attempted to be released from memory if this flag is not set to TRUE.

CComSafeDeleteCriticalSection::Term

Calls the base class implementation of [CComCriticalSection::Term](#) if the internal `CRITICAL_SECTION` object is valid.

```
HRESULT Term() throw();
```

Return Value

Returns the result of [CComCriticalSection::Term](#), or S_OK if `m_bInitialized` was set to FALSE upon entry.

Remarks

It is safe to call this method even if the internal `CRITICAL_SECTION` object is not valid. The destructor of this class calls this method if the `m_bInitialized` data member is set to TRUE.

See also

[CComCriticalSection Class](#)

[Class Overview](#)

CComSimpleThreadAllocator Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class manages thread selection for the class `CComAutoThreadModule`.

Syntax

```
class CComSimpleThreadAllocator
```

Members

Public Methods

NAME	DESCRIPTION
CComSimpleThreadAllocator::GetThread	Selects a thread.

Remarks

`CComSimpleThreadAllocator` manages thread selection for [CComAutoThreadModule](#).

`CComSimpleThreadAllocator::GetThread` simply cycles through each thread and returns the next one in the sequence.

Requirements

Header: atlbase.h

CComSimpleThreadAllocator::GetThread

Selects a thread by specifying the next thread in the sequence.

```
int GetThread(CComApartment* /* pApt */, int nThreads);
```

Parameters

pApt

Not used in ATL's default implementation.

nThreads

The maximum number of threads in the EXE module.

Return Value

An integer between zero and (*nThreads* - 1). Identifies one of the threads in the EXE module.

Remarks

You can override `GetThread` to provide a different method of selection or to make use of the *pApt* parameter.

`GetThread` is called by [CComAutoThreadModule::CreateInstance](#).

See also

[CComApartment Class](#)

[Class Overview](#)

CComSingleThreadModel Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods for incrementing and decrementing the value of a variable.

Syntax

```
class CComSingleThreadModel
```

Members

Public Typedefs

NAME	DESCRIPTION
CComSingleThreadModel::AutoCriticalSection	References class CComFakeCriticalSection .
CComSingleThreadModel::CriticalSection	References class <code>CComFakeCriticalSection</code> .
CComSingleThreadModel::ThreadModelNoCS	References <code>CComSingleThreadModel</code> .

Public Methods

NAME	DESCRIPTION
CComSingleThreadModel::Decrement	Decrements the value of the specified variable. This implementation is not thread-safe.
CComSingleThreadModel::Increment	Increments the value of the specified variable. This implementation is not thread-safe.

Remarks

`CComSingleThreadModel` provides methods for incrementing and decrementing the value of a variable. Unlike [CComMultiThreadModel](#) and [CComMultiThreadModelNoCS](#), these methods are not thread-safe.

Typically, you use `CComSingleThreadModel` through one of two **typedef** names, either [CComObjectThreadModel](#) or [CComGlobalsThreadModel](#). The class referenced by each **typedef** depends on the threading model used, as shown in the following table:

TYPEDF	SINGLE THREADING MODEL	APARTMENT THREADING MODEL	FREE THREADING MODEL
<code>CComObjectThreadModel</code>	S	S	M
<code>CComGlobalsThreadModel</code>	S	M	M

S = `CComSingleThreadModel`; M = `CComMultiThreadModel`

`CComSingleThreadModel` itself defines three **typedef** names. `ThreadModelNoCS` references `CComSingleThreadModel`. `AutoCriticalSection` and `CriticalSection` reference class [CComFakeCriticalSection](#), which provides empty methods associated with obtaining and releasing ownership of a critical section.

Requirements

Header: `atlbase.h`

CComSingleThreadModel::AutoCriticalSection

When using `CComSingleThreadModel`, the **typedef** name `AutoCriticalSection` references class [CComFakeCriticalSection](#).

```
typedef CComFakeCriticalSection AutoCriticalSection;
```

Remarks

Because `CComFakeCriticalSection` does not provide a critical section, its methods do nothing.

[CComMultiThreadModel](#) and [CComMultiThreadModelNoCS](#) contain definitions for `AutoCriticalSection`. The following table shows the relationship between the threading model class and the critical section class referenced by `AutoCriticalSection`:

CLASS DEFINED IN	CLASS REFERENCED
<code>CComSingleThreadModel</code>	<code>CComFakeCriticalSection</code>
<code>CComMultiThreadModel</code>	<code>CComAutoCriticalSection</code>
<code>CComMultiThreadModelNoCS</code>	<code>CComFakeCriticalSection</code>

In addition to `AutoCriticalSection`, you can use the **typedef** name [CriticalSection](#). You should not specify `AutoCriticalSection` in global objects or static class members if you want to eliminate the CRT startup code.

Example

See [CComMultiThreadModel::AutoCriticalSection](#).

CComSingleThreadModel::CriticalSection

When using `CComSingleThreadModel`, the **typedef** name `CriticalSection` references class [CComFakeCriticalSection](#).

```
typedef CComFakeCriticalSection CriticalSection;
```

Remarks

Because `CComFakeCriticalSection` does not provide a critical section, its methods do nothing.

[CComMultiThreadModel](#) and [CComMultiThreadModelNoCS](#) contain definitions for `CriticalSection`. The following table shows the relationship between the threading model class and the critical section class referenced by `CriticalSection`:

CLASS DEFINED IN	CLASS REFERENCED
<code>CComSingleThreadModel</code>	<code>CComFakeCriticalSection</code>
<code>CComMultiThreadModel</code>	<code>CComCriticalSection</code>
<code>CComMultiThreadModelNoCS</code>	<code>CComFakeCriticalSection</code>

In addition to `CriticalSection`, you can use the **typedef** name `AutoCriticalSection`. You should not specify `AutoCriticalSection` in global objects or static class members if you want to eliminate the CRT startup code.

Example

See `CComMultiThreadModel::AutoCriticalSection`.

CComSingleThreadModel::Decrement

This static function decrements the value of the variable pointed to by *p*.

```
static ULONG WINAPI Decrement(LPLONG p) throw();
```

Parameters

p
[in] Pointer to the variable to be decremented.

Return Value

The result of the decrement.

CComSingleThreadModel::Increment

This static function decrements the value of the variable pointed to by *p*.

```
static ULONG WINAPI Increment(LPLONG p) throw();
```

Parameters

p
[in] Pointer to the variable to be incremented.

Return Value

The result of the increment.

CComSingleThreadModel::ThreadModelNoCS

When using `CComSingleThreadModel`, the **typedef** name `ThreadModelNoCS` simply references `CComSingleThreadModel`.

```
typedef CComSingleThreadModel ThreadModelNoCS;
```

Remarks

`CComMultiThreadModel` and `CComMultiThreadModelNoCS` contain definitions for `ThreadModelNoCS`. The following table shows the relationship between the threading model class and the class referenced by `ThreadModelNoCS`:

CLASS DEFINED IN	CLASS REFERENCED
<code>CComSingleThreadModel</code>	<code>CComSingleThreadModel</code>
<code>CComMultiThreadModel</code>	<code>CComMultiThreadModelNoCS</code>
<code>CComMultiThreadModelNoCS</code>	<code>CComMultiThreadModelNoCS</code>

Example

See [CComMultiThreadModel::AutoCriticalSection](#).

See also

[Class Overview](#)

CComTearOffObject Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

This class implements a tear-off interface.

Syntax

```
template<class Base>
class CComTearOffObject : public Base
```

Parameters

Base

Your tear-off class, derived from `CComTearOffObjectBase` and the interfaces you want your tear-off object to support.

ATL implements its tear-off interfaces in two phases — the `CComTearOffObjectBase` methods handle the reference count and `QueryInterface`, while `CComTearOffObject` implements [IUnknown](#).

Members

Public Constructors

NAME	DESCRIPTION
CComTearOffObject::CComTearOffObject	The constructor.
CComTearOffObject::~~CComTearOffObject	The destructor.

Public Methods

NAME	DESCRIPTION
CComTearOffObject::AddRef	Increments the reference count for a <code>CComTearOffObject</code> object.
CComTearOffObject::QueryInterface	Returns a pointer to the requested interface on either your tear-off class or the owner class.
CComTearOffObject::Release	Decrements the reference count for a <code>CComTearOffObject</code> object and destroys it.

CComTearOffObjectBase Methods

CComTearOffObjectBase	Constructor.

CComTearOffObjectBase Data Members

m_pOwner	A pointer to a <code>CComObject</code> derived from the owner class.

Remarks

`CComTearOffObject` implements a tear-off interface as a separate object that is instantiated only when that interface is queried for. The tear-off is deleted when its reference count becomes zero. Typically, you build a tear-off interface for an interface that is rarely used, since using a tear-off saves a vtable pointer in all the instances of your main object.

You should derive the class implementing the tear-off from `CComTearOffObjectBase` and from whichever interfaces you want your tear-off object to support. `CComTearOffObjectBase` is templated on the owner class and the thread model. The owner class is the class of the object for which a tear-off is being implemented. If you do not specify a thread model, the default thread model is used.

You should create a COM map for your tear-off class. When ATL instantiates the tear-off, it will create

`CComTearOffObject<CYourTearOffClass>` or `CComCachedTearOffObject<CYourTearOffClass>`.

For example, in the BEEPER sample, the `CBeeper2` class is the tear-off class and the `CBeeper` class is the owner class:


```

class CBeeper2 :
    public ISupportErrorInfo,
    public CComTearOffObjectBase<CBeeper>
{
public:
    CBeeper2() {}
    STDMETHOD(InterfaceSupportsErrorInfo)(REFIID riid)
    {
        return (InlineIsEqualGUID(IID_IBeeper, riid)) ? S_OK : S_FALSE;
    }

BEGIN_COM_MAP(CBeeper2)
    COM_INTERFACE_ENTRY(ISupportErrorInfo)
END_COM_MAP()
};

class ATL_NO_VTABLE CBeeper :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CBeeper, &CLSID_Beeper>,
    public IDispatchImpl<IBeeper, &IID_IBeeper, &LIBID_NVC_ATL_COMLib, /*wMajor =*/ 1, /*wMinor =*/ 0>
{
public:
    CBeeper()
    {
    }

DECLARE_REGISTRY_RESOURCEID(IDR_BEEPER)

DECLARE_NOT_AGGREGATABLE(CBeeper)

BEGIN_COM_MAP(CBeeper)
    COM_INTERFACE_ENTRY(IBeeper)
    COM_INTERFACE_ENTRY(IDispatch)
    COM_INTERFACE_ENTRY_TEAR_OFF(IID_ISupportErrorInfo, CBeeper2)
END_COM_MAP()

// ISupportsErrorInfo
    STDMETHOD(InterfaceSupportsErrorInfo)(REFIID riid);

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    HRESULT FinalConstruct()
    {
        return S_OK;
    }

    void FinalRelease()
    {
    }

public:
};

```

Inheritance Hierarchy

Base

CComTearOffObject

Requirements

Header: atlcom.h

CComTearOffObject::AddRef

Increments the reference count of the `CComTearOffObject` object by one.

```
STDMETHOD_(ULONG, AddRef)();
```

Return Value

A value that may be useful for diagnostics and testing.

CComTearOffObject::CComTearOffObject

The constructor.

```
CComTearOffObject(void* pv);
```

Parameters

pv

[in] Pointer that will be converted to a pointer to a `CComObject<Owner>` object.

Remarks

Increments the owner's reference count by one.

CComTearOffObject::~~CComTearOffObject

The destructor.

```
~CComTearOffObject();
```

Remarks

Frees all allocated resources, calls `FinalRelease`, and decrements the module lock count.

CComTearOffObject::CComTearOffObjectBase

The constructor.

```
CComTearOffObjectBase();
```

Remarks

Initializes the `m_pOwner` member to `NULL`.

CComTearOffObject::m_pOwner

A pointer to a `CComObject` object derived from *Owner*.

```
CComObject<Owner>* m_pOwner;
```

Parameters

Owner

[in] The class for which a tear-off is being implemented.

Remarks

The pointer is initialized to NULL during construction.

CComTearOffObject::QueryInterface

Retrieves a pointer to the requested interface.

```
STDMETHOD(QueryInterface)(REFIID iid, void** ppvObject);
```

Parameters

iid

[in] The IID of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by *iid*, or NULL if the interface is not found.

Return Value

A standard HRESULT value.

Remarks

Queries first for interfaces on your tear-off class. If the interface is not there, queries for the interface on the owner object. If the requested interface is `IUnknown`, returns the `IUnknown` of the owner.

CComTearOffObject::Release

Decrements the reference count by one and, if the reference count is zero, deletes the `CComTearOffObject`.

```
STDMETHOD_ULONG Release();
```

Return Value

In non-debug builds, always returns zero. In debug builds, returns a value that may be useful for diagnostics or testing.

See also

[CComCachedTearOffObject Class](#)

[Class Overview](#)

CComUnkArray Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

This class stores `IUnknown` pointers, and is designed to be used as a parameter to the `IConnectionPointImpl` template class.

Syntax

```
template<unsigned int nMaxSize>
class CComUnkArray
```

Parameters

nMaxSize

The maximum number of `IUnknown` pointers that can be held in the static array.

Members

Public Constructors

NAME	DESCRIPTION
CComUnkArray::CComUnkArray	Constructor.

Public Methods

NAME	DESCRIPTION
CComUnkArray::Add	Call this method to add an <code>IUnknown</code> pointer to the array.
CComUnkArray::begin	Returns a pointer to the first <code>IUnknown</code> pointer in the collection.
CComUnkArray::end	Returns a pointer to one past the last <code>IUnknown</code> pointer in the collection.
CComUnkArray::GetCookie	Call this method to get the cookie associated with a given <code>IUnknown</code> pointer.
CComUnkArray::GetUnknown	Call this method to get the <code>IUnknown</code> pointer associated with a given cookie.
CComUnkArray::Remove	Call this method to remove an <code>IUnknown</code> pointer from the array.

Remarks

`CComUnkArray` holds a fixed number of `IUnknown` pointers, each an interface on a connection point. `CComUnkArray` can be used as a parameter to the `IConnectionPointImpl` template class. `CComUnkArray<1>` is a template specialization of `CComUnkArray` that has been optimized for one connection point.

The `CComUnkArray` methods [begin](#) and [end](#) can be used to loop through all connection points (for example, when an event is fired).

See [Adding Connection Points to an Object](#) for details on automating creation of connection point proxies.

NOTE

Note The class `CComDynamicUnkArray` is used by the **Add Class** wizard when creating a control which has Connection Points. If you wish to specify the number of Connection Points manually, change the reference from

`CComDynamicUnkArray` to `CComUnkArray< n >`, where n is the number of connection points required.

Requirements

Header: `atlcom.h`

CComUnkArray::Add

Call this method to add an `IUnknown` pointer to the array.

```
DWORD Add(IUnknown* pUnk);
```

Parameters

pUnk

Call this method to add an `IUnknown` pointer to the array.

Return Value

Returns the cookie associated with the newly added pointer, or 0 if the array is not large enough to contain the new pointer.

CComUnkArray::begin

Returns a pointer to the beginning of the collection of `IUnknown` interface pointers.

```
IUnknown**  
begin();
```

Return Value

A pointer to an `IUnknown` interface pointer.

Remarks

The collection contains pointers to interfaces stored locally as `IUnknown`. You cast each `IUnknown` interface to the real interface type and then call through it. You do not need to query for the interface first.

Before using the `IUnknown` interface, you should check that it is not NULL.

CComUnkArray::CComUnkArray

The constructor.

```
CComUnkArray();
```

Remarks

Sets the collection to hold `nMaxSize` `IUnknown` pointers, and initializes the pointers to NULL.

CComUnkArray::end

Returns a pointer to one past the last `IUnknown` pointer in the collection.

```
IUnknown**  
    end();
```

Return Value

A pointer to an `IUnknown` interface pointer.

Remarks

The `CComUnkArray` methods `begin` and `end` can be used to loop through all connection points, for example, when an event is fired.

```
IUnknown** p = m_vec.begin();  
while(p != m_vec.end())  
{  
    // Do something with *p  
    p++;  
}
```

CComUnkArray::GetCookie

Call this method to get the cookie associated with a given `IUnknown` pointer.

```
DWORD WINAPI GetCookie(IUnknown** ppFind);
```

Parameters

ppFind

The `IUnknown` pointer for which the associated cookie is required.

Return Value

Returns the cookie associated with the `IUnknown` pointer, or 0 if no matching `IUnknown` pointer is found.

Remarks

If there is more than one instance of the same `IUnknown` pointer, this function returns the cookie for the first one.

CComUnkArray::GetUnknown

Call this method to get the `IUnknown` pointer associated with a given cookie.

```
IUnknown* WINAPI GetUnknown(DWORD dwCookie);
```

Parameters

dwCookie

The cookie for which the associated `IUnknown` pointer is required.

Return Value

Returns the `IUnknown` pointer, or NULL if no matching cookie is found.

CComUnkArray::Remove

Call this method to remove an `IUnknown` pointer from the array.

```
BOOL Remove(DWORD dwCookie);
```

Parameters

dwCookie

The cookie referencing the `IUnknown` pointer to be removed from the array.

Return Value

Returns TRUE if the pointer is removed, FALSE otherwise.

See also

[CComDynamicUnkArray Class](#)

[Class Overview](#)

CComVariant Class

3/4/2019 • 10 minutes to read • [Edit Online](#)

This class wraps the VARIANT type, providing a member indicating the type of data stored.

Syntax

```
class CComVariant : public tagVARIANT
```

Members

Public Constructors

NAME	DESCRIPTION
CComVariant::CComVariant	The constructor.
CComVariant::~~CComVariant	The destructor.

Public Methods

NAME	DESCRIPTION
CComVariant::Attach	Attaches a VARIANT to the <code>CComVariant</code> object.
CComVariant::ChangeType	Converts the <code>CComVariant</code> object to a new type.
CComVariant::Clear	Clears the <code>CComVariant</code> object.
CComVariant::Copy	Copies a VARIANT to the <code>CComVariant</code> object.
CComVariant::CopyTo	Copies the contents of the <code>CComVariant</code> object.
CComVariant::Detach	Detaches the underlying VARIANT from the <code>CComVariant</code> object.
CComVariant::GetSize	Returns the size in number of bytes of the contents of the <code>CComVariant</code> object.
CComVariant::ReadFromStream	Loads a VARIANT from a stream.
CComVariant::SetByRef	Initializes the <code>CComVariant</code> object and sets the <code>vt</code> member to VT_BYREF.
CComVariant::WriteToStream	Saves the underlying VARIANT to a stream.

Public Operators

<code>CComVariant::operator <</code>	Indicates whether the <code>CComVariant</code> object is less than the specified VARIANT.
<code>CComVariant::operator ></code>	Indicates whether the <code>CComVariant</code> object is greater than the specified VARIANT.
<code>operator !=</code>	Indicates whether the <code>CComVariant</code> object does not equal the specified VARIANT.
<code>operator =</code>	Assigns a value to the <code>CComVariant</code> object.
<code>operator ==</code>	Indicates whether the <code>CComVariant</code> object equals the specified VARIANT.

Remarks

`CComVariant` wraps the VARIANT and VARIANTARG type, which consists of a union and a member indicating the type of the data stored in the union. VARIANTs are typically used in Automation.

`CComVariant` derives from the VARIANT type so it can be used wherever a VARIANT can be used. You can, for example, use the V_VT macro to extract the type of a `CComVariant` or you can access the `vt` member directly just as you can with a VARIANT.

Inheritance Hierarchy

`tagVARIANT`

`CComVariant`

Requirements

Header: atlcomcli.h

CComVariant::Attach

Safely clears the current contents of the `CComVariant` object, copies the contents of *pSrc* into this object, then sets the variant type of *pSrc* to VT_EMPTY.

```
HRESULT Attach(VARIANT* pSrc);
```

Parameters

pSrc

[in] Points to the [VARIANT](#) to be attached to the object.

Return Value

A standard HRESULT value.

Remarks

Ownership of the data held by *pSrc* is transferred to the `CComVariant` object.

CComVariant::CComVariant

Each constructor handles the safe initialization of the `CComVariant` object by calling the `VariantInit` Win32 function or by setting the object's value and type according to the parameters passed.

```
CComVariant() throw();
CComVariant(const CComVariant& varSrc);
CComVariant(const VARIANT& varSrc);
CComVariant(LPCOLESTR lpszSrc);
CComVariant(LPCSTR lpszSrc);
CComVariant(bool bSrc);
CComVariant(BYTE nSrc) throw();
CComVariant(int nSrc, VARTYPE vtSrc = VT_I4) throw();
CComVariant(unsigned int nSrc, VARTYPE vtSrc = VT_UI4) throw();
CComVariant(short nSrc) throw();
CComVariant(unsigned short nSrc) throw();
CComVariant(long nSrc, VARTYPE vtSrc = VT_I4) throw();
CComVariant(unsigned long nSrc) throw();
CComVariant(LONGLONG nSrc) throw();
CComVariant(ULONGLONG nSrc) throw();
CComVariant(float fltSrc) throw();
CComVariant(double dblSrc, VARTYPE vtSrc = VT_R8) throw();
CComVariant(CY cySrc) throw();
CComVariant(IDispatch* pSrc) throw();
CComVariant(IUnknown* pSrc) throw();
CComVariant(const SAFEARRAY* pSrc);
CComVariant(char cSrc) throw();
CComVariant(const CComBSTR& bstrSrc);
```

Parameters

varSrc

[in] The `CComVariant` or `VARIANT` used to initialize the `CComVariant` object. The contents of the source variant are copied to the destination without conversion.

lpszSrc

[in] The character string used to initialize the `CComVariant` object. You can pass a zero-terminated wide (Unicode) character string to the `LPCOLESTR` version of the constructor or an ANSI string to the `LPCSTR` version. In either case the string is converted to a Unicode `BSTR` allocated using `SysAllocString`. The type of the `CComVariant` object will be `VT_BSTR`.

bSrc

[in] The **bool** used to initialize the `CComVariant` object. The **bool** argument is converted to a `VARIANT_BOOL` before being stored. The type of the `CComVariant` object will be `VT_BOOL`.

nSrc

[in] The **int**, **BYTE**, **short**, **long**, `LONGLONG`, `ULONGLONG`, **unsigned short**, **unsigned long**, or **unsigned int** used to initialize the `CComVariant` object. The type of the `CComVariant` object will be `VT_I4`, `VT_UI1`, `VT_I2`, `VT_I4`, `VT_I8`, `VT_UI8`, `VT_UI2`, `VT_UI4`, or `VT_UI4`, respectively.

vtSrc

[in] The type of the variant. When the first parameter is **int**, valid types are `VT_I4` and `VT_INT`. When the first parameter is **long**, valid types are `VT_I4` and `VT_ERROR`. When the first parameter is **double**, valid types are `VT_R8` and `VT_DATE`. When the first parameter is **unsigned int**, valid types are `VT_UI4` and `VT_UINT`.

fltSrc

[in] The **float** used to initialize the `CComVariant` object. The type of the `CComVariant` object will be `VT_R4`.

dblSrc

[in] The **double** used to initialize the `CComVariant` object. The type of the `CComVariant` object will be `VT_R8`.

cySrc

[in] The `cy` used to initialize the `CComVariant` object. The type of the `CComVariant` object will be VT_CY.

pSrc

[in] The `IDispatch` or `IUnknown` pointer used to initialize the `CComVariant` object. `AddRef` will be called on the interface pointer. The type of the `CComVariant` object will be VT_DISPATCH or VT_UNKNOWN, respectively.

Or, the SAFERARRAY pointer used to initialize the `CComVariant` object. A copy of the SAFEARRAY is stored in the `CComVariant` object. The type of the `CComVariant` object will be a combination of the original type of the SAFEARRAY and VT_ARRAY.

cSrc

[in] The **char** used to initialize the `CComVariant` object. The type of the `CComVariant` object will be VT_I1.

bstrSrc

[in] The BSTR used to initialize the `CComVariant` object. The type of the `CComVariant` object will be VT_BSTR.

Remarks

The destructor manages cleanup by calling [CComVariant::Clear](#).

CComVariant::~CComVariant

The destructor.

```
~CComVariant() throw();
```

Remarks

This method manages cleanup by calling [CComVariant::Clear](#).

CComVariant::ChangeType

Converts the `CComVariant` object to a new type.

```
HRESULT ChangeType(VARTYPE vtNew, const VARIANT* pSrc = NULL);
```

Parameters

vtNew

[in] The new type for the `CComVariant` object.

pSrc

[in] A pointer to the VARIANT whose value will be converted to the new type. The default value is NULL, meaning the `CComVariant` object will be converted in place.

Return Value

A standard HRESULT value.

Remarks

If you pass a value for *pSrc*, `ChangeType` will use this VARIANT as the source for the conversion. Otherwise, the `CComVariant` object will be the source.

CComVariant::Clear

Clears the `CComVariant` object by calling the `VariantClear` Win32 function.

```
HRESULT Clear();
```

Return Value

A standard HRESULT value.

Remarks

The destructor automatically calls `Clear`.

CComVariant::Copy

Frees the `CComVariant` object and then assigns it a copy of the specified VARIANT.

```
HRESULT Copy(const VARIANT* pSrc);
```

Parameters

pSrc

[in] A pointer to the [VARIANT](#) to be copied.

Return Value

A standard HRESULT value.

CComVariant::CopyTo

Copies the contents of the `CComVariant` object.

```
HRESULT CopyTo(BSTR* pstrDest);
```

Parameters

pstrDest

Points to a BSTR that will receive a copy of the contents of the `CComVariant` object.

Return Value

A standard HRESULT value.

Remarks

The `CComVariant` object must be of type VT_BSTR.

CComVariant::Detach

Detaches the underlying VARIANT from the `CComVariant` object and sets the object's type to VT_EMPTY.

```
HRESULT Detach(VARIANT* pDest);
```

Parameters

pDest

[out] Returns the underlying VARIANT value of the object.

Return Value

A standard HRESULT value.

Remarks

Note that the contents of the VARIANT referenced by *pDest* will automatically be cleared before being assigned the value and type of the calling `CComVariant` object.

CComVariant::GetSize

For simple-fixed size VARIANTs, this method returns the **sizeof** the underlying data type plus **sizeof(VARTYPE)**.

```
ULONG GetSize() const;
```

Return Value

The size in bytes of the current contents of the `CComVariant` object.

Remarks

If the VARIANT contains an interface pointer, `GetSize` queries for `IPersistStream` or `IPersistStreamInit`. If successful, the return value is the low-order 32 bits of the value returned by `GetSizeMax` plus the **sizeof** a CLSID and **sizeof(VARTYPE)**. If the interface pointer is NULL, `GetSize` returns the **sizeof** a CLSID plus **sizeof(VARTYPE)**. If the total size is larger than `ULONG_MAX`, `GetSize` returns **sizeof(VARTYPE)** which indicates an error.

In all other cases, a temporary VARIANT of type `VT_BSTR` is coerced from the current VARIANT. The length of this BSTR is calculated as the size of the length of the string plus the length of the string itself plus the size of the null character plus **sizeof(VARTYPE)**. If the VARIANT cannot be coerced to a VARIANT of type `VT_BSTR`, `GetSize` returns **sizeof(VARTYPE)**.

The size returned by this method matches the number of bytes used by `CComVariant::WriteToStream` under successful conditions.

CComVariant::operator =

Assigns a value and corresponding type to the `CComVariant` object.

```
CComVariant& operator=(const CComVariant& varSrc);
CComVariant& operator=(const VARIANT& varSrc);
CComVariant& operator=(const CComBSTR& bstrSrc);
CComVariant& operator=(LPCOLESTR lpszSrc);
CComVariant& operator=(LPCSTR lpszSrc);
CComVariant& operator=(bool bSrc);
CComVariant& operator=(BYTE nSrc) throw();
CComVariant& operator=(int nSrc) throw();
CComVariant& operator=(unsigned int nSrc) throw();
CComVariant& operator=(short nSrc) throw();
CComVariant& operator=(unsigned short nSrc) throw();
CComVariant& operator=(long nSrc) throw();
CComVariant& operator=(unsigned long nSrc) throw();
CComVariant& operator=(LONGLONG nSrc) throw();
CComVariant& operator=(ULONGLONG nSrc) throw();
CComVariant& operator=(float fltSrc) throw();
CComVariant& operator=(double dblSrc) throw();
CComVariant& operator=(CY cySrc) throw();
CComVariant& operator=(IDispatch* pSrc) throw();
CComVariant& operator=(IUnknown* pSrc) throw();
CComVariant& operator=(const SAFEARRAY* pSrc);
CComVariant& operator=(char cSrc) throw();
```

Parameters

varSrc

[in] The `CComVariant` or `VARIANT` to be assigned to the `CComVariant` object. The contents of the source variant

are copied to the destination without conversion.

bstrSrc

[in] The BSTR to be assigned to the `CComVariant` object. The type of the `CComVariant` object will be VT_BSTR.

lpzSrc

[in] The character string to be assigned to the `CComVariant` object. You can pass a zero-terminated wide (Unicode) character string to the LPCOLESTR version of the operator or an ANSI string to the LPCSTR version. In either case, the string is converted to a Unicode BSTR allocated using `SysAllocString`. The type of the `CComVariant` object will be VT_BSTR.

bSrc

[in] The **bool** to be assigned to the `CComVariant` object. The **bool** argument is converted to a VARIANT_BOOL before being stored. The type of the `CComVariant` object will be VT_BOOL.

nSrc

[in] The **int**, **BYTE**, **short**, **long**, **LONGLONG**, **ULONGLONG**, **unsigned short**, **unsigned long**, or **unsigned int** to be assigned to the `CComVariant` object. The type of the `CComVariant` object will be VT_I4, VT_UI1, VT_I2, VT_I4, VT_I8, VT_UI8, VT_UI2, VT_UI4, or VT_UI4, respectively.

fltSrc

[in] The **float** to be assigned to the `CComVariant` object. The type of the `CComVariant` object will be VT_R4.

dblSrc

[in] The **double** to be assigned to the `CComVariant` object. The type of the `CComVariant` object will be VT_R8.

cySrc

[in] The **CY** to be assigned to the `CComVariant` object. The type of the `CComVariant` object will be VT_CY.

pSrc

[in] The `IDispatch` or `IUnknown` pointer to be assigned to the `CComVariant` object. `AddRef` will be called on the interface pointer. The type of the `CComVariant` object will be VT_DISPATCH or VT_UNKNOWN, respectively.

Or, a SAFEARRAY pointer to be assigned to the `CComVariant` object. A copy of the SAFEARRAY is stored in the `CComVariant` object. The type of the `CComVariant` object will be a combination of the original type of the SAFEARRAY and VT_ARRAY.

cSrc

[in] The char to be assigned to the `CComVariant` object. The type of the `CComVariant` object will be VT_I1.

CComVariant::operator ==

Indicates whether the `CComVariant` object equals the specified VARIANT.

```
bool operator==(const VARIANT& varSrc) const throw();
```

Remarks

Returns TRUE if the value and type of *varSrc* are equal to the value and type, respectively, of the `CComVariant` object. Otherwise, FALSE. The operator uses the user's default locale to perform the comparison.

The operator compares only the value of the variant types. It compares strings, integers, and floating points, but not arrays or records.

CComVariant::operator !=

Indicates whether the `CComVariant` object does not equal the specified VARIANT.

```
bool operator!=(const VARIANT& varSrc) const throw();
```

Remarks

Returns TRUE if either the value or type of *varSrc* is not equal to the value or type, respectively, of the `CComVariant` object. Otherwise, FALSE. The operator uses the user's default locale to perform the comparison.

The operator compares only the value of the variant types. It compares strings, integers, and floating points, but not arrays or records.

CComVariant::operator <

Indicates whether the `CComVariant` object is less than the specified VARIANT.

```
bool operator<(const VARIANT& varSrc) const throw();
```

Remarks

Returns TRUE if the value of the `CComVariant` object is less than the value of *varSrc*. Otherwise, FALSE. The operator uses the user's default locale to perform the comparison.

CComVariant::operator >

Indicates whether the `CComVariant` object is greater than the specified VARIANT.

```
bool operator>(const VARIANT& varSrc) const throw();
```

Remarks

Returns TRUE if the value of the `CComVariant` object is greater than the value of *varSrc*. Otherwise, FALSE. The operator uses the user's default locale to perform the comparison.

CComVariant::ReadFromStream

Sets the underlying VARIANT to the VARIANT contained in the specified stream.

```
HRESULT ReadFromStream(IStream* pStream);
```

Parameters

pStream

[in] A pointer to the [IStream](#) interface on the stream containing the data.

Return Value

A standard HRESULT value.

Remarks

`ReadToStream` requires a previous call to [WriteToStream](#).

CComVariant::SetByRef

Initializes the `CComVariant` object and sets the `vt` member to VT_BYREF.

```
template < typename T >
void SetByRef(T* pT) throw();
```

Parameters

T

The type of VARIANT, for example, BSTR, **int**, or **char**.

pT

The pointer used to initialize the `CComVariant` object.

Remarks

`SetByRef` is a function template that initializes the `CComVariant` object to the pointer *pT* and sets the `vt` member to VT_BYREF. For example:

```
CComVariant var;
int nData = 10;
var.SetByRef(&nData);
```

CComVariant::WriteToStream

Saves the underlying VARIANT to a stream.

```
HRESULT WriteToStream(IStream* pStream);
```

Parameters

pStream

[in] A pointer to the [IStream](#) interface on a stream.

Return Value

A standard HRESULT value.

See also

[Class Overview](#)

CContainedWindowT Class

3/5/2019 • 9 minutes to read • [Edit Online](#)

This class implements a window contained within another object.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class TBase = CWindow, class TWinTraits = CControlWinTraits>
class CContainedWindowT : public TBase
```

Parameters

TBase

The base class of your new class. The default base class is `CWindow`.

TWinTraits

A traits class that defines styles for your window. The default is `CControlWinTraits`.

NOTE

`CContainedWindow` is a specialization of `CContainedWindowT`. If you want to change the base class or traits, use `CContainedWindowT` directly.

Members

Public Constructors

NAME	DESCRIPTION
<code>CContainedWindowT::CContainedWindowT</code>	Constructor. Initializes data members to specify which message map will process the contained window's messages.

Public Methods

NAME	DESCRIPTION
<code>CContainedWindowT::Create</code>	Creates a window.
<code>CContainedWindowT::DefWindowProc</code>	Provides default message processing.
<code>CContainedWindowT::GetCurrentMessage</code>	Returns the current message.
<code>CContainedWindowT::RegisterWndSuperclass</code>	Registers the window class of the contained window.
<code>CContainedWindowT::SubclassWindow</code>	Subclasses a window.

NAME	DESCRIPTION
CContainedWindowT::SwitchMessageMap	Changes which message map is used to process the contained window's messages.
CContainedWindowT::UnsubclassWindow	Restores a previously subclassed window.
CContainedWindowT::WindowProc	(Static) Processes messages sent to the contained window.

Public Data Members

NAME	DESCRIPTION
CContainedWindowT::m_dwMsgMapID	Identifies which message map will process the contained window's messages.
CContainedWindowT::m_lpszClassName	Specifies the name of an existing window class on which a new window class will be based.
CContainedWindowT::m_pfnSuperWindowProc	Points to the window class's original window procedure.
CContainedWindowT::m_pObject	Points to the containing object.

Remarks

[CContainedWindowT](#) implements a window contained within another object. [CContainedWindowT](#)'s window procedure uses a message map in the containing object to direct messages to the appropriate handlers. When constructing a [CContainedWindowT](#) object, you specify which message map should be used.

[CContainedWindowT](#) allows you to create a new window by superclassing an existing window class. The [Create](#) method first registers a window class that is based on an existing class but uses [CContainedWindowT::WindowProc](#). [Create](#) then creates a window based on this new window class. Each instance of [CContainedWindowT](#) can superclass a different window class.

[CContainedWindowT](#) also supports window subclassing. The [SubclassWindow](#) method attaches an existing window to the [CContainedWindowT](#) object and changes the window procedure to [CContainedWindowT::WindowProc](#). Each instance of [CContainedWindowT](#) can subclass a different window.

NOTE

For any given [CContainedWindowT](#) object, call either [Create](#) or [SubclassWindow](#). You should not invoke both methods on the same object.

When you use the **Add control based on** option in the ATL Project Wizard, the wizard will automatically add a [CContainedWindowT](#) data member to the class implementing the control. The following example shows how the contained window is declared:

```

public:
    // Declare a contained window data member
    CContainedWindow m_ctlEdit;

    // Initialize the contained window:
    // 1. Pass "Edit" to specify that the contained
    //    window should be based on the standard
    //    Windows Edit box
    // 2. Pass 'this' pointer to specify that CAtlEdit
    //    contains the message map to be used for the
    //    contained window's message processing
    // 3. Pass the identifier of the message map. '1'
    //    identifies the alternate message map declared
    //    with ALT_MSG_MAP(1)
    CAtlEdit()
        : m_ctlEdit(_T("Edit"), this, 1)
    {
        m_bWindowOnly = TRUE;
    }

```

```

// Declare the default message map, identified by '0'
BEGIN_MSG_MAP(CAtlEdit)
    MESSAGE_HANDLER(WM_CREATE, OnCreate)
    MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
    CHAIN_MSG_MAP(CComControl<CAtlEdit>)
// Declare an alternate message map, identified by '1'
ALT_MSG_MAP(1)
    MESSAGE_HANDLER(WM_CHAR, OnChar)
END_MSG_MAP()

```

```

// Define OnCreate handler
// When the containing window receives a WM_CREATE
// message, create the contained window by calling
// CContainedWindow::Create
LRESULT OnCreate(UINT /*uMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
    BOOL& /*bHandled*/)
{
    RECT rc;
    GetWindowRect(&rc);
    rc.right -= rc.left;
    rc.bottom -= rc.top;
    rc.top = rc.left = 0;
    m_ctlEdit.Create(m_hWnd, rc, _T("hello"), WS_CHILD | WS_VISIBLE |
        ES_MULTILINE | ES_AUTOVSCROLL);
    return 0;
}

```

FOR MORE INFORMATION ABOUT	SEE
Creating controls	ATL Tutorial
Using windows in ATL	ATL Window Classes
ATL Project Wizard	Creating an ATL Project
Windows	Windows and subsequent topics in the Windows SDK

Inheritance Hierarchy

TBase

CContainedWindowT

Requirements

Header: atlwin.h

CContainedWindowT::CContainedWindowT

The constructor initializes data members.

```
CContainedWindowT(  
    LPTSTR lpszClassName,  
    CMessageMap* pObject,  
    DWORD dwMsgMapID = 0);  
  
CContainedWindowT(  
    CMessageMap* pObject,  
    DWORD dwMsgMapID = 0)  
    CContainedWindowT();
```

Parameters

lpszClassName

[in] The name of an existing window class on which the contained window will be based.

pObject

[in] A pointer to the containing object that declares the message map. This object's class must derive from [CMessageMap](#).

dwMsgMapID

[in] Identifies the message map that will process the contained window's messages. The default value, 0, specifies the default message map declared with [BEGIN_MSG_MAP](#). To use an alternate message map declared with [ALT_MSG_MAP\(msgMapID\)](#), pass `msgMapID`.

Remarks

If you want to create a new window through [Create](#), you must pass the name of an existing window class for the *lpszClassName* parameter. For an example, see the [CContainedWindow](#) overview.

There are three constructors:

- The constructor with three arguments is the one typically called.
- The constructor with two arguments uses the class name from `TBase::GetWndClassName`.
- The constructor with no arguments is used if you want to supply the arguments later. You must supply the window class name, message map object, and message map ID when you later call `Create`.

If you subclass an existing window through [SubclassWindow](#), the *lpszClassName* value will not be used; therefore, you can pass NULL for this parameter.

CContainedWindowT::Create

Calls [RegisterWndSuperclass](#) to register a window class that is based on an existing class but uses [CContainedWindowT::WindowProc](#).

```

HWND Create(
    HWND hWndParent,
    _U_RECT rect,
    LPCTSTR szWindowName = NULL,
    DWORD dwStyle = 0,
    DWORD dwExStyle = 0,
    _U_MENUorID MenuOrID = 0U,
    LPVOID lpCreateParam = NULL);

HWND Create(
    CMessageMap* pObject,
    DWORD dwMsgMapID,
    HWND hWndParent,
    _U_RECT rect,
    LPCTSTR szWindowName = NULL,
    DWORD dwStyle = 0,
    DWORD dwExStyle = 0,
    _U_MENUorID MenuOrID = 0U,
    LPVOID lpCreateParam = NULL);

HWND Create(
    LPCTSTR lpszClassName,
    CMessageMap* pObject,
    DWORD dwMsgMapID,
    HWND hWndParent,
    _U_RECT rect,
    LPCTSTR szWindowName = NULL,
    DWORD dwStyle = 0,
    DWORD dwExStyle = 0,
    _U_MENUorID MenuOrID = 0U,
    LPVOID lpCreateParam = NULL);

```

Parameters

lpszClassName

[in] The name of an existing window class on which the contained window will be based.

pObject

[in] A pointer to the containing object that declares the message map. This object's class must derive from [CMessageMap](#).

dwMsgMapID

[in] Identifies the message map that will process the contained window's messages. The default value, 0, specifies the default message map declared with [BEGIN_MSG_MAP](#). To use an alternate message map declared with [ALT_MSG_MAP\(msgMapID\)](#), pass `msgMapID`.

hWndParent

[in] The handle to the parent or owner window.

rect

[in] A [RECT](#) structure specifying the position of the window. The `RECT` can be passed by pointer or by reference.

szWindowName

[in] Specifies the name of the window. The default value is NULL.

dwStyle

[in] The style of the window. The default value is `WS_CHILD | WS_VISIBLE`. For a list of possible values, see [CreateWindow](#) in the Windows SDK.

dwExStyle

[in] The extended window style. The default value is 0, meaning no extended style. For a list of possible values, see [CreateWindowEx](#) in the Windows SDK.

MenuOrID

[in] For a child window, the window identifier. For a top-level window, a menu handle for the window. The default value is **0U**.

lpCreateParam

[in] A pointer to window-creation data. For a full description, see the description for the final parameter to [CreateWindowEx](#).

Return Value

If successful, the handle to the newly created window; otherwise, NULL.

Remarks

The existing window class name is saved in [m_lpszClassName](#). `Create` then creates a window based on this new class. The newly created window is automatically attached to the `CContainedWindowT` object.

NOTE

Do not call `Create` if you have already called [SubclassWindow](#).

NOTE

If 0 is used as the value for the *MenuOrID* parameter, it must be specified as 0U (the default value) to avoid a compiler error.

CContainedWindowT::DefWindowProc

Called by [WindowProc](#) to process messages not handled by the message map.

```
LRESULT DefWindowProc()  
LRESULT DefWindowProc(  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam);
```

Parameters

uMsg

[in] The message sent to the window.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

Return Value

The result of the message processing.

Remarks

By default, `DefWindowProc` calls the [CallWindowProc](#) Win32 function to send the message information to the window procedure specified in [m_pfnSuperWindowProc](#).

CContainedWindowT::GetCurrentMessage

Returns the current message (`m_pCurrentMsg`).

```
const _ATL_MSG* GetCurrentMessage();
```

Return Value

The current message, packaged in the `MSG` structure.

CContainedWindowT::m_dwMsgMapID

Holds the identifier of the message map currently being used for the contained window.

```
DWORD m_dwMsgMapID;
```

Remarks

This message map must be declared in the containing object.

The default message map, declared with [BEGIN_MSG_MAP](#), is always identified by zero. An alternate message map, declared with [ALT_MSG_MAP\(msgMapID\)](#), is identified by `msgMapID`.

`m_dwMsgMapID` is first initialized by the constructor and can be changed by calling [SwitchMessageMap](#). For an example, see the [CContainedWindowT Overview](#).

CContainedWindowT::m_lpszClassName

Specifies the name of an existing window class.

```
LPTSTR m_lpszClassName;
```

Remarks

When you create a window, [Create](#) registers a new window class that is based on this existing class but uses [CContainedWindowT::WindowProc](#).

`m_lpszClassName` is initialized by the constructor. For an example, see the [CContainedWindowT](#) overview.

CContainedWindowT::m_pfnSuperWindowProc

If the contained window is subclassed, `m_pfnSuperWindowProc` points to the original window procedure of the window class.

```
WNDPROC m_pfnSuperWindowProc;
```

Remarks

If the contained window is superclassed, meaning it is based on a window class that modifies an existing class, `m_pfnSuperWindowProc` points to the existing window class's window procedure.

The [DefWindowProc](#) method sends message information to the window procedure saved in `m_pfnSuperWindowProc`.

CContainedWindowT::m_pObject

Points to the object containing the `CContainedWindowT` object.

```
CMessageMap* m_pObject;
```

Remarks

This container, whose class must derive from [CMessageMap](#), declares the message map used by the contained window.

`m_pObject` is initialized by the constructor. For an example, see the [CContainedWindowT](#) overview.

CContainedWindowT::RegisterWndSuperclass

Called by [Create](#) to register the window class of the contained window.

```
ATOM RegisterWndSuperClass();
```

Return Value

If successful, an atom that uniquely identifies the window class being registered; otherwise, zero.

Remarks

This window class is based on an existing class but uses [CContainedWindowT::WindowProc](#). The existing window class's name and window procedure are saved in [m_lpszClassName](#) and [m_pfnSuperWindowProc](#), respectively.

CContainedWindowT::SubclassWindow

Subclasses the window identified by *hWnd* and attaches it to the `CContainedWindowT` object.

```
BOOL SubclassWindow(HWND hWnd);
```

Parameters

hWnd

[in] The handle to the window being subclassed.

Return Value

TRUE if the window is successfully subclassed; otherwise, FALSE.

Remarks

The subclassed window now uses [CContainedWindowT::WindowProc](#). The original window procedure is saved in [m_pfnSuperWindowProc](#).

NOTE

Do not call `SubclassWindow` if you have already called [Create](#).

CContainedWindowT::SwitchMessageMap

Changes which message map will be used to process the contained window's messages.

```
void SwitchMessageMap(DWORD dwMsgMapID);
```

Parameters

dwMsgMapID

[in] The message map identifier. To use the default message map declared with [BEGIN_MSG_MAP](#), pass zero. To use an alternate message map declared with [ALT_MSG_MAP\(msgMapID\)](#), pass `msgMapID`.

Remarks

The message map must be defined in the containing object.

You initially specify the message map identifier in the constructor.

CContainedWindowT::UnsubclassWindow

Detaches the subclassed window from the `CContainedWindowT` object and restores the original window procedure, saved in [m_pfnSuperWindowProc](#).

```
HWND UnsubclassWindow(BOOL bForce = FALSE);
```

Parameters

bForce

[in] Set to TRUE to force the original window procedure to be restored even if the window procedure for this `CContainedWindowT` object is not currently active. If *bForce* is set to FALSE and the window procedure for this `CContainedWindowT` object is not currently active, the original window procedure will not be restored.

Return Value

The handle to the window previously subclassed. If *bForce* is set to FALSE and the window procedure for this `CContainedWindowT` object is not currently active, returns NULL.

Remarks

Use this method only if you want to restore the original window procedure before the window is destroyed. Otherwise, [WindowProc](#) will automatically do this when the window is destroyed.

CContainedWindowT::WindowProc

This static method implements the window procedure.

```
static LRESULT CALLBACK WindowProc(  
    HWND hWnd,  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam);
```

Parameters

hWnd

[in] The handle to the window.

uMsg

[in] The message sent to the window.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

Return Value

The result of the message processing.

Remarks

`WindowProc` directs messages to the message map identified by `m_dwMsgMapID`. If necessary, `WindowProc` calls [DefWindowProc](#) for additional message processing.

See also

[CWindow Class](#)

[CWindowImpl Class](#)

[CMessageMap Class](#)

[BEGIN_MSG_MAP](#)

[ALT_MSG_MAP\(msgMapID\)](#)

[Class Overview](#)

CCRTAllocator Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods for managing memory using CRT memory routines.

Syntax

```
class ATL::CCRTAllocator
```

Members

Public Methods

NAME	DESCRIPTION
CCRTAllocator::Allocate	(Static) Call this method to allocate memory.
CCRTAllocator::Free	(Static) Call this method to free memory.
CCRTAllocator::Reallocate	(Static) Call this method to reallocate memory.

Remarks

This class is used by [CHeapPtr](#) to provide the CRT memory allocation routines. The counterpart class, [CComAllocator](#), provides the same methods using COM routines.

Requirements

Header: atlcore.h

CCRTAllocator::Allocate

Call this static function to allocate memory.

```
static __declspec(allocator) void* Allocate(size_t nBytes) throw();
```

Parameters

nBytes

The number of bytes to allocate.

Return Value

Returns a void pointer to the allocated space, or NULL if there is insufficient memory available.

Remarks

Allocates memory. See [malloc](#) for more details.

CCRTAllocator::Free

Call this static function to free memory.

```
static void Free(void* p) throw();
```

Parameters

p

Pointer to the allocated memory.

Remarks

Frees the allocated memory. See [free](#) for more details.

CCRTAllocator::Reallocate

Call this static function to reallocate memory.

```
static __declspec(allocator) void* Reallocate(void* p, size_t nBytes) throw();
```

Parameters

p

Pointer to the allocated memory.

nBytes

The number of bytes to reallocate.

Return Value

Returns a void pointer to the allocated space, or NULL if there is insufficient memory.

Remarks

Resizes the amount of allocated memory. See [realloc](#) for more details.

See also

[CHepPtr Class](#)

[CComAllocator Class](#)

[Class Overview](#)

CCRTHeap Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements [IAtlMemMgr](#) using the CRT heap functions.

Syntax

```
class CCRTHeap : public IAtlMemMgr
```

Members

Public Methods

NAME	DESCRIPTION
CCRTHeap::Allocate	Call this method to allocate a block of memory.
CCRTHeap::Free	Call this method to free a block of memory allocated by this memory manager.
CCRTHeap::GetSize	Call this method to get the allocated size of a memory block allocated by this memory manager.
CCRTHeap::Reallocate	Call this method to reallocate memory allocated by this memory manager.

Remarks

[CCRTHeap](#) implements memory allocation functions using the CRT heap functions, including [malloc](#), [free](#), [realloc](#), and [_msize](#).

Example

See the example for [IAtlMemMgr](#).

Inheritance Hierarchy

[IAtlMemMgr](#)

[CCRTHeap](#)

Requirements

Header: `atlmem.h`

CCRTHeap::Allocate

Call this method to allocate a block of memory.

```
virtual __declspec(allocator) void* Allocate(size_t nBytes) throw();
```

Parameters

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call [CCRTHeap::Free](#) or [CCRTHeap::Reallocate](#) to free the memory allocated by this method.

Implemented using [malloc](#).

CCRTHeap::Free

Call this method to free a block of memory allocated by this memory manager.

```
virtual void Free(void* p) throw();
```

Parameters

p

Pointer to memory previously allocated by this memory manager. NULL is a valid value and does nothing.

Remarks

Implemented using [free](#).

CCRTHeap::GetSize

Call this method to get the allocated size of a memory block allocated by this memory manager.

```
virtual size_t GetSize(void* p) throw();
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

Return Value

Returns the size of the allocated memory block in bytes.

Remarks

Implemented using [_msize](#).

CCRTHeap::Reallocate

Call this method to reallocate memory allocated by this memory manager.

```
virtual __declspec(allocator) void* Reallocate(void* p, size_t nBytes) throw();
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call [CCRTHeap::Free](#) to free the memory allocated by this method. Implemented using [realloc](#).

See also

[Class Overview](#)

[CComHeap Class](#)

[CWin32Heap Class](#)

[CLocalHeap Class](#)

[CGlobalHeap Class](#)

[IAtlMemMgr Class](#)

CDacl Class

4/1/2019 • 4 minutes to read • [Edit Online](#)

This class is a wrapper for a DACL (discretionary access-control list) structure.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CDacl : public CAcl
```

Members

Public Constructors

NAME	DESCRIPTION
CDacl::CDacl	The constructor.
CDacl::~~CDacl	The destructor.

Public Methods

NAME	DESCRIPTION
CDacl::AddAllowedAce	Adds an allowed ACE (access-control entry) to the <code>CDacl</code> object.
CDacl::AddDeniedAce	Adds a denied ACE to the <code>CDacl</code> object.
CDacl::GetAceCount	Returns the number of ACEs (access-control entries) in the <code>CDacl</code> object.
CDacl::RemoveAce	Removes a specific ACE (access-control entry) from the <code>CDacl</code> object.
CDacl::RemoveAllAces	Removes all of the ACEs contained in the <code>CDacl</code> object.

Public Operators

NAME	DESCRIPTION
CDacl::operator =	Assignment operator.

Remarks

An object's security descriptor can contain a DACL. A DACL contains zero or more ACEs (access-control entries) that identify the users and groups who can access the object. If a DACL is empty (that is, it contains zero ACEs), no access is explicitly granted, so access is implicitly denied. However, if an object's security descriptor does not have a DACL, the object is unprotected and everyone has complete access.

To retrieve an object's DACL, you must be the object's owner or have `READ_CONTROL` access to the object. To change an object's DACL, you must have `WRITE_DAC` access to the object.

Use the class methods provided to create, add, remove, and delete ACEs from the `CDacl` object. See also [AtlGetDacl](#) and [AtlSetDacl](#).

For an introduction to the access control model in Windows, see [Access Control](#) in the Windows SDK.

Inheritance Hierarchy

[CAcl](#)

`CDacl`

Requirements

Header: `atlsecurity.h`

CDacl::AddAllowedAce

Adds an allowed ACE (access-control entry) to the `CDacl` object.

```
bool AddAllowedAce(
    const CSid& rSid,
    ACCESS_MASK AccessMask,
    BYTE AceFlags = 0) throw(...);

bool AddAllowedAce(
    const CSid& rSid,
    ACCESS_MASK AccessMask,
    BYTE AceFlags,
    const GUID* pObjectType,
    const GUID* pInheritedObjectType) throw(...);
```

Parameters

rSid

A [CSid](#) object.

AccessMask

Specifies the mask of access rights to be allowed for the specified `CSid` object.

AceFlags

A set of bit flags that control ACE inheritance.

pObjectType

The object type.

pInheritedObjectType

The inherited object type.

Return Value

Returns `TRUE` if the ACE is added to the `CDacl` object, `FALSE` on failure.

Remarks

A `CDacl` object contains zero or more ACEs (access-control entries) that identify the users and groups who can access the object. This method adds an ACE that allows access to the `CDacl` object.

See [ACE_HEADER](#) for a description of the various flags which can be set in the `AceFlags` parameter.

CDacl::AddDeniedAce

Adds a denied ACE (access-control entry) to the `CDacl` object.

```
bool AddDeniedAce(
    const CSid& rSid,
    ACCESS_MASK AccessMask,
    BYTE AceFlags = 0) throw(...);

bool AddDeniedAce(
    const CSid& rSid,
    ACCESS_MASK AccessMask,
    BYTE AceFlags,
    const GUID* pObjectType,
    const GUID* pInheritedObjectType) throw(...);
```

Parameters

rSid

A `CSid` object.

AccessMask

Specifies the mask of access rights to be denied for the specified `CSid` object.

AceFlags

A set of bit flags that control ACE inheritance. Defaults to 0 in the first form of the method.

pObjectType

The object type.

pInheritedObjectType

The inherited object type.

Return Value

Returns TRUE if the ACE is added to the `CDacl` object, FALSE on failure.

Remarks

A `CDacl` object contains zero or more ACEs (access-control entries) that identify the users and groups who can access the object. This method adds an ACE that denies access to the `CDacl` object.

See [ACE_HEADER](#) for a description of the various flags which can be set in the `AceFlags` parameter.

CDacl::CDacl

The constructor.

```
CDacl (const ACL& rhs) throw(...);
CDacl () throw();
```

Parameters

rhs

An existing `ACL` (access-control list) structure.

Remarks

The `CDacl` object can be optionally created using an existing `ACL` structure. It is important to note that only a DACL (discretionary access-control list), and not a SACL (system access-control list), should be passed as this parameter. In debug builds, passing a SACL will cause an ASSERT. In release builds, passing a SACL will cause the ACEs (access-control entries) in the ACL to be ignored, and no error will occur.

CDacl::~~CDacl

The destructor.

```
~CDacl () throw();
```

Remarks

The destructor frees any resources acquired by the object, including all ACEs (access-control entries) using [CDacl::RemoveAllAces](#).

CDacl::GetAceCount

Returns the number of ACEs (access-control entries) in the `CDacl` object.

```
UINT GetAceCount() const throw();
```

Return Value

Returns the number of ACEs contained in the `CDacl` object.

CDacl::operator =

Assignment operator.

```
CDacl& operator= (const ACL& rhs) throw(...);
```

Parameters

rhs

The ACL (access-control list) to assign to the existing object.

Return Value

Returns a reference to the updated `CDacl` object.

Remarks

You should ensure that you only pass a DACL (discretionary access-control list) to this function. Passing a SACL (system access-control list) to this function will cause an ASSERT in debug builds but will cause no error in release builds.

CDacl::RemoveAce

Removes a specific ACE (access-control entry) from the `CDacl` object.

```
void RemoveAce(UINT nIndex) throw();
```

Parameters

nIndex

Index to the ACE entry to remove.

Remarks

This method is derived from [CAtlArray::RemoveAt](#).

CDacl::RemoveAllAces

Removes all of the ACEs (access-control entries) contained in the `CDacl` object.

```
void RemoveAllAces() throw();
```

Remarks

Removes every `ACE` (access-control entry) structure (if any) in the `CDacl` object.

See also

[Security Sample](#)

[CAcl Class](#)

[ACLs](#)

[ACEs](#)

[Class Overview](#)

[Security Global Functions](#)

CDebugReportHook Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

Use this class to send debug reports to a named pipe.

Syntax

```
class CDebugReportHook
```

Members

Public Constructors

NAME	DESCRIPTION
CDebugReportHook::CDebugReportHook	Calls SetPipeName , SetTimeout , and SetHook .
CDebugReportHook::~~CDebugReportHook	Calls CDebugReportHook::RemoveHook .

Public Methods

NAME	DESCRIPTION
CDebugReportHook::CDebugReportHookProc	(Static) The custom reporting function that is hooked into the C run-time debug reporting process.
CDebugReportHook::RemoveHook	Call this method to stop sending debug reports to the named pipe and restore the previous report hook.
CDebugReportHook::SetHook	Call this method to start sending debug reports to the named pipe.
CDebugReportHook::SetPipeName	Call this method to set the machine and name of the pipe to which the debug reports will be sent.
CDebugReportHook::SetTimeout	Call this method to set the time in milliseconds that this class will wait for the named pipe to become available.

Remarks

Create an instance of this class in debug builds of your services or applications to send debug reports to a named pipe. Debug reports are generated by calling [_CrtDbgReport](#) or using a wrapper for this function such as the [ATLTRACE](#) and [ATLASSERT](#) macros.

Use of this class allows you to interactively debug components running in non-interactive [window stations](#).

Note that debug reports are sent using the underlying security context of the thread. Impersonation is temporarily disabled so that debug reports can be viewed in situations where impersonation of low privilege users is taking place, such as in web applications.

Requirements

Header: atlutil.h

CDebugReportHook::CDebugReportHook

Calls [SetPipeName](#), [SetTimeout](#), and [SetHook](#).

```
CDebugReportHook(  
    LPCSTR szMachineName = ".",  
    LPCSTR szPipeName = "AtlsDbgPipe",  
    DWORD dwTimeout = 20000) throw();
```

Parameters

szMachineName

The name of the machine to which the debug output should be sent. Defaults to the local machine.

szPipeName

The name of the named pipe to which the debug output should be sent.

dwTimeout

The time in milliseconds that this class will wait for the named pipe to become available.

CDebugReportHook::~~CDebugReportHook

Calls [CDebugReportHook::RemoveHook](#).

```
~CDebugReportHook() throw();
```

CDebugReportHook::CDebugReportHookProc

The custom reporting function that is hooked into the C run-time debug reporting process.

```
static int __cdecl CDebugReportHookProc(  
    int reportType,  
    char* message,  
    int* returnValue) throw();
```

Parameters

reportType

The type of the report (`_CRT_WARN`, `_CRT_ERROR`, or `_CRT_ASSERT`).

message

The message string.

returnValue

The value that should be returned by [_CrtDbgReport](#).

Return Value

Returns FALSE if the hook handles the message in question completely so that no further reporting is required.

Returns TRUE if `_CrtDbgReport` should report the message in the normal way.

Remarks

The reporting function attempts to open the named pipe and communicate with the process at the other end. If the

pipe is busy, the reporting function will wait until the pipe is free or the timeout expires. The timeout can be set by the constructor or a call to [CDebugReportHook::SetTimeout](#).

The code in this function is executed in the underlying security context of the calling thread, that is, impersonation is disabled for the duration of this function.

CDebugReportHook::RemoveHook

Call this method to stop sending debug reports to the named pipe and restore the previous report hook.

```
void RemoveHook() throw();
```

Remarks

Calls [_CrtSetReportHook2](#) to restore the previous report hook.

CDebugReportHook::SetHook

Call this method to start sending debug reports to the named pipe.

```
void SetHook() throw();
```

Remarks

Calls [_CrtSetReportHook2](#) to have debug reports routed through [CDebugReportHookProc](#) to the named pipe. This class keeps track of the previous report hook so that it can be restored when [RemoveHook](#) is called.

CDebugReportHook::SetPipeName

Call this method to set the machine and name of the pipe to which the debug reports will be sent.

```
BOOL SetPipeName(  
    LPCSTR szMachineName = ".",  
    LPCSTR szPipeName = "AtlsDbgPipe") throw();
```

Parameters

szMachineName

The name of the machine to which the debug output should be sent.

szPipeName

The name of the named pipe to which the debug output should be sent.

Return Value

Returns TRUE on success, FALSE on failure.

CDebugReportHook::SetTimeout

Call this method to set the time in milliseconds that this class will wait for the named pipe to become available.

```
void SetTimeout(DWORD dwTimeout);
```

Parameters

dwTimeout

The time in milliseconds that this class will wait for the named pipe to become available.

See also

[Classes](#)

CDefaultCharTraits Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides two static functions for converting characters between uppercase and lowercase.

Syntax

```
template <typename T>
class CDefaultCharTraits
```

Parameters

T

The type of data to be stored in the collection.

Members

Public Methods

NAME	DESCRIPTION
CDefaultCharTraits::CharToLower	(Static) Call this function to convert a character to uppercase.
CDefaultCharTraits::CharToUpper	(Static) Call this function to convert a character to lowercase.

Remarks

This class provides functions that are utilized by the class [CStringElementTraits](#).

Requirements

Header: atcoll.h

CDefaultCharTraits::CharToLower

Call this function to convert a character to lowercase.

```
static wchar_t CharToLower(wchar_t x);
static char CharToLower(char x);
```

Parameters

x

The character to convert to lowercase.

Example

```
printf_s("%c\n", CDefaultCharTraits<char>::CharToLower('A'));
```

CDefaultCharTraits::CharToUpper

Call this function to convert a character to uppercase.

```
static wchar_t CharToUpper(wchar_t x);  
static char CharToUpper(char x);
```

Parameters

x

The character to convert to uppercase.

See also

[Class Overview](#)

CDefaultCompareTraits Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides default element comparison functions.

Syntax

```
template<typename T>
class CDefaultCompareTraits
```

Parameters

T

The type of data to be stored in the collection.

Members

Public Methods

NAME	DESCRIPTION
CDefaultCompareTraits::CompareElements	(Static) Call this function to compare two elements for equality.
CDefaultCompareTraits::CompareElementsOrdered	(Static) Call this function to determine the greater and lesser element.

Remarks

This class contains two static functions for comparing elements stored in a collection class object. This class is utilized by the [CDefaultElementTraits Class](#).

For more information, see [ATL Collection Classes](#).

Requirements

Header: atlcoll.h

CDefaultCompareTraits::CompareElements

Call this function to compare two elements for equality.

```
static bool CompareElements(const T& element1, const T& element2);
```

Parameters

element1

The first element.

element2

The second element.

Return Value

Returns true if the elements are equal, false otherwise.

Remarks

The default implementation of this function is the equality (==) operator. For objects other than simple data types, this function may need to be overridden.

CDefaultCompareTraits::CompareElementsOrdered

Call this function to determine the greater and lesser element.

```
static int CompareElementsOrdered(const T& element1, const T& element2);
```

Parameters

element1
The first element.

element2
The second element.

Return Value

Returns an integer based on the following table:

CONDITION	RETURN VALUE
<i>element1</i> < <i>element2</i>	<0
<i>element1</i> == <i>element2</i>	0
<i>element1</i> > <i>element2</i>	>0

Remarks

The default implementation of this function uses the ==, <, and > operators. For objects other than simple data types, this function may need to be overridden.

See also

[Class Overview](#)

CDefaultElementTraits Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides default methods and functions for a collection class.

Syntax

```
template <typename T>
class CDefaultElementTraits : public CElementTraitsBase<T>,
    public CDefaultHashTraits<T>,
    public CDefaultCompareTraits<T>
```

Parameters

T

The type of data to be stored in the collection.

Remarks

This class provides default static functions and methods for moving, copying, comparing, and hashing elements stored in a collection class object. This class derives its functions and methods from [CElementTraitsBase](#), [CDefaultHashTraits](#), and [CDefaultCompareTraits](#), and is utilized by [CElementTraits](#), [CPrimitiveElementTraits](#), and [CHeapPtrElementTraits](#).

For more information, see [ATL Collection Classes](#).

Requirements

Header: atcoll.h

See also

[Class Overview](#)

CDefaultHashTraits Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides a static function for calculating hash values.

Syntax

```
template<typename T>
class CDefaultHashTraits
```

Parameters

T

The type of data to be stored in the collection.

Members

Public Methods

NAME	DESCRIPTION
CDefaultHashTraits::Hash	(Static) Call this function to calculate a hash value for a given element.

Remarks

This class contains a single static function that returns a hash value for a given element. This class is utilized by the [CDefaultElementTraits Class](#).

For more information, see [ATL Collection Classes](#).

Requirements

Header: atcoll.h

CDefaultHashTraits::Hash

Call this function to calculate a hash value for a given element.

```
static ULONG Hash(const T& element) throw();
```

Parameters

element

The element.

Return Value

Returns the hash value.

Remarks

The default hashing algorithm is very simple: the return value is the element number. Override this function if a

more complicated algorithm is required.

See also

[Class Overview](#)

CDialogImpl Class

3/5/2019 • 4 minutes to read • [Edit Online](#)

This class provides methods for creating a modal or modeless dialog box.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class T,  
    class TBase = CWindow>  
class ATL_NO_VTABLE CDialogImpl : public CDialogImplBaseT<TBase>
```

Parameters

T

Your class, derived from `CDialogImpl`.

TBase

The base class of your new class. The default base class is [CWindow](#).

Members

Methods

Create	Creates a modeless dialog box.
DestroyWindow	Destroys a modeless dialog box.
DoModal	Creates a modal dialog box.
EndDialog	Destroys a modal dialog box.

CDialogImplBaseT Methods

GetDialogProc	Returns the current dialog box procedure.
MapDialogRect	Maps the dialog-box units of the specified rectangle to screen units (pixels).
OnFinalMessage	Called after receiving the last message, typically WM_NCDESTROY.

Static Functions

DialogProc	Processes messages sent to the dialog box.
StartDialogProc	Called when the first message is received to process messages sent to the dialog box.

Remarks

With `CDialogImpl` you can create a modal or modeless dialog box. `CDialogImpl` provides the dialog box procedure, which uses the default message map to direct messages to the appropriate handlers.

The base class destructor `~CWindowImplRoot` ensures that the window is gone before destroying the object.

`CDialogImpl` derives from `CDialogImplBaseT`, which in turn derives from `CWindowImplRoot`.

NOTE
Your class must define an `IDD` member that specifies the dialog template resource ID. For example, the ATL Project Wizard automatically adds the following line to your class:

```
enum { IDD = IDD_MYDLG };
```

where `MyDlg` is the **Short name** entered in the wizard's **Names** page.

FOR MORE INFORMATION ABOUT	SEE
Creating controls	ATL Tutorial
Using dialog boxes in ATL	ATL Window Classes
ATL Project Wizard	Creating an ATL Project
Dialog boxes	Dialog Boxes and subsequent topics in the Windows SDK

Requirements

Header: atlwin.h

CDialogImpl::Create

Creates a modeless dialog box.

```
HWND Create(  
    HWND hWndParent,  
    LPARAM dwInitParam = NULL );  
  
HWND Create(  
    HWND hWndParent,  
    RECT&,  
    LPARAM dwInitParam = NULL);
```

Parameters

hWndParent

[in] The handle to the owner window.

RECT& rect [in] A [RECT](#) structure specifying the dialog's size and position.

dwInitParam

[in] Specifies the value to pass to the dialog box in the *lParam* parameter of the WM_INITDIALOG message.

Return Value

The handle to the newly created dialog box.

Remarks

This dialog box is automatically attached to the `CDialogImpl` object. To create a modal dialog box, call [DoModal](#).

The second override above is used only with [CComControl](#).

CDialogImpl::DestroyWindow

Destroys a modeless dialog box.

```
BOOL DestroyWindow();
```

Return Value

TRUE if the dialog box was successfully destroyed; otherwise FALSE.

Remarks

Returns TRUE if the dialog box was successfully destroyed; otherwise FALSE.

CDialogImpl::DialogProc

This static function implements the dialog box procedure.

```
static LRESULT CALLBACK DialogProc(  
    HWND hWnd,  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam);
```

Parameters

hWnd

[in] The handle to the dialog box.

uMsg

[in] The message sent to the dialog box.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

Return Value

TRUE if the message is processed; otherwise, FALSE.

Remarks

`DialogProc` uses the default message map to direct messages to the appropriate handlers.

You can override `DialogProc` to provide a different mechanism for handling messages.

CDialogImpl::DoModal

Creates a modal dialog box.

```
INT_PTR DoModal(  
    HWND hWndParent = ::GetActiveWindow(),  
    LPARAM dwInitParam = NULL);
```

Parameters

hWndParent

[in] The handle to the owner window. The default value is the return value of the [GetActiveWindow](#) Win32 function.

dwInitParam

[in] Specifies the value to pass to the dialog box in the *lParam* parameter of the WM_INITDIALOG message.

Return Value

If successful, the value of the *nRetCode* parameter specified in the call to [EndDialog](#). Otherwise, -1.

Remarks

This dialog box is automatically attached to the `CDialogImpl` object.

To create a modeless dialog box, call [Create](#).

CDialogImpl::EndDialog

Destroys a modal dialog box.

```
BOOL EndDialog(int nRetCode);
```

Parameters

nRetCode

[in] The value to be returned by [CDialogImpl::DoModal](#).

Return Value

TRUE if the dialog box is destroyed; otherwise, FALSE.

Remarks

`EndDialog` must be called through the dialog procedure. After the dialog box is destroyed, Windows uses the value of *nRetCode* as the return value for `DoModal`, which created the dialog box.

NOTE

Do not call `EndDialog` to destroy a modeless dialog box. Call [CWindow::DestroyWindow](#) instead.

CDialogImpl::GetDialogProc

Returns `DialogProc`, the current dialog box procedure.

```
virtual WNDPROC GetDialogProc();
```

Return Value

The current dialog box procedure.

Remarks

Override this method to replace the dialog procedure with your own.

CDialogImpl::MapDialogRect

Converts (maps) the dialog-box units of the specified rectangle to screen units (pixels).

```
BOOL MapDialogRect(LPRECT lpRect);
```

Parameters

lpRect

Points to a `CRect` object or `RECT` structure that is to receive the client coordinates of the update that encloses the update region.

Return Value

Nonzero if the update succeeds; 0 if the update fails. To get extended error information, call `GetLastError`.

Remarks

The function replaces the coordinates in the specified `RECT` structure with the converted coordinates, which allows the structure to be used to create a dialog box or position a control within a dialog box.

CDialogImpl::OnFinalMessage

Called after receiving the last message (typically `WM_NCDESTROY`).

```
virtual void OnFinalMessage(HWND hWnd);
```

Parameters

hWnd

[in] A handle to the window being destroyed.

Remarks

Note that if you want to automatically delete your object upon the window destruction, you can call **delete this**; here.

CDialogImpl::StartDialogProc

Called only once, when the first message is received, to process messages sent to the dialog box.

```
static LRESULT CALLBACK StartDialogProc(  
    HWND hWnd,  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam);
```

Parameters

hWnd

[in] The handle to the dialog box.

uMsg

[in] The message sent to the dialog box.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

Return Value

The window procedure.

Remarks

After the initial call to `StartDialogProc`, `DialogProc` is set as a dialog procedure, and further calls go there.

See also

[BEGIN_MSG_MAP](#)

[Class Overview](#)

CDynamicChain Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

This class provides methods supporting the dynamic chaining of message maps.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CDynamicChain
```

Members

Public Constructors

NAME	DESCRIPTION
CDynamicChain::CDynamicChain	The constructor.
CDynamicChain::~~CDynamicChain	The destructor.

Public Methods

NAME	DESCRIPTION
CDynamicChain::CallChain	Directs a Windows message to another object's message map.
CDynamicChain::RemoveChainEntry	Removes a message map entry from the collection.
CDynamicChain::SetChainEntry	Adds a message map entry to the collection or modifies an existing entry.

Remarks

[CDynamicChain](#) manages a collection of message maps, enabling a Windows message to be directed, at run time, to another object's message map.

To add support for dynamic chaining of message maps, do the following:

- Derive your class from [CDynamicChain](#). In the message map, specify the [CHAIN_MSG_MAP_DYNAMIC](#) macro to chain to another object's default message map.
- Derive every class you want to chain to from [CMessageMap](#). [CMessageMap](#) allows an object to expose its message maps to other objects.
- Call [CDynamicChain::SetChainEntry](#) to identify which object and which message map you want to chain to.

For example, suppose your class is defined as follows:

```

class CMyChainWnd : public CWindowImpl<CMyChainWnd>,
    public CDynamicChain
{
public:
    CMyChainWnd() {}

    BEGIN_MSG_MAP(CMyChainWnd)
        MESSAGE_HANDLER(WM_PAINT, OnPaint)
        MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
        // dynamically chain to the default
        // message map in another object
        CHAIN_MSG_MAP_DYNAMIC(1313)
            // '1313' identifies the object
            // and the message map that will be
            // chained to. '1313' is defined
            // through the SetChainEntry method
    END_MSG_MAP()

    LRESULT OnPaint(UINT /*nMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
        BOOL& /*bHandled*/)
    {
        // Do some painting code
        return 0;
    }

    LRESULT OnSetFocus(UINT /*uMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
        BOOL& /*bHandled*/)
    {
        return 0;
    }
};

```

The client then calls `CMyWindow::SetChainEntry` :

```
myCtl.SetChainEntry(1313, &chainedObj);
```

where `chainedObj` is the chained object and is an instance of a class derived from `CMessageMap`. Now, if `myCtl` receives a message that is not handled by `OnPaint` or `OnSetFocus`, the window procedure directs the message to `chainedObj`'s default message map.

For more information about message map chaining, see [Message Maps](#) in the article "ATL Window Classes."

Requirements

Header: atlwin.h

CDynamicChain::CallChain

Directs the Windows message to another object's message map.

```

BOOL CallChain(
    DWORD dwChainID,
    HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam,
    LRESULT& lResult);

```

Parameters

dwChainID

[in] The unique identifier associated with the chained object and its message map.

hWnd

[in] The handle to the window receiving the message.

uMsg

[in] The message sent to the window.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

lResult

[out] The result of the message processing.

Return Value

TRUE if the message is fully processed; otherwise, FALSE.

Remarks

For the window procedure to invoke `CallChain`, you must specify the `CHAIN_MSG_MAP_DYNAMIC` macro in your message map. For an example, see the [CDynamicChain](#) overview.

`CallChain` requires a previous call to [SetChainEntry](#) to associate the *dwChainID* value with an object and its message map.

CDynamicChain::CDynamicChain

The constructor.

```
CDynamicChain();
```

CDynamicChain::~~CDynamicChain

The destructor.

```
~CDynamicChain();
```

Remarks

Frees all allocated resources.

CDynamicChain::RemoveChainEntry

Removes the specified message map from the collection.

```
BOOL RemoveChainEntry(DWORD dwChainID);
```

Parameters

dwChainID

[in] The unique identifier associated with the chained object and its message map. You originally define this value through a call to [SetChainEntry](#).

Return Value

TRUE if the message map is successfully removed from the collection. Otherwise, FALSE.

CDynamicChain::SetChainEntry

Adds the specified message map to the collection.

```
BOOL SetChainEntry(  
    DWORD dwChainID,  
    CMessageMap* pObject,  
    DWORD dwMsgMapID = 0);
```

Parameters

dwChainID

[in] The unique identifier associated with the chained object and its message map.

pObject

[in] A pointer to the chained object declaring the message map. This object must derive from [CMessageMap](#).

dwMsgMapID

[in] The identifier of the message map in the chained object. The default value is 0, which identifies the default message map declared with [BEGIN_MSG_MAP](#). To specify an alternate message map declared with [ALT_MSG_MAP\(msgMapID\)](#), pass `msgMapID`.

Return Value

TRUE if the message map is successfully added to the collection. Otherwise, FALSE.

Remarks

If the *dwChainID* value already exists in the collection, its associated object and message map are replaced by *pObject* and *dwMsgMapID*, respectively. Otherwise, a new entry is added.

See also

[CWindowImpl Class](#)

[Class Overview](#)

CElementTraits Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class is used by collection classes to provide methods and functions for moving, copying, comparison, and hashing operations.

Syntax

```
template<typename T>
class CElementTraits : public CDefaultElementTraits<T>
```

Parameters

T

The type of data to be stored in the collection.

Remarks

This class provides default static functions and methods for moving, copying, comparing, and hashing elements stored in a collection class object. `CElementTraits` is specified as the default provider of these operations by the collection classes [CAtlArray](#), [CAtlList](#), [CRBMap](#), [CRBMultiMap](#), and [CRBTree](#).

The default implementations will suffice for simple data types, but if the collection classes are used to store more complex objects, the functions and methods must be overridden by user-supplied implementations.

For more information, see [ATL Collection Classes](#).

Requirements

Header: atlcoll.h

See also

[CDefaultElementTraits Class](#)

[Class Overview](#)

CElementTraitsBase Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides default copy and move methods for a collection class.

Syntax

```
template<typename T>
class CElementTraitsBase
```

Parameters

T

The type of data to be stored in the collection.

Members

Public Typedefs

NAME	DESCRIPTION
CElementTraitsBase::INARGETYPE	The data type to use for adding elements to the collection class object.
CElementTraitsBase::OUTARGETYPE	The data type to use for retrieving elements from the collection class object.

Public Methods

NAME	DESCRIPTION
CElementTraitsBase::CopyElements	Call this method to copy elements stored in a collection class object.
CElementTraitsBase::RelocateElements	Call this method to relocate elements stored in a collection class object.

Remarks

This base class defines methods for copying and relocating elements in a collection class. It is utilized by the classes [CDefaultElementTraits](#), [CStringRefElementTraits](#), and [CStringElementTraits](#).

For more information, see [ATL Collection Classes](#).

Requirements

Header: atlcoll.h

CElementTraitsBase::CopyElements

Call this method to copy elements stored in a collection class object.

```
static void CopyElements(  
    T* pDest,  
    const T* pSrc,  
    size_t nElements);
```

Parameters

pDest

Pointer to the first element that will receive the copied data.

pSrc

Pointer to the first element to copy.

nElements

The number of elements to copy.

Remarks

The source and destination elements should not overlap.

CElementTraitsBase::INARGTYPE

The data type to use for adding elements to the collection.

```
typedef const T& INARGTYPE;
```

CElementTraitsBase::OUTARGTYPE

The data type to use for retrieving elements from the collection.

```
typedef T& OUTARGTYPE;
```

CElementTraitsBase::RelocateElements

Call this method to relocate elements stored in a collection class object.

```
static void RelocateElements(  
    T* pDest,  
    T* pSrc,  
    size_t nElements);
```

Parameters

pDest

Pointer to the first element that will receive the relocated data.

pSrc

Pointer to the first element to relocate.

nElements

The number of elements to relocate.

Remarks

This method calls [memmove](#), which is sufficient for most data types. If the objects being moved contain pointers to their own members, this method will need to be overridden.

See also

[Class Overview](#)

CFirePropNotifyEvent Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods for notifying the container's sink regarding control property changes.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CFirePropNotifyEvent
```

Members

Public Methods

NAME	DESCRIPTION
CFirePropNotifyEvent::FireOnChanged	(Static) Notifies the container's sink that a control property has changed.
CFirePropNotifyEvent::FireOnRequestEdit	(Static) Notifies the container's sink that a control property is about to change.

Remarks

`CFirePropNotifyEvent` has two methods that notify the container's sink that a control property has changed or is about to change.

If the class implementing your control is derived from `IPropertyNotifySink`, the `CFirePropNotifyEvent` methods are invoked when you call `FireOnRequestEdit` or `FireOnChanged`. If your control class is not derived from `IPropertyNotifySink`, calls to these functions return `S_OK`.

For more information about creating controls, see the [ATL Tutorial](#).

Requirements

Header: atlctl.h

CFirePropNotifyEvent::FireOnChanged

Notifies all connected [IPropertyNotifySink](#) interfaces (on every connection point of the object) that the specified object property has changed.

```
static HRESULT FireOnChanged(IUnknown* pUnk, DISPID dispID);
```

Parameters

pUnk

[in] Pointer to the `IUnknown` of the object sending the notification.

dispID

[in] Identifier of the property that has changed.

Return Value

One of the standard HRESULT values.

Remarks

This function is safe to call even if your control does not support connection points.

CFirePropNotifyEvent::FireOnRequestEdit

Notifies all connected [IPropertyNotifySink](#) interfaces (on every connection point of the object) that the specified object property is about to change.

```
static HRESULT FireOnRequestEdit(IUnknown* pUnk, DISPID dispID);
```

Parameters

pUnk

[in] Pointer to the `IUnknown` of the object sending the notification.

dispID

[in] Identifier of the property about to change.

Return Value

One of the standard HRESULT values.

Remarks

This function is safe to call even if your control does not support connection points.

See also

[Class Overview](#)

CGlobalHeap Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements [IAtlMemMgr](#) using the Win32 global heap functions.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CGlobalHeap : public IAtlMemMgr
```

Members

Public Methods

NAME	DESCRIPTION
CGlobalHeap::Allocate	Call this method to allocate a block of memory.
CGlobalHeap::Free	Call this method to free a block of memory allocated by this memory manager.
CGlobalHeap::GetSize	Call this method to get the allocated size of a memory block allocated by this memory manager.
CGlobalHeap::Reallocate	Call this method to reallocate memory allocated by this memory manager.

Remarks

[CGlobalHeap](#) implements memory allocation functions using the Win32 global heap functions.

NOTE

The global heap functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the [heap functions](#). These are available in the [CWin32Heap](#) class. Global functions are still used by DDE and the clipboard functions.

Example

See the example for [IAtlMemMgr](#).

Inheritance Hierarchy

[IAtlMemMgr](#)

Requirements

Header: atlmem.h

CGlobalHeap::Allocate

Call this method to allocate a block of memory.

```
virtual __declspec(allocator) void* Allocate(size_t nBytes) throw();
```

Parameters

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call [CGlobalHeap::Free](#) or [CGlobalHeap::Reallocate](#) to free the memory allocated by this method.

Implemented using [GlobalAlloc](#) with a flag parameter of GMEM_FIXED.

CGlobalHeap::Free

Call this method to free a block of memory allocated by this memory manager.

```
virtual void Free(void* p) throw();
```

Parameters

p

Pointer to memory previously allocated by this memory manager. NULL is a valid value and does nothing.

Remarks

Implemented using [GlobalFree](#).

CGlobalHeap::GetSize

Call this method to get the allocated size of a memory block allocated by this memory manager.

```
virtual size_t GetSize(void* p) throw();
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

Return Value

Returns the size of the allocated memory block in bytes.

Remarks

Implemented using [GlobalSize](#).

CGlobalHeap::Reallocate

Call this method to reallocate memory allocated by this memory manager.

```
virtual __declspec(allocator) void* Reallocate(void* p, size_t nBytes) throw();
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call [CGlobalHeap::Free](#) to free the memory allocated by this method.

Implemented using [GlobalReAlloc](#).

See also

[Class Overview](#)

[CComHeap Class](#)

[CWin32Heap Class](#)

[CLocalHeap Class](#)

[CCRTHeap Class](#)

[IAtlMemMgr Class](#)

CHandle Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods for creating and using a handle object.

Syntax

```
class CHandle
```

Members

Public Constructors

NAME	DESCRIPTION
CHandle::CHandle	The constructor.
CHandle::~~CHandle	The destructor.

Public Methods

NAME	DESCRIPTION
CHandle::Attach	Call this method to attach the <code>CHandle</code> object to an existing handle.
CHandle::Close	Call this method to close a <code>CHandle</code> object.
CHandle::Detach	Call this method to detach a handle from a <code>CHandle</code> object.

Public Operators

NAME	DESCRIPTION
CHandle::operator HANDLE	Returns the value of the stored handle.
CHandle::operator =	Assignment operator.

Public Data Members

NAME	DESCRIPTION
CHandle::m_h	The member variable that stores the handle.

Remarks

A `CHandle` object can be used whenever a handle is required: the main difference is that the `CHandle` object will automatically be deleted.

NOTE

Some API functions will use NULL as an empty or invalid handle, while others use INVALID_HANDLE_VALUE. `CHandle` only uses NULL and will treat INVALID_HANDLE_VALUE as a real handle. If you call an API which can return INVALID_HANDLE_VALUE, you should check for this value before calling [CHandle::Attach](#) or passing it to the `CHandle` constructor, and instead pass NULL.

Requirements

Header: `atlbase.h`

CHandle::Attach

Call this method to attach the `CHandle` object to an existing handle.

```
void Attach(HANDLE h) throw();
```

Parameters

h

`CHandle` will take ownership of the handle *h*.

Remarks

Assigns the `CHandle` object to the *h* handle. In debugs builds, an ATLASSERT will be raised if *h* is NULL. No other check as to the validity of the handle is made.

CHandle::CHandle

The constructor.

```
CHandle() throw();  
CHandle(CHandle& h) throw();  
explicit CHandle(HANDLE h) throw();
```

Parameters

h

An existing handle or `CHandle`.

Remarks

Creates a new `CHandle` object, optionally using an existing handle or `CHandle` object.

CHandle::~CHandle

The destructor.

```
~CHandle() throw();
```

Remarks

Frees the `CHandle` object by calling [CHandle::Close](#).

CHandle::Close

Call this method to close a `CHandle` object.

```
void Close() throw();
```

Remarks

Closes an open object handle. If the handle is NULL, which will be the case if `Close` has already been called, an ATLASSERT will be raised in debug builds.

CHandle::Detach

Call this method to detach a handle from a `CHandle` object.

```
HANDLE Detach() throw();
```

Return Value

Returns the handle being detached.

Remarks

Releases ownership of the handle.

CHandle::m_h

The member variable that stores the handle.

```
HANDLE m_h;
```

CHandle::operator =

The assignment operator.

```
CHandle& operator=(CHandle& h) throw();
```

Parameters

h

`CHandle` will take ownership of the handle *h*.

Return Value

Returns a reference to the new `CHandle` object.

Remarks

If the `CHandle` object currently contains a handle, it will be closed. The `CHandle` object being passed in will have its handle reference set to NULL. This ensures that two `CHandle` objects will never contain the same active handle.

CHandle::operator HANDLE

Returns the value of the stored handle.

```
operator HANDLE() const throw();
```

Remarks

Returns the value stored in [CHandle::m_h](#).

See also

[Class Overview](#)

CHeapPtr Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

A smart pointer class for managing heap pointers.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<typename T, class Allocator=CCRTAllocator>
class CHeapPtr : public CHeapPtrBase<T, Allocator>
```

Parameters

T

The object type to be stored on the heap.

Allocator

The memory allocation class to use.

Members

Public Constructors

NAME	DESCRIPTION
CHeapPtr::CHeapPtr	The constructor.

Public Methods

NAME	DESCRIPTION
CHeapPtr::Allocate	Call this method to allocate memory on the heap to store objects.
CHeapPtr::Reallocate	Call this method to reallocate the memory on the heap.

Public Operators

NAME	DESCRIPTION
CHeapPtr::operator =	The assignment operator.

Remarks

`CHeapPtr` is derived from [CHeapPtrBase](#) and by default uses the CRT routines (in [CCRTAllocator](#)) to allocate and free memory. The class [CHeapPtrList](#) may be used to construct a list of heap pointers. See also [CComHeapPtr](#), which uses COM memory allocation routines.

Inheritance Hierarchy

[CHeapPtrBase](#)

[CHeapPtr](#)

Requirements

Header: `atlccore.h`

CHeapPtr::Allocate

Call this method to allocate memory on the heap to store objects.

```
bool Allocate(size_t nElements = 1) throw();
```

Parameters

nElements

The number of elements used to calculate the amount of memory to allocate. The default value is 1.

Return Value

Returns true if the memory was successfully allocated, false on failure.

Remarks

The allocator routines are used to reserve enough memory on the heap to store *nElement* objects of a type defined in the constructor.

Example

```
// Create a new CHeapPtr object
CHeapPtr <int> myHP;
// Allocate space for 10 integers on the heap
myHP.Allocate(10);
```

CHeapPtr::CHeapPtr

The constructor.

```
CHeapPtr() throw();
explicit CHeapPtr(T* p) throw();
CHeapPtr(CHeapPtr<T, Allocator>& p) throw();
```

Parameters

p

An existing heap pointer or [CHeapPtr](#).

Remarks

The heap pointer can optionally be created using an existing pointer, or a [CHeapPtr](#) object. If so, the new [CHeapPtr](#) object assumes responsibility for managing the new pointer and resources.

Example


```
// Create a new CHeapPtr object
CHeapPtr <int> myHP;
// Create a new CHeapPtr from the first
CHeapPtr <int> myHP2(myHP);
```

CHeapPtr::operator =

Assignment operator.

```
CHeapPtr<T, Allocator>& operator=(
    CHeapPtr<T, Allocator>& p) throw();
```

Parameters

p

An existing `CHeapPtr` object.

Return Value

Returns a reference to the updated `CHeapPtr`.

Example

```
// Create a new CHeapPtr object
CHeapPtr <int> myHP;
// Allocate space for 10 integers on the heap
myHP.Allocate(10);
// Create a second heap pointer
// and assign it to the first pointer.
CHeapPtr <int> myHP2;
myHP2 = myHP;
```

CHeapPtr::Reallocate

Call this method to reallocate the memory on the heap.

```
bool Reallocate(size_t nElements) throw();
```

Parameters

nElements

The new number of elements used to calculate the amount of memory to allocate.

Return Value

Returns true if the memory was successfully allocated, false on failure.

Example

```
// Create a new CHeapPtr object
CHeapPtr <int> myHP;
// Allocate space for 10 integers on the heap
myHP.Allocate(10);
// Resize the allocated memory for 20 integers
myHP.Reallocate(20);
```

See also

[CHeapPtrBase Class](#)

[CCRTAllocator Class](#)

[Class Overview](#)

CHeapPtrBase Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class forms the basis for several smart heap pointer classes.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class T, class Allocator = CCRTAllocator>
class CHeapPtrBase
```

Parameters

T

The object type to be stored on the heap.

Allocator

The memory allocation class to use. By default CRT routines are used to allocate and free memory.

Members

Public Constructors

NAME	DESCRIPTION
CHeapPtrBase::~CHeapPtrBase	The destructor.

Public Methods

NAME	DESCRIPTION
CHeapPtrBase::AllocateBytes	Call this method to allocate memory.
CHeapPtrBase::Attach	Call this method to take ownership of an existing pointer.
CHeapPtrBase::Detach	Call this method to release ownership of a pointer.
CHeapPtrBase::Free	Call this method to delete an object pointed to by a <code>CHeapPtrBase</code> .
CHeapPtrBase::ReallocateBytes	Call this method to reallocate memory.

Public Operators

NAME	DESCRIPTION
CHeapPtrBase::operator T*	The cast operator.

NAME	DESCRIPTION
CHeapPtrBase::operator &	The & operator.
CHeapPtrBase::operator ->	The pointer-to-member operator.

Public Data Members

NAME	DESCRIPTION
CHeapPtrBase::m_pData	The pointer data member variable.

Remarks

This class forms the basis for several smart heap pointer classes. The derived classes, for example, [CHeapPtr](#) and [CComHeapPtr](#), add their own constructors and operators. See these classes for implementation examples.

Requirements

Header: atlc core.h

CHeapPtrBase::AllocateBytes

Call this method to allocate memory.

```
bool AllocateBytes(size_t nBytes) throw();
```

Parameters

nBytes

The number of bytes of memory to allocate.

Return Value

Returns true if the memory is successfully allocated, false otherwise.

Remarks

In debug builds, an assertion failure will occur if the [CHeapPtrBase::m_pData](#) member variable currently points to an existing value; that is, it is not equal to NULL.

CHeapPtrBase::Attach

Call this method to take ownership of an existing pointer.

```
void Attach(T* pData) throw();
```

Parameters

pData

The `CHeapPtrBase` object will take ownership of this pointer.

Remarks

When a `CHeapPtrBase` object takes ownership of a pointer, it will automatically delete the pointer and any allocated data when it goes out of scope.

In debug builds, an assertion failure will occur if the [CHeapPtrBase::m_pData](#) member variable currently points to an existing value; that is, it is not equal to NULL.

CHeapPtrBase::~~CHeapPtrBase

The destructor.

```
~CHeapPtrBase() throw();
```

Remarks

Frees all allocated resources.

CHeapPtrBase::Detach

Call this method to release ownership of a pointer.

```
T* Detach() throw();
```

Return Value

Returns a copy of the pointer.

Remarks

Releases ownership of a pointer, sets the [CHeapPtrBase::m_pData](#) member variable to NULL, and returns a copy of the pointer.

CHeapPtrBase::Free

Call this method to delete an object pointed to by a `CHeapPtrBase`.

```
void Free() throw();
```

Remarks

The object pointed to by the `CHeapPtrBase` is freed, and the [CHeapPtrBase::m_pData](#) member variable is set to NULL.

CHeapPtrBase::m_pData

The pointer data member variable.

```
T* m_pData;
```

Remarks

This member variable holds the pointer information.

CHeapPtrBase::operator &

The & operator.

```
T** operator&() throw();
```

Return Value

Returns the address of the object pointed to by the `CHeapPtrBase` object.

CHeapPtrBase::operator ->

The pointer-to-member operator.

```
T* operator->() const throw();
```

Return Value

Returns the value of the `CHeapPtrBase::m_pData` member variable.

Remarks

Use this operator to call a method in a class pointed to by the `CHeapPtrBase` object. In debug builds, an assertion failure will occur if the `CHeapPtrBase` points to NULL.

CHeapPtrBase::operator T*

The cast operator.

```
operator T*() const throw();
```

Remarks

Returns `CHeapPtrBase::m_pData`.

CHeapPtrBase::ReallocateBytes

Call this method to reallocate memory.

```
bool ReallocateBytes(size_t nBytes) throw();
```

Parameters

nBytes

The new amount of memory to allocate, in bytes.

Return Value

Returns true if the memory is successfully allocated, false otherwise.

See also

[CHeapPtr Class](#)

[CComHeapPtr Class](#)

[Class Overview](#)

CHeapPtrElementTraits Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods, static functions, and typedefs useful when creating collections of heap pointers.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<typename T, class Allocator = ATL::CCRTAllocator>
class CHeapPtrElementTraits :
    public CDefaultElementTraits<ATL::CHeapPtr<T, Allocator>>
```

Parameters

T

The object type to be stored in the collection class.

Allocator

The memory allocation class to use. The default is [CCRTAllocator](#).

Members

Public Typedefs

NAME	DESCRIPTION
CHeapPtrElementTraits::INARGTYPE	The data type to use for adding elements to the collection class object.
CHeapPtrElementTraits::OUTARGTYPE	The data type to use for retrieving elements from the collection class object.

Remarks

This class provides methods, static functions, and typedefs for aiding the creation of collection class objects containing heap pointers. The class `CHeapPtrList` derives from `CHeapPtrElementTraits`.

For more information, see [ATL Collection Classes](#).

Inheritance Hierarchy

[CDefaultCompareTraits](#)

[CDefaultHashTraits](#)

[CElementTraitsBase](#)

[CDefaultElementTraits](#)

Requirements

Header: atcoll.h

CHeapPtrElementTraits::INARGTYPE

The data type to use for adding elements to the collection class object.

```
typedef CHeapPtr<T, Allocator>& INARGTYPE;
```

CHeapPtrElementTraits::OUTARGTYPE

The data type to use for retrieving elements from the collection class object.

```
typedef T *& OUTARGTYPE;
```

See also

[CDefaultElementTraits Class](#)

[CComHeapPtr Class](#)

[Class Overview](#)

CHeapPtrList Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods useful when constructing a list of heap pointers.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<typename E, class Allocator = ATL::CCRTAllocator>
class CHeapPtrList
    : public CATlList<ATL::CHeapPtr<E, Allocator>,
        CHeapPtrElementTraits<E, Allocator>>
```

Parameters

E

The object type to be stored in the collection class.

Allocator

The memory allocation class to use. The default is [CCRTAllocator](#).

Members

Public Constructors

NAME	DESCRIPTION
CHeapPtrList::CHeapPtrList	The constructor.

Remarks

This class provides a constructor and derives methods from [CATlList](#) and [CHeapPtrElementTraits](#) to aid the creation of a collection class object storing heap pointers.

Inheritance Hierarchy

[CATlList](#)

CHeapPtrList

Requirements

Header: atlcoll.h

CHeapPtrList::CHeapPtrList

The constructor.

```
CHeapPtrList(UINT nBlockSize = 10) throw();
```

Parameters

nBlockSize

The block size.

Remarks

The block size is a measure of the amount of memory allocated when a new element is required. Larger block sizes reduce calls to memory allocation routines, but use more resources.

See also

[CAtlList Class](#)

[CHeapPtr Class](#)

[CHeapPtrElementTraits Class](#)

[Class Overview](#)

CInterfaceArray Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods useful when constructing an array of COM interface pointers.

Syntax

```
template <class I, const IID* piid=& __uuidof(I)>
class CInterfaceArray :
    public CATLArray<ATL::CComQIPtr<I, piid>,
                    CComQIPtrElementTraits<I, piid>>
```

Parameters

I

A COM interface specifying the type of pointer to be stored.

piid

A pointer to the IID of *I*.

Members

Public Constructors

NAME	DESCRIPTION
CInterfaceArray::CInterfaceArray	The constructor for the interface array.

Remarks

This class provides a constructor and derived methods for creating an array of COM interface pointers. Use [CInterfaceList](#) when a list is required.

For more information, see [ATL Collection Classes](#).

Inheritance Hierarchy

CATLArray

CInterfaceArray

Requirements

Header: atlcoll.h

CInterfaceArray::CInterfaceArray

The constructor.

```
CInterfaceArray() throw();
```

Remarks

Initializes the smart pointer array.

See also

[CAtlArray Class](#)

[CComQIPtr Class](#)

[CComQIPtrElementTraits Class](#)

[Class Overview](#)

CInterfaceList Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods useful when constructing a list of COM interface pointers.

Syntax

```
template<class I, const IID* piid =& __uuidof(I)>
class CInterfaceList
: public CATlList<ATL::CComQIPtr<I, piid>,
    CComQIPtrElementTraits<I, piid>>
```

Parameters

I

A COM interface specifying the type of pointer to be stored.

piid

A pointer to the IID of *I*.

Members

Public Constructors

NAME	DESCRIPTION
CInterfaceList::CInterfaceList	The constructor for the interface list.

Remarks

This class provides a constructor and derived methods for creating a list of COM interface pointers. Use [CInterfaceArray](#) when an array is required.

For more information, see [ATL Collection Classes](#).

Inheritance Hierarchy

[CATlList](#)

CInterfaceList

Requirements

Header: atlcoll.h

CInterfaceList::CInterfaceList

The constructor for the interface list.

```
CInterfaceList(UINT nBlockSize = 10) throw();
```

Parameters

nBlockSize

The block size, with a default of 10.

Remarks

The block size is a measure of the amount of memory allocated when a new element is required. Larger block sizes reduce calls to memory allocation routines, but use more resources.

See also

[CAtlList Class](#)

[CComQIPtr Class](#)

[CComQIPtrElementTraits Class](#)

[Class Overview](#)

CLocalHeap Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements [IAtlMemMgr](#) using the Win32 local heap functions.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CLocalHeap : public IAtlMemMgr
```

Members

Public Methods

NAME	DESCRIPTION
CLocalHeap::Allocate	Call this method to allocate a block of memory.
CLocalHeap::Free	Call this method to free a block of memory allocated by this memory manager.
CLocalHeap::GetSize	Call this method to get the allocated size of a memory block allocated by this memory manager.
CLocalHeap::Reallocate	Call this method to reallocate memory allocated by this memory manager.

Remarks

[CLocalHeap](#) implements memory allocation functions using the Win32 local heap functions.

NOTE

The local heap functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the [heap functions](#). These are available in the [CWin32Heap](#) class.

Example

See the example for [IAtlMemMgr](#).

Inheritance Hierarchy

[IAtlMemMgr](#)

[CLocalHeap](#)

Requirements

Header: `atlmem.h`

CLocalHeap::Allocate

Call this method to allocate a block of memory.

```
virtual __declspec(allocator) void* Allocate(size_t nBytes) throw();
```

Parameters

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call [CLocalHeap::Free](#) or [CLocalHeap::Reallocate](#) to free the memory allocated by this method.

Implemented using [LocalAlloc](#) with a flag parameter of `LMEM_FIXED`.

CLocalHeap::Free

Call this method to free a block of memory allocated by this memory manager.

```
virtual void Free(void* p) throw();
```

Parameters

p

Pointer to memory previously allocated by this memory manager. `NULL` is a valid value and does nothing.

Remarks

Implemented using [LocalFree](#).

CLocalHeap::GetSize

Call this method to get the allocated size of a memory block allocated by this memory manager.

```
virtual size_t GetSize(void* p) throw();
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

Return Value

Returns the size of the allocated memory block in bytes.

Remarks

Implemented using [LocalSize](#).

CLocalHeap::Reallocate

Call this method to reallocate memory allocated by this memory manager.

```
virtual __declspec(allocator) void* Reallocate(void* p, size_t nBytes) throw();
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call [CLocalHeap::Free](#) to free the memory allocated by this method.

Implemented using [LocalReAlloc](#).

See also

[Class Overview](#)

[CComHeap Class](#)

[CWin32Heap Class](#)

[CGlobalHeap Class](#)

[CCRTHeap Class](#)

[IAtlMemMgr Class](#)

CMessageMap Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class allows an object's message maps to be access by another object.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class ATL_NO_VTABLE CMessageMap
```

Members

Public Methods

NAME	DESCRIPTION
CMessageMap::ProcessWindowMessage	Accesses a message map in the <code>CMessageMap</code> -derived class.

Remarks

`CMessageMap` is an abstract base class that allows an object's message maps to be accessed by another object. In order for an object to expose its message maps, its class must derive from `CMessageMap`.

ATL uses `CMessageMap` to support contained windows and dynamic message map chaining. For example, any class containing a `CContainedWindow` object must derive from `CMessageMap`. The following code is taken from the `SUBEDIT` sample. Through `CComControl`, the `CAtlEdit` class automatically derives from `CMessageMap`.

```

class ATL_NO_VTABLE CATlEdit :
    OtherInheritedClasses
    public CComControl<CATlEdit>
    // CComControl derives from CWindowImpl, which derives from CMessageMap
{
public:
    // Declare a contained window data member
    CContainedWindow m_ctlEdit;

    // Initialize the contained window:
    // 1. Pass "Edit" to specify that the contained
    //    window should be based on the standard
    //    Windows Edit box
    // 2. Pass 'this' pointer to specify that CATlEdit
    //    contains the message map to be used for the
    //    contained window's message processing
    // 3. Pass the identifier of the message map. '1'
    //    identifies the alternate message map declared
    //    with ALT_MSG_MAP(1)
    CATlEdit()
        : m_ctlEdit(_T("Edit"), this, 1)
    {
        m_bWindowOnly = TRUE;
    }

    // Declare the default message map, identified by '0'
    BEGIN_MSG_MAP(CATlEdit)
        MESSAGE_HANDLER(WM_CREATE, OnCreate)
        MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
        CHAIN_MSG_MAP(CComControl<CATlEdit>)
    // Declare an alternate message map, identified by '1'
    ALT_MSG_MAP(1)
        MESSAGE_HANDLER(WM_CHAR, OnChar)
    END_MSG_MAP()
}

```

Because the contained window, `m_EditCtrl`, will use a message map in the containing class, `CATlEdit` derives from `CMessageMap`.

For more information about message maps, see [Message Maps](#) in the article "ATL Window Classes."

Requirements

Header: `atlwin.h`

CMessageMap::ProcessWindowMessage

Accesses the message map identified by *dwMsgMapID* in a `CMessageMap`-derived class.

```

virtual BOOL ProcessWindowMessage(
    HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam,
    LRESULT& lResult,
    DWORD dwMsgMapID) = 0;

```

Parameters

hWnd

[in] The handle to the window receiving the message.

uMsg

[in] The message sent to the window.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

lResult

[out] The result of the message processing.

dwMsgMapID

[in] The identifier of the message map that will process the message. The default message map, declared with [BEGIN_MSG_MAP](#), is identified by 0. An alternate message map, declared with [ALT_MSG_MAP\(msgMapID\)](#), is identified by `msgMapID`.

Return Value

TRUE if the message is fully handled; otherwise, FALSE.

Remarks

Called by the window procedure of a [CContainedWindow](#) object or of an object that is dynamically chaining to the message map.

See also

[CDynamicChain Class](#)

[BEGIN_MSG_MAP](#)

[ALT_MSG_MAP\(msgMapID\)](#)

[Class Overview](#)

CNonStatelessWorker Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Receives requests from a thread pool and passes them on to a worker object that is created and destroyed on each request.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class Worker>
class CNonStatelessWorker
```

Parameters

Worker

A worker thread class conforming to the [worker archetype](#) suitable for handling requests queued on [CThreadPool](#).

Members

Public Typedefs

NAME	DESCRIPTION
CNonStatelessWorker::RequestType	Implementation of WorkerArchetype::RequestType .

Public Methods

NAME	DESCRIPTION
CNonStatelessWorker::Execute	Implementation of WorkerArchetype::Execute .
CNonStatelessWorker::Initialize	Implementation of WorkerArchetype::Initialize .
CNonStatelessWorker::Terminate	Implementation of WorkerArchetype::Terminate .

Remarks

This class is a simple worker thread for use with [CThreadPool](#). This class doesn't provide any request-handling capabilities of its own. Instead, it instantiates one instance of *Worker* per request and delegates the implementation of its methods to that instance.

The benefit of this class is that it provides a convenient way to change the state model for existing worker thread classes. `CThreadPool` will create a single worker for the lifetime of the thread, so if the worker class holds state, it will hold it across multiple requests. By simply wrapping that class in the `CNonStatelessWorker` template before using it with `CThreadPool`, the lifetime of the worker and the state it holds is limited to a single request.

Requirements

Header: atlutil.h

CNonStatelessWorker::Execute

Implementation of [WorkerArchetype::Execute](#).

```
void Execute(  
    Worker::RequestType request,  
    void* pvWorkerParam,  
    OVERLAPPED* pOverlapped);
```

Remarks

This method creates an instance of the *Worker* class on the stack and calls [Initialize](#) on that object. If the initialization is successful, this method also calls [Execute](#) and [Terminate](#) on the same object.

CNonStatelessWorker::Initialize

Implementation of [WorkerArchetype::Initialize](#).

```
BOOL Initialize(void* /* pvParam */) throw();
```

Return Value

Always returns TRUE.

Remarks

This class does not do any initialization in `Initialize`.

CNonStatelessWorker::RequestType

Implementation of [WorkerArchetype::RequestType](#).

```
typedef Worker::RequestType RequestType;
```

Remarks

This class handles the same type of work item as the class used for the *Worker* template parameter. See [CNonStatelessWorker Overview](#) for details.

CNonStatelessWorker::Terminate

Implementation of [WorkerArchetype::Terminate](#).

```
void Terminate(void* /* pvParam */) throw();
```

Remarks

This class does not do any cleanup in `Terminate`.

See also

[CThreadPool Class](#)

CNoWorkerThread Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Use this class as the argument for the `MonitorClass` template parameter to cache classes if you want to disable dynamic cache maintenance.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CNoWorkerThread
```

Members

Public Methods

NAME	DESCRIPTION
CNoWorkerThread::AddHandle	Non-functional equivalent of CWorkerThread::AddHandle .
CNoWorkerThread::AddTimer	Non-functional equivalent of CWorkerThread::AddTimer .
CNoWorkerThread::GetThreadHandle	Non-functional equivalent of CWorkerThread::GetThreadHandle .
CNoWorkerThread::GetThreadId	Non-functional equivalent of CWorkerThread::GetThreadId .
CNoWorkerThread::Initialize	Non-functional equivalent of CWorkerThread::Initialize .
CNoWorkerThread::RemoveHandle	Non-functional equivalent of CWorkerThread::RemoveHandle .
CNoWorkerThread::Shutdown	Non-functional equivalent of CWorkerThread::Shutdown .

Remarks

This class provides the same public interface as [CWorkerThread](#). This interface is expected to be provided by the `MonitorClass` template parameter to cache classes.

The methods in this class are implemented to do nothing. The methods that return an HRESULT always return S_OK, and the methods that return a HANDLE or thread ID always return 0.

Requirements

Header: atlutil.h

CNoWorkerThread::AddHandle

Non-functional equivalent of [CWorkerThread::AddHandle](#).

```
HRESULT AddHandle(HANDLE /* hObject */,  
    IWorkerThreadClient* /* pClient */,  
    DWORD_PTR /* dwParam */) throw();
```

Return Value

Always returns S_OK.

Remarks

The implementation provided by this class does nothing.

CNoWorkerThread::AddTimer

Non-functional equivalent of [CWorkerThread::AddTimer](#).

```
HRESULT AddTimer(DWORD /* dwInterval */,  
    IWorkerThreadClient* /* pClient */,  
    DWORD_PTR /* dwParam */,  
    HANDLE* /* phTimer */) throw();
```

Return Value

Always returns S_OK.

Remarks

The implementation provided by this class does nothing.

CNoWorkerThread::GetThreadHandle

Non-functional equivalent of [CWorkerThread::GetThreadHandle](#).

```
HANDLE GetThreadHandle() throw();
```

Return Value

Always returns NULL.

Remarks

The implementation provided by this class does nothing.

CNoWorkerThread::GetThreadId

Non-functional equivalent of [CWorkerThread::GetThreadId](#).

```
DWORD GetThreadId() throw();
```

Return Value

Always returns 0.

Remarks

The implementation provided by this class does nothing.

CNoWorkerThread::Initialize

Non-functional equivalent of [CWorkerThread::Initialize](#).

```
HRESULT Initialize() throw();
```

Return Value

Always returns S_OK.

Remarks

The implementation provided by this class does nothing.

CNoWorkerThread::RemoveHandle

Non-functional equivalent of [CWorkerThread::RemoveHandle](#).

```
HRESULT RemoveHandle(HANDLE /* hObject */) throw();
```

Return Value

Always returns S_OK.

Remarks

The implementation provided by this class does nothing.

CNoWorkerThread::Shutdown

Non-functional equivalent of [CWorkerThread::Shutdown](#).

```
HRESULT Shutdown(DWORD dwWait = ATL_WORKER_THREAD_WAIT) throw();
```

Return Value

Always returns S_OK.

Remarks

The implementation provided by this class does nothing.

CPathT Class

3/28/2019 • 13 minutes to read • [Edit Online](#)

This class represents a path.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <typename StringType>
class CPathT
```

Parameters

StringType

The ATL/MFC string class to use for the path (see [CStringT](#)).

Members

Public Typedefs

NAME	DESCRIPTION
CPathT::PCWSTR	A constant string type.
CPathT::PWSTR	A string type.
CPathT::WCHAR	A character type.

Public Constructors

NAME	DESCRIPTION
CPathT::CPathT	The constructor for the path.

Public Methods

NAME	DESCRIPTION
CPathT::AddBackslash	Call this method to add a backslash to the end of a string to create the correct syntax for a path.
CPathT::AddExtension	Call this method to add a file extension to a path.
CPathT::Append	Call this method to append a string to the current path.
CPathT::BuildRoot	Call this method to create a root path from a given drive number.

NAME	DESCRIPTION
CPathT::Canonicalize	Call this method to convert the path to canonical form.
CPathT::Combine	Call this method to concatenate a string representing a directory name and a string representing a file path name into one path.
CPathT::CommonPrefix	Call this method to determine whether the specified path shares a common prefix with the current path.
CPathT::CompactPath	Call this method to truncate a file path to fit within a given pixel width by replacing path components with ellipses.
CPathT::CompactPathEx	Call this method to truncate a file path to fit within a given number of characters by replacing path components with ellipses.
CPathT::FileExists	Call this method to check whether the file at this path name exists.
CPathT::FindExtension	Call this method to find the position of the file extension within the path.
CPathT::FindFileName	Call this method to find the position of the file name within the path.
CPathT::GetDriveNumber	Call this method to search the path for a drive letter within the range of 'A' to 'Z' and return the corresponding drive number.
CPathT::GetExtension	Call this method to get the file extension from the path.
CPathT::IsDirectory	Call this method to check whether the path is a valid directory.
CPathT::IsFileSpec	Call this method to search a path for any path-delimiting characters (for example, ':' or '\'). If there are no path-delimiting characters present, the path is considered to be a File Spec path.
CPathT::IsPrefix	Call this method to determine whether a path contains a valid prefix of the type passed by <i>pszPrefix</i> .
CPathT::IsRelative	Call this method to determine if the path is relative.
CPathT::IsRoot	Call this method to determine if the path is a directory root.
CPathT::IsSameRoot	Call this method to determine whether another path has a common root component with the current path.
CPathT::IsUNC	Call this method to determine whether the path is a valid UNC (universal naming convention) path for a server and share.

NAME	DESCRIPTION
CPathT::IsUNCServer	Call this method to determine whether the path is a valid UNC (universal naming convention) path for a server only.
CPathT::IsUNCServerShare	Call this method to determine whether the path is a valid UNC (universal naming convention) share path, <code>\\server\share</code> .
CPathT::MakePretty	Call this method to convert a path to all lowercase characters to give the path a consistent appearance.
CPathT::MatchSpec	Call this method to search the path for a string containing a wildcard match type.
CPathT::QuoteSpaces	Call this method to enclose the path in quotation marks if it contains any spaces.
CPathT::RelativePathTo	Call this method to create a relative path from one file or folder to another.
CPathT::RemoveArgs	Call this method to remove any command-line arguments from the path.
CPathT::RemoveBackslash	Call this method to remove the trailing backslash from the path.
CPathT::RemoveBlanks	Call this method to remove all leading and trailing spaces from the path.
CPathT::RemoveExtension	Call this method to remove the file extension from the path, if there is one.
CPathT::RemoveFileSpec	Call this method to remove the trailing file name and backslash from the path, if it has them.
CPathT::RenameExtension	Call this method to replace the file name extension in the path with a new extension. If the file name does not contain an extension, the extension will be attached to the end of the string.
CPathT::SkipRoot	Call this method to parse a path, ignoring the drive letter or UNC server/share path parts.
CPathT::StripPath	Call this method to remove the path portion of a fully qualified path and file name.
CPathT::StripToRoot	Call this method to remove all parts of the path except for the root information.
CPathT::UnquoteSpaces	Call this method to remove quotation marks from the beginning and end of a path.

Public Operators

NAME	DESCRIPTION
CPathT::operator const StringType &	This operator allows the object to be treated like a string.
CPathT::operator CPathT::PCWSTR	This operator allows the object to be treated like a string.
CPathT::operator StringType &	This operator allows the object to be treated like a string.
CPathT::operator +=	This operator appends a string to the path.

Public Data Members

NAME	DESCRIPTION
CPathT::m_strPath	The path.

Remarks

`CPath` , `CPathA` , and `CPathW` are instantiations of `CPathT` defined as follows:

```
typedef CPathT< CString > CPath;
```

```
typedef CPathT< CStringA > CPathA;
```

```
typedef CPathT< CStringW > CPathW;
```

Requirements

Header: `atlpath.h`

CPathT::AddBackslash

Call this method to add a backslash to the end of a string to create the correct syntax for a path. If the path already has a trailing backslash, no backslash will be added.

```
void AddBackslash();
```

Remarks

For more information, see [PathAddBackSlash](#).

CPathT::AddExtension

Call this method to add a file extension to a path.

```
BOOL AddExtension(PCWSTR pszExtension);
```

Parameters

pszExtension
The file extension to add.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

For more information, see [PathAddExtension](#).

CPathT::Append

Call this method to append a string to the current path.

```
BOOL Append(PCXSTR pszMore);
```

Parameters

pszMore

The string to append.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

For more information, see [PathAppend](#).

CPathT::BuildRoot

Call this method to create a root path from a given drive number.

```
void BuildRoot(int iDrive);
```

Parameters

iDrive

The drive number (0 is A:, 1 is B:, and so on).

Remarks

For more information, see [PathBuildRoot](#).

CPathT::Canonicalize

Call this method to convert the path to canonical form.

```
void Canonicalize();
```

Remarks

For more information, see [PathCanonicalize](#).

CPathT::Combine

Call this method to concatenate a string representing a directory name and a string representing a file path name into one path.

```
void Combine(PCXSTR pszDir, PCXSTR pszFile);
```

Parameters

pszDir

The directory path.

pszFile

The file path.

Remarks

For more information, see [PathCombine](#).

CPathT::CommonPrefix

Call this method to determine whether the specified path shares a common prefix with the current path.

```
CPathT<StringType> CommonPrefix(PCXSTR pszOther);
```

Parameters

pszOther

The path to compare to the current one.

Return Value

Returns the common prefix.

Remarks

A prefix is one of these types: "C:\\", ".", "..", "..\\". For more information, see [PathCommonPrefix](#).

CPathT::CompactPath

Call this method to truncate a file path to fit within a given pixel width by replacing path components with ellipses.

```
BOOL CompactPath(HDC hDC, UINT nWidth);
```

Parameters

hDC

The device context used for font metrics.

nWidth

The width, in pixels, that the string will be forced to fit in.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

For more information, see [PathCompactPath](#).

CPathT::CompactPathEx

Call this method to truncate a file path to fit within a given number of characters by replacing path components with ellipses.

```
BOOL CompactPathEx(UINT nMaxChars, DWORD dwFlags = 0);
```

Parameters

nMaxChars

The maximum number of characters to be contained in the new string, including the terminating NULL

character.

dwFlags

Reserved.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

For more information, see [PathCompactPathEx](#).

CPathT::CPathT

The constructor.

```
CPathT(PCXSTR pszPath);  
CPathT(const CPathT<StringType>& path);  
CPathT() throw();
```

Parameters

pszPath

The pointer to a path string.

path

The path string.

CPathT::FileExists

Call this method to check whether the file at this path name exists.

```
BOOL FileExists() const;
```

Return Value

Returns TRUE if the file exists, FALSE otherwise.

Remarks

For more information, see [PathFileExists](#).

CPathT::FindExtension

Call this method to find the position of the file extension within the path.

```
int FindExtension() const;
```

Return Value

Returns the position of the "." preceding the extension. If no extension is found, returns -1.

Remarks

For more information, see [PathFindExtension](#).

CPathT::FindFileName

Call this method to find the position of the file name within the path.

```
int FindFileName() const;
```

Return Value

Returns the position of the file name. If no file name is found, returns -1.

Remarks

For more information, see [PathFindFileName](#).

CPathT::GetDriveNumber

Call this method to search the path for a drive letter within the range of 'A' to 'Z' and return the corresponding drive number.

```
int GetDriveNumber() const;
```

Return Value

Returns the drive number as an integer from 0 through 25 (corresponding to 'A' through 'Z') if the path has a drive letter, or -1 otherwise.

Remarks

For more information, see [PathGetDriveNumber](#).

CPathT::GetExtension

Call this method to get the file extension from the path.

```
StringType GetExtension() const;
```

Return Value

Returns the file extension.

CPathT::IsDirectory

Call this method to check whether the path is a valid directory.

```
BOOL IsDirectory() const;
```

Return Value

Returns a non-zero value (16) if the path is a directory, FALSE otherwise.

Remarks

For more information, see [PathIsDirectory](#).

CPathT::IsFileSpec

Call this method to search a path for any path-delimiting characters (for example, ':' or '\'). If there are no path-delimiting characters present, the path is considered to be a File Spec path.

```
BOOL IsFileSpec() const;
```

Return Value

Returns TRUE if there are no path-delimiting characters within the path, or FALSE if there are path-delimiting characters.

Remarks

For more information, see [PathIsFileSpec](#).

CPathT::IsPrefix

Call this method to determine whether a path contains a valid prefix of the type passed by *pszPrefix*.

```
BOOL IsPrefix(PCWSTR pszPrefix) const;
```

Parameters

pszPrefix

The prefix for which to search. A prefix is one of these types: "C:\\", ".", "..", "..\\".

Return Value

Returns TRUE if the path contains the prefix, or FALSE otherwise.

Remarks

For more information, see [PathIsPrefix](#).

CPathT::IsRelative

Call this method to determine if the path is relative.

```
BOOL IsRelative() const;
```

Return Value

Returns TRUE if the path is relative, or FALSE if it is absolute.

Remarks

For more information, see [PathIsRelative](#).

CPathT::IsRoot

Call this method to determine if the path is a directory root.

```
BOOL IsRoot() const;
```

Return Value

Returns TRUE if the path is a root, or FALSE otherwise.

Remarks

For more information, see [PathIsRoot](#).

CPathT::IsSameRoot

Call this method to determine whether another path has a common root component with the current path.

```
BOOL IsSameRoot(PCXSTR pszOther) const;
```

Parameters

pszOther

The other path.

Return Value

Returns TRUE if both strings have the same root component, or FALSE otherwise.

Remarks

For more information, see [PathIsSameRoot](#).

CPathT::IsUNC

Call this method to determine whether the path is a valid UNC (universal naming convention) path for a server and share.

```
BOOL IsUNC() const;
```

Return Value

Returns TRUE if the path is a valid UNC path, or FALSE otherwise.

Remarks

For more information, see [PathIsUNC](#).

CPathT::IsUNCServer

Call this method to determine whether the path is a valid UNC (universal naming convention) path for a server only.

```
BOOL IsUNCServer() const;
```

Return Value

Returns TRUE if the string is a valid UNC path for a server only (no share name), or FALSE otherwise.

Remarks

For more information, see [PathIsUNCServer](#).

CPathT::IsUNCServerShare

Call this method to determine whether the path is a valid UNC (universal naming convention) share path, *\\server\share*.

```
BOOL IsUNCServerShare() const;
```

Return Value

Returns TRUE if the path is in the form *\\server\share*, or FALSE otherwise.

Remarks

For more information, see [PathIsUNCServerShare](#).

CPathT::m_strPath

The path.

```
StringType m_strPath;
```

Remarks

`StringType` is the template parameter to `CPathT`.

CPathT::MakePretty

Call this method to convert a path to all lowercase characters to give the path a consistent appearance.

```
BOOL MakePretty();
```

Return Value

Returns TRUE if the path has been converted, or FALSE otherwise.

Remarks

For more information, see [PathMakePretty](#).

CPathT::MatchSpec

Call this method to search the path for a string containing a wildcard match type.

```
BOOL MatchSpec(PCXSTR pszSpec) const;
```

Parameters

pszSpec

Pointer to a null-terminated string with the file type for which to search. For example, to test whether the file at the current path is a DOC file, *pszSpec* should be set to "*.doc".

Return Value

Returns TRUE if the string matches, or FALSE otherwise.

Remarks

For more information, see [PathMatchSpec](#).

CPathT::operator +=

This operator appends a string to the path.

```
CPathT<StringType>& operator+=(PCXSTR pszMore);
```

Parameters

pszMore

The string to append.

Return Value

Returns the updated path.

CPathT::operator const StringType &

This operator allows the object to be treated like a string.

```
operator const StringType&() const throw();
```

Return Value

Returns a string representing the current path managed by this object.

CPathT::operator CPathT::PCXSTR

This operator allows the object to be treated like a string.

```
operator PCXSTR() const throw();
```

Return Value

Returns a string representing the current path managed by this object.

CPathT::operator StringType &

This operator allows the object to be treated like a string.

```
operator StringType&() throw();
```

Return Value

Returns a string representing the current path managed by this object.

CPathT::PCXSTR

A constant string type.

```
typedef StringType::PCXSTR PCXSTR;
```

Remarks

`StringType` is the template parameter to `CPathT`.

CPathT::PXSTR

A string type.

```
typedef StringType::PXSTR PXSTR;
```

Remarks

`StringType` is the template parameter to `CPathT`.

CPathT::QuoteSpaces

Call this method to enclose the path in quotation marks if it contains any spaces.

```
void QuoteSpaces();
```

Remarks

For more information, see [PathQuoteSpaces](#).

CPathT::RelativePathTo

Call this method to create a relative path from one file or folder to another.

```
BOOL RelativePathTo(  
    PCXSTR pszFrom,  
    DWORD dwAttrFrom,  
    PCXSTR pszTo,  
    DWORD dwAttrTo);
```

Parameters

pszFrom

The start of the relative path.

dwAttrFrom

The File attributes of *pszFrom*. If this value contains FILE_ATTRIBUTE_DIRECTORY, *pszFrom* is assumed to be a directory; otherwise, *pszFrom* is assumed to be a file.

pszTo

The end point of the relative path.

dwAttrTo

The File attributes of *pszTo*. If this value contains FILE_ATTRIBUTE_DIRECTORY, *pszTo* is assumed to be a directory; otherwise, *pszTo* is assumed to be a file.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

For more information, see [PathRelativePathTo](#).

CPathT::RemoveArgs

Call this method to remove any command-line arguments from the path.

```
void RemoveArgs();
```

Remarks

For more information, see [PathRemoveArgs](#).

CPathT::RemoveBackslash

Call this method to remove the trailing backslash from the path.

```
void RemoveBackslash();
```

Remarks

For more information, see [PathRemoveBackslash](#).

CPathT::RemoveBlanks

Call this method to remove all leading and trailing spaces from the path.

```
void RemoveBlanks();
```

Remarks

For more information, see [PathRemoveBlanks](#).

CPathT::RemoveExtension

Call this method to remove the file extension from the path, if there is one.

```
void RemoveExtension();
```

Remarks

For more information, see [PathRemoveExtension](#).

CPathT::RemoveFileSpec

Call this method to remove the trailing file name and backslash from the path, if it has them.

```
BOOL RemoveFileSpec();
```

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

For more information, see [PathRemoveFileSpec](#).

CPathT::RenameExtension

Call this method to replace the file name extension in the path with a new extension. If the file name does not contain an extension, the extension will be attached to the end of the path.

```
BOOL RenameExtension(PCXSTR pszExtension);
```

Parameters

pszExtension

The new file name extension, preceded by a "." character.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

For more information, see [PathRenameExtension](#).

CPathT::SkipRoot

Call this method to parse a path, ignoring the drive letter or UNC (universal naming convention) server/share path parts.

```
int SkipRoot() const;
```

Return Value

Returns the position of the beginning of the subpath that follows the root (drive letter or UNC server/share).

Remarks

For more information, see [PathSkipRoot](#).

CPathT::StripPath

Call this method to remove the path portion of a fully qualified path and file name.

```
void StripPath();
```

Remarks

For more information, see [PathStripPath](#).

CPathT::StripToRoot

Call this method to remove all parts of the path except for the root information.

```
BOOL StripToRoot();
```

Return Value

Returns TRUE if a valid drive letter was found in the path, or FALSE otherwise.

Remarks

For more information, see [PathStripToRoot](#).

CPathT::UnquoteSpaces

Call this method to remove quotation marks from the beginning and end of a path.

```
void UnquoteSpaces();
```

Remarks

For more information, see [PathUnquoteSpaces](#).

CPathT::XCHAR

A character type.

```
typedef StringType::XCHAR XCHAR;
```

Remarks

`StringType` is the template parameter to `CPathT`.

See also

[Classes](#)

[CStringT Class](#)

CPrimitiveElementTraits Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides default methods and functions for a collection class composed of primitive data types.

Syntax

```
template <typename T>
class CPrimitiveElementTraits : public CDefaultElementTraits<T>
```

Parameters

T

The type of data to be stored in the collection class object.

Members

Public Typedefs

NAME	DESCRIPTION
CPrimitiveElementTraits::INARGTYPE	The data type to use for adding elements to the collection class object.
CPrimitiveElementTraits::OUTARGTYPE	The data type to use for retrieving elements from the collection class object.

Remarks

This class provides default static functions and methods for moving, copying, comparing, and hashing primitive data type elements stored in a collection class object.

For more information, see [ATL Collection Classes](#).

Inheritance Hierarchy

[CDefaultCompareTraits](#)

[CDefaultHashTraits](#)

[CElementTraitsBase](#)

[CDefaultElementTraits](#)

[CPrimitiveElementTraits](#)

Requirements

Header: atlcoll.h

CPrimitiveElementTraits::INARGTYPE

The data type to use for adding elements to the collection class object.

```
typedef T INARGTYPE;
```

CPrimitiveElementTraits::OUTARGTYPE

The data type to use for retrieving elements from the collection class object.

```
typedef T& OUTARGTYPE;
```

See also

[CDefaultElementTraits Class](#)

[Class Overview](#)

CPrivateObjectSecurityDesc Class

3/4/2019 • 5 minutes to read • [Edit Online](#)

This class represents a private object security descriptor object.

Syntax

```
class CPrivateObjectSecurityDesc : public CSecurityDesc
```

Members

Public Constructors

NAME	DESCRIPTION
CPrivateObjectSecurityDesc::CPrivateObjectSecurityDesc	The constructor.
CPrivateObjectSecurityDesc::~~CPrivateObjectSecurityDesc	The destructor.

Public Methods

NAME	DESCRIPTION
CPrivateObjectSecurityDesc::ConvertToAutoInherit	Call this method to convert a security descriptor and its access-control lists (ACLs) to a format that supports automatic propagation of inheritable access-control entries (ACEs).
CPrivateObjectSecurityDesc::Create	Call this method to allocate and initialize a self-relative security descriptor for the private object created by the calling resource manager.
CPrivateObjectSecurityDesc::Get	Call this method to retrieve information from a private object's security descriptor.
CPrivateObjectSecurityDesc::Set	Call this method to modify a private object's security descriptor.

Operators

operator =	Assignment operator.

Remarks

This class, derived from [CSecurityDesc](#), provides methods for creating and managing the security descriptor of a private object.

For an introduction to the access control model in Windows, see [Access Control](#) in the Windows SDK.

Inheritance Hierarchy

[CSecurityDesc](#)

CPrivateObjectSecurityDesc

Requirements

Header: atlsecurity.h

CPrivateObjectSecurityDesc::ConvertToAutoInherit

Call this method to convert a security descriptor and its access-control lists (ACLs) to a format that supports automatic propagation of inheritable access-control entries (ACEs).

```
bool ConvertToAutoInherit(  
    const CSecurityDesc* pParent,  
    GUID* ObjectType,  
    bool bIsDirectoryObject,  
    PGENERIC_MAPPING GenericMapping) throw();
```

Parameters

pParent

Pointer to a [CSecurityDesc](#) object referencing the parent container of the object. If there is no parent container, this parameter is NULL.

ObjectType

Pointer to a `GUID` structure that identifies the type of object associated with the current object. Set *ObjectType* to NULL if the object does not have a GUID.

bIsDirectoryObject

Specifies whether the new object can contain other objects. A value of true indicates that the new object is a container. A value of false indicates that the new object is not a container.

GenericMapping

Pointer to a [GENERIC_MAPPING](#) structure that specifies the mapping from each generic right to specific rights for the object.

Return Value

Returns true on success, false on failure.

Remarks

This method attempts to determine whether the ACEs in the discretionary access-control list (DACL) and system access-control list (SACL) of the current security descriptor were inherited from the parent security descriptor. It calls the [ConvertToAutoInheritPrivateObjectSecurity](#) function.

CPrivateObjectSecurityDesc::CPrivateObjectSecurityDesc

The constructor.

```
CPrivateObjectSecurityDesc() throw();
```

Remarks

Initializes the `CPrivateObjectSecurityDesc` object.

CPrivateObjectSecurityDesc::~CPrivateObjectSecurityDesc

The destructor.

```
~CPrivateObjectSecurityDesc() throw();
```

Remarks

The destructor frees all allocated resources and deletes the private object's security descriptor.

CPrivateObjectSecurityDesc::Create

Call this method to allocate and initialize a self-relative security descriptor for the private object created by the calling resource manager.

```
bool Create(  
    const CSecurityDesc* pParent,  
    const CSecurityDesc* pCreator,  
    bool bIsDirectoryObject,  
    const CAccessToken& Token,  
    PGENERIC_MAPPING GenericMapping) throw();  
  
bool Create(  
    const CSecurityDesc* pParent,  
    const CSecurityDesc* pCreator,  
    GUID* ObjectType,  
    bool bIsContainerObject,  
    ULONG AutoInheritFlags,  
    const CAccessToken& Token,  
    PGENERIC_MAPPING GenericMapping) throw();
```

Parameters

pParent

Pointer to a [CSecurityDesc](#) object referencing the parent directory in which a new object is being created. Set to NULL if there is no parent directory.

pCreator

Pointer to a security descriptor provided by the creator of the object. If the object's creator does not explicitly pass security information for the new object, set this parameter to NULL.

bIsDirectoryObject

Specifies whether the new object can contain other objects. A value of true indicates that the new object is a container. A value of false indicates that the new object is not a container.

Token

Reference to the [CAccessToken](#) object for the client process on whose behalf the object is being created.

GenericMapping

Pointer to a [GENERIC_MAPPING](#) structure that specifies the mapping from each generic right to specific rights for the object.

ObjectType

Pointer to a [GUID](#) structure that identifies the type of object associated with the current object. Set *ObjectType* to NULL if the object does not have a GUID.

bIsContainerObject

Specifies whether the new object can contain other objects. A value of true indicates that the new object is a container. A value of false indicates that the new object is not a container.

AutoInheritFlags

A set of bit flags that control how access-control entries (ACEs) are inherited from *pParent*. See [CreatePrivateObjectSecurityEx](#) for more details.

Return Value

Returns true on success, false on failure.

Remarks

This method calls [CreatePrivateObjectSecurity](#) or [CreatePrivateObjectSecurityEx](#).

The second method permits specifying the object type GUID of the new object or controlling how ACEs are inherited.

NOTE

A self-relative security descriptor is a security descriptor that stores all of its security information in a contiguous block of memory.

CPrivateObjectSecurityDesc::Get

Call this method to retrieve information from a private object's security descriptor.

```
bool Get(  
    SECURITY_INFORMATION si,  
    CSecurityDesc* pResult) const throw();
```

Parameters

si

A set of bit flags that indicate the parts of the security descriptor to retrieve. This value can be a combination of the [SECURITY_INFORMATION](#) bit flags.

pResult

Pointer to a [CSecurityDesc](#) object that receives a copy of the requested information from the specified security descriptor.

Return Value

Returns true on success, false on failure.

Remarks

The security descriptor is a structure and associated data that contains the security information for a securable object.

CPrivateObjectSecurityDesc::operator =

Assignment operator.

```
CPrivateObjectSecurityDesc& operator= (const CPrivateObjectSecurityDesc& rhs) throw(...);
```

Parameters

rhs

The `CPrivateObjectSecurityDesc` object to assign to the current object.

Return Value

Returns the updated `CPrivateObjectSecurityDesc` object.

CPrivateObjectSecurityDesc::Set

Call this method to modify a private object's security descriptor.

```
bool Set(
    SECURITY_INFORMATION si,
    const CSecurityDesc& Modification,
    PGENERIC_MAPPING GenericMapping,
    const CAccessToken& Token) throw();

bool Set(
    SECURITY_INFORMATION si,
    const CSecurityDesc& Modification,
    ULONG AutoInheritFlags,
    PGENERIC_MAPPING GenericMapping,
    const CAccessToken& Token) throw();
```

Parameters

si

A set of bit flags that indicate the parts of the security descriptor to set. This value can be a combination of the [SECURITY_INFORMATION](#) bit flags.

Modification

Pointer to a [CSecurityDesc](#) object. The parts of this security descriptor indicated by the *si* parameter are applied to the object's security descriptor.

GenericMapping

Pointer to a [GENERIC_MAPPING](#) structure that specifies the mapping from each generic right to specific rights for the object.

Token

Reference to the [CAccessToken](#) object for the client process on whose behalf the object is being created.

AutoInheritFlags

A set of bit flags that control how access-control entries (ACEs) are inherited from *pParent*. See [CreatePrivateObjectSecurityEx](#) for more details.

Return Value

Returns true on success, false on failure.

Remarks

The second method permits specifying the object type GUID of the object or controlling how ACEs are inherited.

See also

[SECURITY_DESCRIPTOR](#)

[Class Overview](#)

[Security Global Functions](#)

[CSecurityDesc Class](#)

CRBMap Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

This class represents a mapping structure, using a Red-Black binary tree.

Syntax

```
template <typename K,
          typename V,
          class KTraits = CElementTraits<K>,
          class VTraits = CElementTraits<V>>
class CRBMap : public CRBTree<K, V, KTraits, VTraits>
```

Parameters

K

The key element type.

V

The value element type.

KTraits

The code used to copy or move key elements. See [CElementTraits Class](#) for more details.

VTraits

The code used to copy or move value elements.

Members

Public Constructors

NAME	DESCRIPTION
CRBMap::CRBMap	The constructor.
CRBMap::~~CRBMap	The destructor.

Public Methods

NAME	DESCRIPTION
CRBMap::Lookup	Call this method to look up keys or values in the <code>CRBMap</code> object.
CRBMap::RemoveKey	Call this method to remove an element from the <code>CRBMap</code> object, given the key.
CRBMap::SetAt	Call this method to insert an element pair into the map.

Remarks

`CRBMap` provides support for a mapping array of any given type, managing an ordered array of key elements and

their associated values. Each key can have only one associated value. Elements (consisting of a key and a value) are stored in a binary tree structure, using the [CRBMap::SetAt](#) method. Elements can be removed using the [CRBMap::RemoveKey](#) method, which deletes the element with the given key value.

Traversing the tree is made possible with methods such as [CRBTree::GetHeadPosition](#), [CRBTree::GetNext](#), and [CRBTree::GetNextValue](#).

The *KTraits* and *VTraits* parameters are traits classes that contain any supplemental code needed to copy or move elements.

`CRBMap` is derived from [CRBTree](#), which implements a binary tree using the Red-Black algorithm. [CRBMultiMap](#) is a variation that allows multiple values for each key. It too is derived from `CRBTree`, and so shares many features with `CRBMap`.

An alternative to both `CRBMap` and `CRBMultiMap` is offered by the [CAtlMap](#) class. When only a small number of elements needs to be stored, consider using the [CSimpleMap](#) class instead.

For a more complete discussion of the various collection classes and their features and performance characteristics, see [ATL Collection Classes](#).

Inheritance Hierarchy

[CRBTree](#)

`CRBMap`

Requirements

Header: `atcoll.h`

CRBMap::CRBMap

The constructor.

```
explicit CRBMap(size_t nBlockSize = 10) throw();
```

Parameters

nBlockSize

The block size.

Remarks

The *nBlockSize* parameter is a measure of the amount of memory allocated when a new element is required. Larger block sizes reduce calls to memory allocation routines, but use more resources. The default will allocate space for 10 elements at a time.

See the documentation for the base class [CRBTree](#) for information on the other methods available.

Example

```
// Define a map object which has an
// integer key, a double value, and a
// block size of 5
CRBMap<int, double> myMap(5);
```

CRBMap::~~CRBMap

The destructor.

```
~CRBMap() throw();
```

Remarks

Frees any allocated resources.

See the documentation for the base class [CRBTree](#) for information on the other methods available.

CRBMap::Lookup

Call this method to look up keys or values in the `CRBMap` object.

```
bool Lookup(KINARGTYPE key, VOUTARGTYPE value) const throw(...);  
const CPair* Lookup(KINARGTYPE key) const throw();  
CPair* Lookup(KINARGTYPE key) throw();
```

Parameters

key

Specifies the key that identifies the element to be looked up.

value

Variable that receives the looked-up value.

Return Value

The first form of the method returns true if the key is found, otherwise false. The second and third forms return a pointer to a [CPair](#).

Remarks

See the documentation for the base class [CRBTree](#) for information on the other methods available.

Example

```
// Look up the value for a key of 0  
double v;  
myMap.Lookup(0,v);
```

CRBMap::RemoveKey

Call this method to remove an element from the `CRBMap` object, given the key.

```
bool RemoveKey(KINARGTYPE key) throw();
```

Parameters

key

The key corresponding to the element pair you want to remove.

Return Value

Returns true if the key is found and removed, false on failure.

Remarks

See the documentation for the base class [CRBTree](#) for information on the other methods available.

Example

```
// Remove an element, based on the key of 0
ATLVERIFY(myMap.RemoveKey(0) == true);
```

CRBMap::SetAt

Call this method to insert an element pair into the map.

```
POSITION SetAt(
    KINARGTYPE key,
    VINARGTYPE value) throw(...);
```

Parameters

key

The key value to add to the `CRBMap` object.

value

The value to add to the `CRBMap` object.

Return Value

Returns the position of the key/value element pair in the `CRBMap` object.

Remarks

`SetAt` replaces an existing element if a matching key is found. If the key is not found, a new key/value pair is created.

See the documentation for the base class [CRBTree](#) for information on the other methods available.

Example

```
// Add an element to the map, with a key of 0
myMap.SetAt(0,1.1);
```

See also

[CRBTree Class](#)

[CAtlMap Class](#)

[CRBMultiMap Class](#)

[Class Overview](#)

CRBMultiMap Class

3/4/2019 • 5 minutes to read • [Edit Online](#)

This class represents a mapping structure that allows each key can be associated with more than one value, using a Red-Black binary tree.

Syntax

```
template<typename K,  
        typename V,  
        class KTraits = CElementTraits<K>,  
        class VTraits = CElementTraits<V>>  
class CRBMultiMap : public CRBTree<K, V, KTraits, VTraits>
```

Parameters

K

The key element type.

V

The value element type.

KTraits

The code used to copy or move key elements. See [CElementTraits Class](#) for more details.

VTraits

The code used to copy or move value elements.

Members

Public Constructors

NAME	DESCRIPTION
CRBMultiMap::CRBMultiMap	The constructor.
CRBMultiMap::~~CRBMultiMap	The destructor.

Public Methods

NAME	DESCRIPTION
CRBMultiMap::FindFirstWithKey	Call this method to find the position of the first element with a given key.
CRBMultiMap::GetNextValueWithKey	Call this method to get the value associated with a given key, and update the position value.
CRBMultiMap::GetNextWithKey	Call this method to get the element associated with a given key, and update the position value.
CRBMultiMap::Insert	Call this method to insert an element pair into the map.

NAME	DESCRIPTION
CRBMultiMap::RemoveKey	Call this method to remove all of the key/value elements for a given key.

Remarks

`CRBMultiMap` provides support for a mapping array of any given type, managing an ordered array of key elements and values. Unlike the [CRBMap](#) class, each key can be associated with more than one value.

Elements (consisting of a key and a value) are stored in a binary tree structure, using the [CRBMultiMap::Insert](#) method. Elements can be removed using the [CRBMultiMap::RemoveKey](#) method, which deletes all elements which match the given key.

Traversing the tree is made possible with methods such as [CRBTree::GetHeadPosition](#), [CRBTree::GetNext](#), and [CRBTree::GetNextValue](#). Accessing the potentially multiple values per key is possible using the [CRBMultiMap::FindFirstWithKey](#), [CRBMultiMap::GetNextValueWithKey](#), and [CRBMultiMap::GetNextWithKey](#) methods. See the example for [CRBMultiMap::CRBMultiMap](#) for an illustration of this in practice.

The *KTraits* and *VTraits* parameters are traits classes that contain any supplemental code needed to copy or move elements.

`CRBMultiMap` is derived from [CRBTree](#), which implements a binary tree using the Red-Black algorithm. An alternative to `CRBMultiMap` and `CRBMap` is offered by the [CAtiMap](#) class. When only a small number of elements needs to be stored, consider using the [CSimpleMap](#) class instead.

For a more complete discussion of the various collection classes and their features and performance characteristics, see [ATL Collection Classes](#).

Inheritance Hierarchy

[CRBTree](#)

`CRBMultiMap`

Requirements

Header: `atcoll.h`

CRBMultiMap::CRBMultiMap

The constructor.

```
explicit CRBMultiMap(size_t nBlockSize = 10) throw();
```

Parameters

nBlockSize

The block size.

Remarks

The *nBlockSize* parameter is a measure of the amount of memory allocated when a new element is required. Larger block sizes reduce calls to memory allocation routines, but use more resources. The default will allocate space for 10 elements at a time.

See the documentation for the base class [CRBTree](#) for information on the other methods available.

Example

```
// Define a multimap object which has an integer
// key, a double value, and a block size of 5
CRBMultiMap<int, double> myMap(5);

// Add some key/values. Notice how three
// different values are associated with
// one key. In a CRBMap object, the values
// would simply overwrite each other.
myMap.Insert(0, 1.1);
myMap.Insert(0, 1.2);
myMap.Insert(0, 1.3);
myMap.Insert(1, 2.1);

// Look up a key and iterate through
// all associated values

double v;
POSITION myPos = myMap.FindFirstWithKey(0);

while (myPos != NULL)
{
    v = myMap.GetNextValueWithKey(myPos,0);
    // As the loop iterates, v
    // contains the values 1.3, 1.2, 1.1
}

// Remove all of the values associated with that key
size_t i = myMap.RemoveKey(0);

// Confirm all three values were deleted
ATLASSERT(i == 3);
```

CRBMultiMap::~~CRBMultiMap

The destructor.

```
~CRBMultiMap() throw();
```

Remarks

Frees any allocated resources.

See the documentation for the base class [CRBTree](#) for information on the other methods available.

CRBMultiMap::FindFirstWithKey

Call this method to find the position of the first element with a given key.

```
POSITION FindFirstWithKey(KINARGTYPE key) const throw();
```

Parameters

key

Specifies the key that identifies the element to be found.

Return Value

Returns the POSITION of the first key/value element if the key is found, NULL otherwise.

Remarks

A key in the `CRBMultiMap` can have one or more associated values. This method will provide the position value of the first value (which may, in fact, be the only value) associated with that particular key. The position value returned can then be used with `CRBMultiMap::GetValueWithKey` or `CRBMultiMap::GetNextWithKey` to obtain the value and update the position.

See the documentation for the base class `CRBTree` for information on the other methods available.

Example

See the example for `CRBMultiMap::CRBMultiMap`.

CRBMultiMap::GetValueWithKey

Call this method to get the value associated with a given key and update the position value.

```
const V& GetValueWithKey(  
    POSITION& pos,  
    KINARGTYPE key) const throw();  
V& GetValueWithKey(  
    POSITION& pos,  
    KINARGTYPE key) throw();
```

Parameters

pos

The position value, obtained with either a call to `CRBMultiMap::FindFirstWithKey` or `CRBMultiMap::GetNextWithKey`, or a previous call to `GetValueWithKey`.

key

Specifies the key that identifies the element to be found.

Return Value

Returns the element pair associated with the given key.

Remarks

The position value is updated to point to the next value associated with the key. If no more values exist, the position value is set to NULL.

See the documentation for the base class `CRBTree` for information on the other methods available.

Example

See the example for `CRBMultiMap::CRBMultiMap`.

CRBMultiMap::GetNextWithKey

Call this method to get the element associated with a given key and update the position value.

```
const CPair* GetNextWithKey(  
    POSITION& pos,  
    KINARGTYPE key) const throw();  
CPair* GetNextWithKey(  
    POSITION& pos,  
    KINARGTYPE key) throw();
```

Parameters

pos

The position value, obtained with either a call to `CRBMultiMap::FindFirstWithKey` or

[CRBMultiMap::GetNextValueWithKey](#), or a previous call to `GetNextWithKey` .

key

Specifies the key that identifies the element to be found.

Return Value

Returns the next [CRBTree::CPair Class](#) element associated with the given key.

Remarks

The position value is updated to point to the next value associated with the key. If no more values exist, the position value is set to NULL.

See the documentation for the base class [CRBTree](#) for information on the other methods available.

CRBMultiMap::Insert

Call this method to insert an element pair into the map.

```
POSITION Insert(KINARGTYPE key, VINARGTYPE value) throw(...);
```

Parameters

key

The key value to add to the `CRBMultiMap` object.

value

The value to add to the `CRBMultiMap` object, associated with *key*.

Return Value

Returns the position of the key/value element pair in the `CRBMultiMap` object.

Remarks

See the documentation for the base class [CRBTree](#) for information on the other methods available.

Example

See the example for [CRBMultiMap::CRBMultiMap](#).

CRBMultiMap::RemoveKey

Call this method to remove all of the key/value elements for a given key.

```
size_t RemoveKey(KINARGTYPE key) throw();
```

Parameters

key

Specifies the key that identifies the element(s) to be deleted.

Return Value

Returns the number of values associated with the given key.

Remarks

`RemoveKey` deletes all of the key/value elements that have a key that matches *key*.

See the documentation for the base class [CRBTree](#) for information on the other methods available.

Example

See the example for [CRBMultiMap::CRBMultiMap](#).

See also

[CRBTree Class](#)

[CAtlMap Class](#)

[CRBMap Class](#)

[Class Overview](#)

CRBTree Class

3/4/2019 • 8 minutes to read • [Edit Online](#)

This class provides methods for creating and utilizing a Red-Black tree.

Syntax

```
template <typename K,  
          typename V,  
          class KTraits = CElementTraits<K>,  
          class VTraits = CElementTraits<V>>  
class CRBTree
```

Parameters

K

The key element type.

V

The value element type.

KTraits

The code used to copy or move key elements. See [CElementTraits Class](#) for more details.

VTraits

The code used to copy or move value elements.

Members

Public Typedefs

NAME	DESCRIPTION
CRBTree::KINARGTYPE	Type used when a key is passed as an input argument.
CRBTree::KOUTARGTYPE	Type used when a key is returned as an output argument.
CRBTree::VINARGTYPE	Type used when a value is passed as an input argument.
CRBTree::VOUTARGTYPE	Type used when a value is passed as an output argument.

Public Classes

NAME	DESCRIPTION
CRBTree::CPair Class	A class containing the key and value elements.

Public Constructors

NAME	DESCRIPTION
CRBTree::~~CRBTree	The destructor.

Public Methods

NAME	DESCRIPTION
CRBTree::FindFirstKeyAfter	Call this method to find the position of the element that uses the next available key.
CRBTree::GetAt	Call this method to get the element at a given position in the tree.
CRBTree::GetCount	Call this method to get the number of elements in the tree.
CRBTree::GetHeadPosition	Call this method to get the position value for the element at the head of the tree.
CRBTree::GetKeyAt	Call this method to get the key from a given position in the tree.
CRBTree::GetNext	Call this method to obtain a pointer to an element stored in the <code>CRBTree</code> object, and advance the position to the next element.
CRBTree::GetNextAssoc	Call this method to get the key and value of an element stored in the map and advance the position to the next element.
CRBTree::GetNextKey	Call this method to get the key of an element stored in the tree and advance the position to the next element.
CRBTree::GetNextValue	Call this method to get the value of an element stored in the tree and advance the position to the next element.
CRBTree::GetPrev	Call this method to obtain a pointer to an element stored in the <code>CRBTree</code> object, and then update the position to the previous element.
CRBTree::GetTailPosition	Call this method to get the position value for the element at the tail of the tree.
CRBTree::GetValueAt	Call this method to retrieve the value stored at a given position in the <code>CRBTree</code> object.
CRBTree::IsEmpty	Call this method to test for an empty tree object.
CRBTree::RemoveAll	Call this method to remove all elements from the <code>CRBTree</code> object.
CRBTree::RemoveAt	Call this method to remove the element at the given position in the <code>CRBTree</code> object.
CRBTree::SetValueAt	Call this method to change the value stored at a given position in the <code>CRBTree</code> object.

Remarks

A Red-Black tree is a binary search tree that uses an extra bit of information per node to ensure that it

remains "balanced," that is, the tree height doesn't grow disproportionately large and affect performance.

This template class is designed to be used by [CRBMap](#) and [CRBMultiMap](#). The bulk of the methods that make up these derived classes are provided by `CRBTree`.

For a more complete discussion of the various collection classes and their features and performance characteristics, see [ATL Collection Classes](#).

Requirements

Header: `atlcoll.h`

CRBTree::CPair Class

A class containing the key and value elements.

```
class CPair : public __POSITION
```

Remarks

This class is used by the methods [CRBTree::GetAt](#), [CRBTree::GetNext](#), and [CRBTree::GetPrev](#) to access the key and value elements stored in the tree structure.

The members are as follows:

<code>m_key</code>	The data member storing the key element.
<code>m_value</code>	The data member storing the value element.

CRBTree::~~CRBTree

The destructor.

```
~CRBTree() throw();
```

Remarks

Frees any allocated resources. Calls [CRBTree::RemoveAll](#) to delete all elements.

CRBTree::FindFirstKeyAfter

Call this method to find the position of the element that uses the next available key.

```
POSITION FindFirstKeyAfter(KINARGTYPE key) const throw();
```

Parameters

key

A key value.

Return Value

Returns the position value of the element that uses the next available key. If there are no more elements, NULL is returned.

Remarks

This method makes it easy to traverse the tree without having to calculate position values beforehand.

CRBTree::GetAt

Call this method to get the element at a given position in the tree.

```
CPair* GetAt(POSITION pos) throw();  
const CPair* GetAt(POSITION pos) const throw();  
void GetAt(POSITION pos, KOUTARGETTYPE key, VOUTARGETTYPE value) const;
```

Parameters

pos

The position value.

key

The variable that receives the key.

value

The variable that receives the value.

Return Value

The first two forms return a pointer to a [CPair](#). The third form obtains a key and a value for the given position.

Remarks

The position value can be previously determined with a call to a method such as [CRBTree::GetHeadPosition](#) or [CRBTree::GetTailPosition](#).

In debug builds, an assertion failure will occur if *pos* is equal to NULL.

CRBTree::GetCount

Call this method to get the number of elements in the tree.

```
size_t GetCount() const throw();
```

Return Value

Returns the number of elements (each key/value pair is one element) stored in the tree.

CRBTree::GetHeadPosition

Call this method to get the position value for the element at the head of the tree.

```
POSITION GetHeadPosition() const throw();
```

Return Value

Returns the position value for the element at the head of the tree.

Remarks

The value returned by `GetHeadPosition` can be used with methods such as [CRBTree::GetKeyAt](#) or [CRBTree::GetNext](#) to traverse the tree and retrieve values.

CRBTree::GetKeyAt

Call this method to get the key from a given position in the tree.

```
const K& GetKeyAt(POSITION pos) const throw();
```

Parameters

pos

The position value.

Return Value

Returns the key stored at position *pos* in the tree.

Remarks

If *pos* is not a valid position value, results are unpredictable. In debug builds, an assertion failure will occur if *pos* is equal to NULL.

CRBTree::GetNext

Call this method to obtain a pointer to an element stored in the `CRBTree` object, and advance the position to the next element.

```
const CPair* GetNext(POSITION& pos) const throw();  
CPair* GetNext(POSITION& pos) throw();
```

Parameters

pos

The position counter, returned by a previous call to methods such as [CRBTree::GetHeadPosition](#) or [CRBTree::FindFirstKeyAfter](#).

Return Value

Returns a pointer to the next [CPair](#) value in the tree.

Remarks

The *pos* position counter is updated after each call. If the retrieved element is the last in the tree, *pos* is set to NULL.

CRBTree::GetNextAssoc

Call this method to get the key and value of an element stored in the map and advance the position to the next element.

```
void GetNextAssoc(  
    POSITION& pos,  
    KOUTARGTYPE key,  
    VOUTARGTYPE value) const;
```

Parameters

pos

The position counter, returned by a previous call to methods such as [CRBTree::GetHeadPosition](#) or [CRBTree::FindFirstKeyAfter](#).

key

Template parameter specifying the type of the tree's key.

value

Template parameter specifying the type of the tree's value.

Remarks

The *pos* position counter is updated after each call. If the retrieved element is the last in the tree, *pos* is set to NULL.

CRBTree::GetNextKey

Call this method to get the key of an element stored in the tree and advance the position to the next element.

```
const K& GetNextKey(POSITION& pos) const throw();
```

Parameters

pos

The position counter, returned by a previous call to methods such as [CRBTree::GetHeadPosition](#) or [CRBTree::FindFirstKeyAfter](#).

Return Value

Returns a reference to the next key in the tree.

Remarks

Updates the current position counter, *pos*. If there are no more entries in the tree, the position counter is set to NULL.

CRBTree::GetNextValue

Call this method to get the value of an element stored in the tree and advance the position to the next element.

```
const V& GetNextValue(POSITION& pos) const throw();  
V& GetNextValue(POSITION& pos) throw();
```

Parameters

pos

The position counter, returned by a previous call to methods such as [CRBTree::GetHeadPosition](#) or [CRBTree::FindFirstKeyAfter](#).

Return Value

Returns a reference to the next value in the tree.

Remarks

Updates the current position counter, *pos*. If there are no more entries in the tree, the position counter is set to NULL.

CRBTree::GetPrev

Call this method to obtain a pointer to an element stored in the `CRBTree` object, and then update the position to the previous element.

```
const CPair* GetPrev(POSITION& pos) const throw();
CPair* GetPrev(POSITION& pos) throw();
```

Parameters

pos

The position counter, returned by a previous call to methods such as [CRBTree::GetHeadPosition](#) or [CRBTree::FindFirstKeyAfter](#).

Return Value

Returns a pointer to the previous [CPair](#) value stored in the tree.

Remarks

Updates the current position counter, *pos*. If there are no more entries in the tree, the position counter is set to NULL.

CRBTree::GetTailPosition

Call this method to get the position value for the element at the tail of the tree.

```
POSITION GetTailPosition() const throw();
```

Return Value

Returns the position value for the element at the tail of the tree.

Remarks

The value returned by `GetTailPosition` can be used with methods such as [CRBTree::GetKeyAt](#) or [CRBTree::GetPrev](#) to traverse the tree and retrieve values.

CRBTree::GetValueAt

Call this method to retrieve the value stored at a given position in the `CRBTree` object.

```
const V& GetValueAt(POSITION pos) const throw();
V& GetValueAt(POSITION pos) throw();
```

Parameters

pos

The position counter, returned by a previous call to methods such as [CRBTree::GetHeadPosition](#) or [CRBTree::FindFirstKeyAfter](#).

Return Value

Returns a reference to the value stored at the given position in the `CRBTree` object.

CRBTree::IsEmpty

Call this method to test for an empty tree object.

```
bool IsEmpty() const throw();
```

Return Value

Returns TRUE if the tree is empty, FALSE otherwise.

CRBTree::KINARGTYPE

Type used when a key is passed as an input argument.

```
typedef KTraits::INARGTYPE KINARGTYPE;
```

CRBTree::KOUTARGTYPE

Type used when a key is returned as an output argument.

```
typedef KTraits::OUTARGTYPE KOUTARGTYPE;
```

CRBTree::RemoveAll

Call this method to remove all elements from the `CRBTree` object.

```
void RemoveAll() throw();
```

Remarks

Clears out the `CRBTree` object, freeing the memory used to store the elements.

CRBTree::RemoveAt

Call this method to remove the element at the given position in the `CRBTree` object.

```
void RemoveAt(POSITION pos) throw();
```

Parameters

pos

The position counter, returned by a previous call to methods such as [CRBTree::GetHeadPosition](#) or [CRBTree::FindFirstKeyAfter](#).

Remarks

Removes the key/value pair stored at the specified position. The memory used to store the element is freed. The POSITION referenced by *pos* becomes invalid, and while the POSITION of any other elements in the tree remains valid, they do not necessarily retain the same order.

CRBTree::SetValueAt

Call this method to change the value stored at a given position in the `CRBTree` object.

```
void SetValueAt(POSITION pos, VINARGTYPE value);
```

Parameters

pos

The position counter, returned by a previous call to methods such as [CRBTree::GetHeadPosition](#) or [CRBTree::FindFirstKeyAfter](#).

value

The value to add to the `CRBTree` object.

Remarks

Changes the value element stored at the given position in the `CRBTree` object.

CRBTree::VINARGTYPE

Type used when a value is passed as an input argument.

```
typedef VTraits::INARGTYPE VINARGTYPE;
```

CRBTree::VOUTARGTYPE

Type used when a value is passed as an output argument.

```
typedef VTraits::OUTARGTYPE VOUTARGTYPE;
```

See also

[Class Overview](#)

CRegKey Class

4/1/2019 • 24 minutes to read • [Edit Online](#)

This class provides methods for manipulating entries in the system registry.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CRegKey
```

Members

Public Constructors

NAME	DESCRIPTION
CRegKey::CRegKey	The constructor.
CRegKey::~~CRegKey	The destructor.

Public Methods

NAME	DESCRIPTION
CRegKey::Attach	Call this method to attach an HKEY to the <code>CRegKey</code> object by setting the <code>m_hKey</code> member handle to <code>hKey</code> .
CRegKey::Close	Call this method to release the <code>m_hKey</code> member handle and set it to NULL.
CRegKey::Create	Call this method to create the specified key, if it does not exist as a subkey of <code>hKeyParent</code> .
CRegKey::DeleteSubKey	Call this method to remove the specified key from the registry.
CRegKey::DeleteValue	Call this method to remove a value field from <code>m_hKey</code> .
CRegKey::Detach	Call this method to detach the <code>m_hKey</code> member handle from the <code>CRegKey</code> object and set <code>m_hKey</code> to NULL.
CRegKey::EnumKey	Call this method to enumerate the subkeys of the open registry key.

NAME	DESCRIPTION
CRegKey::Flush	Call this method to write all of the attributes of the open registry key into the registry.
CRegKey::GetKeySecurity	Call this method to retrieve a copy of the security descriptor protecting the open registry key.
CRegKey::NotifyChangeKeyValue	This method notifies the caller about changes to the attributes or contents of the open registry key.
CRegKey::Open	Call this method to open the specified key and set m_hKey to the handle of this key.
CRegKey::QueryBinaryValue	Call this method to retrieve the binary data for a specified value name.
CRegKey::QueryDWORDValue	Call this method to retrieve the DWORD data for a specified value name.
CRegKey::QueryGUIDValue	Call this method to retrieve the GUID data for a specified value name.
CRegKey::QueryMultiStringValue	Call this method to retrieve the multistring data for a specified value name.
CRegKey::QueryQWORDValue	Call this method to retrieve the QWORD data for a specified value name.
CRegKey::QueryStringValue	Call this method to retrieve the string data for a specified value name.
CRegKey::QueryValue	Call this method to retrieve the data for the specified value field of m_hKey . Earlier versions of this method are no longer supported and are marked as ATL_DEPRECATED.
CRegKey::RecurseDeleteKey	Call this method to remove the specified key from the registry and explicitly remove any subkeys.
CRegKey::SetBinaryValue	Call this method to set the binary value of the registry key.
CRegKey::SetDWORDValue	Call this method to set the DWORD value of the registry key.
CRegKey::SetGUIDValue	Call this method to set the GUID value of the registry key.
CRegKey::SetKeySecurity	Call this method to set the security of the registry key.
CRegKey::SetKeyValue	Call this method to store data in a specified value field of a specified key.
CRegKey::SetMultiStringValue	Call this method to set the multistring value of the registry key.
CRegKey::SetQWORDValue	Call this method to set the QWORD value of the registry key.

NAME	DESCRIPTION
CRegKey::SetStringValue	Call this method to set the string value of the registry key.
CRegKey::SetValue	Call this method to store data in the specified value field of m_hKey . Earlier versions of this method are no longer supported and are marked as ATL_DEPRECATED.

Public Operators

NAME	DESCRIPTION
CRegKey::operator HKEY	Converts a <code>CRegKey</code> object to an HKEY.
CRegKey::operator =	Assignment operator.

Public Data Members

NAME	DESCRIPTION
CRegKey::m_hKey	Contains a handle of the registry key associated with the <code>CRegKey</code> object.
CRegKey::m_pTM	Pointer to <code>CAtlTransactionManager</code> object

Remarks

`CRegKey` provides methods for creating and deleting keys and values in the system registry. The registry contains an installation-specific set of definitions for system components, such as software version numbers, logical-to-physical mappings of installed hardware, and COM objects.

`CRegKey` provides a programming interface to the system registry for a given machine. For example, to open a particular registry key, call `CRegKey::Open`. To retrieve or modify a data value, call `CRegKey::QueryValue` or `CRegKey::SetValue`, respectively. To close a key, call `CRegKey::Close`.

When you close a key, its registry data is written (flushed) to the hard disk. This process may take several seconds. If your application must explicitly write registry data to the hard disk, you can call the [RegFlushKey](#) Win32 function. However, `RegFlushKey` uses many system resources and should be called only when absolutely necessary.

IMPORTANT

Any methods that allow the caller to specify a registry location have the potential to read data that cannot be trusted. Methods that make use of [RegQueryValueEx](#) should take into consideration that this function does not explicitly handle strings which are NULL terminated. Both conditions should be checked for by the calling code.

Requirements

Header: atlbase.h

CRegKey::Attach

Call this method to attach an HKEY to the `CRegKey` object by setting the [m_hKey](#) member handle to *hKey*.

```
void Attach(HKEY hKey) throw();
```

Parameters

hKey

The handle of a registry key.

Remarks

`Attach` will assert if `m_hKey` is non-NULL.

CRegKey::Close

Call this method to release the `m_hKey` member handle and set it to NULL.

```
LONG Close() throw();
```

Return Value

If successful, returns `ERROR_SUCCESS`; otherwise returns an error value.

CRegKey::Create

Call this method to create the specified key, if it does not exist as a subkey of *hKeyParent*.

```
LONG Create(  
    HKEY hKeyParent,  
    LPCTSTR lpszKeyName,  
    LPTSTR lpszClass = REG_NONE,  
    DWORD dwOptions = REG_OPTION_NON_VOLATILE,  
    REGSAM samDesired = KEY_READ | KEY_WRITE,  
    LPSECURITY_ATTRIBUTES lpSecAttr = NULL,  
    LPDWORD lpdwDisposition = NULL) throw();
```

Parameters

hKeyParent

The handle of an open key.

lpszKeyName

Specifies the name of a key to be created or opened. This name must be a subkey of *hKeyParent*.

lpszClass

Specifies the class of the key to be created or opened. The default value is `REG_NONE`.

dwOptions

Options for the key. The default value is `REG_OPTION_NON_VOLATILE`. For a list of possible values and descriptions, see [RegCreateKeyEx](#) in the Windows SDK.

samDesired

The security access for the key. The default value is `KEY_READ | KEY_WRITE`. For a list of possible values and descriptions, see `RegCreateKeyEx`.

lpSecAttr

A pointer to a `SECURITY_ATTRIBUTES` structure that indicates whether the handle of the key can be inherited by a child process. By default, this parameter is NULL (meaning the handle cannot be inherited).

lpdwDisposition

[out] If non-NULL, retrieves either REG_CREATED_NEW_KEY (if the key did not exist and was created) or REG_OPENED_EXISTING_KEY (if the key existed and was opened).

Return Value

If successful, returns ERROR_SUCCESS and opens the key. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

Remarks

`Create` sets the `m_hKey` member to the handle of this key.

CRegKey::CRegKey

The constructor.

```
CRegKey() throw();  
CRegKey(CRegKey& key) throw();  
explicit CRegKey(HKEY hKey) throw();  
CRegKey(CAtlTransactionManager* pTM) throw();
```

Parameters

key

A reference to a `CRegKey` object.

hKey

A handle to a registry key.

pTM

Pointer to CAtlTransactionManager object

Remarks

Creates a new `CRegKey` object. The object can be created from an existing `CRegKey` object, or from a handle to a registry key.

CRegKey::~~CRegKey

The destructor.

```
~CRegKey() throw();
```

Remarks

The destructor releases `m_hKey`.

CRegKey::DeleteSubKey

Call this method to remove the specified key from the registry.

```
LONG DeleteSubKey(LPCTSTR lpszSubKey) throw();
```

Parameters

lpszSubKey

Specifies the name of the key to delete. This name must be a subkey of `m_hKey`.

Return Value

If successful, returns `ERROR_SUCCESS`. If the method fails, the return value is a nonzero error code defined in `WINERROR.H`.

Remarks

`DeleteSubKey` can only delete a key that has no subkeys. If the key has subkeys, call [RecurseDeleteKey](#) instead.

CRegKey::DeleteValue

Call this method to remove a value field from [m_hKey](#).

```
LONG DeleteValue(LPCTSTR lpszValue) throw();
```

Parameters

lpszValue

Specifies the value field to remove.

Return Value

If successful, returns `ERROR_SUCCESS`. If the method fails, the return value is a nonzero error code defined in `WINERROR.H`.

CRegKey::Detach

Call this method to detach the [m_hKey](#) member handle from the `CRegKey` object and set `m_hKey` to `NULL`.

```
HKEY Detach() throw();
```

Return Value

The `HKEY` associated with the `CRegKey` object.

CRegKey::EnumKey

Call this method to enumerate the subkeys of the open registry key.

```
LONG EnumKey(  
    DWORD iIndex,  
    LPTSTR pszName,  
    LPDWORD pNameLength,  
    FILETIME* pftLastWriteTime = NULL) throw();
```

Parameters

iIndex

The subkey index. This parameter should be zero for the first call and then incremented for subsequent calls

pszName

Pointer to a buffer that receives the name of the subkey, including the terminating null character. Only the name of the subkey is copied to the buffer, not the full key hierarchy.

pNameLength

Pointer to a variable that specifies the size, in `TCHARs`, of the buffer specified by the *pszName* parameter. This size should include the terminating null character. When the method returns, the variable pointed to by *pNameLength* contains the number of characters stored in the buffer. The count returned does not include the terminating null character.

pftLastWriteTime

Pointer to a variable that receives the time the enumerated subkey was last written to.

Return Value

If the method succeeds, the return value is `ERROR_SUCCESS`. If the method fails, the return value is a nonzero error code defined in `WINERROR.H`.

Remarks

To enumerate the subkeys, call `CRegKey::EnumKey` with an index of zero. Increment the index value and repeat until the method returns `ERROR_NO_MORE_ITEMS`. For more information, see [RegEnumKeyEx](#) in the Windows SDK.

CRegKey::Flush

Call this method to write all of the attributes of the open registry key into the registry.

```
LONG Flush() throw();
```

Return Value

If the method succeeds, the return value is `ERROR_SUCCESS`. If the method fails, the return value is a nonzero error code defined in `WINERROR.H`.

Remarks

For more information, see [RegEnumFlush](#) in the Windows SDK.

CRegKey::GetKeySecurity

Call this method to retrieve a copy of the security descriptor protecting the open registry key.

```
LONG GetKeySecurity(  
    SECURITY_INFORMATION si,  
    PSECURITY_DESCRIPTOR psd,  
    LPDWORD pnBytes) throw();
```

Parameters

si

The [SECURITY_INFORMATION](#) value that indicates the requested security information.

psd

A pointer to a buffer that receives a copy of the requested security descriptor.

pnBytes

The size, in bytes, of the buffer pointed to by *psd*.

Return Value

If the method succeeds, the return value is `ERROR_SUCCESS`. If the method fails, the return value is a nonzero error code is defined in `WINERROR.H`.

Remarks

For more information, see [RegGetKeySecurity](#).

CRegKey::m_hKey

Contains a handle of the registry key associated with the `CRegKey` object.

```
HKEY m_hKey;
```

CRegKey::m_pTM

Pointer to a `CAtlTransactionManager` object.

```
CAtlTransactionManager* m_pTM;
```

Remarks

CRegKey::NotifyChangeKeyValue

This method notifies the caller about changes to the attributes or contents of the open registry key.

```
LONG NotifyChangeKeyValue(  
    BOOL bWatchSubtree,  
    DWORD dwNotifyFilter,  
    HANDLE hEvent,  
    BOOL bAsync = TRUE) throw();
```

Parameters

bWatchSubtree

Specifies a flag that indicates whether to report changes in the specified key and all of its subkeys or only in the specified key. If this parameter is `TRUE`, the method reports changes in the key and its subkeys. If the parameter is `FALSE`, the method reports changes only in the key.

dwNotifyFilter

Specifies a set of flags that control which changes should be reported. This parameter can be a combination of the following values:

VALUE	MEANING
REG_NOTIFY_CHANGE_NAME	Notify the caller if a subkey is added or deleted.
REG_NOTIFY_CHANGE_ATTRIBUTES	Notify the caller of changes to the attributes of the key, such as the security descriptor information.
REG_NOTIFY_CHANGE_LAST_SET	Notify the caller of changes to a value of the key. This can include adding or deleting a value, or changing an existing value.
REG_NOTIFY_CHANGE_SECURITY	Notify the caller of changes to the security descriptor of the key.

hEvent

Handle to an event. If the *bAsync* parameter is `TRUE`, the method returns immediately and changes are reported by signaling this event. If *bAsync* is `FALSE`, *hEvent* is ignored.

bAsync

Specifies a flag that indicates how the method reports changes. If this parameter is `TRUE`, the method returns immediately and reports changes by signaling the specified event. When this parameter is `FALSE`, the method does not return until a change has occurred. If *hEvent* does not specify a valid event, the *bAsync* parameter cannot be `TRUE`.

Return Value

If the method succeeds, the return value is `ERROR_SUCCESS`. If the method fails, the return value is a nonzero error code defined in `WINERROR.H`.

Remarks

NOTE

This method does not notify the caller if the specified key is deleted.

For more details and a sample program, see [RegNotifyChangeKeyValue](#).

CRegKey::Open

Call this method to open the specified key and set `m_hKey` to the handle of this key.

```
LONG Open(  
    HKEY hKeyParent,  
    LPCTSTR lpszKeyName,  
    REGSAM samDesired = KEY_READ | KEY_WRITE) throw();
```

Parameters

hKeyParent

The handle of an open key.

lpszKeyName

Specifies the name of a key to be created or opened. This name must be a subkey of *hKeyParent*.

samDesired

The security access for the key. The default value is `KEY_ALL_ACCESS`. For a list of possible values and descriptions, see [RegCreateKeyEx](#) in the Windows SDK.

Return Value

If successful, returns `ERROR_SUCCESS`; otherwise, a non-zero error value defined in `WINERROR.H`.

Remarks

If the *lpszKeyName* parameter is `NULL` or points to an empty string, `Open` opens a new handle of the key identified by *hKeyParent*, but does not close any previously opened handle.

Unlike [CRegKey::Create](#), `Open` will not create the specified key if it does not exist.

CRegKey::operator HKEY

Converts a `CRegKey` object to an `HKEY`.

```
operator HKEY() const throw();
```

CRegKey::operator =

Assignment operator.

```
CRegKey& operator= (CRegKey& key) throw();
```

Parameters

key

The key to copy.

Return Value

Returns a reference to the new key.

Remarks

This operator detaches *key* from its current object and assigns it to the `CRegKey` object instead.

CRegKey::QueryBinaryValue

Call this method to retrieve the binary data for a specified value name.

```
LONG QueryBinaryValue(  
    LPCTSTR pszValueName,  
    void* pValue,  
    ULONG* pnBytes) throw();
```

Parameters

pszValueName

Pointer to a null-terminated string containing the name of the value to query.

pValue

Pointer to a buffer that receives the value's data.

pnBytes

Pointer to a variable that specifies the size, in bytes, of the buffer pointed to by the *pValue* parameter. When the method returns, this variable contains the size of the data copied to the buffer.

Return Value

If the method succeeds, `ERROR_SUCCESS` is returned. If the method fails to read a value, it returns a nonzero error code defined in `WINERROR.H`. If the data referenced is not of type `REG_BINARY`, `ERROR_INVALID_DATA` is returned.

Remarks

This method makes use of `RegQueryValueEx` and confirms that the correct type of data is returned. See [RegQueryValueEx](#) for more details.

IMPORTANT

This method allows the caller to specify any registry location, potentially reading data which cannot be trusted. Also, the [RegQueryValueEx](#) function used by this method does not explicitly handle strings which are NULL terminated. Both conditions should be checked for by the calling code.

CRegKey::QueryDWORDValue

Call this method to retrieve the DWORD data for a specified value name.

```
LONG QueryDWORDValue(  
    LPCTSTR pszValueName,  
    DWORD& dwValue) throw();
```

Parameters

pszValueName

Pointer to a null-terminated string containing the name of the value to query.

dwValue

Pointer to a buffer that receives the DWORD.

Return Value

If the method succeeds, `ERROR_SUCCESS` is returned. If the method fails to read a value, it returns a nonzero error code defined in `WINERROR.H`. If the data referenced is not of type `REG_DWORD`, `ERROR_INVALID_DATA` is returned.

Remarks

This method makes use of `RegQueryValueEx` and confirms that the correct type of data is returned. See [RegQueryValueEx](#) for more details.

IMPORTANT

This method allows the caller to specify any registry location, potentially reading data which cannot be trusted. Also, the [RegQueryValueEx](#) function used by this method does not explicitly handle strings which are NULL terminated. Both conditions should be checked for by the calling code.

CRegKey::QueryGUIDValue

Call this method to retrieve the GUID data for a specified value name.

```
LONG QueryGUIDValue(  
    LPCTSTR pszValueName,  
    GUID& guidValue) throw();
```

Parameters

pszValueName

Pointer to a null-terminated string containing the name of the value to query.

guidValue

Pointer to a variable that receives the GUID.

Return Value

If the method succeeds, `ERROR_SUCCESS` is returned. If the method fails to read a value, it returns a nonzero error code defined in `WINERROR.H`. If the data referenced is not a valid GUID, `ERROR_INVALID_DATA` is returned.

Remarks

This method makes use of `CRegKey::QueryStringValue` and converts the string into a GUID using [CLSIDFromString](#).

IMPORTANT

This method allows the caller to specify any registry location, potentially reading data which cannot be trusted.

CRegKey::QueryMultiStringValue

Call this method to retrieve the multistring data for a specified value name.

```
LONG QueryMultiStringValue(  
    LPCTSTR pszValueName,  
    LPTSTR pszValue,  
    ULONG* pnChars) throw();
```

Parameters

pszValueName

Pointer to a null-terminated string containing the name of the value to query.

pszValue

Pointer to a buffer that receives the multistring data. A multistring is an array of null-terminated strings, terminated by two null characters.

pnChars

The size, in TCHARs, of the buffer pointed to by *pszValue*. When the method returns, *pnChars* contains the size, in TCHARs, of the multistring retrieved, including a terminating null character.

Return Value

If the method succeeds, `ERROR_SUCCESS` is returned. If the method fails to read a value, it returns a nonzero error code defined in `WINERROR.H`. If the data referenced is not of type `REG_MULTI_SZ`, `ERROR_INVALID_DATA` is returned.

Remarks

This method makes use of `RegQueryValueEx` and confirms that the correct type of data is returned. See [RegQueryValueEx](#) for more details.

IMPORTANT

This method allows the caller to specify any registry location, potentially reading data which cannot be trusted. Also, the [RegQueryValueEx](#) function used by this method does not explicitly handle strings which are NULL terminated. Both conditions should be checked for by the calling code.

CRegKey::QueryQWORDValue

Call this method to retrieve the QWORD data for a specified value name.

```
LONG QueryQWORDValue(  
    LPCTSTR pszValueName,  
    ULONGLONG& qwValue) throw();
```

Parameters

pszValueName

Pointer to a null-terminated string containing the name of the value to query.

qwValue

Pointer to a buffer that receives the QWORD.

Return Value

If the method succeeds, `ERROR_SUCCESS` is returned. If the method fails to read a value, it returns a nonzero error code defined in `WINERROR.H`. If the data referenced is not of type `REG_QWORD`, `ERROR_INVALID_DATA` is returned.

Remarks

This method makes use of `RegQueryValueEx` and confirms that the correct type of data is returned. See [RegQueryValueEx](#) for more details.

IMPORTANT

This method allows the caller to specify any registry location, potentially reading data which cannot be trusted. Also, the [RegQueryValueEx](#) function used by this method does not explicitly handle strings which are NULL terminated. Both conditions should be checked for by the calling code.

CRegKey::QueryStringValue

Call this method to retrieve the string data for a specified value name.

```
LONG QueryStringValue(  
    LPCTSTR pszValueName,  
    LPTSTR pszValue,  
    ULONG* pnChars) throw();
```

Parameters

pszValueName

Pointer to a null-terminated string containing the name of the value to query.

pszValue

Pointer to a buffer that receives the string data.

pnChars

The size, in TCHARs, of the buffer pointed to by *pszValue*. When the method returns, *pnChars* contains the size, in TCHARs, of the string retrieved, including a terminating null character.

Return Value

If the method succeeds, `ERROR_SUCCESS` is returned. If the method fails to read a value, it returns a nonzero error code defined in `WINERROR.H`. If the data referenced is not of type `REG_SZ`, `ERROR_INVALID_DATA` is returned. If the method returns `ERROR_MORE_DATA`, *pnChars* equals zero, not the required buffer size in bytes.

Remarks

This method makes use of `RegQueryValueEx` and confirms that the correct type of data is returned. See [RegQueryValueEx](#) for more details.

IMPORTANT

This method allows the caller to specify any registry location, potentially reading data which cannot be trusted. Also, the [RegQueryValueEx](#) function used by this method does not explicitly handle strings which are NULL terminated. Both conditions should be checked for by the calling code.

CRegKey::QueryValue

Call this method to retrieve the data for the specified value field of [m_hKey](#). Earlier versions of this method are no longer supported and are marked as `ATL_DEPRECATED`.

```

LONG QueryValue(
    LPCTSTR pszValueName,
    DWORD* pdwType,
    void* pData,
    ULONG* pnBytes) throw();

ATL_DEPRECATED LONG QueryValue(
    DWORD& dwValue,
    LPCTSTR lpszValueName);

ATL_DEPRECATED LONG QueryValue(
    LPTSTR szValue,
    LPCTSTR lpszValueName,
    DWORD* pdwCount);

```

Parameters

pszValueName

Pointer to a null-terminated string containing the name of the value to query. If *pszValueName* is NULL or an empty string, "", the method retrieves the type and data for the key's unnamed or default value, if any.

pdwType

Pointer to a variable that receives a code indicating the type of data stored in the specified value. The *pdwType* parameter can be NULL if the type code is not required.

pData

Pointer to a buffer that receives the value's data. This parameter can be NULL if the data is not required.

pnBytes

Pointer to a variable that specifies the size, in bytes, of the buffer pointed to by the *pData* parameter. When the method returns, this variable contains the size of the data copied to *pData*.

dwValue

The value field's numerical data.

lpszValueName

Specifies the value field to be queried.

szValue

The value field's string data.

pdwCount

The size of the string data. Its value is initially set to the size of the *szValue* buffer.

Return Value

If successful, returns ERROR_SUCCESS; otherwise, a nonzero error code defined in WINERROR.H.

Remarks

The two original versions of `QueryValue` are no longer supported and are marked as ATL_DEPRECATED. The compiler will issue a warning if these forms are used.

The remaining method calls RegQueryValueEx.

IMPORTANT

This method allows the caller to specify any registry location, potentially reading data which cannot be trusted. Also, the RegQueryValueEx function used by this method does not explicitly handle strings which are NULL terminated. Both conditions should be checked for by the calling code.

CRegKey::RecurseDeleteKey

Call this method to remove the specified key from the registry and explicitly remove any subkeys.

```
LONG RecurseDeleteKey(LPCTSTR lpszKey) throw();
```

Parameters

lpszKey

Specifies the name of the key to delete. This name must be a subkey of [m_hKey](#).

Return Value

If successful, returns ERROR_SUCCESS; otherwise, a non-zero error value defined in WINERROR.H.

Remarks

If the key has subkeys, you must call this method to delete the key.

CRegKey::SetBinaryValue

Call this method to set the binary value of the registry key.

```
LONG SetBinaryValue(
    LPCTSTR pszValueName,
    const void* pValue,
    ULONG nBytes) throw();
```

Parameters

pszValueName

Pointer to a string containing the name of the value to set. If a value with this name is not already present, the method adds it to the key.

pValue

Pointer to a buffer containing the data to be stored with the specified value name.

nBytes

Specifies the size, in bytes, of the information pointed to by the *pValue* parameter.

Return Value

If the method succeeds, the return value is ERROR_SUCCESS. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

Remarks

This method uses [RegSetValueEx](#) to write the value to the registry.

CRegKey::SetDWORDValue

Call this method to set the DWORD value of the registry key.

```
LONG SetDWORDValue(LPCTSTR pszValueName, DWORD dwValue) throw();
```

Parameters

pszValueName

Pointer to a string containing the name of the value to set. If a value with this name is not already present, the method adds it to the key.

dwValue

The DWORD data to be stored with the specified value name.

Return Value

If the method succeeds, the return value is `ERROR_SUCCESS`. If the method fails, the return value is a nonzero error code defined in `WINERROR.H`.

Remarks

This method uses [RegSetValueEx](#) to write the value to the registry.

CRegKey::SetGUIDValue

Call this method to set the GUID value of the registry key.

```
LONG SetGUIDValue(LPCTSTR pszValueName, REFGUID guidValue) throw();
```

Parameters

pszValueName

Pointer to a string containing the name of the value to set. If a value with this name is not already present, the method adds it to the key.

guidValue

Reference to the GUID to be stored with the specified value name.

Return Value

If the method succeeds, the return value is `ERROR_SUCCESS`. If the method fails, the return value is a nonzero error code defined in `WINERROR.H`.

Remarks

This method makes use of `CRegKey::SetStringValue` and converts the GUID into a string using [StringFromGUID2](#).

CRegKey::SetKeyValue

Call this method to store data in a specified value field of a specified key.

```
LONG SetKeyValue(  
    LPCTSTR lpszKeyName,  
    LPCTSTR lpszValue,  
    LPCTSTR lpszValueName = NULL) throw();
```

Parameters

lpszKeyName

Specifies the name of the key to be created or opened. This name must be a subkey of [m_hKey](#).

lpszValue

Specifies the data to be stored. This parameter must be non-NULL.

lpszValueName

Specifies the value field to be set. If a value field with this name does not already exist in the key, it is added.

Return Value

If successful, returns `ERROR_SUCCESS`; otherwise, a nonzero error code defined in `WINERROR.H`.

Remarks

Call this method to create or open the *lpzKeyName* key and store the *lpzValue* data in the *lpzValueName* value field.

CRegKey::SetKeySecurity

Call this method to set the security of the registry key.

```
LONG SetKeySecurity(SEURITY_INFORMATION si, PSECURITY_DESCRIPTOR psd) throw();
```

Parameters

si

Specifies the components of the security descriptor to set. The value can be a combination of the following values:

VALUE	MEANING
DACL_SECURITY_INFORMATION	Sets the key's discretionary access-control list (DACL). The key must have WRITE_DAC access, or the calling process must be the object's owner.
GROUP_SECURITY_INFORMATION	Sets the key's primary group security identifier (SID). The key must have WRITE_OWNER access, or the calling process must be the object's owner.
OWNER_SECURITY_INFORMATION	Sets the key's owner SID. The key must have WRITE_OWNER access, or the calling process must be the object's owner or have the SE_TAKE_OWNERSHIP_NAME privilege enabled.
SACL_SECURITY_INFORMATION	Sets the key's system access-control list (SACL). The key must have ACCESS_SYSTEM_SECURITY access. The proper way to get this access is to enable the SE_SECURITY_NAME privilege in the caller's current access token, open the handle for ACCESS_SYSTEM_SECURITY access, and then disable the privilege.

psd

Pointer to a [SECURITY_DESCRIPTOR](#) structure that specifies the security attributes to set for the specified key.

Return Value

If the method succeeds, the return value is ERROR_SUCCESS. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

Remarks

Sets the key's security attributes. See [RegSetKeySecurity](#) for more details.

CRegKey::SetMultiStringValue

Call this method to set the multistring value of the registry key.

```
LONG SetMultiStringValue(LPCTSTR pszValueName, LPCTSTR pszValue) throw();
```

Parameters

pszValueName

Pointer to a string containing the name of the value to set. If a value with this name is not already present, the method adds it to the key.

pszValue

Pointer to the multistring data to be stored with the specified value name. A multistring is an array of null-terminated strings, terminated by two null characters.

Return Value

If the method succeeds, the return value is `ERROR_SUCCESS`. If the method fails, the return value is a nonzero error code defined in `WINERROR.H`.

Remarks

This method uses [RegSetValueEx](#) to write the value to the registry.

CRegKey::SetQWORDValue

Call this method to set the QWORD value of the registry key.

```
LONG SetQWORDValue(LPCTSTR pszValueName, ULONGLONG qwValue) throw();
```

Parameters

pszValueName

Pointer to a string containing the name of the value to set. If a value with this name is not already present, the method adds it to the key.

qwValue

The QWORD data to be stored with the specified value name.

Return Value

If the method succeeds, the return value is `ERROR_SUCCESS`. If the method fails, the return value is a nonzero error code defined in `WINERROR.H`.

Remarks

This method uses [RegSetValueEx](#) to write the value to the registry.

CRegKey::SetStringValue

Call this method to set the string value of the registry key.

```
LONG SetStringValue(  
    LPCTSTR pszValueName,  
    LPCTSTR pszValue,  
    DWORD dwType = REG_SZ) throw();
```

Parameters

pszValueName

Pointer to a string containing the name of the value to set. If a value with this name is not already present, the method adds it to the key.

pszValue

Pointer to the string data to be stored with the specified value name.

dwType

The type of the string to write to the registry: either `REG_SZ` (the default) or `REG_EXPAND_SZ` (for multistrings).

Return Value

If the method succeeds, the return value is `ERROR_SUCCESS`. If the method fails, the return value is a nonzero

error code defined in WINERROR.H.

Remarks

This method uses [RegSetValueEx](#) to write the value to the registry.

CRegKey::SetValue

Call this method to store data in the specified value field of [m_hKey](#). Earlier versions of this method are no longer supported and are marked as ATL_DEPRECATED.

```
LONG SetValue(
    LPCTSTR pszValueName,
    DWORD dwType,
    const void* pValue,
    ULONG nBytes) throw();

static LONG WINAPI SetValue(
    HKEY hKeyParent,
    LPCTSTR lpszKeyName,
    LPCTSTR lpszValue,
    LPCTSTR lpszValueName = NULL);

ATL_DEPRECATED LONG SetValue(
    DWORD dwValue,
    LPCTSTR lpszValueName);

ATL_DEPRECATED LONG SetValue(
    LPCTSTR lpszValue,
    LPCTSTR lpszValueName = NULL,
    bool bMulti = false,
    int nValueLen = -1);
```

Parameters

pszValueName

Pointer to a string containing the name of the value to set. If a value with this name is not already present in the key, the method adds it to the key. If *pszValueName* is NULL or an empty string, "", the method sets the type and data for the key's unnamed or default value.

dwType

Specifies a code indicating the type of data pointed to by the *pValue* parameter.

pValue

Pointer to a buffer containing the data to be stored with the specified value name.

nBytes

Specifies the size, in bytes, of the information pointed to by the *pValue* parameter. If the data is of type REG_SZ, REG_EXPAND_SZ, or REG_MULTI_SZ, *nBytes* must include the size of the terminating null character.

hKeyParent

The handle of an open key.

lpszKeyName

Specifies the name of a key to be created or opened. This name must be a subkey of *hKeyParent*.

lpszValue

Specifies the data to be stored. This parameter must be non-NULL.

lpszValueName

Specifies the value field to be set. If a value field with this name does not already exist in the key, it is added.

dwValue

Specifies the data to be stored.

bMulti

If false, indicates the string is of type REG_SZ. If true, indicates the string is a multistring of type REG_MULTI_SZ.

nValueLen

If *bMulti* is true, *nValueLen* is the length of the *lpzValue* string in characters. If *bMulti* is false, a value of -1 indicates that the method will calculate the length automatically.

Return Value

If successful, returns ERROR_SUCCESS; otherwise, a nonzero error code defined in WINERROR.H.

Remarks

The two original versions of `SetValue` are marked as ATL_DEPRECATED and should no longer be used. The compiler will issue a warning if these forms are used.

The third method calls [RegSetValueEx](#).

See also

[DCOM Sample](#)

[Class Overview](#)

CRTThreadTraits Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides the creation function for a CRT thread. Use this class if the thread will use CRT functions.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CRTThreadTraits
```

Members

Public Methods

NAME	DESCRIPTION
CRTThreadTraits::CreateThread	(Static) Call this function to create a thread that can use CRT functions.

Remarks

Thread traits are classes that provide a creation function for a particular type of thread. The creation function has the same signature and semantics as the Windows [CreateThread](#) function.

Thread traits are used by the following classes:

- [CThreadPool](#)
- [CWorkerThread](#)

If the thread will not be using CRT functions, use [Win32ThreadTraits](#) instead.

Requirements

Header: `atlbase.h`

CRTThreadTraits::CreateThread

Call this function to create a thread that can use CRT functions.

```
static HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES lpsa,  
    DWORD dwStackSize,  
    LPTHREAD_START_ROUTINE pfnThreadProc,  
    void* pvParam,  
    DWORD dwCreationFlags,  
    DWORD* pdwThreadId) throw();
```

Parameters

lpSa

The security attributes for the new thread.

dwStackSize

The stack size for the new thread.

pfnThreadProc

The thread procedure of the new thread.

pvParam

The parameter to be passed to the thread procedure.

dwCreationFlags

The creation flags (0 or CREATE_SUSPENDED).

pdwThreadId

[out] Address of the DWORD variable that, on success, receives the thread ID of the newly created thread.

Return Value

Returns the handle to the newly created thread or NULL on failure. Call [GetLastError](#) to get extended error information.

Remarks

See [CreateThread](#) for further information on the parameters to this function.

This function calls [_beginthreadex](#) to create the thread.

See also

[Class Overview](#)

CSacl Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

This class is a wrapper for a SACL (system access-control list) structure.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CSacl : public CAcl
```

Members

Public Constructors

NAME	DESCRIPTION
CSacl::CSacl	The constructor.
CSacl::~~CSacl	The destructor.

Public Methods

NAME	DESCRIPTION
CSacl::AddAuditAce	Adds an audit access-control entry (ACE) to the <code>CSacl</code> object.
CSacl::GetAceCount	Returns the number of access-control entries (ACEs) in the <code>CSacl</code> object.
CSacl::RemoveAce	Removes a specific ACE (access-control entry) from the <code>CSacl</code> object.
CSacl::RemoveAllAces	Removes all of the ACEs contained in the <code>CSacl</code> object.

Public Operators

NAME	DESCRIPTION
CSacl::operator =	Assignment operator.

Remarks

A SACL contains access-control entries (ACEs) that specify the types of access attempts that generate audit records in the security event log of a domain controller. Note that a SACL generates log entries only on the

domain controller where the access attempt occurred, not on every domain controller that contains a replica of the object.

To set or retrieve the SACL in an object's security descriptor, the SE_SECURITY_NAME privilege must be enabled in the access token of the requesting thread. The administrators group has this privilege granted by default, and it can be granted to other users or groups. Having the privilege granted is not all that is required: before the operation defined by the privilege can be performed, the privilege must be enabled in the security access token in order to take effect. The model allows privileges to be enabled only for specific system operations, and then disabled when they are no longer needed. See [AtlGetSacl](#) and [AtlSetSacl](#) for examples of enabling SE_SECURITY_NAME.

Use the class methods provided to add, remove, create, and delete ACEs from the `SACL` object. See also [AtlGetSacl](#) and [AtlSetSacl](#).

For an introduction to the access control model in Windows, see [Access Control](#) in the Windows SDK.

Inheritance Hierarchy

[CAcl](#)

`CSacl`

Requirements

Header: atlsecurity.h

CSacl::AddAuditAce

Adds an audit access-control entry (ACE) to the `CSacl` object.

```
bool AddAuditAce(
    const CSid& rSid,
    ACCESS_MASK AccessMask,
    bool bSuccess,
    bool bFailure,
    BYTE AceFlags = 0) throw(...);

bool AddAuditAce(
    const CSid& rSid,
    ACCESS_MASK AccessMask,
    bool bSuccess,
    bool bFailure,
    BYTE AceFlags,
    const GUID* pObjectType,
    const GUID* pInheritedObjectType) throw(...);
```

Parameters

rSid

The [CSid](#) object.

AccessMask

Specifies the mask of access rights to be audited for the specified `csid` object.

bSuccess

Specifies whether allowed access attempts are to be audited. Set this flag to true to enable auditing; otherwise, set it to false.

bFailure

Specifies whether denied access attempts are to be audited. Set this flag to true to enable auditing; otherwise, set it to false.

AceFlags

A set of bit flags that control ACE inheritance.

pObjectType

The object type.

pInheritedObjectType

The inherited object type.

Return Value

Returns TRUE if the ACE is added to the `CSacl` object, FALSE on failure.

Remarks

A `CSacl` object contains access-control entries (ACEs) that specify the types of access attempts that generate audit records in the security event log. This method adds such an ACE to the `CSacl` object.

See [ACE_HEADER](#) for a description of the various flags which can be set in the *AceFlags* parameter.

CSacl::CSacl

The constructor.

```
CSacl() throw();  
CSacl(const ACL& rhs) throw(...);
```

Parameters

rhs

An existing `ACL` (access-control list) structure.

Remarks

The `CSacl` object can be optionally created using an existing `ACL` structure. Ensure that this parameter is a system access-control list (SACL) and not a discretionary access-control list (DACL). In debug builds, if a DACL is supplied an assertion will occur. In release builds any entries from a DACL are ignored.

CSacl::~CSacl

The destructor.

```
~CSacl() throw();
```

Remarks

The destructor frees any resources acquired by the object, including all access-control entries (ACEs).

CSacl::GetAceCount

Returns the number of access-control entries (ACEs) in the `CSacl` object.

```
UINT GetAceCount() const throw();
```

Return Value

Returns the number of ACEs contained in the `CSacl` object.

CSacl::operator =

Assignment operator.

```
CSacl& operator=(const ACL& rhs) throw(...);
```

Parameters

rhs

The `ACL` (access-control list) to assign to the existing object.

Return Value

Returns a reference to the updated `CSacl` object. Ensure that the `ACL` parameter is actually a system access-control list (SACL) and not a discretionary access-control list (DACL). In debug builds an assertion will occur, and in release builds the `ACL` parameter will be ignored.

CSacl::RemoveAce

Removes a specific ACE (access-control entry) from the `CSacl` object.

```
void RemoveAce(UINT nIndex) throw();
```

Parameters

nIndex

Index to the ACE entry to remove.

Remarks

This method is derived from [CAclArray::RemoveAt](#).

CSacl::RemoveAllAces

Removes all of the access-control entries (ACEs) contained in the `CSacl` object.

```
void RemoveAllAces() throw();
```

Remarks

Removes every `ACE` structure (if any) in the `CSacl` object.

See also

[CAcl Class](#)

[ACLs](#)

[ACEs](#)

[Class Overview](#)

[Security Global Functions](#)

CSecurityAttributes Class

4/1/2019 • 2 minutes to read • [Edit Online](#)

This class is a thin wrapper for the security attributes structure.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CSecurityAttributes : public SECURITY_ATTRIBUTES
```

Members

Public Constructors

NAME	DESCRIPTION
CSecurityAttributes::CSecurityAttributes	The constructor.

Public Methods

NAME	DESCRIPTION
CSecurityAttributes::Set	Call this method to set the attributes of the <code>CSecurityAttributes</code> object.

Remarks

The `SECURITY_ATTRIBUTES` structure contains a [security descriptor](#) used for the creation of an object and specifies whether the handle retrieved by specifying this structure is inheritable.

For an introduction to the access control model in Windows, see [Access Control](#) in the Windows SDK.

Inheritance Hierarchy

`SECURITY_ATTRIBUTES`

`CSecurityAttributes`

Requirements

Header: atlsecurity.h

CSecurityAttributes::CSecurityAttributes

The constructor.

```
CSecurityAttributes() throw();  
explicit CSecurityAttributes(const CSecurityDesc& rSecurityDescriptor, bool bInheritsHandle = false)  
throw(...);
```

Parameters

rSecurityDescriptor

Reference to a security descriptor.

bInheritsHandle

Specifies whether the returned handle is inherited when a new process is created. If this member is true, the new process inherits the handle.

CSecurityAttributes::Set

Call this method to set the attributes of the `CSecurityAttributes` object.

```
void Set(const CSecurityDesc& rSecurityDescriptor, bool bInheritHandle = false) throw(...);
```

Parameters

rSecurityDescriptor

Reference to a security descriptor.

bInheritHandle

Specifies whether the returned handle is inherited when a new process is created. If this member is true, the new process inherits the handle.

Remarks

This method is used by the constructor to initialize the `CSecurityAttributes` object.

See also

[Security Sample](#)

[SECURITY_ATTRIBUTES](#)

[security descriptor](#)

[Class Overview](#)

[Security Global Functions](#)

CSecurityDesc Class

4/1/2019 • 15 minutes to read • [Edit Online](#)

This class is a wrapper for the `SECURITY_DESCRIPTOR` structure.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CSecurityDesc
```

Members

Public Constructors

NAME	DESCRIPTION
CSecurityDesc::CSecurityDesc	The constructor.
CSecurityDesc::~~CSecurityDesc	The destructor.

Public Methods

NAME	DESCRIPTION
CSecurityDesc::FromString	Converts a string-format security descriptor into a valid, functional security descriptor.
CSecurityDesc::GetControl	Retrieves control information from the security descriptor.
CSecurityDesc::GetDacl	Retrieves discretionary access-control list (DACL) information from the security descriptor.
CSecurityDesc::GetGroup	Retrieves the primary group information from the security descriptor.
CSecurityDesc::GetOwner	Retrieves owner information from the security descriptor.
CSecurityDesc::GetPSECURITY_DESCRIPTOR	Returns a pointer to the <code>SECURITY_DESCRIPTOR</code> structure.
CSecurityDesc::GetSacl	Retrieves system access-control list (SACL) information from the security descriptor.
CSecurityDesc::IsDaclAutoInherited	Determines if the DACL is configured to support automatic propagation.

NAME	DESCRIPTION
CSecurityDesc::IsDaclDefaulted	Determines if the security descriptor is configured with a default DACL.
CSecurityDesc::IsDaclPresent	Determines if the security descriptor contains a DACL.
CSecurityDesc::IsDaclProtected	Determines if the DACL is configured to prevent modifications.
CSecurityDesc::IsGroupDefaulted	Determines if the security descriptor's group security identifier (SID) was set by default.
CSecurityDesc::IsOwnerDefaulted	Determines if the security descriptor's owner SID was set by default.
CSecurityDesc::IsSaclAutoInherited	Determines if the SACL is configured to support automatic propagation.
CSecurityDesc::IsSaclDefaulted	Determines if the security descriptor is configured with a default SACL.
CSecurityDesc::IsSaclPresent	Determines if the security descriptor contains a SACL.
CSecurityDesc::IsSaclProtected	Determines if the SACL is configured to prevent modifications.
CSecurityDesc::IsSelfRelative	Determines if the security descriptor is in self-relative format.
CSecurityDesc::MakeAbsolute	Call this method to convert the security descriptor to absolute format.
CSecurityDesc::MakeSelfRelative	Call this method to convert the security descriptor to self-relative format.
CSecurityDesc::SetControl	Sets the control bits of a security descriptor.
CSecurityDesc::SetDacl	Sets information in a DACL. If a DACL is already present in the security descriptor, it is replaced.
CSecurityDesc::SetGroup	Sets the primary group information of an absolute format security descriptor, replacing any primary group information already present.
CSecurityDesc::SetOwner	Sets the owner information of an absolute format security descriptor, replacing any owner information already present.
CSecurityDesc::SetSacl	Sets information in a SACL. If a SACL is already present in the security descriptor, it is replaced.
CSecurityDesc::ToString	Converts a security descriptor to a string format.

Public Operators

NAME	DESCRIPTION
<code>CSecurityDesc::operator const SECURITY_DESCRIPTOR *</code>	Returns a pointer to the <code>SECURITY_DESCRIPTOR</code> structure.
<code>CSecurityDesc::operator =</code>	Assignment operator.

Remarks

The `SECURITY_DESCRIPTOR` structure contains the security information associated with an object. Applications use this structure to set and query an object's security status. See also [AtlGetSecurityDescriptor](#).

Applications should not modify the `SECURITY_DESCRIPTOR` structure directly, and instead should use the class methods provided.

For an introduction to the access control model in Windows, see [Access Control](#) in the Windows SDK.

Requirements

Header: `atlsecurity.h`

CSecurityDesc::CSecurityDesc

The constructor.

```
CSecurityDesc() throw();
CSecurityDesc(const CSecurityDesc& rhs) throw(... );
CSecurityDesc(const SECURITY_DESCRIPTOR& rhs) throw(...);
```

Parameters

rhs

The `CSecurityDesc` object or `SECURITY_DESCRIPTOR` structure to assign to the new `CSecurityDesc` object.

Remarks

The `CSecurityDesc` object can optionally be created using a `SECURITY_DESCRIPTOR` structure or a previously defined `CSecurityDesc` object.

CSecurityDesc::~~CSecurityDesc

The destructor.

```
virtual ~CSecurityDesc() throw();
```

Remarks

The destructor frees all allocated resources.

CSecurityDesc::FromString

Converts a string-format security descriptor into a valid, functional security descriptor.

```
bool FromString(LPCTSTR pstr) throw(...);
```

Parameters

pstr

Pointer to a null-terminated string that contains the [string-format security descriptor](#) to be converted.

Return Value

Returns true on success. Throws an exception on failure.

Remarks

The string can be created by using [CSecurityDesc::ToString](#). Converting the security descriptor into a string makes it easier to store and transmit.

This method calls [ConvertStringSecurityDescriptorToSecurityDescriptor](#).

CSecurityDesc::GetControl

Retrieves control information from the security descriptor.

```
bool GetControl(SEURITY_DESCRIPTOR_CONTROL* psdc) const throw();
```

Parameters

psdc

Pointer to a `SECURITY_DESCRIPTOR_CONTROL` structure that receives the security descriptor's control information.

Return Value

Returns true if the method succeeds, false if it fails.

Remarks

This method calls [GetSecurityDescriptorControl](#).

CSecurityDesc::GetDacl

Retrieves discretionary access-control list (DACL) information from the security descriptor.

```
bool GetDacl(  
    CDacl* pDacl,  
    bool* pbPresent = NULL,  
    bool* pbDefaulted = NULL) const throw(...);
```

Parameters

pDacl

Pointer to an `CDacl` structure in which to store a copy of the security descriptor's DACL. If a discretionary ACL exists, the method sets *pDacl* to the address of the security descriptor's discretionary ACL. If a discretionary ACL does not exist, no value is stored.

pbPresent

Pointer to a value that indicates the presence of a discretionary ACL in the specified security descriptor. If the security descriptor contains a discretionary ACL, this parameter is set to true. If the security descriptor does not contain a discretionary ACL, this parameter is set to false.

pbDefaulted

Pointer to a flag set to the value of the SE_DACL_DEFAULTED flag in the `SECURITY_DESCRIPTOR_CONTROL` structure if a discretionary ACL exists for the security descriptor. If this flag is true, the discretionary ACL was retrieved by a default mechanism; if false, the discretionary ACL was explicitly specified by a user.

Return Value

Returns true if the method succeeds, false if it fails.

CSecurityDesc::GetGroup

Retrieves the primary group information from the security descriptor.

```
bool GetGroup(  
    CSid* pSid,  
    bool* pbDefaulted = NULL) const throw(...);
```

Parameters

pSid

Pointer to a [CSid](#) (security identifier) that receives a copy of the group stored in the CDacl.

pbDefaulted

Pointer to a flag set to the value of the SE_GROUP_DEFAULTED flag in the `SECURITY_DESCRIPTOR_CONTROL` structure when the method returns.

Return Value

Returns true if the method succeeds, false if it fails.

CSecurityDesc::GetOwner

Retrieves owner informaton from the security descriptor.

```
bool GetOwner(  
    CSid* pSid,  
    bool* pbDefaulted = NULL) const throw(...);
```

Parameters

pSid

Pointer to a [CSid](#) (security identifier) that receives a copy of the group stored in the CDacl.

pbDefaulted

Pointer to a flag set to the value of the SE_OWNER_DEFAULTED flag in the `SECURITY_DESCRIPTOR_CONTROL` structure when the method returns.

Return Value

Returns true if the method succeeds, false if it fails.

CSecurityDesc::GetPSECURITY_DESCRIPTOR

Returns a pointer to the `SECURITY_DESCRIPTOR` structure.

```
const SECURITY_DESCRIPTOR* GetPSECURITY_DESCRIPTOR() const throw();
```

Return Value

Returns a pointer to the [SECURITY_DESCRIPTOR](#) structure.

CSecurityDesc::GetSacl

Retrieves system access-control list (SACL) information from the security descriptor.

```
bool GetSacl(
    CSacl* pSacl,
    bool* pbPresent = NULL,
    bool* pbDefaulted = NULL) const throw(...);
```

Parameters

pSacl

Pointer to an `CSacl` structure in which to store a copy of the security descriptor's SACL. If a system ACL exists, the method sets *pSacl* to the address of the security descriptor's system ACL. If a system ACL does not exist, no value is stored.

pbPresent

Pointer to a flag the method sets to indicate the presence of a system ACL in the specified security descriptor. If the security descriptor contains a system ACL, this parameter is set to true. If the security descriptor does not contain a system ACL, this parameter is set to false.

pbDefaulted

Pointer to a flag set to the value of the SE_SACL_DEFAULTED flag in the `SECURITY_DESCRIPTOR_CONTROL` structure if a system ACL exists for the security descriptor.

Return Value

Returns true if the method succeeds, false if it fails.

CSecurityDesc::IsDaclAutoInherited

Determines if the discretionary access-control list (DACL) is configured to support automatic propagation.

```
bool IsDaclAutoInherited() const throw();
```

Return Value

Returns true if the security descriptor contains a DACL which is set up to support automatic propagation of inheritable access-control entries (ACEs) to existing child objects. Returns false otherwise.

Remarks

The system sets this bit when it performs the automatic inheritance algorithm for the object and its existing child objects.

CSecurityDesc::IsDaclDefaulted

Determines if the security descriptor is configured with a default discretionary access-control list (DACL).

```
bool IsDaclDefaulted() const throw();
```

Return Value

Returns true if the security descriptor contains a default DACL, false otherwise.

Remarks

This flag can affect how the system treats the DACL, with respect to access-control entry (ACE) inheritance. For example, if an object's creator does not specify a DACL, the object receives the default DACL from the creator's access token. The system ignores this flag if the SE_DACL_PRESENT flag is not set.

This flag is used to determine how the final DACL on the object is to be computed and is not stored physically in the security descriptor control of the securable object.

To set this flag, use the [CSecurityDesc::SetDacl](#) method.

CSecurityDesc::IsDaclPresent

Determines if the security descriptor contains a discretionary access-control list (DACL).

```
bool IsDaclPresent() const throw();
```

Return Value

Returns true if the security descriptor contains a DACL, false otherwise.

Remarks

If this flag is not set, or if this flag is set and the DACL is NULL, the security descriptor allows full access to everyone.

This flag is used to hold the security information specified by a caller until the security descriptor is associated with a securable object. Once the security descriptor is associated with a securable object, the SE_DACL_PRESENT flag is always set in the security descriptor control.

To set this flag, use the [CSecurityDesc::SetDacl](#) method.

CSecurityDesc::IsDaclProtected

Determines if the discretionary access-control list (DACL) is configured to prevent modifications.

```
bool IsDaclProtected() const throw();
```

Return Value

Returns true if the DACL is configured to prevent the security descriptor from being modified by inheritable access-control entries (ACEs). Returns false otherwise.

Remarks

To set this flag, use the [CSecurityDesc::SetDacl](#) method.

This method supports automatic propagation of inheritable ACEs.

CSecurityDesc::IsGroupDefaulted

Determines if the security descriptor's group security identifier (SID) was set by default.

```
bool IsGroupDefaulted() const throw();
```

Return Value

Returns true if a default mechanism, rather than the original provider of the security descriptor, provided the security descriptor's group SID. Returns false otherwise.

Remarks

To set this flag, use the [CSecurityDesc::SetGroup](#) method.

CSecurityDesc::IsOwnerDefaulted

Determines if the security descriptor's owner security identifier (SID) was set by default.

```
bool IsOwnerDefaulted() const throw();
```

Return Value

Returns true if a default mechanism, rather than the original provider of the security descriptor, provided the security descriptor's owner SID. Returns false otherwise.

Remarks

To set this flag, use the [CSecurityDesc::SetOwner](#) method.

CSecurityDesc::IsSaclAutoInherited

Determines if the system access-control list (SACL) is configured to support automatic propagation.

```
bool IsSaclAutoInherited() const throw();
```

Return Value

Returns true if the security descriptor contains a SACL which is set up to support automatic propagation of inheritable access-control entries (ACEs) to existing child objects. Returns false otherwise.

Remarks

The system sets this bit when it performs the automatic inheritance algorithm for the object and its existing child objects.

CSecurityDesc::IsSaclDefaulted

Determines if the security descriptor is configured with a default system access-control list (SACL).

```
bool IsSaclDefaulted() const throw();
```

Return Value

Returns true if the security descriptor contains a default SACL, false otherwise.

Remarks

This flag can affect how the system treats the SACL, with respect to access-control entry (ACE) inheritance. The system ignores this flag if the SE_SACL_PRESENT flag is not set.

To set this flag, use the [CSecurityDesc::SetSacl](#) method.

CSecurityDesc::IsSaclPresent

Determines if the security descriptor contains a system access-control list (SACL).

```
bool IsSaclPresent() const throw();
```

Return Value

Returns true if the security descriptor contains a SACL, false otherwise.

Remarks

To set this flag, use the [CSecurityDesc::SetSacl](#) method.

CSecurityDesc::IsSaclProtected

Determines if the system access-control list (SACL) is configured to prevent modifications.

```
bool IsSaclProtected() const throw();
```

Return Value

Returns true if the SACL is configured to prevent the security descriptor from being modified by inheritable access-control entries (ACEs). Returns false otherwise.

Remarks

To set this flag, use the [CSecurityDesc::SetSacl](#) method.

This method supports automatic propagation of inheritable ACEs.

CSecurityDesc::IsSelfRelative

Determines if the security descriptor is in self-relative format.

```
bool IsSelfRelative() const throw();
```

Return Value

Returns true if the security descriptor is in self-relative format with all the security information in a contiguous block of memory. Returns false if the security descriptor is in absolute format. For more information, see [Absolute and Self-Relative Security Descriptors](#).

CSecurityDesc::MakeAbsolute

Call this method to convert the security descriptor to absolute format.

```
bool MakeAbsolute() throw(...);
```

Return Value

Returns true if the method succeeds, false otherwise.

Remarks

A security descriptor in absolute format contains pointers to the information it contains, rather than the information itself. A security descriptor in self-relative format contains the information in a contiguous block of memory. In a self-relative security descriptor, a `SECURITY_DESCRIPTOR` structure always starts the information, but the security descriptor's other components can follow the structure in any order. Instead of using memory addresses, the components of the self-relative security descriptor are identified by offsets from the beginning of the security descriptor. This format is useful when a security descriptor must be stored on a disk or transmitted by means of a communications protocol. For more information, see [Absolute and Self-Relative Security Descriptors](#).

CSecurityDesc::MakeSelfRelative

Call this method to convert the security descriptor to self-relative format.

```
bool MakeSelfRelative() throw(...);
```

Return Value

Returns true if the method succeeds, false otherwise.

Remarks

A security descriptor in absolute format contains pointers to the information it contains, rather than containing the information itself. A security descriptor in self-relative format contains the information in a contiguous block of memory. In a self-relative security descriptor, a `SECURITY_DESCRIPTOR` structure always starts the information, but the security descriptor's other components can follow the structure in any order. Instead of using memory addresses, the components of the security descriptor are identified by offsets from the beginning of the security descriptor. This format is useful when a security descriptor must be stored on a disk or transmitted by means of a communications protocol. For more information, see [Absolute and Self-Relative Security Descriptors](#).

CSecurityDesc::operator =

Assignment operator.

```
CSecurityDesc& operator= (const SECURITY_DESCRIPTOR& rhs) throw(...);  
CSecurityDesc& operator= (const CSecurityDesc& rhs) throw(...);
```

Parameters

rhs

The `SECURITY_DESCRIPTOR` structure or `CSecurityDesc` object to assign to the `CSecurityDesc` object.

Return Value

Returns the updated `CSecurityDesc` object.

CSecurityDesc::operator const SECURITY_DESCRIPTOR *

Casts a value to a pointer to the `SECURITY_DESCRIPTOR` structure.

```
operator const SECURITY_DESCRIPTOR *() const throw();
```

CSecurityDesc::SetControl

Sets the control bits of a security descriptor.

```
bool SetControl(  
    SECURITY_DESCRIPTOR_CONTROL ControlBitsOfInterest,  
    SECURITY_DESCRIPTOR_CONTROL ControlBitsToSet) throw();
```

Parameters

ControlBitsOfInterest

A `SECURITY_DESCRIPTOR_CONTROL` mask that indicates the control bits to set. For a list of the flags which can be set, see [SetSecurityDescriptorControl](#).

ControlBitsToSet

A `SECURITY_DESCRIPTOR_CONTROL` mask that indicates the new values for the control bits specified by the *ControlBitsOfInterest* mask. This parameter can be a combination of the flags listed for the *ControlBitsOfInterest* parameter.

Return Value

Returns true on success, false on failure.

Remarks

This method calls [SetSecurityDescriptorControl](#).

CSecurityDesc::SetDacl

Sets information in a discretionary access-control list (DACL). If a DACL is already present in the security descriptor, it is replaced.

```
inline void SetDacl(
    bool bPresent = true,
    bool bDefaulted = false) throw(...);

inline void SetDacl(
    const CDacl& Dacl,
    bool bDefaulted = false) throw(...);
```

Parameters

Dacl

Reference to a `CDacl` object specifying the DACL for the security descriptor. This parameter must not be NULL. To set a NULL DACL in the security descriptor, the first form of the method should be used with *bPresent* set to false.

bPresent

Specifies a flag indicating the presence of a DACL in the security descriptor. If this parameter is true, the method sets the SE_DACL_PRESENT flag in the `SECURITY_DESCRIPTOR_CONTROL` structure and uses the values in the *Dacl* and *bDefaulted* parameters. If it is false, the method clears the SE_DACL_PRESENT flag, and *bDefaulted* is ignored.

bDefaulted

Specifies a flag indicating the source of the DACL. If this flag is true, the DACL has been retrieved by some default mechanism. If false, the DACL has been explicitly specified by a user. The method stores this value in the SE_DACL_DEFAULTED flag of the `SECURITY_DESCRIPTOR_CONTROL` structure. If this parameter is not specified, the SE_DACL_DEFAULTED flag is cleared.

Return Value

Returns true on success, false on failure.

Remarks

There is an important difference between an empty and a nonexistent DACL. When a DACL is empty, it contains no access-control entries and no access rights have been explicitly granted. As a result, access to the object is implicitly denied. When an object has no DACL, on the other hand, no protection is assigned to the object, and any access request is granted.

CSecurityDesc::SetGroup

Sets the primary group information of an absolute format security descriptor, replacing any primary group information already present.

```
bool SetGroup(const CSid& Sid, bool bDefaulted = false) throw(...);
```

Parameters

Sid

Reference to a [CSid](#) object for the security descriptor's new primary group. This parameter must not be NULL. A security descriptor can be marked as not having a DACL or a SACL, but it must have a group and an owner, even

it these are the NULL SID (which is a built-in SID with a special meaning).

bDefaulted

Indicates whether the primary group information was derived from a default mechanism. If this value is true, it is default information, and the method stores this value as the SE_GROUP_DEFAULTED flag in the `SECURITY_DESCRIPTOR_CONTROL` structure. If this parameter is zero, the SE_GROUP_DEFAULTED flag is cleared.

Return Value

Returns true on success, false on failure.

CSecurityDesc::SetOwner

Sets the owner information of an absolute format security descriptor. It replaces any owner information already present.

```
bool SetOwner(const CSid& Sid, bool bDefaulted = false) throw(...);
```

Parameters

Sid

The `CSid` object for the security descriptor's new primary owner. This parameter must not be NULL.

bDefaulted

Indicates whether the owner information is derived from a default mechanism. If this value is true, it is default information. The method stores this value as the SE_OWNER_DEFAULTED flag in the `SECURITY_DESCRIPTOR_CONTROL` structure. If this parameter is zero, the SE_OWNER_DEFAULTED flag is cleared.

Return Value

Returns true on success, false on failure.

CSecurityDesc::SetSacl

Sets information in a system access-control list (SACL). If a SACL is already present in the security descriptor, it is replaced.

```
bool SetSacl(const CSacl& Sacl, bool bDefaulted = false) throw(...);
```

Parameters

Sacl

Pointer to an `CSacl` object specifying the SACL for the security descriptor. This parameter must not be NULL, and must be a CSacl object. Unlike DACLs, there is no difference between NULL and an empty SACL, as SACL objects do not specify access rights, only auditing information.

bDefaulted

Specifies a flag indicating the source of the SACL. If this flag is true, the SACL has been retrieved by some default mechanism. If false, the SACL has been explicitly specified by a user. The method stores this value in the SE_SACL_DEFAULTED flag of the `SECURITY_DESCRIPTOR_CONTROL` structure. If this parameter is not specified, the SE_SACL_DEFAULTED flag is cleared.

Return Value

Returns true on success, false on failure.

CSecurityDesc::ToString

Converts a security descriptor to a string format.

```
bool ToString(  
    CString* pstr, SECURITY_INFORMATION si = OWNER_SECURITY_INFORMATION |  
    GROUP_SECURITY_INFORMATION | DACL_SECURITY_INFORMATION |  
    SACL_SECURITY_INFORMATION) const throw(...);
```

Parameters

pstr
Pointer to a null-terminated string which will receive the [string-format security descriptor](#).

si
Specifies a combination of SECURITY_INFORMATION bit flags to indicate the components of the security descriptor to include in the output string.

Return Value

Returns true on success, false on failure.

Remarks

Once the security descriptor is in string format, it can more easily be stored or transmitted. Use the `CSecurityDesc::FromString` method to convert the string back into a security descriptor.

The *si* parameter can contain the following SECURITY_INFORMATION flags:

VALUE	MEANING
OWNER_SECURITY_INFORMATION	Include the owner.
GROUP_SECURITY_INFORMATION	Include the primary group.
DACL_SECURITY_INFORMATION	Include the DACL.
SACL_SECURITY_INFORMATION	Include the SACL.

If the DACL is NULL and the SE_DACL_PRESENT control bit is set in the input security descriptor, the method fails.

If the DACL is NULL and the SE_DACL_PRESENT control bit is not set in the input security descriptor, the resulting security descriptor string does not have a D: component. See [Security Descriptor String Format](#) for more details.

This method calls [ConvertStringSecurityDescriptorToSecurityDescriptor](#).

See also

- [Security Sample](#)
- [SECURITY_DESCRIPTOR](#)
- [Class Overview](#)
- [Security Global Functions](#)

CSid Class

4/1/2019 • 9 minutes to read • [Edit Online](#)

This class is a wrapper for a `SID` (security identifier) structure.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CSid
```

Members

Public Typedefs

NAME	DESCRIPTION
<code>CSid::CSidArray</code>	An array of <code>CSid</code> objects.

Public Constructors

NAME	DESCRIPTION
<code>CSid::CSid</code>	The constructor.
<code>CSid::~~CSid</code>	The destructor.

Public Methods

NAME	DESCRIPTION
<code>CSid::AccountName</code>	Returns the name of the account associated with the <code>CSid</code> object.
<code>CSid::Domain</code>	Returns the name of the domain associated with the <code>CSid</code> object.
<code>CSid::EqualPrefix</code>	Tests <code>SID</code> (security identifier) prefixes for equality.
<code>CSid::GetLength</code>	Returns the length of the <code>CSid</code> object.
<code>CSid::GetPSID</code>	Returns a pointer to a <code>SID</code> structure.
<code>CSid::GetPSID_IDENTIFIER_AUTHORITY</code>	Returns a pointer to the <code>SID_IDENTIFIER_AUTHORITY</code> structure.

NAME	DESCRIPTION
<code>CSid::GetSubAuthority</code>	Returns a specified subauthority in a <code>SID</code> structure.
<code>CSid::GetSubAuthorityCount</code>	Returns the subauthority count.
<code>CSid::IsValid</code>	Tests the <code>CSid</code> object for validity.
<code>CSid::LoadAccount</code>	Updates the <code>CSid</code> object given the account name and domain, or an existing <code>SID</code> structure.
<code>CSid::Sid</code>	Returns the ID string.
<code>CSid::SidNameUse</code>	Returns a description of the state of the <code>CSid</code> object.

Operators

<code>operator =</code>	Assignment operator.
<code>operator const SID *</code>	Casts a <code>CSid</code> object to a pointer to a <code>SID</code> structure.

Global Operators

<code>operator ==</code>	Tests two security descriptor objects for equality
<code>operator !=</code>	Tests two security descriptor objects for inequality
<code>operator <</code>	Compares relative value of two security descriptor objects.
<code>operator ></code>	Compares relative value of two security descriptor objects.
<code>operator <=</code>	Compares relative value of two security descriptor objects.
<code>operator >=</code>	Compares relative value of two security descriptor objects.

Remarks

The `SID` structure is a variable-length structure used to uniquely identify users or groups.

Applications should not modify the `SID` structure directly, but instead use the methods provided in this wrapper class. See also [AtlGetOwnerSid](#), [AtlSetGroupSid](#), [AtlGetGroupSid](#), and [AtlSetOwnerSid](#).

For an introduction to the access control model in Windows, see [Access Control](#) in the Windows SDK.

Requirements

Header: `atlsecurity.h`

CSid::AccountName

Returns the name of the account associated with the `CSid` object.

```
LPCTSTR AccountName() const throw(...);
```

Return Value

Returns the LPCTSTR pointing to the name of the account.

Remarks

This method attempts to find a name for the specified `SID` (security identifier). For full details, see [LookupAccountSid](#).

If no account name for the `SID` can be found, `AccountName` returns an empty string. This can occur if a network timeout prevents this method from finding the name. It also occurs for security identifiers with no corresponding account name, such as an `SID` that identifies a sign-in session.

CSid::CSid

The constructor.

```
CSid() throw();
CSid(const SID& rhs) throw(...);
CSid(const CSid& rhs) throw(...);

CSid(
    const SID_IDENTIFIER_AUTHORITY& IdentifierAuthority,
    BYTE nSubAuthorityCount,
    ...) throw(...);

explicit CSid(
    LPCTSTR pszAccountName,
    LPCTSTR pszSystem = NULL) throw(...);

explicit CSid(
    const SID* pSid,
    LPCTSTR pszSystem = NULL) throw(...);
```

Parameters

rhs

An existing `csid` object or `SID` (security identifier) structure.

IdentifierAuthority

The authority.

nSubAuthorityCount

The subauthority count.

pszAccountName

The account name.

pszSystem

The system name. This string can be the name of a remote computer. If this string is NULL, the local system is used instead.

pSid

A pointer to a `SID` structure.

Remarks

The constructor initializes the `csid` object, setting an internal data member to *SidTypeInvalid*, or by copying the settings from an existing `csid`, `SID`, or existing account.

If initialization fails, the constructor will throw a [CAtlException Class](#).

CSid::~~CSid

The destructor.

```
virtual ~CSid() throw();
```

Remarks

The destructor frees any resources acquired by the object.

CSid::CSidArray

An array of [CSid](#) objects.

```
typedef CAtlArray<CSid> CSidArray;
```

Remarks

This typedef specifies the array type that can be used to retrieve security identifiers from an ACL (access-control list). See [CAcl::GetAclEntries](#).

CSid::Domain

Returns the name of the domain associated with the `CSid` object.

```
LPCTSTR Domain() const throw(...);
```

Return Value

Returns the `LPCTSTR` pointing to the domain.

Remarks

This method attempts to find a name for the specified `SID` (security identifier). For full details, see [LookupAccountSid](#).

If no account name for the `SID` can be found, `Domain` returns the domain as an empty string. This can occur if a network timeout prevents this method from finding the name. It also occurs for security identifiers with no corresponding account name, such as an `SID` that identifies a sign-in session.

CSid::EqualPrefix

Tests `SID` (security identifier) prefixes for equality.

```
bool EqualPrefix(const SID& rhs) const throw();  
bool EqualPrefix(const CSid& rhs) const throw();
```

Parameters

rhs

The `SID` (security identifier) structure or `CSid` object to compare.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

See [EqualPrefixSid](#) in the Windows SDK for more details.

CSid::GetLength

Returns the length of the `CSid` object.

```
UINT GetLength() const throw();
```

Return Value

Returns the length in bytes of the `CSid` object.

Remarks

If the `CSid` structure is not valid, the return value is undefined. Before calling `GetLength`, use the [CSid::IsValid](#) member function to verify that `CSid` is valid.

NOTE

Under debug builds the function will cause an ASSERT if the `CSid` object is not valid.

CSid::GetPSID

Returns a pointer to a `SID` (security identifier) structure.

```
const SID* GetPSID() const throw(...);
```

Return Value

Returns the address of the `CSid` object's underlying `SID` structure.

CSid::GetPSID_IDENTIFIER_AUTHORITY

Returns a pointer to the `SID_IDENTIFIER_AUTHORITY` structure.

```
const SID_IDENTIFIER_AUTHORITY* GetPSID_IDENTIFIER_AUTHORITY() const throw();
```

Return Value

If the method succeeds, it returns the address of the `SID_IDENTIFIER_AUTHORITY` structure. If it fails, the return value is undefined. Failure may occur if the `CSid` object is not valid, in which case the [CSid::IsValid](#) method returns FALSE. The function `GetLastError` can be called for extended error information.

NOTE

Under debug builds the function will cause an ASSERT if the `CSid` object is not valid.

CSid::GetSubAuthority

Returns a specified subauthority in a `SID` (security identifier) structure.

```
DWORD GetSubAuthority(DWORD nSubAuthority) const throw();
```

Parameters

nSubAuthority

The subauthority.

Return Value

Returns the subauthority referenced by *nSubAuthority*. The subauthority value is a relative identifier (RID).

Remarks

The *nSubAuthority* parameter specifies an index value identifying the subauthority array element the method will return. The method performs no validation tests on this value. An application can call [CSid::GetSubAuthorityCount](#) to discover the range of acceptable values.

NOTE

Under debug builds the function will cause an ASSERT if the `CSid` object is not valid.

CSid::GetSubAuthorityCount

Returns the subauthority count.

```
UCHAR GetSubAuthorityCount() const throw();
```

Return Value

If the method succeeds, the return value is the subauthority count.

If the method fails, the return value is undefined. The method fails if the `CSid` object is invalid. To get extended error information, call `GetLastError`.

NOTE

Under debug builds the function will cause an ASSERT if the `CSid` object is not valid.

CSid::IsValid

Tests the `CSid` object for validity.

```
bool IsValid() const throw();
```

Return Value

Returns TRUE if the `CSid` object is valid, FALSE if not. There is no extended error information for this method; do not call `GetLastError`.

Remarks

The `IsValid` method validates the `CSid` object by verifying that the revision number is within a known range and that the number of subauthorities is less than the maximum.

CSid::LoadAccount

Updates the `CSid` object given the account name and domain, or an existing SID (security identifier) structure.

```
bool LoadAccount(
    LPCTSTR pszAccountName,
    LPCTSTR pszSystem = NULL) throw(...);

bool LoadAccount(
    const SID* pSid,
    LPCTSTR pszSystem = NULL) throw(...);
```

Parameters

pszAccountName

The account name.

pszSystem

The system name. This string can be the name of a remote computer. If this string is NULL, the local system is used instead.

pSid

A pointer to a [SID](#) structure.

Return Value

Returns TRUE on success, FALSE on failure. To get extended error information, call `GetLastError`.

Remarks

`LoadAccount` attempts to find a security identifier for the specified name. See [LookupAccountSid](#) for more details.

CSid::operator =

Assignment operator.

```
CSid& operator= (const CSid& rhs) throw(...);
CSid& operator= (const SID& rhs) throw(...);
```

Parameters

rhs

The `SID` (security identifier) or `CSid` to assign to the `CSid` object.

Return Value

Returns a reference to the updated `CSid` object.

CSid::operator ==

Tests two security descriptor objects for equality.

```
bool operator==(
    const CSid& lhs,
    const CSid& rhs) throw();
```

Parameters

lhs

The `SID` (security identifier) or `CSid` that appears on the left side of the == operator.

rhs

The `SID` (security identifier) or `csid` that appears on the right side of the `==` operator.

Return Value

TRUE if the security descriptors are equal, otherwise FALSE.

CSid::operator !=

Tests two security descriptor objects for inequality.

```
bool operator!=(  
    const CSid& lhs,  
    const CSid& rhs) throw();
```

Parameters

lhs

The `SID` (security identifier) or `csid` that appears on the left side of the `!=` operator.

rhs

The `SID` (security identifier) or `csid` that appears on the right side of the `!=` operator.

Return Value

TRUE if the security descriptors are not equal, otherwise FALSE.

CSid::operator <

Compares relative value of two security descriptor objects.

```
bool operator<(  
    const CSid& lhs,  
    const CSid& rhs) throw();
```

Parameters

lhs

The `SID` (security identifier) or `csid` that appears on the left side of the `!=` operator.

rhs

The `SID` (security identifier) or `csid` that appears on the right side of the `!=` operator.

Return Value

TRUE if *lhs* is less than *rhs*, otherwise FALSE.

CSid::operator <=

Compares relative value of two security descriptor objects.

```
bool operator<=(  
    const CSid& lhs,  
    const CSid& rhs) throw();
```

Parameters

lhs

The `SID` (security identifier) or `csid` that appears on the left side of the `!=` operator.

rhs

The `SID` (security identifier) or `csid` that appears on the right side of the `!=` operator.

Return Value

TRUE if *lhs* is less than or equal to *rhs*, otherwise FALSE.

CSid::operator >

Compares relative value of two security descriptor objects.

```
bool operator>(
    const CSid& lhs,
    const CSid& rhs) throw();
```

Parameters

lhs

The `SID` (security identifier) or `csid` that appears on the left side of the `!=` operator.

rhs

The `SID` (security identifier) or `csid` that appears on the right side of the `!=` operator.

Return Value

TRUE if *lhs* is greater than *rhs*, otherwise FALSE.

CSid::operator >=

Compares relative value of two security descriptor objects.

```
bool operator>=(
    const CSid& lhs,
    const CSid& rhs) throw();
```

Parameters

lhs

The `SID` (security identifier) or `csid` that appears on the left side of the `!=` operator.

rhs

The `SID` (security identifier) or `csid` that appears on the right side of the `!=` operator.

Return Value

TRUE if *lhs* is greater than or equal to *rhs*, otherwise FALSE.

CSid::operator const SID *

Casts a `CSid` object to a pointer to a `SID` (security identifier) structure.

```
operator const SID *() const throw(...);
```

Remarks

Returns the address of the `SID` structure.

CSid::Sid

Returns the `SID` (security identifier) structure as a string.

```
LPCTSTR Sid() const throw(...);
```

Return Value

Returns the `SID` structure as a string in a format suitable for display, storage, or transmission. Equivalent to [ConvertSidToStringSid](#).

CSid::SidNameUse

Returns a description of the state of the `CSid` object.

```
SID_NAME_USE SidNameUse() const throw();
```

Return Value

Returns the value of the data member that stores a value describing the state of the `CSid` object.

VALUE	DESCRIPTION
SidTypeUser	Indicates a user <code>SID</code> (security identifier).
SidTypeGroup	Indicates a group <code>SID</code> .
SidTypeDomain	Indicates a domain <code>SID</code> .
SidTypeAlias	Indicates an alias <code>SID</code> .
SidTypeWellKnownGroup	Indicates a <code>SID</code> for a well-known group.
SidTypeDeletedAccount	Indicates a <code>SID</code> for a deleted account.
SidTypeInvalid	Indicates an invalid <code>SID</code> .
SidTypeUnknown	Indicates an unknown <code>SID</code> type.
SidTypeComputer	Indicates a <code>SID</code> for a computer.

Remarks

Call [CSid::LoadAccount](#) to update the `CSid` object before calling `SidNameUse` to return its state. `SidNameUse` does not change the state of the object (by calling to `LookupAccountName` or `LookupAccountSid`), but only returns the current state.

See also

- [Security Sample](#)
- [Class Overview](#)
- [Security Global Functions](#)
- [Operators](#)

CSimpleArray Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

This class provides methods for managing a simple array.

Syntax

```
template <class T, class TEqual = CSimpleArrayEqualHelper<T>>
class CSimpleArray
```

Parameters

T

The type of data to store in the array.

TEqual

A trait object, defining the equality test for elements of type *T*.

Members

Public Constructors

NAME	DESCRIPTION
CSimpleArray::CSimpleArray	The constructor for the simple array.
CSimpleArray::~~CSimpleArray	The destructor for the simple array.

Public Methods

NAME	DESCRIPTION
CSimpleArray::Add	Adds a new element to the array.
CSimpleArray::Find	Finds an element in the array.
CSimpleArray::GetData	Returns a pointer to the data stored in the array.
CSimpleArray::GetSize	Returns the number of elements stored in the array.
CSimpleArray::Remove	Removes a given element from the array.
CSimpleArray::RemoveAll	Removes all elements from the array.
CSimpleArray::RemoveAt	Removes the specified element from the array.
CSimpleArray::SetAtIndex	Sets the specified element in the array.

Public Operators

NAME	DESCRIPTION
CSimpleArray::operator[]	Retrieves an element from the array.
CSimpleArray::operator =	Assignment operator.

Remarks

[CSimpleArray](#) provides methods for creating and managing a simple array, of any given type [T](#).

The parameter [TEqual](#) provides a means of defining an equality function for two elements of type [T](#). By creating a class similar to [CSimpleArrayEqualHelper](#), it is possible to alter the behavior of the equality test for any given array. For example, when dealing with an array of pointers, it may be useful to define the equality as depending on the values the pointers reference. The default implementation utilizes **operator=()**.

Both [CSimpleArray](#) and [CSimpleMap](#) are designed for a small number of elements. [CAtlArray](#) and [CAtlMap](#) should be used when the array contains a large number of elements.

Requirements

Header: atlsimpcoll.h

Example

```
// Create an array of integers
CSimpleArray<int> iArray;

// Create an array of char pointers
// and use a new equality function
CSimpleArray<char *, MyEqualityEqualHelper<char *> > cMyArray;
```

CSimpleArray::Add

Adds a new element to the array.

```
BOOL Add(const T& t);
```

Parameters

t

The element to add to the array.

Return Value

Returns TRUE if the element is successfully added to the array, FALSE otherwise.

Example

```
// Create an array of integers and add some elements
CSimpleArray<int> iMyArray;
for (int i = 0; i < 10; i++)
    iMyArray.Add(i);
```

CSimpleArray::CSimpleArray

The constructor for the array object.

```
CSimpleArray(const CSimpleArray<T, TEqual>& src);
CSimpleArray();
```

Parameters

src

An existing `CSimpleArray` object.

Remarks

Initializes the data members, creating a new empty `CSimpleArray` object, or a copy of an existing `CSimpleArray` object.

CSimpleArray::~CSimpleArray

The destructor.

```
~CSimpleArray();
```

Remarks

Frees all allocated resources.

CSimpleArray::Find

Finds an element in the array.

```
int Find(const T& t) const;
```

Parameters

t

The element for which to search.

Return Value

Returns the index of the found element, or -1 if the element is not found.

Example

```
// Create an array of floats and search for a particular element

CSimpleArray<float> fMyArray;

for (int i = 0; i < 10; i++)
    fMyArray.Add((float)i * 100);

int e = fMyArray.Find(200);
if (e == -1)
    _tprintf_s(_T("Could not find element\n"));
else
    _tprintf_s(_T("Found the element at location %d\n"), e);
```

CSimpleArray::GetData

Returns a pointer to the data stored in the array.

```
T* GetData() const;
```

Return Value

Returns a pointer to the data in the array.

CSimpleArray::GetSize

Returns the number of elements stored in the array.

```
int GetSize() const;
```

Return Value

Returns the number of elements stored in the array.

CSimpleArray::operator []

Retrieves an element from the array.

```
T& operator[](int nIndex);
```

Parameters

nIndex

The element index.

Return Value

Returns the element of the array referenced by *nIndex*.

Example

```
// Create an array and display its contents
CSimpleArray<int> iMySampleArray;

for (int i = 0; i < 10; i++)
    iMySampleArray.Add(i);

for (int i = 0; i < iMySampleArray.GetSize(); i++)
    _tprintf_s(_T("Array index %d contains %d\n"), i, iMySampleArray[i]);
```

CSimpleArray::operator =

Assignment operator.

```
CSimpleArray<T, TEqual>
& operator=(
    const CSimpleArray<T, TEqual>& src);
```

Parameters

src

The array to copy.

Return Value

Returns a pointer to the updated `CSimpleArray` object.

Remarks

Copies all elements from the `CSimpleArray` object referenced by `src` into the current array object, replacing all existing data.

Example

```
// Create an array of chars and copy it to a second array
CSimpleArray<char> cMyArray1;
cMyArray1.Add('a');
CSimpleArray<char> cMyArray2;
cMyArray2 = cMyArray1;
ATLASSERT(cMyArray2[0] == 'a');
```

CSimpleArray::Remove

Removes a given element from the array.

```
BOOL Remove(const T& t);
```

Parameters

t

The element to remove from the array.

Return Value

Returns TRUE if the element is found and removed, FALSE otherwise.

Remarks

When an element is removed, the remaining elements in the array are renumbered to fill the empty space.

CSimpleArray::RemoveAll

Removes all elements from the array.

```
void RemoveAll();
```

Remarks

Removes all elements currently stored in the array.

CSimpleArray::RemoveAt

Removes the specified element from the array.

```
BOOL RemoveAt(int nIndex);
```

Parameters

nIndex

Index pointing to the element to remove.

Return Value

Returns TRUE if the element was removed, FALSE if the index was invalid.

Remarks

When an element is removed, the remaining elements in the array are renumbered to fill the empty space.

CSimpleArray::SetAtIndex

Set the specified element in the array.

```
BOOL SetAtIndex(  
    int nIndex,  
    const T& t);
```

Parameters

nIndex

The index of the element to change.

t

The value to assign to the specified element.

Return Value

Returns TRUE if successful, FALSE if the index was not valid.

See also

[Class Overview](#)

CSimpleArrayEqualHelper Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class is a helper for the [CSimpleArray](#) class.

Syntax

```
template <class T>
class CSimpleArrayEqualHelper
```

Parameters

T

A derived class.

Members

Public Methods

NAME	DESCRIPTION
CSimpleArrayEqualHelper::IsEqual	(Static) Tests two CSimpleArray object elements for equality.

Remarks

This traits class is a supplement to the [CSimpleArray](#) class. It provides a method for comparing two elements stored in a [CSimpleArray](#) object. By default, the elements are compared using **operator=()**, but if the array contains complex data types that lack their own equality operator, you will need to override this class.

Requirements

Header: atlsimpcoll.h

CSimpleArrayEqualHelper::IsEqual

Tests two [CSimpleArray](#) object elements for equality.

```
static bool IsEqual(
    const T& t1,
    const T& t2);
```

Parameters

t1

An object of type T.

t2

An object of type T.

Return Value

Returns true if the elements are equal, false otherwise.

See also

[CSimpleArray Class](#)

[CSimpleArrayEqualHelperFalse Class](#)

[Class Overview](#)

CSimpleArrayEqualHelperFalse Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class is a helper for the [CSimpleArray](#) class.

Syntax

```
template <class T>
class CSimpleArrayEqualHelperFalse
```

Parameters

T

A derived class.

Members

Public Methods

NAME	DESCRIPTION
CSimpleArrayEqualHelperFalse::IsEqual	(Static) Returns false.

Remarks

This traits class is a complement to the `CSimpleArray` class. It always returns false, and in addition, will call `ATLASSERT` with an argument of false if it is ever referenced. In situations where the equality test is not sufficiently defined, this class allows an array containing elements to operate correctly for most methods but fail in a well-defined manner for methods that depend on comparisons such as [CSimpleArray::Find](#).

Requirements

Header: atlsimpcoll.h

CSimpleArrayEqualHelperFalse::IsEqual

Returns false.

```
static bool IsEqual(const T&, const T&);
```

Return Value

Returns false.

Remarks

This method always returns false, and will call `ATLASSERT` with an argument of false if referenced. The purpose of `CSimpleArrayEqualHelperFalse::IsEqual` is to force methods using comparisons to fail in a well-defined manner when equality tests have not been adequately defined.

See also

CSimpleArrayEqualHelper Class

Class Overview

CSimpleDialog Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements a basic modal dialog box.

Syntax

```
template <WORD t_wDlgTemplateID, BOOL t_bCenter = TRUE>
class CSimpleDialog : public CDialogImplBase
```

Parameters

t_wDlgTemplateID

The resource ID of the dialog template resource.

t_bCenter

TRUE if the dialog object is to be centered on the owner window; otherwise FALSE.

Members

Public Methods

NAME	DESCRIPTION
CSimpleDialog::DoModal	Creates a modal dialog box.

Remarks

Implements a modal dialog box with basic functionality. `CSimpleDialog` provides support for Windows common controls only. To create and display a modal dialog box, create an instance of this class, providing the name of an existing resource template for the dialog box. The dialog box object closes when the user clicks any control with a pre-defined value (such as IDOK or IDCANCEL).

`CSimpleDialog` allows you to create only modal dialog boxes. `CSimpleDialog` provides the dialog box procedure, which uses the default message map to direct messages to the appropriate handlers.

See [Implementing a Dialog Box](#) for more information.

Inheritance Hierarchy

`CDialogImplBase`

`CSimpleDialog`

Requirements

Header: atlwin.h

CSimpleDialog::DoModal

Invokes a modal dialog box and returns the dialog-box result when done.

```
INT_PTR DoModal(HWND hWndParent = ::GetActiveWindow());
```

Parameters

hWndParent

A handle to the parent of the dialog box. If no value is provided, the parent is set to the current active window.

Return Value

If successful, the return value is the resource ID of the control that dismissed the dialog box.

If the function fails, the return value is -1. To get extended error information, call `GetLastError`.

Remarks

This method handles all interaction with the user while the dialog box is active. This is what makes the dialog box modal; that is, the user cannot interact with other windows until the dialog box is closed.

See also

[Class Overview](#)

CSimpleMap Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

This class provides support for a simple mapping array.

Syntax

```
template <class TKey, class TVal, class TEqual = CSimpleMapEqualHelper<TKey, TVal>>
class CSimpleMap
```

Parameters

TKey

The key element type.

TVal

The value element type.

TEqual

A trait object, defining the equality test for elements of type `T`.

Members

Public Typedefs

NAME	DESCRIPTION
<code>CSimpleMap::_ArrayElementType</code>	Typedef for the value type.
<code>CSimpleMap::_ArrayKeyType</code>	Typedef for the key type.

Public Constructors

NAME	DESCRIPTION
<code>CSimpleMap::CSimpleMap</code>	The constructor.
<code>CSimpleMap::~CSimpleMap</code>	The destructor.

Public Methods

NAME	DESCRIPTION
<code>CSimpleMap::Add</code>	Adds a key and associated value to the map array.
<code>CSimpleMap::FindKey</code>	Finds a specific key.
<code>CSimpleMap::FindVal</code>	Finds a specific value.
<code>CSimpleMap::GetKeyAt</code>	Retrieves the specified key.

NAME	DESCRIPTION
CSimpleMap::GetSize	Returns the number of entries in the mapping array.
CSimpleMap::GetValueAt	Retrieves the specified value.
CSimpleMap::Lookup	Returns the value associated with the given key.
CSimpleMap::Remove	Removes a key and matching value.
CSimpleMap::RemoveAll	Removes all keys and values.
CSimpleMap::RemoveAt	Removes a specific key and matching value.
CSimpleMap::ReverseLookup	Returns the key associated with the given value.
CSimpleMap::SetAt	Sets the value associated with the given key.
CSimpleMap::SetAtIndex	Sets the specific key and value.

Remarks

[CSimpleMap](#) provides support for a simple mapping array of any given type [T](#), managing an unordered array of key elements and their associated values.

The parameter [TEqual](#) provides a means of defining an equality function for two elements of type [T](#). By creating a class similar to [CSimpleMapEqualHelper](#), it is possible to alter the behavior of the equality test for any given array. For example, when dealing with an array of pointers, it may be useful to define the equality as depending on the values the pointers reference. The default implementation utilizes **operator==(())**.

Both [CSimpleMap](#) and [CSimpleArray](#) are provided for compatibility with previous ATL releases, and more complete and efficient collection implementations are provided by [CAtlArray](#) and [CAtlMap](#).

Unlike other map collections in ATL and MFC, this class is implemented with a simple array, and lookup searches require a linear search. [CAtlMap](#) should be used when the array contains a large number of elements.

Requirements

Header: `atlsimpcoll.h`

Example

```
// Create a map with an integer key and character pointer value
CSimpleMap<int, char *> iArray;
```

CSimpleMap::Add

Adds a key and associated value to the map array.

```
BOOL Add(const TKey& key, const TVal& val);
```

Parameters

key

The key.

val

The associated value.

Return Value

Returns TRUE if the key and value were successfully added, FALSE otherwise.

Remarks

Each key and value pair added causes the mapping array memory to be freed and reallocated, in order to ensure the data for each is always stored contiguously. That is, the second key element always directly follows the first key element in memory and so on.

CSimpleMap::_ArrayElementType

A typedef for the key type.

```
typedef TVal _ArrayElementType;
```

CSimpleMap::_ArrayKeyType

A typedef for the value type.

```
typedef TKey _ArrayKeyType;
```

CSimpleMap::CSimpleMap

The constructor.

```
CSimpleMap();
```

Remarks

Initializes the data members.

CSimpleMap::~~CSimpleMap

The destructor.

```
~CSimpleMap();
```

Remarks

Frees all allocated resources.

CSimpleMap::FindKey

Finds a specific key.

```
int FindKey(const TKey& key) const;
```

Parameters

key

The key to search for.

Return Value

Returns the index of the key if found, otherwise returns -1.

CSimpleMap::FindVal

Finds a specific value.

```
int FindVal(const TVal& val) const;
```

Parameters

val

The value for which to search.

Return Value

Returns the index of the value if it is found, otherwise returns -1.

CSimpleMap::GetKeyAt

Retrieves the key at the specified index.

```
TKey& GetKeyAt(int nIndex) const;
```

Parameters

nIndex

The index of the key to return.

Return Value

Returns the key referenced by *nIndex*.

Remarks

The index passed by *nIndex* must be valid for the return value to be meaningful.

CSimpleMap::GetSize

Returns the number of entries in the mapping array.

```
int GetSize() const;
```

Return Value

Returns the number of entries (a key and value is one entry) in the mapping array.

CSimpleMap::GetValueAt

Retrieves the value at the specific index.

```
TVal& GetValueAt(int nIndex) const;
```

Parameters

nIndex

The index of the value to return.

Return Value

Returns the value referenced by *nIndex*.

Remarks

The index passed by *nIndex* must be valid for the return value to be meaningful.

CSimpleMap::Lookup

Returns the value associated with the given key.

```
TVa1 Lookup(const TKey& key) const;
```

Parameters

key

The key.

Return Value

Returns the associated value. If no matching key is found, NULL is returned.

CSimpleMap::Remove

Removes a key and matching value.

```
BOOL Remove(const TKey& key);
```

Parameters

key

The key.

Return Value

Returns TRUE if the key, and matching value, were successfully removed, FALSE otherwise.

CSimpleMap::RemoveAll

Removes all keys and values.

```
void RemoveAll();
```

Remarks

Removes all keys and values from the mapping array object.

CSimpleMap::RemoveAt

Removes a key and associated value at the specified index.

```
BOOL RemoveAt(int nIndex);
```


Parameters

nIndex

The index of the key and associated value to remove.

Return Value

Returns TRUE on success, FALSE if the index specified is an invalid index.

CSimpleMap::ReverseLookup

Returns the key associated with the given value.

```
TKey ReverseLookup(const TVal& val) const;
```

Parameters

val

The value.

Return Value

Returns the associated key. If no matching key is found, NULL is returned.

CSimpleMap::SetAt

Sets the value associated with the given key.

```
BOOL SetAt(const TKey& key, const TVal& val);
```

Parameters

key

The key.

val

The new value to assign.

Return Value

Returns TRUE if the key was found, and the value was successfully changed, FALSE otherwise.

CSimpleMap::SetAtIndex

Sets the key and value at a specified index.

```
BOOL SetAtIndex(  
    int nIndex,  
    const TKey& key,  
    const TVal& val);
```

Parameters

nIndex

The index, referencing the key and value pairing to change.

key

The new key.

val

The new value.

Return Value

Returns TRUE if successful, FALSE if the index was not valid.

Remarks

Updates both the key and value pointed to by *nIndex*.

See also

[Class Overview](#)

CSimpleMapEqualHelper Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class is a helper for the [CSimpleMap](#) class.

Syntax

```
template <class TKey, class TVal>
class CSimpleMapEqualHelper
```

Parameters

TKey

The key element.

TVal

The value element.

Members

Public Methods

NAME	DESCRIPTION
CSimpleMapEqualHelper::IsEqualKey	(Static) Tests two keys for equality.
CSimpleMapEqualHelper::IsEqualValue	(Static) Tests two values for equality.

Remarks

This traits class is a supplement to the [CSimpleMap](#) class. It provides methods for comparing two [CSimpleMap](#) object elements (specifically, the key and value components) for equality. By default, the keys and values are compared using **operator==()**, but if the map contains complex data types that lack their own equality operator, this class can be overridden to provide the extra required functionality.

Requirements

Header: atlsimpcoll.h

CSimpleMapEqualHelper::IsEqualKey

Tests two keys for equality.

```
static bool IsEqualKey(const TKey& k1, const TKey& k2);
```

Parameters

k1

The first key.

k2

The second key.

Return Value

Returns true if the keys are equal, false otherwise.

CSimpleMapEqualHelper::IsEqualValue

Tests two values for equality.

```
static bool IsEqualValue(const TVal& v1, const TVal& v2);
```

Parameters

v1

The first value.

v2

The second value.

Return Value

Returns true if the values are equal, false otherwise.

See also

[CSimpleMapEqualHelperFalse Class](#)

[Class Overview](#)

CSimpleMapEqualHelperFalse Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class is a helper for the [CSimpleMap](#) class.

Syntax

```
template <class TKey, class TVal>
class CSimpleMapEqualHelperFalse
```

Members

Public Methods

NAME	DESCRIPTION
CSimpleMapEqualHelperFalse::IsEqualKey	(Static) Tests two keys for equality.
CSimpleMapEqualHelperFalse::IsEqualValue	(Static) Returns false.

Remarks

This traits class is a supplement to the `CSimpleMap` class. It provides a method for comparing two elements contained in the `CSimpleMap` object, specifically two value elements or two key elements.

The value comparison will always return false, and in addition, will call `ATLASSERT` with an argument of false if it is ever referenced. In situations where the equality test is not sufficiently defined, this class allows a map containing key/value pairs to operate correctly for most methods but fail in a well-defined manner for methods that depend on comparisons such as [CSimpleMap::FindVal](#).

Requirements

Header: atlsimpcoll.h

CSimpleMapEqualHelperFalse::IsEqualKey

Tests two keys for equality.

```
static bool IsEqualKey(const TKey& k1, const TKey& k2);
```

Parameters

k1

The first key.

k2

The second key.

Return Value

Returns true if the keys are equal, false otherwise.

Remarks

This method calls [CSimpleArrayEqualHelper](#).

CSimpleMapEqualHelperFalse::IsEqualValue

Returns false.

```
static bool IsEqualValue(const TVal&, const TVal&);
```

Return Value

Returns false.

Remarks

This method always returns false, and will call `ATLASSERT` with an argument of false if it is ever referenced. The purpose of `CSimpleMapEqualHelperFalse::IsEqualValue` is to force methods using comparisons to fail in a well-defined manner when equality tests have not been adequately defined.

See also

[CSimpleMapEqualHelper Class](#)

[Class Overview](#)

CSnapInItemImpl Class

3/4/2019 • 10 minutes to read • [Edit Online](#)

This class provides methods for implementing a snap-in node object.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class T, BOOL bIsExtension = FALSE>
class ATL_NO_VTABLE CSnapInItemImpl : public CSnapInItem
```

Parameters

T

Your class, derived from `CSnapInItemImpl`.

bIsExtension

TRUE if the object is a snap-in extension; otherwise FALSE.

Members

Public Constructors

NAME	DESCRIPTION
CSnapInItemImpl::CSnapInItemImpl	Constructor.

Public Methods

NAME	DESCRIPTION
CSnapInItemImpl::AddMenuItems	Adds menu items to a context menu.
CSnapInItemImpl::Command	Called by the console when a custom menu item is selected.
CSnapInItemImpl::CreatePropertyPages	Adds pages to the property sheet of the snap-in.
CSnapInItemImpl::FillData	Copies information on the snap-in object into a specified stream.
CSnapInItemImpl::GetResultPanelInfo	Retrieves the <code>RESULTDATAITEM</code> structure of the snap-in.
CSnapInItemImpl::GetResultViewType	Determines the type of view used by the result pane.
CSnapInItemImpl::GetScopePanelInfo	Retrieves the <code>SCOPEDATAITEM</code> structure of the snap-in.

NAME	DESCRIPTION
C SnapInItemImpl::Notify	Called by the console to notify the snap-in of actions taken by the user.
C SnapInItemImpl::QueryPagesFor	Called to see if the snap-in node supports property pages.
C SnapInItemImpl::SetMenuInsertionFlags	Modifies the menu insertion flags for a snap-in object.
C SnapInItemImpl::SetToolBarButtonInfo	Sets the information of the specified toolbar button.
C SnapInItemImpl::UpdateMenuState	Updates the state of a context menu item.
C SnapInItemImpl::UpdateToolBarButton	Updates the state of the specified toolbar button.

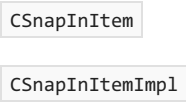
Public Data Members

NAME	DESCRIPTION
C SnapInItemImpl::m_bstrDisplayName	The name of the snap-in object.
C SnapInItemImpl::m_resultDataItem	The Windows <code>RESULTDATAITEM</code> structure used by the <code>C SnapInItemImpl</code> object.
C SnapInItemImpl::m_scopeDataItem	The Windows <code>SCOPEDATAITEM</code> structure used by the <code>C SnapInItemImpl</code> object.

Remarks

`C SnapInItemImpl` provides a basic implementation for a snap-in node object, such as adding menu items and toolbars, and forwarding commands for the snap-in node to the appropriate handler function. These features are implemented using several different interfaces and map types. The default implementation handles notifications sent to the node object by determining the correct instance of the derived class and then forwarding the message to the correct instance.

Inheritance Hierarchy



Requirements

Header: atlsnap.h

C SnapInItemImpl::AddMenuItems

This method implements the Win32 function [IExtendContextMenu::AddMenuItems](#).

```
AddMenuItems(  
    LPCONTEXTMENUCALLBACK piCallback,  
    long* pInsertionAllowed,  
    DATA_OBJECT_TYPES type);
```


Parameters

piCallback

[in] Pointer to the `IContextMenuCallback` that can add items to the context menu.

plInsertionAllowed

[in, out] Identifies Microsoft Management Console (MMC)-defined, menu-item insertion points that can be used. This can be a combination of the following flags:

- `CCM_INSERTIONALLOWED_TOP` Items can be inserted at the top of a context menu.
- `CCM_INSERTIONALLOWED_NEW` Items can be inserted in the Create New submenu.
- `CCM_INSERTIONALLOWED_TASK` Items can be inserted in the Task submenu.
- `CCM_INSERTIONALLOWED_VIEW` Items can be inserted in the toolbar view menu or in the View submenu of the result pane context menu.

type

[in] Specifies the type of object. It can have one of the following values:

- `CCT_SCOPE` Data object for scope pane context.
- `CCT_RESULT` Data object for result pane context.
- `CCT_SNAPIN_MANAGER` Data object for snap-in manager context.
- `CCT_UNINITIALIZED` Data object has an invalid type.

CSnapInItemImpl::Command

This method implements the Win32 function [IExtendContextMenu::Command](#).

```
Command(long lCommandID, DATA_OBJECT_TYPES type);
```

Parameters

lCommandID

[in] Specifies the command identifier of the menu item.

type

[in] Specifies the type of object. It can have one of the following values:

- `CCT_SCOPE` Data object for scope pane context.
- `CCT_RESULT` Data object for result pane context.
- `CCT_SNAPIN_MANAGER` Data object for snap-in manager context.
- `CCT_UNINITIALIZED` Data object has an invalid type.

CSnapInItemImpl::CreatePropertyPages

This method implements the Win32 function [IExtendPropertySheet::CreatePropertyPages](#).

```
CreatePropertyPages(  
    LPPROPERTYSHEETCALLBACK lpProvider,  
    long handle,  
    IUnknown* pUnk,  
    DATA_OBJECT_TYPES type);
```

Parameters

lpProvider

[in] Pointer to the `IPropertySheetCallback` interface.

handle

[in] Specifies the handle used to route the MMCN_PROPERTY_CHANGE notification message to the appropriate data class.

pUnk

[in] Pointer to the `IExtendPropertySheet` interface on the object that contains context information about the node.

type

[in] Specifies the type of object. It can have one of the following values:

- CCT_SCOPE Data object for scope pane context.
- CCT_RESULT Data object for result pane context.
- CCT_SNAPIN_MANAGER Data object for snap-in manager context.
- CCT_UNINITIALIZED Data object has an invalid type.

CSnapInItemImpl::CSnapInItemImpl

Constructs a `CSnapInItemImpl` object.

```
CSnapInItemImpl();
```

CSnapInItemImpl::FillData

This function is called to retrieve information about the item.

```
FillData(CLIPFORMAT cf, LPSTREAM pStream);
```

Parameters

cf

[in] The format (text, rich text, or rich text with OLE items) of the Clipboard.

pStream

[in] A pointer to the stream containing the object data.

Remarks

To properly implement this function, copy the correct information into the stream (*pStream*), depending on the Clipboard format indicated by *cf*.

CSnapInItemImpl::GetResultViewType

Call this function to retrieve the type of view for the result pane of the snap-in object.

```
GetResultViewType(  
    LPOLESTR* ppViewType,  
    long* pViewOptions);
```

Parameters

ppViewType

[out] Pointer to the address of the returned view type.

pViewOptions

[out] Pointer to the MMC_VIEW_OPTIONS enumeration, which provides the console with options specified by the owning snap-in. This value can be one of the following:

- MMC_VIEW_OPTIONS_NOLISTVIEWS = 0x00000001 Tells the console to refrain from presenting standard list view choices in the **View** menu. Allows the snap-in to display its own custom views only in the result view pane. This is the only option flag defined at this time.
- MMC_VIEW_OPTIONS_NONE = 0 Allows the default view options.

CSnapInItemImpl::GetScopePanelInfo

Call this function to retrieve the `SCOPEDATAITEM` structure of the snap-in.

```
GetScopePaneInfo (SCOPEDATAITEM* pScopeDataItem);
```

Parameters

pScopeDataItem

[out] A pointer to the `SCOPEDATAITEM` structure of the `CSnapInItemImpl` object.

CSnapInItemImpl::GetResultPanelInfo

Call this function to retrieve the `RESULTDATAITEM` structure of the snap-in.

```
GetResultPaneInfo (RESULTDATAITEM* pResultDataItem);
```

Parameters

pResultDataItem

[out] A pointer to the `RESULTDATAITEM` structure of the `CSnapInItemImpl` object.

CSnapInItemImpl::m_bstrDisplayName

Contains the string displayed for the node item.

```
COMBSTR m_bstrDisplayName;
```

CSnapInItemImpl::m_scopeDataItem

The `SCOPEDATAITEM` structure of the snap-in data object.

```
SCOPEDATAITEM m_scopeDataItem;
```

CSnapInItemImpl::m_resultDataItem

The `RESULTDATAITEM` structure of the snap-in data object.

```
RESULTDATAITEM m_resultDataItem;
```

CSnapInItemImpl::Notify

Called when the snap-in object is acted upon by the user.

```
STDMETHOD(Notify)(  
    MMC_NOTIFY_TYPE event,  
    long arg,  
    long param,  
    IComponentData* pComponentData,  
    IComponent* pComponent,  
    DATA_OBJECT_TYPES type) = 0;
```

Parameters

event

[in] Identifies an action taken by a user. The following notifications are possible:

- MMCN_ACTIVATE Sent when a window is being activated and deactivated.
- MMCN_ADD_IMAGES Sent to add images to the result pane.
- MMCN_BTN_CLICK Sent when the user clicks one of the toolbar buttons.
- MMCN_CLICK Sent when a user clicks a mouse button on a list view item.
- MMCN_DBLCLICK Sent when a user double clicks a mouse button on a list view item.
- MMCN_DELETE Sent to inform the snap-in that the object should be deleted.
- MMCN_EXPAND Sent when a folder needs to be expanded or contracted.
- MMCN_MINIMIZED Sent when a window is being minimized or maximized.
- MMCN_PROPERTY_CHANGE Sent to notify a snap-in object that the snap-in object's view is about to change.
- MMCN_REMOVE_CHILDREN Sent when the snap-in must delete the entire subtree it has added below the specified node.
- MMCN_RENAME Sent the first time to query for a rename and the second time to do the rename.
- MMCN_SELECT Sent when an item in the scope or result view pane is selected.
- MMCN_SHOW Sent when a scope item is selected or deselected for the first time.
- MMCN_VIEW_CHANGE Sent when the snap-in can update all views when a change occurs.

arg

[in] Depends on the notification type.

param

[in] Depends on the notification type.

pComponentData

[out] A pointer to the object implementing `IComponentData`. This parameter is NULL if the notification is not being forwarded from `IComponentData::Notify`.

pComponent

[out] A pointer to the object that implements `IComponent`. This parameter is NULL if the notification is not being forwarded from `IComponent::Notify`.

type

[in] Specifies the type of object. It can have one of the following values:

- CCT_SCOPE Data object for scope pane context.
- CCT_RESULT Data object for result pane context.
- CCT_SNAPIN_MANAGER Data object for snap-in manager context.
- CCT_UNINITIALIZED Data object has an invalid type.

CSnapInItemImpl::QueryPagesFor

Called to see if the snap-in node supports property pages.

```
QueryPagesFor(DATA_OBJECT_TYPES type);
```

CSnapInItemImpl::SetMenuInsertionFlags

Call this function to modify the menu insertion flags, specified by *pInsertionAllowed*, for the snap-in object.

```
void SetMenuInsertionFlags(  
    bool bBeforeInsertion,  
    long* pInsertionAllowed);
```

Parameters

bBeforeInsertion

[in] Nonzero if the function should be called before items are added to the context menu; otherwise 0.

pInsertionAllowed

[in, out] Identifies Microsoft Management Console (MMC)-defined, menu-item insertion points that can be used. This can be a combination of the following flags:

- CCM_INSERTIONALLOWED_TOP Items can be inserted at the top of a context menu.
- CCM_INSERTIONALLOWED_NEW Items can be inserted in the Create New submenu.
- CCM_INSERTIONALLOWED_TASK Items can be inserted in the Task submenu.
- CCM_INSERTIONALLOWED_VIEW Items can be inserted in the toolbar view menu or in the View submenu of the result pane context menu.

Remarks

If you are developing a primary snap-in, you can reset any of the insertion flags as a way of restricting the kind of menu items that a third-party extension can add. For example, the primary snap-in can clear the CCM_INSERTIONALLOWED_NEW flag to prevent extensions from adding their own Create New menu items.

You should not attempt to set bits in *pInsertionAllowed* that were originally cleared. Future versions of MMC may use bits not currently defined so you should not change bits that are currently not defined.

CSnapInItemImpl::SetToolbarButtonInfo

Call this function to modify any toolbar button styles, of the snap-in object, before the toolbar is created.

```
void SetToolbarButtonInfo(  
    UINT id,  
    BYTE* fsState,  
    BYTE* fsType);
```

Parameters

id

[in] The ID of the toolbar button to be set.

fsState

[in] The state flags of the button. Can be one or more of the following:

- TBSTATE_CHECKED The button has the TBSTYLE_CHECKED style and is being pressed.
- TBSTATE_ENABLED The button accepts user input. A button that does not have this state does not accept user input and is grayed.
- TBSTATE_HIDDEN The button is not visible and cannot receive user input.
- TBSTATE_INDETERMINATE The button is grayed.
- TBSTATE_PRESSED The button is being pressed.
- TBSTATE_WRAP A line break follows the button. The button must also have the TBSTATE_ENABLED.

fsType

[in] The state flags of the button. Can be one or more of the following:

- TBSTYLE_BUTTON Creates a standard push button.
- TBSTYLE_CHECK Creates a button that toggles between the pressed and not-pressed states each time the user clicks it. The button has a different background color when it is in the pressed state.
- TBSTYLE_CHECKGROUP Creates a check button that stays pressed until another button in the group is pressed.
- TBSTYLE_GROUP Creates a button that stays pressed until another button in the group is pressed.
- TBSTYLE_SEP Creates a separator, providing a small gap between button groups. A button that has this style does not receive user input.

CSnapInItemImpl::UpdateMenuState

Call this function to modify a menu item before it is inserted into the context menu of the snap-in object.

```
void UpdateMenuState(  
    UINT id,  
    LPTSTR pBuf,  
    UINT* flags);
```

Parameters

id

[in] The ID of the menu item to be set.

pBuf

[in] A pointer to the string for the menu item to be updated.

flags

[in] Specifies the new state flags. This can be a combination of the following flags:

- MF_POPUP Specifies that this is a submenu within the context menu. Menu items, insertion points, and further submenus may be added to this submenu using its `ICommandID` as their `IInsertionPointID`.
- MF_BITMAP and MF_OWNERDRAW These flags are not permitted and will result in a return value of `E_INVALIDARG`.
- MF_SEPARATOR Draws a horizontal dividing line. Only `IContextMenuProvider` is allowed to add menu items with MF_SEPARATOR set.
- MF_CHECKED Places a check mark next to the menu item.
- MF_DISABLED Disables the menu item so it cannot be selected, but the flag does not gray it.
- MF_ENABLED Enables the menu item so it can be selected, restoring it from its grayed state.
- MF_GRAYED Disables the menu item, graying it so it cannot be selected.
- MF_MENUBARBREAK Functions the same as the MF_MENUBREAK flag for a menu bar. For a drop-down menu, submenu, or shortcut menu, the new column is separated from the old column by a vertical line.
- MF_MENUBREAK Places the item on a new line (for a menu bar) or in a new column (for a drop-down menu, submenu, or shortcut menu) without separating columns.
- MF_UNCHECKED Does not place a check mark next to the item (default).

The following groups of flags cannot be used together:

- MF_DISABLED, MF_ENABLED, and MF_GRAYED.
- MF_MENUBARBREAK and MF_MENUBREAK.
- MF_CHECKED and MF_UNCHECKED.

CSnapInItemImpl::UpdateToolBarButton

Call this function to modify a toolbar button, of the snap-in object, before it is displayed.

```
BOOL UpdateToolBarButton(UINT id, BYTE fsState);
```

Parameters

id

Specifies the button ID of the toolbar button to be updated.

fsState

Specifies a toolbar button state. If this state is to be set, return TRUE. This can be a combination of the following flags:

- ENABLED The button accepts user input. A button that does not have this state does not accept user input and is grayed.
- CHECKED The button has the CHECKED style and is being pressed.
- HIDDEN The button is not visible and cannot receive user input.
- INDETERMINATE The button is grayed.
- BUTTONPRESSED The button is being pressed.

See also

[Class Overview](#)

CSnapInPropertyPageImpl Class

3/4/2019 • 6 minutes to read • [Edit Online](#)

This class provides methods for implementing a snap-in property page object.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
CSnapInPropertyPageImpl : public CDialogImplBase
```

Members

Public Constructors

NAME	DESCRIPTION
CSnapInPropertyPageImpl::CSnapInPropertyPageImpl	Constructor.

Public Methods

NAME	DESCRIPTION
CSnapInPropertyPageImpl::CancelToClose	Changes the status of the OK and Cancel buttons.
CSnapInPropertyPageImpl::Create	Initializes a newly created <code>CSnapInPropertyPageImpl</code> object.
CSnapInPropertyPageImpl::OnApply	Called by the framework when the user clicks the Apply Now button while using a wizard-type property sheet.
CSnapInPropertyPageImpl::OnHelp	Called by the framework when the user clicks the Help button while using a wizard-type property sheet.
CSnapInPropertyPageImpl::OnKillActive	Called by the framework when the current page is no longer active.
CSnapInPropertyPageImpl::OnQueryCancel	Called by the framework when the user clicks the Cancel button and before the cancel has taken place.
CSnapInPropertyPageImpl::OnReset	Called by the framework when the user clicks the Reset button while using a wizard-type property sheet.
CSnapInPropertyPageImpl::OnSetActive	Called by the framework when the current page becomes active.
CSnapInPropertyPageImpl::OnWizardBack	Called by the framework when the user clicks the Back button while using a wizard-type property sheet.

NAME	DESCRIPTION
CSnapInPropertyPageImpl::OnWizardFinish	Called by the framework when the user clicks the Finish button while using a wizard-type property sheet.
CSnapInPropertyPageImpl::OnWizardNext	Called by the framework when the user clicks the Next button while using a wizard-type property sheet.
CSnapInPropertyPageImpl::QuerySiblings	Forwards the current message to all pages of the property sheet.
CSnapInPropertyPageImpl::SetModified	Call to activate or deactivate the Apply Now button.

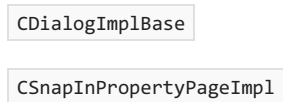
Public Data Members

NAME	DESCRIPTION
CSnapInPropertyPageImpl::m_psp	The Windows <code>PROPSHEETPAGE</code> structure used by the <code>CSnapInPropertyPageImpl</code> object.

Remarks

`CSnapInPropertyPageImpl` provides a basic implementation for a snap-in property page object. The basic features of a snap-in property page are implemented using several different interfaces and map types.

Inheritance Hierarchy



Requirements

Header: atlsnap.h

CSnapInPropertyPageImpl::CancelToClose

Call this function after an unrecoverable change has been made to the data in a page of a modal property sheet.

```
void CancelToClose();
```

Remarks

This function will change the **OK** button to **Close** and disable the **Cancel** button. This change alerts the user that a change is permanent and the modifications cannot be cancelled.

The `CancelToClose` member function does nothing in a modeless property sheet, because a modeless property sheet does not have a **Cancel** button by default.

CSnapInPropertyPageImpl::CSnapInPropertyPageImpl

Constructs a `CSnapInPropertyPageImpl` object.

```
CSnapInPropertyPageImpl(LPCTSTR lpszTitle = NULL);
```

Parameters

lpszTitle

[in] The title of the property page.

Remarks

To initialize the underlying structure, call [CSnapInPropertyPageImpl::Create](#).

CSnapInPropertyPageImpl::Create

Call this function to initialize the underlying structure of the property page.

```
HPROPSHEETPAGE Create();
```

Return Value

A handle to a `PROPSHEETPAGE` structure containing the attributes of the newly created property sheet.

Remarks

You should first call [CSnapInPropertyPageImpl::CSnapInPropertyPageImpl](#) before calling this function.

CSnapInPropertyPageImpl::m_psp

`m_psp` is a structure whose members store the characteristics of `PROPSHEETPAGE`.

```
PROPSHEETPAGE m_psp;
```

Remarks

Use this structure to initialize the appearance of a property page after it is constructed.

For more information on this structure, including a listing of its members, see [PROPSHEETPAGE](#) in the Windows SDK.

CSnapInPropertyPageImpl::OnApply

This member function is called when the user clicks the **OK** or the **Apply Now** button.

```
BOOL OnApply();
```

Return Value

Nonzero if the changes are accepted; otherwise 0.

Remarks

Before `OnApply` can be called by the framework, you must have called `SetModified` and set its parameter to TRUE. This will activate the **Apply Now** button as soon as the user makes a change on the property page.

Override this member function to specify what action your program takes when the user clicks the **Apply Now** button. When overriding, the function should return TRUE to accept changes and FALSE to prevent changes from taking effect.

The default implementation of `OnApply` returns TRUE.

CSnapInPropertyPageImpl::OnHelp

This member function is called when the user clicks the **Help** button for the property page.

```
void OnHelp();
```

Remarks

Override this member function to display help for the property page.

CSnapInPropertyPageImpl::OnKillActive

This member function is called when the page is no longer the active page.

```
BOOL OnKillActive();
```

Return Value

Nonzero if data was updated successfully; otherwise 0.

Remarks

Override this member function to perform special data validation tasks.

CSnapInPropertyPageImpl::OnQueryCancel

This member function is called when the user clicks the **Cancel** button and before the cancel action has taken place.

```
BOOL OnQueryCancel();
```

Return Value

Nonzero to allow the cancel operation; otherwise 0.

Remarks

Override this member function to specify an action the program takes when the user clicks the **Cancel** button.

The default implementation of `OnQueryCancel` returns TRUE.

CSnapInPropertyPageImpl::OnReset

This member function is called when the user clicks the **Cancel** button.

```
void OnReset();
```

Remarks

When this function is called, changes to all property pages that were made by the user previously clicking the **Apply Now** button are discarded, and the property sheet retains focus.

Override this member function to specify what action the program takes when the user clicks the **Cancel** button.

CSnapInPropertyPageImpl::OnSetActive

This member function is called when the page is chosen by the user and becomes the active page.

```
BOOL OnSetActive();
```

Return Value

Nonzero if the page was successfully set active; otherwise 0.

Remarks

Override this member function to perform tasks when a page is activated. Your override of this member function should call the default version before any other processing is done.

The default implementation returns TRUE.

CSnapInPropertyPageImpl::OnWizardBack

This member function is called when the user clicks the **Back** button in a wizard.

```
BOOL OnWizardBack();
```

Return Value

- 0 to automatically advance to the previous page.
- -1 to prevent the page from changing.

To jump to a page other than the next one, return the identifier of the dialog box to be displayed.

Remarks

Override this member function to specify some action the user must take when the **Back** button is clicked.

CSnapInPropertyPageImpl::OnWizardFinish

This member function is called when the user clicks the **Finish** button in a wizard.

```
BOOL OnWizardFinish();
```

Return Value

Nonzero if the property sheet is destroyed when the wizard finishes; otherwise zero.

Remarks

Override this member function to specify some action the user must take when the **Finish** button is clicked.

CSnapInPropertyPageImpl::OnWizardNext

This member function is called when the user clicks the **Next** button in a wizard.

```
BOOL OnWizardNext();
```

Return Value

- 0 to automatically advance to the next page.
- -1 to prevent the page from changing.

To jump to a page other than the next one, return the identifier of the dialog box to be displayed.

Remarks

Override this member function to specify some action the user must take when the **Next** button is clicked.

CSnapInPropertyPageImpl::QuerySiblings

Call this member function to forward a message to each page in the property sheet.

```
LRESULT QuerySiblings(WPARAM wParam, LPARAM lParam);
```

Parameters

wParam

[in] Specifies additional message-dependent information.

lParam

[in] Specifies additional message-dependent information.

Return Value

Nonzero if the message should not be forwarded to the next property page; otherwise zero.

Remarks

If a page returns a nonzero value, the property sheet does not send the message to subsequent pages.

CSnapInPropertyPageImpl::SetModified

Call this member function to enable or disable the **Apply Now** button, based on whether the settings in the property page should be applied to the appropriate external object.

```
void SetModified(BOOL bChanged = TRUE);
```

Parameters

bChanged

[in] TRUE to indicate that the property page settings have been modified since the last time they were applied;

FALSE to indicate that the property page settings have been applied, or should be ignored.

Remarks

The property sheet keeps track of which pages are "dirty," that is, property pages for which you have called

`SetModified( TRUE )`. The **Apply Now** button will always be enabled if you call `SetModified( TRUE )` for one of the pages. The **Apply Now** button will be disabled when you call `SetModified( FALSE )` for one of the pages, but only if none of the other pages is "dirty."

See also

[Class Overview](#)

CSocketAddr Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

This class provides methods for converting host names to host addresses, supporting both IPv4 and IPV6 formats.

Syntax

```
class CSocketAddr
```

Members

Public Constructors

NAME	DESCRIPTION
CSocketAddr::CSocketAddr	The constructor.

Public Methods

NAME	DESCRIPTION
CSocketAddr::FindAddr	Call this method to convert the provided host name to the host address.
CSocketAddr::FindINET4Addr	Call this method to convert the IPv4 host name to the host address.
CSocketAddr::FindINET6Addr	Call this method to convert the IPv6 host name to the host address.
CSocketAddr::GetAddrInfo	Call this method to return a pointer to a specific element in the <code>addrinfo</code> list.
CSocketAddr::GetAddrInfoList	Call this method to return a pointer to the <code>addrinfo</code> list.

Remarks

This class provides an IP version agnostic approach for looking up network addresses for use with Windows sockets API functions and socket wrappers in libraries.

The members of this class that are used to look up network addresses use the Win32 API function [getaddrinfo](#). The ANSI or UNICODE version of the function is called depending on whether your code is compiled for ANSI or UNICODE.

This class supports both IPv4 andIPv6 network addresses.

Requirements

Header: `atlsocket.h`

CSocketAddr::CSocketAddr

The constructor.

```
CSocketAddr();
```

Remarks

Creates a new `CSocketAddr` object and initializes the linked list containing response information about the host.

CSocketAddr::FindAddr

Call this method to convert the provided host name to the host address.

```
int FindAddr(
    const TCHAR *szHost,
    const TCHAR *szPortOrServiceName,
    int flags,
    int addr_family,
    int sock_type,
    int ai_proto);

int FindAddr(
    const TCHAR *szHost,
    int nPortNo,
    int flags,
    int addr_family,
    int sock_type,
    int ai_proto);
```

Parameters

szHost

The host name or dotted IP address.

szPortOrServiceName

The port number or name of service on host.

nPortNo

The port number.

flags

0 or combination of AI_PASSIVE, AI_CANONNAME or AI_NUMERICHOST.

addr_family

Address family (such as PF_INET).

sock_type

Socket type (such as SOCK_STREAM).

ai_proto

Protocol (such as IPPROTO_IP or IPPROTO_IPV6).

Return Value

Returns zero if the address is calculated successfully. Returns a nonzero Windows Socket error code on failure. If successful, the calculated address is stored in a linked list that may be referenced using

`CSocketAddr::GetAddrInfoList` and `CSocketAddr::GetAddrInfo`.

Remarks

The host name parameter may be in either IPv4 or IPv6 format. This method calls the Win32 API function [getaddrinfo](#) to perform the conversion.

CSocketAddr::FindINET4Addr

Call this method to convert the IPv4 host name to the host address.

```
int FindINET4Addr(  
    const TCHAR *szHost,  
    int nPortNo,  
    int flags = 0,  
    int sock_type = SOCK_STREAM);
```

Parameters

szHost

The host name or dotted IP address.

nPortNo

The port number.

flags

0 or combination of AI_PASSIVE, AI_CANONNAME or AI_NUMERICHOST.

sock_type

Socket type (such as SOCK_STREAM).

Return Value

Returns zero if the address is calculated successfully. Returns a nonzero Windows Socket error code on failure. If successful, the calculated address is stored in a linked list that may be referenced using

`CSocketAddr::GetAddrInfoList` and `CSocketAddr::GetAddrInfo`.

Remarks

This method calls the Win32 API function [getaddrinfo](#) to perform the conversion.

CSocketAddr::FindINET6Addr

Call this method to convert the IPv6 host name to the host address.

```
int FindINET6Addr(  
    const TCHAR *szHost,  
    int nPortNo,  
    int flags = 0,  
    int sock_type = SOCK_STREAM);
```

Parameters

szHost

The host name or dotted IP address.

nPortNo

The port number.

flags

0 or combination of AI_PASSIVE, AI_CANONNAME or AI_NUMERICHOST.

sock_type

Socket type (such as SOCK_STREAM).

Return Value

Returns zero if the address is calculated successfully. Returns a nonzero Windows Socket error code on failure. If successful, the calculated address is stored in a linked list that may be referenced using

`CSocketAddr::GetAddrInfoList` and `CSocketAddr::GetAddrInfo`.

Remarks

This method calls the Win32 API function [getaddrinfo](#) to perform the conversion.

CSocketAddr::GetAddrInfo

Call this method to return a pointer to a specific element in the `addrinfo` list.

```
addrinfo* const GetAddrInfo(int nIndex = 0) const;
```

Parameters

nIndex

A reference to a specific element in the [addrinfo](#) list.

Return Value

Returns a pointer to the `addrinfo` structure referenced by *nIndex* in the linked list containing response information about the host.

CSocketAddr::GetAddrInfoList

Call this method to return a pointer to the `addrinfo` list.

```
addrinfo* const GetAddrInfoList() const;
```

Return Value

Pointer to a linked list of one or more `addrinfo` structures containing response information about the host. For more information, see [addrinfo structure](#).

See also

[Class Overview](#)

CStockPropImpl Class

5/7/2019 • 17 minutes to read • [Edit Online](#)

This class provides methods for supporting stock property values.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <
    class T,
    class InterfaceName,
    const IID* piid = &_ATL_IIDOF(InterfaceName),
    const GUID* plibid = &CComModule::m_libid,
    WORD wMajor = 1,
    WORD wMinor = 0,
    class tihclass = CcomTypeInfoHolder>
class ATL_NO_VTABLE CStockPropImpl :
    public IDispatchImpl<InterfaceName, piid, plibid, wMajor, wMinor, tihclass>
```

Parameters

T

The class implementing the control and deriving from `CStockPropImpl`.

InterfaceName

A dual interface exposing the stock properties.

piid

A pointer to the IID of `InterfaceName`.

plibid

A pointer to the LIBID of the type library containing the definition of `InterfaceName`.

wMajor

The major version of the type library. The default value is 1.

wMinor

The minor version of the type library. The default value is 0.

tihclass

The class used to manage the type information for *T*. The default value is `CComTypeInfoHolder`.

Members

Public Methods

<code>get_Appearance</code>	Call this method to get the paint style used by the control, for example, flat or 3D.

get_AutoSize	Call this method to get the status of the flag that indicates if the control cannot be any other size.
get_BackColor	Call this method to get the control's background color.
get_BackStyle	Call this method to get the control's background style, either transparent or opaque.
get_BorderColor	Call this method to get the control's border color.
get_BorderStyle	Call this method to get the control's border style.
get_BorderVisible	Call this method to get the status of the flag that indicates if the control's border is visible or not.
get_BorderWidth	Call this method to get the width (in pixels) of the control's border.
get_Caption	Call this method to get the text specified in an object's caption.
get_DrawMode	Call this method to get the control's drawing mode, for example, XOR Pen or Invert Colors.
get_DrawStyle	Call this method to get the control's drawing style, for example, solid, dashed, or dotted.
get_DrawWidth	Call this method to get the drawing width (in pixels) used by the control's drawing methods.
get_Enabled	Call this method to get the status of the flag that indicates if the control is enabled.
get_FillColor	Call this method to get the control's fill color.
get_FillStyle	Call this method to get the control's fill style, for example, solid, transparent, or cross-hatched.
get_Font	Call this method to get a pointer to the control's font properties.
get_ForeColor	Call this method to get the control's foreground color.
get_HWND	Call this method to get the window handle associated with the control.
get_MouseIcon	Call this method to get the picture properties of the graphic (icon, bitmap, or metafile) to be displayed when the mouse is over the control.
get_MousePointer	Call this method to get the type of mouse pointer displayed when the mouse is over the control, for example, arrow, cross, or hourglass.

get_Picture	Call this method to get a pointer to the picture properties of a graphic (icon, bitmap, or metafile) to be displayed.
get_ReadyState	Call this method to get the control's ready state, for example, loading or loaded.
get_TabStop	Call this method to get the flag that indicates if the control is a tab stop or not.
get_Text	Call this method to get the text that is displayed with the control.
getvalid	Call this method to get the status of the flag that indicates if the control is valid or not.
get_Window	Call this method to get the window handle associated with the control. Identical to CStockPropImpl::get_HWND .
put_Appearance	Call this method to set the paint style used by the control, for example, flat or 3D.
put_AutoSize	Call this method to set the value of the flag that indicates if the control cannot be any other size.
put_BackColor	Call this method to set the control's background color.
put_BackStyle	Call this method to set the control's background style.
put_BorderColor	Call this method to set the control's border color.
put_BorderStyle	Call this method to set the control's border style.
put_BorderVisible	Call this method to set the value of the flag that indicates if the control's border is visible or not.
put_BorderWidth	Call this method to set the width of the control's border.
put_Caption	Call this method to set the text to be displayed with the control.
put_DrawMode	Call this method to set the control's drawing mode, for example, XOR Pen or Invert Colors.
put_DrawStyle	Call this method to set the control's drawing style, for example, solid, dashed, or dotted.
put_DrawWidth	Call this method to set the width (in pixels) used by the control's drawing methods.
put_Enabled	Call this method to set the flag that indicates if the control is enabled.
put_FillColor	Call this method to set the control's fill color.

put_FillStyle	Call this method to set the control's fill style, for example, solid, transparent, or cross-hatched.
put_Font	Call this method to set the control's font properties.
put_ForeColor	Call this method to set the control's foreground color.
put_HWND	This method returns E_FAIL.
put_MouseIcon	Call this method to set the picture properties of the graphic (icon, bitmap, or metafile) to be displayed when the mouse is over the control.
put_MousePointer	Call this method to set the type of mouse pointer displayed when the mouse is over the control, for example, arrow, cross, or hourglass.
put_Picture	Call this method to set the picture properties of a graphic (icon, bitmap, or metafile) to be displayed.
put_ReadyState	Call this method to set the control's ready state, for example, loading or loaded.
put_TabStop	Call this method to set the value of the flag that indicates if the control is a tab stop or not.
put_Text	Call this method to set the text that is displayed with the control.
putvalid	Call this method to set the flag that indicates if the control is valid or not.
put_Window	This method calls CStockPropImpl::put_HWND , which returns E_FAIL.
putref_Font	Call this method to set the control's font properties, with a reference count.
putref_MouseIcon	Call this method to set the picture properties of the graphic (icon, bitmap, or metafile) to be displayed when the mouse is over the control, with a reference count.
putref_Picture	Call this method to set the picture properties of a graphic (icon, bitmap, or metafile) to be displayed, with a reference count.

Remarks

[CStockPropImpl](#) provides **put** and **get** methods for each stock property. These methods provide the code necessary to set or get the data member associated with each property and to notify and synchronize with the container when any property changes.

Visual Studio provides support for stock properties through its wizards. For more information about adding stock properties to a control, see the [ATL Tutorial](#).

For backward compatibility, `CStockPropImpl` also exposes `get_Window` and `put_Window` methods that simply call `get_HWND` and `put_HWND`, respectively. The default implementation of `put_HWND` returns `E_FAIL` since `HWND` should be a read-only property.

The following properties also have a **putref** implementation:

- Font
- Mouselcon
- Picture

The same three stock properties require their corresponding data member to be of type `CComPtr` or some other class that provides correct interface reference counting by means of the assignment operator.

Inheritance Hierarchy

T

[IDispatchImpl](#)

`CStockPropImpl`

Requirements

Header: atlctl.h

CStockPropImpl::get_Appearance

Call this method to get the paint style used by the control, for example, flat or 3D.

```
HRESULT STDMETHODCALLTYPE get_Appearance(SHORT pnAppearance);
```

Parameters

pnAppearance

Variable that receives the control's paint style.

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure.

CStockPropImpl::get_AutoSize

Call this method to get the status of the flag that indicates if the control cannot be any other size.

```
HRESULT STDMETHODCALLTYPE get_Autosize(VARIANT_BOOL* pbAutoSize);
```

Parameters

pbAutoSize

Variable that receives the flag status. `TRUE` indicates that the control cannot be any other size.

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure.

CStockPropImpl::get_BackColor

Call this method to get the control's background color.

```
HRESULT STDMETHODCALLTYPE get_BackColor(OLE_COLOR* pClrBackColor);
```

Parameters

pClrBackColor

Variable that receives the control's background color.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_BackStyle

Call this method to get the control's background style, either transparent or opaque.

```
HRESULT STDMETHODCALLTYPE get_BackStyle(LONG* pnBackStyle);
```

Parameters

pnBackStyle

Variable that receives the control's background style.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_BorderColor

Call this method to get the control's border color.

```
HRESULT STDMETHODCALLTYPE get_BorderColor(OLE_COLOR* pClrBorderColor);
```

Parameters

pClrBorderColor

Variable that receives the control's border color.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_BorderStyle

Call this method to get the control's border style.

```
HRESULT STDMETHODCALLTYPE get_BorderStyle(LONG* pnBorderStyle);
```

Parameters

pnBorderStyle

Variable that receives the control's border style.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_BorderVisible

Call this method to get the status of the flag that indicates if the control's border is visible or not.

```
HRESULT STDMETHODCALLTYPE get_BorderVisible(VARIANT_BOOL* pbBorderVisible);
```

Parameters

pbBorderVisible

Variable that receives the flag status. TRUE indicates that the control's border is visible.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_BorderWidth

Call this method to get the width of the control's border.

```
HRESULT STDMETHODCALLTYPE get_BorderWidth(LONG* pnBorderWidth);
```

Parameters

pnBorderWidth

Variable that receives the control's border width.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_Caption

Call this method to get the text specified in an object's caption.

```
HRESULT STDMETHODCALLTYPE get_Caption(BSTR* pbstrCaption);
```

Parameters

pbstrCaption

The text to be displayed with the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_DrawMode

Call this method to get the control's drawing mode, for example, XOR Pen or Invert Colors.

```
HRESULT STDMETHODCALLTYPE get_DrawMode(LONG* pnDrawMode);
```

Parameters

pnDrawMode

Variable that receives the control's drawing mode.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_DrawStyle

Call this method to get the control's drawing style, for example, solid, dashed, or dotted.

```
HRESULT STDMETHODCALLTYPE get_DrawStyle(LONG* pnDrawStyle);
```

Parameters

pnDrawStyle

Variable that receives the control's drawing style.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_DrawWidth

Call this method to get the drawing width (in pixels) used by the control's drawing methods.

```
HRESULT STDMETHODCALLTYPE get_DrawWidth(LONG* pnDrawWidth);
```

Parameters

pnDrawWidth

Variable that receives the control's width value, in pixels.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_Enabled

Call this method to get the status of the flag that indicates if the control is enabled.

```
HRESULT STDMETHODCALLTYPE get_Enabled(VARIANT_BOOL* pbEnabled);
```

Parameters

pbEnabled

Variable that receives the flag status. TRUE indicates that the control is enabled.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_FillColor

Call this method to get the control's fill color.

```
HRESULT STDMETHODCALLTYPE get_FillColor(OLE_COLOR* pClrFillColor);
```

Parameters

pClrFillColor

Variable that receives the control's fill color.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_FillStyle

Call this method to get the control's fill style, for example, solid, transparent, or crosshatched.

```
HRESULT STDMETHODCALLTYPE get_FillStyle(LONG* pnFillStyle);
```

Parameters

pnFillStyle

Variable that receives the control's fill style.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_Font

Call this method to get a pointer to the control's font properties.

```
HRESULT STDMETHODCALLTYPE get_Font(IFontDisp** ppFont);
```

Parameters

ppFont

Variable that receives a pointer to the control's font properties.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_ForeColor

Call this method to get the control's foreground color.

```
HRESULT STDMETHODCALLTYPE get_ForeColor(OLE_COLOR* pclrForeColor);
```

Parameters

pclrForeColor

Variable that receives the controls foreground color.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_HWND

Call this method to get the window handle associated with the control.

```
HRESULT STDMETHODCALLTYPE get_HWND(LONG_PTR* phWnd);
```

Parameters

phWnd

The window handle associated with the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_MouseIcon

Call this method to get the picture properties of the graphic (icon, bitmap, or metafile) to be displayed when the mouse is over the control.

```
HRESULT STDMETHODCALLTYPE get_MouseIcon(IPictureDisp** ppPicture);
```

Parameters

ppPicture

Variable that receives a pointer to the picture properties of the graphic.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_MousePointer

Call this method to get the type of mouse pointer displayed when the mouse is over the control, for example, arrow, cross, or hourglass.

```
HRESULT STDMETHODCALLTYPE get_MousePointer(LONG* pnMousePointer);
```

Parameters

pnMousePointer

Variable that receives the type of mouse pointer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_Picture

Call this method to get a pointer to the picture properties of a graphic (icon, bitmap, or metafile) to be displayed.

```
HRESULT STDMETHODCALLTYPE get_Picture(IPictureDisp** ppPicture);
```

Parameters

ppPicture

Variable that receives a pointer to the picture's properties. See [IPictureDisp](#) for more details.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_ReadyState

Call this method to get the control's ready state, for example, loading or loaded.

```
HRESULT STDMETHODCALLTYPE get_ReadyState(LONG* pnReadyState);
```

Parameters

pnReadyState

Variable that receives the control's ready state.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_TabStop

Call this method to get the status of the flag that indicates if the control is a tab stop or not.

```
HRESULT STDMETHODCALLTYPE get_TabStop(VARIANT_BOOL* pbTabStop);
```

Parameters

pbTabStop

Variable that receives the flag status. TRUE indicates that the control is a tab stop.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_Text

Call this method to get the text that is displayed with the control.

```
HRESULT STDMETHODCALLTYPE get_Text(BSTR* pbstrText);
```

Parameters

pbstrText

The text that is displayed with the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::getvalid

Call this method to get the status of the flag that indicates if the control is valid or not.

```
HRESULT STDMETHODCALLTYPE getvalid(VARIANT_BOOL* pbValid);
```

Parameters

pbValid

Variable that receives the flag status. TRUE indicates that the control is valid.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::get_Window

Call this method to get the window handle associated with the control. Identical to [CStockPropImpl::get_HWND](#).

```
HRESULT STDMETHODCALLTYPE get_Window(LONG_PTR* phWnd);
```

Parameters

phWnd

The window handle associated with the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_Appearance

Call this method to set the paint style used by the control, for example, flat or 3D.

```
HRESULT STDMETHODCALLTYPE put_Appearance(SHORT nAppearance);
```

Parameters

nAppearance

The new paint style to be used by the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_AutoSize

Call this method to set the value of flag that indicates if the control cannot be any other size.

```
HRESULT STDMETHODCALLTYPE put_AutoSize(VARIANT_BOOL bAutoSize,);
```

Parameters

bAutoSize

TRUE if the control cannot be any other size.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_BackColor

Call this method to set the control's background color.

```
HRESULT STDMETHODCALLTYPE put_BackColor(OLE_COLOR clrBackColor);
```

Parameters

clrBackColor

The new control background color.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_BackStyle

Call this method to set the control's background style.

```
HRESULT STDMETHODCALLTYPE put_BackStyle(LONG nBackStyle);
```

Parameters

nBackStyle

The new control background style.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_BorderColor

Call this method to set the control's border color.

```
HRESULT STDMETHODCALLTYPE put_BorderColor(OLE_COLOR clrBorderColor);
```

Parameters

clrBorderColor

The new border color. The OLE_COLOR data type is internally represented as a 32-bit long integer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_BorderStyle

Call this method to set the control's border style.

```
HRESULT STDMETHODCALLTYPE put_BorderStyle(LONG nBorderStyle);
```

Parameters

nBorderStyle

The new border style.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_BorderVisible

Call this method to set the value of the flag that indicates if the control's border is visible or not.

```
HRESULT STDMETHODCALLTYPE put_BorderVisible(VARIANT_BOOL bBorderVisible);
```

Parameters

bBorderVisible

TRUE if the border is to be visible.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_BorderWidth

Call this method to set the width of the control's border.

```
HRESULT STDMETHODCALLTYPE put_BorderWidth(LONG nBorderWidth);
```

Parameters

nBorderWidth

The new width of the control's border.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_Caption

Call this method to set the text to be displayed with the control.

```
HRESULT STDMETHODCALLTYPE put_Caption(BSTR bstrCaption);
```

Parameters

bstrCaption

The text to be displayed with the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_DrawMode

Call this method to set the control's drawing mode, for example, XOR Pen or Invert Colors.

```
HRESULT STDMETHODCALLTYPE put_DrawMode(LONG nDrawMode);
```

Parameters

nDrawMode

The new drawing mode for the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_DrawStyle

Call this method to set the control's drawing style, for example, solid, dashed, or dotted.

```
HRESULT STDMETHODCALLTYPE put_DrawStyle(LONG pnDrawStyle);
```

Parameters

nDrawStyle

The new drawing style for the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_DrawWidth

Call this method to set the width (in pixels) used by the control's drawing methods.

```
HRESULT STDMETHODCALLTYPE put_DrawWidth(LONG nDrawWidth);
```

Parameters

nDrawWidth

The new width to be used by the control's drawing methods.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_Enabled

Call this method to set the value of the flag that indicates if the control is enabled.

```
HRESULT STDMETHODCALLTYPE put_Enabled(VARIANT_BOOL bEnabled);
```

Parameters

bEnabled

TRUE if the control is enabled.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_FillColor

Call this method to set the control's fill color.

```
HRESULT STDMETHODCALLTYPE put_FillColor(OLE_COLOR clrFillColor);
```

Parameters

clrFillColor

The new fill color for the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_FillStyle

Call this method to set the control's fill style, for example, solid, transparent, or cross-hatched.

```
HRESULT STDMETHODCALLTYPE put_FillStyle(LONG nFillStyle);
```

Parameters

nFillStyle

The new fill style for the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_Font

Call this method to set the control's font properties.

```
HRESULT STDMETHODCALLTYPE put_Font(IFontDisp* pFont);
```

Parameters

pFont

A pointer to the control's font properties.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_ForeColor

Call this method to set the control's foreground color.

```
HRESULT STDMETHODCALLTYPE put_ForeColor(OLE_COLOR clrForeColor);
```

Parameters

clrForeColor

The new foreground color of the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_HWND

This method returns E_FAIL.

```
HRESULT STDMETHODCALLTYPE put_HWND(LONG_PTR /* hWnd */);
```

Parameters

hWnd

Reserved.

Return Value

Returns E_FAIL.

Remarks

The window handle is a read-only value.

CStockPropImpl::put_MouseIcon

Call this method to set the picture properties of the graphic (icon, bitmap, or metafile) to be displayed when the mouse is over the control.

```
HRESULT STDMETHODCALLTYPE put_MouseIcon(IPictureDisp* pPicture);
```

Parameters

pPicture

A pointer to the picture properties of the graphic.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_MousePointer

Call this method to set the type of mouse pointer displayed when the mouse is over the control, for example, arrow, cross, or hourglass.

```
HRESULT STDMETHODCALLTYPE put_MousePointer(LONG nMousePointer);
```

Parameters

nMousePointer

The type of mouse pointer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_Picture

Call this method to set the picture properties of a graphic (icon, bitmap, or metafile) to be displayed.

```
HRESULT STDMETHODCALLTYPE put_Picture(IPictureDisp* pPicture);
```

Parameters

pPicture

A pointer to the picture's properties. See [IPictureDisp](#) for more details.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_ReadyState

Call this method to set the control's ready state, for example, loading or loaded.

```
HRESULT STDMETHODCALLTYPE put_ReadyState(LONG nReadyState);
```

Parameters

nReadyState

The control's ready state.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_TabStop

Call this method to set the flag that indicates if the control is a tab stop or not.

```
HRESULT STDMETHODCALLTYPE put_TabStop(VARIANT_BOOL bTabStop);
```

Parameters

bTabStop

TRUE if the control is a tab stop.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_Text

Call this method to set the text that is displayed with the control.

```
HRESULT STDMETHODCALLTYPE put_Text(BSTR bstrText);
```

Parameters

bstrText

The text that is displayed with the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::putvalid

Call this method to set the flag that indicates if the control is valid or not.

```
HRESULT STDMETHODCALLTYPE getvalid(VARIANT_BOOL bValid);
```

Parameters

bValid

TRUE if the control is valid.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CStockPropImpl::put_Window

This method calls [CStockPropImpl::put_HWND](#), which returns E_FAIL.

```
HRESULT STDMETHODCALLTYPE put_Window(LONG_PTR hWnd);
```

Parameters

hWnd

The window handle.

Return Value

Returns E_FAIL.

Remarks

The window handle is a read-only value.

CStockPropImpl::putref_Font

Call this method to set the control's font properties, with a reference count.

```
HRESULT STDMETHODCALLTYPE putref_Font(IFontDisp* pFont);
```

Parameters

pFont

A pointer to the control's font properties.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The same as [CStockPropImpl::put_Font](#), but with a reference count.

CStockPropImpl::putref_MouseIcon

Call this method to set the picture properties of the graphic (icon, bitmap, or metafile) to be displayed when the mouse is over the control, with a reference count.

```
HRESULT STDMETHODCALLTYPE putref_MouseIcon(IPictureDisp* pPicture);
```

Parameters

pPicture

A pointer to the picture properties of the graphic.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The same as [CStockPropImpl::put_MouseIcon](#), but with a reference count.

CStockPropImpl::putref_Picture

Call this method to set the picture properties of a graphic (icon, bitmap, or metafile) to be displayed, with a reference count.

```
HRESULT STDMETHODCALLTYPE putref_Picture(IPictureDisp* pPicture);
```

Parameters

pPicture

A pointer to the picture's properties. See [IPictureDisp](#) for more details.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The same as [CStockPropImpl::put_Picture](#), but with a reference count.

See also

[Class Overview](#)

[IDispatchImpl Class](#)

CStringElementTraits Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides static functions used by collection classes storing `CString` objects.

Syntax

```
template <typename T>
class CStringElementTraits
```

Parameters

T

The type of data to be stored in the collection.

Members

Public Typedefs

NAME	DESCRIPTION
CStringElementTraits::INARGTYPE	The data type to use for adding elements to the collection class object.
CStringElementTraits::OUTARGTYPE	The data type to use for retrieving elements from the collection class object.

Public Methods

NAME	DESCRIPTION
CStringElementTraits::CompareElements	(Static) Call this function to compare two string elements for equality.
CStringElementTraits::CompareElementsOrdered	(Static) Call this function to compare two string elements.
CStringElementTraits::CopyElements	(Static) Call this function to copy <code>CString</code> elements stored in a collection class object.
CStringElementTraits::Hash	(Static) Call this function to calculate a hash value for the given string element.
CStringElementTraits::RelocateElements	(Static) Call this function to relocate <code>CString</code> elements stored in a collection class object.

Remarks

This class provides static functions for copying, moving, and comparing strings and for creating a hash value. These functions are useful when using a collection class to store string-based data. Use [CStringElementTraits](#) when case-insensitive comparisons are required. Use [CStringRefElementTraits](#) when the string objects are to be dealt with as references.

For more information, see [ATL Collection Classes](#).

Requirements

Header: cstringt.h

CStringElementTraits::CompareElements

Call this static function to compare two string elements for equality.

```
static bool CompareElements(INARGTYPE str1, INARGTYPE str2);
```

Parameters

str1

The first string element.

str2

The second string element.

Return Value

Returns true if the elements are equal, false otherwise.

CStringElementTraits::CompareElementsOrdered

Call this static function to compare two string elements.

```
static int CompareElementsOrdered(INARGTYPE str1, INARGTYPE str2);
```

Parameters

str1

The first string element.

str2

The second string element.

Return Value

Zero if the strings are identical, < 0 if *str1* is less than *str2*, or > 0 if *str1* is greater than *str2*. The [CStringT::Compare](#) method is used to perform the comparisons.

CStringElementTraits::CopyElements

Call this static function to copy `CString` elements stored in a collection class object.

```
static void CopyElements(  
    T* pDest,  
    const T* pSrc,  
    size_t nElements);
```

Parameters

pDest

Pointer to the first element that will receive the copied data.

pSrc

Pointer to the first element to copy.

nElements

The number of elements to copy.

Remarks

The source and destination elements should not overlap.

CStringElementTraits::Hash

Call this static function to calculate a hash value for the given string element.

```
static ULONG Hash(INARGTYPE str);
```

Parameters

str

The string element.

Return Value

Returns a hash value, calculated using the string's contents.

CStringElementTraits::INARGTYPE

The data type to use for adding elements to the collection class object.

```
typedef T::PCWSTR INARGTYPE;
```

CStringElementTraits::OUTARGTYPE

The data type to use for retrieving elements from the collection class object.

```
typedef T& OUTARGTYPE;
```

CStringElementTraits::RelocateElements

Call this static function to relocate `CString` elements stored in a collection class object.

```
static void RelocateElements(  
    T* pDest,  
    T* pSrc,  
    size_t nElements);
```

Parameters

pDest

Pointer to the first element that will receive the relocated data.

pSrc

Pointer to the first element to relocate.

nElements

The number of elements to relocate.

Remarks

This static function calls [memmove](#), which is sufficient for most data types. If the objects being moved contain pointers to their own members, this static function will need to be overridden.

See also

[CElementTraitsBase Class](#)

[CStringElementTraitsI Class](#)

[Class Overview](#)

CStringElementTraitsI Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides static functions related to strings stored in collection class objects. It is similar to [CStringElementTraits](#), but performs case-insensitive comparisons.

Syntax

```
template <typename T, class CharTraits = CDefaultCharTraits<T>::XCHAR>>
class CStringElementTraitsI : public CElementTraitsBase<T>
```

Parameters

T

The type of data to be stored in the collection.

Members

Public Typedefs

NAME	DESCRIPTION
CStringElementTraitsI::INARGTYPE	The data type to use for adding elements to the collection class object.
CStringElementTraitsI::OUTARGTYPE	The data type to use for retrieving elements from the collection class object.

Public Methods

NAME	DESCRIPTION
CStringElementTraitsI::CompareElements	Call this static function to compare two string elements for equality, ignoring differences in case.
CStringElementTraitsI::CompareElementsOrdered	Call this static function to compare two string elements, ignoring differences in case.
CStringElementTraitsI::Hash	Call this static function to calculate a hash value for the given string element.

Remarks

This class provides static functions for comparing strings and for creating a hash value. These functions are useful when using a collection class to store string-based data. Use [CStringRefElementTraits](#) when the string objects are to be with dealt with as references.

For more information, see [ATL Collection Classes](#).

Inheritance Hierarchy

Requirements

Header: atcoll.h

CStringElementTraitsI::CompareElements

Call this static function to compare two string elements for equality, ignoring differences in case.

```
static bool CompareElements(INARGTYPE str1, INARGTYPE str2) throw();
```

Parameters

str1

The first string element.

str2

The second string element.

Return Value

Returns true if the elements are equal, false otherwise.

Remarks

Comparisons are case insensitive.

CStringElementTraitsI::CompareElementsOrdered

Call this static function to compare two string elements, ignoring differences in case.

```
static int CompareElementsOrdered(INARGTYPE str1, INARGTYPE str2) throw();
```

Parameters

str1

The first string element.

str2

The second string element.

Return Value

Zero if the strings are identical, < 0 if *str1* is less than *str2*, or > 0 if *str1* is greater than *str2*. The [CStringT::Compare](#) method is used to perform the comparisons.

Remarks

Comparisons are case insensitive.

CStringElementTraitsI::Hash

Call this static function to calculate a hash value for the given string element.

```
static ULONG Hash(INARGTYPE str) throw();
```

Parameters

str

The string element.

Return Value

Returns a hash value, calculated using the string's contents.

CStringElementTraitsI::INARGTYPE

The data type to use for adding elements to the collection class object.

```
typedef T::PCWSTR INARGTYPE;
```

CStringElementTraitsI::OUTARGTYPE

The data type to use for retrieving elements from the collection class object.

```
typedef T& OUTARGTYPE;
```

See also

[CElementTraitsBase Class](#)

[Class Overview](#)

[CStringElementTraits Class](#)

CStringRefElementTraits Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides static functions related to strings stored in collection class objects. The string objects are dealt with as references.

Syntax

```
template <typename T>
class CStringRefElementTraits : public CElementTraitsBase<T>
```

Parameters

T

The type of data to be stored in the collection.

Members

Public Methods

NAME	DESCRIPTION
CStringRefElementTraits::CompareElements	Call this static function to compare two string elements for equality.
CStringRefElementTraits::CompareElementsOrdered	Call this static function to compare two string elements.
CStringRefElementTraits::Hash	Call this static function to calculate a hash value for the given string element.

Remarks

This class provides static functions for comparing strings and for creating a hash value. These functions are useful when using a collection class to store string-based data. Unlike [CStringElementTraits](#) and [CStringElementTraitsI](#),

`CStringRefElementTraits` causes the `CString` arguments to be passed as **const** `CString&` references.

For more information, see [ATL Collection Classes](#).

Inheritance Hierarchy

[CElementTraitsBase](#)

`CStringRefElementTraits`

Requirements

Header: `atlcoll.h`

CStringRefElementTraits::CompareElements

Call this static function to compare two string elements for equality.

```
static bool CompareElements(INARGTYPE element1, INARGTYPE element2) throw();
```

Parameters

element1

The first string element.

element2

The second string element.

Return Value

Returns true if the elements are equal, false otherwise.

CStringRefElementTraits::CompareElementsOrdered

Call this static function to compare two string elements.

```
static int CompareElementsOrdered(INARGTYPE str1, INARGTYPE str2) throw();
```

Parameters

str1

The first string element.

str2

The second string element.

Return Value

Zero if the strings are identical, < 0 if *str1* is less than *str2*, or > 0 if *str1* is greater than *str2*. The [CStringT::Compare](#) method is used to perform the comparisons.

CStringRefElementTraits::Hash

Call this static function to calculate a hash value for the given string element.

```
static ULONG Hash(INARGTYPE str) throw();
```

Parameters

str

The string element.

Return Value

Returns a hash value, calculated using the string's contents.

See also

[CElementTraitsBase Class](#)

[Class Overview](#)

CThreadPool Class

3/4/2019 • 7 minutes to read • [Edit Online](#)

This class provides a pool of worker threads that process a queue of work items.

Syntax

```
template <class Worker, class ThreadTraits = DefaultThreadTraits>
class CThreadPool : public IThreadPoolConfig
```

Parameters

Worker

The class conforming to the [worker archetype](#) providing the code used to process work items queued on the thread pool.

ThreadTraits

The class providing the function used to create the threads in the pool.

Members

Public Constructors

NAME	DESCRIPTION
CThreadPool::CThreadPool	The constructor for the thread pool.
CThreadPool::~~CThreadPool	The destructor for the thread pool.

Public Methods

NAME	DESCRIPTION
CThreadPool::AddRef	Implementation of <code>IUnknown::AddRef</code> .
CThreadPool::GetNumThreads	Call this method to get the number of threads in the pool.
CThreadPool::GetQueueHandle	Call this method to get the handle of the IO completion port used to queue work items.
CThreadPool::GetSize	Call this method to get the number of threads in the pool.
CThreadPool::GetTimeout	Call this method to get the maximum time in milliseconds that the thread pool will wait for a thread to shut down.
CThreadPool::Initialize	Call this method to initialize the thread pool.
CThreadPool::QueryInterface	Implementation of <code>IUnknown::QueryInterface</code> .
CThreadPool::QueueRequest	Call this method to queue a work item to be handled by a thread in the pool.

NAME	DESCRIPTION
CThreadPool::Release	Implementation of <code>IUnknown::Release</code> .
CThreadPool::SetSize	Call this method to set the number of threads in the pool.
CThreadPool::SetTimeout	Call this method to set the maximum time in milliseconds that the thread pool will wait for a thread to shut down.
CThreadPool::Shutdown	Call this method to shut down the thread pool.

Remarks

Threads in the pool are created and destroyed when the pool is initialized, resized, or shut down. An instance of class *Worker* will be created on the stack of each worker thread in the pool. Each instance will live for the lifetime of the thread.

Immediately after creation of a thread, *Worker::Initialize* will be called on the object associated with that thread. Immediately before destruction of a thread, *Worker::Terminate* will be called. Both methods must accept a **void*** argument. The value of this argument is passed to the thread pool through the *pvWorkerParam* parameter of [CThreadPool::Initialize](#).

When there are work items in the queue and worker threads available for work, a worker thread will pull an item off the queue and call the `Execute` method of the *Worker* object for that thread. Three items are then passed to the method: the item from the queue, the same `pvWorkerParam` passed to *Worker::Initialize* and *Worker::Terminate*, and a pointer to the `OVERLAPPED` structure used for the IO completion port queue.

The *Worker* class declares the type of the items that will be queued on the thread pool by providing a typedef, *Worker::RequestType*. This type must be capable of being cast to and from a `ULONG_PTR`.

An example of a *Worker* class is [CNonStatelessWorker Class](#).

Inheritance Hierarchy

`IUnknown`

[IThreadPoolConfig](#)

`CThreadPool`

Requirements

Header: `atlutil.h`

CThreadPool::AddRef

Implementation of `IUnknown::AddRef`.

```
ULONG STDMETHODCALLTYPE AddRef() throw();
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control using reference counting.

CThreadPool::CThreadPool

The constructor for the thread pool.

```
CThreadPool() throw();
```

Remarks

Initializes the timeout value to ATLS_DEFAULT_THREADPOOLSHUTDOWNTIMEOUT. The default time is 36 seconds. If necessary, you can define your own positive integer value for this symbol before including atlutil.h.

CThreadPool::~CThreadPool

The destructor for the thread pool.

```
~CThreadPool() throw();
```

Remarks

Calls [CThreadPool::Shutdown](#).

CThreadPool::GetNumThreads

Call this method to get the number of threads in the pool.

```
int GetNumThreads() throw();
```

Return Value

Returns the number of threads in the pool.

CThreadPool::GetQueueHandle

Call this method to get the handle of the IO completion port used to queue work items.

```
HANDLE GetQueueHandle() throw();
```

Return Value

Returns the queue handle or NULL if the thread pool has not been initialized.

CThreadPool::GetSize

Call this method to get the number of threads in the pool.

```
HRESULT STDMETHODCALLTYPE GetSize(int* pnNumThreads) throw();
```

Parameters

pnNumThreads

[out] Address of the variable that, on success, receives the number of threads in the pool.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CThreadPool::GetTimeout

Call this method to get the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

```
HRESULT STDMETHODCALLTYPE GetTimeout(DWORD* pdwMaxWait) throw();
```

Parameters

pdwMaxWait

[out] Address of the variable that, on success, receives the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This timeout value is used by [CThreadPool::Shutdown](#) if no other value is supplied to that method.

CThreadPool::Initialize

Call this method to initialize the thread pool.

```
HRESULT Initialize(  
    void* pvWorkerParam = NULL,  
    int nNumThreads = 0,  
    DWORD dwStackSize = 0,  
    HANDLE hCompletion = INVALID_HANDLE_VALUE) throw();
```

Parameters

pvWorkerParam

The worker parameter to be passed to the worker thread object's `Initialize`, `Execute`, and `Terminate` methods.

nNumThreads

The requested number of threads in the pool.

If *nNumThreads* is negative, its absolute value will be multiplied by the number of processors in the machine to get the total number of threads.

If *nNumThreads* is zero, `ATLS_DEFAULT_THREADSPERPROC` will be multiplied by the number of processors in the machine to get the total number of threads. The default is 2 threads per processor. If necessary, you can define your own positive integer value for this symbol before including `atlutil.h`.

dwStackSize

The stack size for each thread in the pool.

hCompletion

The handle of an object to associate with the completion port.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

CThreadPool::QueryInterface

Implementation of `IUnknown::QueryInterface` .

```
HRESULT STDMETHODCALLTYPE QueryInterface(REFIID riid, void** ppv) throw();
```

Remarks

Objects of this class can be successfully queried for the `IUnknown` and `IThreadPoolConfig` interfaces.

CThreadPool::QueueRequest

Call this method to queue a work item to be handled by a thread in the pool.

```
BOOL QueueRequest(Worker::RequestType request) throw();
```

Parameters

request

The request to be queued.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This method adds a work item to the queue. The threads in the pool pick items off the queue in the order in which they are received.

CThreadPool::Release

Implementation of `IUnknown::Release` .

```
ULONG STDMETHODCALLTYPE Release() throw();
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control using reference counting.

CThreadPool::SetSize

Call this method to set the number of threads in the pool.

```
HRESULT STDMETHODCALLTYPE SetSize(int nNumThreads) throw();
```

Parameters

nNumThreads

The requested number of threads in the pool.

If *nNumThreads* is negative, its absolute value will be multiplied by the number of processors in the machine to get the total number of threads.

If *nNumThreads* is zero, `ATLS_DEFAULT_THREADSPERPROC` will be multiplied by the number of processors in the machine to get the total number of threads. The default is 2 threads per processor. If necessary, you can define your own positive integer value for this symbol before including `atlutil.h`.

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure.

Remarks

If the number of threads specified is less than the number of threads currently in the pool, the object puts a shutdown message on the queue to be picked up by a waiting thread. When a waiting thread pulls the message off the queue, it notifies the thread pool and exits the thread procedure. This process is repeated until the number of threads in the pool reaches the specified number or until no thread has exited within the period specified by [GetTimeout](#)/ [SetTimeout](#). In this situation the method will return an `HRESULT` corresponding to `WAIT_TIMEOUT` and the pending shutdown message is canceled.

CThreadPool::SetTimeout

Call this method to set the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

```
HRESULT STDMETHODCALLTYPE SetTimeout(DWORD dwMaxWait) throw();
```

Parameters

dwMaxWait

The requested maximum time in milliseconds that the thread pool will wait for a thread to shut down.

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure.

Remarks

The timeout is initialized to `ATLS_DEFAULT_THREADPOOLSHUTDOWNTIMEOUT`. The default time is 36 seconds. If necessary, you can define your own positive integer value for this symbol before including `atlutil.h`.

Note that *dwMaxWait* is the time that the pool will wait for a single thread to shut down. The maximum time that could be taken to remove multiple threads from the pool could be slightly less than *dwMaxWait* multiplied by the number of threads.

CThreadPool::Shutdown

Call this method to shut down the thread pool.

```
void Shutdown(DWORD dwMaxWait = 0) throw();
```

Parameters

dwMaxWait

The requested maximum time in milliseconds that the thread pool will wait for a thread to shut down. If 0 or no value is supplied, this method will use the timeout set by [CThreadPool::SetTimeout](#).

Remarks

This method posts a shutdown request to all threads in the pool. If the timeout expires, this method will call [TerminateThread](#) on any thread that did not exit. This method is called automatically from the destructor of the class.

See also

[IThreadPoolConfig Interface](#)

[DefaultThreadTraits](#)

[Classes](#)

CTokenGroups Class

4/1/2019 • 3 minutes to read • [Edit Online](#)

This class is a wrapper for the `TOKEN_GROUPS` structure.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CTokenGroups
```

Members

Public Constructors

NAME	DESCRIPTION
CTokenGroups::CTokenGroups	The constructor.
CTokenGroups::~~CTokenGroups	The destructor.

Public Methods

NAME	DESCRIPTION
CTokenGroups::Add	Adds a <code>CSid</code> or existing <code>TOKEN_GROUPS</code> structure to the <code>CTokenGroups</code> object.
CTokenGroups::Delete	Deletes a <code>CSid</code> and its associated attributes from the <code>CTokenGroups</code> object.
CTokenGroups::DeleteAll	Deletes all <code>CSid</code> objects and their associated attributes from the <code>CTokenGroups</code> object.
CTokenGroups::GetCount	Returns the number of <code>CSid</code> objects and associated attributes contained in the <code>CTokenGroups</code> object.
CTokenGroups::GetLength	Returns the size of the <code>CTokenGroups</code> object.
CTokenGroups::GetPTOKEN_GROUPS	Retrieves a pointer to the <code>TOKEN_GROUPS</code> structure.
CTokenGroups::GetSidsAndAttributes	Retrieves the <code>CSid</code> objects and attributes belonging to the <code>CTokenGroups</code> object.
CTokenGroups::LookupSid	Retrieves the attributes associated with a <code>CSid</code> object.

Public Operators

NAME	DESCRIPTION
<code>CTokenGroups::operator const TOKEN_GROUPS *</code>	Casts the <code>CTokenGroups</code> object to a pointer to the <code>TOKEN_GROUPS</code> structure.
<code>CTokenGroups::operator =</code>	Assignment operator.

Remarks

An [access token](#) is an object that describes the security context of a process or thread and is allocated to each user logged onto a Windows system.

The `CTokenGroups` class is a wrapper for the [TOKEN_GROUPS](#) structure, containing information about the group security identifiers (SIDs) in an access token.

For an introduction to the access control model in Windows, see [Access Control](#) in the Windows SDK.

Requirements

Header: `atlsecurity.h`

CTokenGroups::Add

Adds a `CSid` or existing `TOKEN_GROUPS` structure to the `CTokenGroups` object.

```
void Add(const CSid& rSid, DWORD dwAttributes) throw(... );  
void Add(const TOKEN_GROUPS& rTokenGroups) throw(... );
```

Parameters

rSid

A [CSid](#) object.

dwAttributes

The attributes to associate with the `CSid` object.

rTokenGroups

A [TOKEN_GROUPS](#) structure.

Remarks

These methods add one or more `CSid` objects and their associated attributes to the `CTokenGroups` object.

CTokenGroups::CTokenGroups

The constructor.

```
CTokenGroups() throw();  
CTokenGroups(const CTokenGroups& rhs) throw(... );  
CTokenGroups(const TOKEN_GROUPS& rhs) throw(... );
```

Parameters

rhs

The `CTokenGroups` object or [TOKEN_GROUPS](#) structure with which to construct the `CTokenGroups` object.

Remarks

The `CTokenGroups` object can optionally be created using a `TOKEN_GROUPS` structure or a previously defined `CTokenGroups` object.

CTokenGroups::~CTokenGroups

The destructor.

```
virtual ~CTokenGroups() throw();
```

Remarks

The destructor frees all allocated resources.

CTokenGroups::Delete

Deletes a `CSid` and its associated attributes from the `CTokenGroups` object.

```
bool Delete(const CSid& rSid) throw();
```

Parameters

rSid

The `CSid` object for which the security identifier (SID) and attributes should be removed.

Return Value

Returns true if the `CSid` is removed, false otherwise.

CTokenGroups::DeleteAll

Deletes all `CSid` objects and their associated attributes from the `CTokenGroups` object.

```
void DeleteAll() throw();
```

CTokenGroups::GetCount

Returns the number of `CSid` objects contained in `CTokenGroups`.

```
UINT GetCount() const throw();
```

Return Value

Returns the number of `CSid` objects and their associated attributes contained in the `CTokenGroups` object.

CTokenGroups::GetLength

Returns the size of the `CTokenGroup` object.

```
UINT GetLength() const throw();
```

Remarks

Returns the total size of the `CTokenGroup` object, in bytes.

CTokenGroups::GetPTOKEN_GROUPS

Retrieves a pointer to the `TOKEN_GROUPS` structure.

```
const TOKEN_GROUPS* GetPTOKEN_GROUPS() const throw(...);
```

Return Value

Retrieves a pointer to the `TOKEN_GROUPS` structure belonging to the `CTokenGroups` access token object.

CTokenGroups::GetSidsAndAttributes

Retrieves the `CSid` objects and (optionally) the attributes belonging to the `CTokenGroups` object.

```
void GetSidsAndAttributes(  
    CSid::CSidArray* pSids,  
    CATLArray<DWORD>* pAttributes = NULL) const throw(...);
```

Parameters

pSids

Pointer to an array of `CSid` objects.

pAttributes

Pointer to an array of DWORDs. If this parameter is omitted or NULL, the attributes are not retrieved.

Remarks

This method will enumerate all of the `CSid` objects contained in the `CTokenGroups` object and place them and (optionally) the attribute flags into array objects.

CTokenGroups::LookupSid

Retrieves the attributes associated with a `CSid` object.

```
bool LookupSid(  
    const CSid& rSid,  
    DWORD* pdwAttributes = NULL) const throw();
```

Parameters

rSid

The `CSid` object.

pdwAttributes

Pointer to a DWORD which will accept the `CSid` object's attribute. If omitted or NULL, the attribute will not be retrieved.

Return Value

Returns true if the `CSid` is found, false otherwise.

Remarks

Setting *pdwAttributes* to NULL provides a way of confirming the existence of the `CSid` without accessing the attribute. Note that this method should not be used to check access rights. Applications should instead use the [CAccessToken::CheckTokenMembership](#) method.

CTokenGroups::operator =

Assignment operator.

```
CTokenGroups& operator= (const TOKEN_GROUPS& rhs) throw(...);  
CTokenGroups& operator= (const CTokenGroups& rhs) throw(...);
```

Parameters

rhs

The `CTokenGroups` object or `TOKEN_GROUPS` structure to assign to the `CTokenGroups` object.

Return Value

Returns the updated `CTokenGroups` object.

CTokenGroups::operator const TOKEN_GROUPS *

Casts a value to a pointer to the `TOKEN_GROUPS` structure.

```
operator const TOKEN_GROUPS *() const throw(...);
```

Remarks

Casts a value to a pointer to the `TOKEN_GROUPS` structure.

See also

[Security Sample](#)

[CSid Class](#)

[Class Overview](#)

[Security Global Functions](#)

CTokenPrivileges Class

4/1/2019 • 5 minutes to read • [Edit Online](#)

This class is a wrapper for the `TOKEN_PRIVILEGES` structure.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CTokenPrivileges
```

Members

Public Constructors

NAME	DESCRIPTION
CTokenPrivileges::CTokenPrivileges	The constructor.
CTokenPrivileges::~~CTokenPrivileges	The destructor.

Public Methods

NAME	DESCRIPTION
CTokenPrivileges::Add	Adds one or more privileges to the <code>CTokenPrivileges</code> object.
CTokenPrivileges::Delete	Deletes a privilege from the <code>CTokenPrivileges</code> object.
CTokenPrivileges::DeleteAll	Deletes all privileges from the <code>CTokenPrivileges</code> object.
CTokenPrivileges::GetCount	Returns the number of privilege entries in the <code>CTokenPrivileges</code> object.
CTokenPrivileges::GetDisplayNames	Retrieves display names for the privileges contained in the <code>CTokenPrivileges</code> object.
CTokenPrivileges::GetLength	Returns the buffer size in bytes required to hold the <code>TOKEN_PRIVILEGES</code> structure represented by the <code>CTokenPrivileges</code> object.
CTokenPrivileges::GetLuidsAndAttributes	Retrieves the locally unique identifiers (LUIDs) and attribute flags from the <code>CTokenPrivileges</code> object.

NAME	DESCRIPTION
CTokenPrivileges::GetNamesAndAttributes	Retrieves the privilege names and attribute flags from the <code>CTokenPrivileges</code> object.
CTokenPrivileges::GetPTOKEN_PRIVILEGES	Returns a pointer to the <code>TOKEN_PRIVILEGES</code> structure.
CTokenPrivileges::LookupPrivilege	Retrieves the attribute associated with a given privilege name.

Public Operators

NAME	DESCRIPTION
CTokenPrivileges::operator const TOKEN_PRIVILEGES *	Casts a value to a pointer to the <code>TOKEN_PRIVILEGES</code> structure.
CTokenPrivileges::operator =	Assignment operator.

Remarks

An [access token](#) is an object that describes the security context of a process or thread and is allocated to each user logged onto a Windows system.

The access token is used to describe the various security privileges granted to each user. A privilege consists of a 64-bit number called a locally unique identifier ([LUID](#)) and a descriptor string.

The `CTokenPrivileges` class is a wrapper for the [TOKEN_PRIVILEGES](#) structure and contains 0 or more privileges. Privileges can be added, deleted, or queried using the supplied class methods.

For an introduction to the access control model in Windows, see [Access Control](#) in the Windows SDK.

Requirements

Header: atlsecurity.h

CTokenPrivileges::Add

Adds one or more privileges to the `CTokenPrivileges` access token object.

```
bool Add(LPCTSTR pszPrivilege, bool bEnable) throw(...);
void Add(const TOKEN_PRIVILEGES& rPrivileges) throw(...);
```

Parameters

pszPrivilege

Pointer to a null-terminated string that specifies the name of the privilege, as defined in the WINNT.H header file.

bEnable

If true, the privilege is enabled. If false, the privilege is disabled.

rPrivileges

Reference to a [TOKEN_PRIVILEGES](#) structure. The privileges and attributes are copied from this structure and added to the `CTokenPrivileges` object.

Return Value

The first form of this method returns true if the privileges are successfully added, false otherwise.

CTokenPrivileges::CTokenPrivileges

The constructor.

```
CTokenPrivileges() throw();  
CTokenPrivileges(const CTokenPrivileges& rhs) throw(... );  
CTokenPrivileges(const TOKEN_PRIVILEGES& rPrivileges) throw(...);
```

Parameters

rhs

The `CTokenPrivileges` object to assign to the new object.

rPrivileges

The `TOKEN_PRIVILEGES` structure to assign to the new `CTokenPrivileges` object.

Remarks

The `CTokenPrivileges` object can optionally be created using a `TOKEN_PRIVILEGES` structure or a previously defined `CTokenPrivileges` object.

CTokenPrivileges::~~CTokenPrivileges

The destructor.

```
virtual ~CTokenPrivileges() throw();
```

Remarks

The destructor frees all allocated resources.

CTokenPrivileges::Delete

Deletes a privilege from the `CTokenPrivileges` access token object.

```
bool Delete(LPCTSTR pszPrivilege) throw();
```

Parameters

pszPrivilege

Pointer to a null-terminated string that specifies the name of the privilege, as defined in the WINNT.H header file. For example, this parameter could specify the constant `SE_SECURITY_NAME`, or its corresponding string, "SeSecurityPrivilege."

Return Value

Returns true if the privilege was successfully deleted, false otherwise.

Remarks

This method is useful as a tool for creating restricted tokens.

CTokenPrivileges::DeleteAll

Deletes all privileges from the `CTokenPrivileges` access token object.

```
void DeleteAll() throw();
```

Remarks

Deletes all privileges contained in the `CTokenPrivileges` access token object.

CTokenPrivileges::GetDisplayNames

Retrieves display names for the privileges contained in the `CTokenPrivileges` access token object.

```
void GetDisplayNames(CNames* pDisplayNames) const throw(...);
```

Parameters

pDisplayNames

A pointer to an array of `CString` objects. `CNames` is defined as a typedef: `CTokenPrivileges::CATlArray<CString>`.

Remarks

The parameter `pDisplayNames` is a pointer to an array of `CString` objects which will receive the display names corresponding to the privileges contained in the `CTokenPrivileges` object. This method retrieves display names only for the privileges specified in the Defined Privileges section of WINNT.H.

This method retrieves a displayable name: for example, if the attribute name is `SE_REMOTE_SHUTDOWN_NAME`, the displayable name is "Force shutdown from a remote system." To obtain the system name, use [CTokenPrivileges::GetNamesAndAttributes](#).

CTokenPrivileges::GetCount

Returns the number of privilege entries in the `CTokenPrivileges` object.

```
UINT GetCount() const throw();
```

Return Value

Returns the number of privileges contained in the `CTokenPrivileges` object.

CTokenPrivileges::GetLength

Returns the length of the `CTokenPrivileges` object.

```
UINT GetLength() const throw();
```

Return Value

Returns the number of bytes required to hold a `TOKEN_PRIVILEGES` structure represented by the `CTokenPrivileges` object, including all of the privilege entries it contains.

CTokenPrivileges::GetLuidsAndAttributes

Retrieves the locally unique identifiers (LUIDs) and attribute flags from the `CTokenPrivileges` object.

```
void GetLuidsAndAttributes(
    CLUIDArray* pPrivileges,
    CAttributes* pAttributes = NULL) const throw(...);
```

Parameters

pPrivileges

Pointer to an array of [LUID](#) objects. `CLUIDArray` is a typedef defined as `CAtlArray<LUID> CLUIDArray`.

pAttributes

Pointer to an array of `DWORD` objects. If this parameter is omitted or `NULL`, the attributes are not retrieved.

`CAttributes` is a typedef defined as `CAtlArray <DWORD> CAttributes`.

Remarks

This method will enumerate all of the privileges contained in the `CTokenPrivileges` access token object and place the individual LUIDs and (optionally) the attribute flags into array objects.

CTokenPrivileges::GetNamesAndAttributes

Retrieves the name and attribute flags from the `CTokenPrivileges` object.

```
void GetNamesAndAttributes(
    CNames* pNames,
    CAttributes* pAttributes = NULL) const throw(...);
```

Parameters

pNames

Pointer to an array of `CString` objects. `CNames` is a typedef defined as `CAtlArray <CString> CNames`.

pAttributes

Pointer to an array of `DWORD` objects. If this parameter is omitted or `NULL`, the attributes are not retrieved.

`CAttributes` is a typedef defined as `CAtlArray <DWORD> CAttributes`.

Remarks

This method will enumerate all of the privileges contained in the `CTokenPrivileges` object, placing the name and (optionally) the attribute flags into array objects.

This method retrieves the attribute name, rather than the displayable name: for example, if the attribute name is `SE_REMOTE_SHUTDOWN_NAME`, the system name is "SeRemoteShutdownPrivilege." To obtain the displayable name, use the method [CTokenPrivileges::GetDisplayNames](#).

CTokenPrivileges::GetPTOKEN_PRIVILEGES

Returns a pointer to the `TOKEN_PRIVILEGES` structure.

```
const TOKEN_PRIVILEGES* GetPTOKEN_PRIVILEGES() const throw(...);
```

Return Value

Returns a pointer to the [TOKEN_PRIVILEGES](#) structure.

CTokenPrivileges::LookupPrivilege

Retrieves the attribute associated with a given privilege name.

```
bool LookupPrivilege(  
    LPCTSTR pszPrivilege,  
    DWORD* pdwAttributes = NULL) const throw(...);
```

Parameters

pszPrivilege

Pointer to a null-terminated string that specifies the name of the privilege, as defined in the WINNT.H header file. For example, this parameter could specify the constant SE_SECURITY_NAME, or its corresponding string, "SeSecurityPrivilege."

pdwAttributes

Pointer to a variable that receives the attributes.

Return Value

Returns true if the attribute is successfully retrieved, false otherwise.

CTokenPrivileges::operator =

Assignment operator.

```
CTokenPrivileges& operator= (const TOKEN_PRIVILEGES& rPrivileges) throw(...);  
CTokenPrivileges& operator= (const CTokenPrivileges& rhs) throw(...);
```

Parameters

rPrivileges

The [TOKEN_PRIVILEGES](#) structure to assign to the `CTokenPrivileges` object.

rhs

The `CTokenPrivileges` object to assign to the object.

Return Value

Returns the updated `CTokenPrivileges` object.

CTokenPrivileges::operator const TOKEN_PRIVILEGES *

Casts a value to a pointer to the `TOKEN_PRIVILEGES` structure.

```
operator const TOKEN_PRIVILEGES *() const throw(...);
```

Remarks

Casts a value to a pointer to the [TOKEN_PRIVILEGES](#) structure.

See also

[Security Sample](#)

[TOKEN_PRIVILEGES](#)

[LUID](#)

[LUID_AND_ATTRIBUTES](#)

[Class Overview](#)

[Security Global Functions](#)

CUrl Class

5/7/2019 • 8 minutes to read • [Edit Online](#)

This class represents a URL. It allows you to manipulate each element of the URL independently of the others whether parsing an existing URL string or building a string from scratch.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CUrl
```

Members

Public Constructors

NAME	DESCRIPTION
CUrl::CUrl	The constructor.
CUrl::~~CUrl	The destructor.

Public Methods

NAME	DESCRIPTION
CUrl::Canonicalize	Call this method to convert the URL string to canonical form.
CUrl::Clear	Call this method to clear all of the URL fields.
CUrl::CrackUrl	Call this method to decode and parse the URL.
CUrl::createUrl	Call this method to create the URL.
CUrl::GetExtrInfo	Call this method to get extra information (such as <i>text</i> or # <i>text</i>) from the URL.
CUrl::GetExtrInfoLength	Call this method to get the length of the extra information (such as <i>text</i> or # <i>text</i>) to retrieve from the URL.
CUrl::GetHostName	Call this method to get the host name from the URL.
CUrl::GetHostNameLength	Call this method to get the length of the host name.
CUrl::GetPassword	Call this method to get the password from the URL.

NAME	DESCRIPTION
CUrl::GetPasswordLength	Call this method to get the length of the password.
CUrl::GetPortNumber	Call this method to get the port number in terms of ATL_URL_PORT.
CUrl::GetScheme	Call this method to get the URL scheme.
CUrl::GetSchemeName	Call this method to get the URL scheme name.
CUrl::GetSchemeNameLength	Call this method to get the length of the URL scheme name.
CUrl::GetUrlLength	Call this method to get the URL length.
CUrl::GetUriPath	Call this method to get the URL path.
CUrl::GetUriPathLength	Call this method to get the URL path length.
CUrl::GetUserName	Call this method to get the user name from the URL.
CUrl::GetUserNameLength	Call this method to get the length of the user name.
CUrl::SetExtraInfo	Call this method to set the extra information (such as <i>text</i> or <i># text</i>) of the URL.
CUrl::SetHostName	Call this method to set the host name.
CUrl::SetPassword	Call this method to set the password.
CUrl::SetPortNumber	Call this method to set the port number in terms of ATL_URL_PORT.
CUrl::SetScheme	Call this method to set the URL scheme.
CUrl::SetSchemeName	Call this method to set the URL scheme name.
CUrl::SetUriPath	Call this method to set the URL path.
CUrl::SetUserName	Call this method to set the user name.

Public Operators

NAME	DESCRIPTION
CUrl::operator =	Assigns the specified <code>cur1</code> object to the current <code>cur1</code> object.

Remarks

`cur1` allows you to manipulate the fields of a URL, such as the path or port number. `cur1` understands URLs of the following form:

```
<Scheme>://<UserName>:<Password>@<HostName>:<PortNumber>/<UriPath> <ExtraInfo>
```

(Some fields are optional.) For example, consider this URL:

```
http://someone:secret@www.microsoft.com:80/visualc/stuff.htm#contents
```

`CUrl::CrackUrl` parses it as follows:

- Scheme: "http" or `ATL_URL_SCHEME_HTTP`
- UserName: "someone"
- Password: "secret"
- HostName: "www.microsoft.com"
- PortNumber: 80
- UriPath: "visualc/stuff.htm"
- ExtraInfo: "#contents"

To manipulate the `UriPath` field (for instance), you would use `GetUriPath`, `GetUriPathLength`, and `SetUriPath`. You would use `CreateUrl` to create the complete URL string.

Requirements

Header: `atlutil.h`

CUrl::Canonicalize

Call this method to convert the URL string to canonical form.

```
inline BOOL Canonicalize(DWORD dwFlags = 0) throw();
```

Parameters

dwFlags

The flags that control canonicalization. If no flags are specified (*dwFlags* = 0), the method converts all unsafe characters and meta sequences (such as `\.`, `\.`, and `\.`) to escape sequences. *dwFlags* can be one of the following values:

- `ATL_URL_BROWSER_MODE`: Does not encode or decode characters after "#" or "" and does not remove trailing white space after "". If this value is not specified, the entire URL is encoded and trailing white space is removed.
- `ATL_URL_DECODE`: Converts all %XX sequences to characters, including escape sequences, before the URL is parsed.
- `ATL_URL_ENCODE_PERCENT`: Encodes any percent signs encountered. By default, percent signs are not encoded.
- `ATL_URL_ENCODE_SPACES_ONLY`: Encodes spaces only.
- `ATL_URL_NO_ENCODE`: Does not convert unsafe characters to escape sequences.
- `ATL_URL_NO_META`: Does not remove meta sequences (such as "." and "..") from the URL.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Converting to canonical form involves converting unsafe characters and spaces to escape sequences.

CUrl::Clear

Call this method to clear all of the URL fields.

```
inline void Clear() throw();
```

CUrl::CrackUrl

Call this method to decode and parse the URL.

```
BOOL CrackUrl(LPCTSTR lpszUrl, DWORD dwFlags = 0) throw();
```

Parameters

lpszUrl

The URL.

dwFlags

Specify ATL_URL_DECODE or ATL_URL_ESCAPE to convert all escape characters in *lpszUrl* to their real values after parsing. (Before Visual C++ 2005, ATL_URL_DECODE converted all escape characters before parsing.)

Return Value

Returns TRUE on success, FALSE on failure.

CUrl::createUrl

This method constructs a URL string from a CUrl object's component fields.

```
inline BOOL CreateUrl(  
    LPTSTR lpszUrl,  
    DWORD* pdwMaxLength,  
    DWORD dwFlags = 0) const throw();
```

Parameters

lpszUrl

A string buffer to hold the complete URL string.

pdwMaxLength

The maximum length of the *lpszUrl* string buffer.

dwFlags

Specify ATL_URL_ESCAPE to convert all escape characters in *lpszUrl* to their real values.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This method appends its individual fields in order to construct the complete URL string using the following format:

<scheme>://<user>:<pass>@<domain>:<port><path><extra>

When calling this method, the *pdwMaxLength* parameter should initially contain the maximum length of the

string buffer referenced by the *lpzUrl* parameter. The value of the *pdwMaxLength* parameter will be updated with the actual length of the URL string.

Example

This sample demonstrates creation of a CUrl object and retrieving its URL string

```
CUrl url;

// Set the CUrl contents
url.CrackUrl(_T("http://someone:secret@www.microsoft.com:8080/visualc/stuff.htm#contents"));

// Obtain the length of the URL string and allocate a buffer to
// hold its contents
DWORD dwUrlLen = url.GetUrlLength() + 1;
TCHAR* szUrl = new TCHAR[dwUrlLen];

// Retrieve the contents of the CUrl object
url.CreateUrl(szUrl, &dwUrlLen, 0L);

// Cleanup
delete[] szUrl;
```

CUrl::CUrl

The constructor.

```
CUrl() throw();
CUrl(const CUrl& urlThat) throw();
```

Parameters

urlThat

The `cur1` object to copy to create the URL.

CUrl::~~CUrl

The destructor.

```
~CUrl() throw();
```

CUrl::GetExtraInfo

Call this method to get extra information (such as *text* or *# text*) from the URL.

```
inline LPCTSTR GetExtraInfo() const throw();
```

Return Value

Returns a string containing the extra information.

CUrl::GetExtraInfoLength

Call this method to get the length of the extra information (such as *text* or *# text*) to retrieve from the URL.

```
inline DWORD GetExtraInfoLength() const throw();
```

Return Value

Returns the length of the string containing the extra information.

CUrl::GetHostName

Call this method to get the host name from the URL.

```
inline LPCTSTR GetHostName() const throw();
```

Return Value

Returns the host name.

CUrl::GetHostNameLength

Call this method to get the length of the host name.

```
inline DWORD GetHostNameLength() const throw();
```

Return Value

Returns the host name length.

CUrl::GetPassword

Call this method to get the password from the URL.

```
inline LPCTSTR GetPassword() const throw();
```

Return Value

Returns the password.

CUrl::GetPasswordLength

Call this method to get the length of the password.

```
inline DWORD GetPasswordLength() const throw();
```

Return Value

Returns the password length.

CUrl::GetPortNumber

Call this method to get the port number.

```
inline ATL_URL_PORT GetPortNumber() const throw();
```

Return Value

Returns the port number.

CUrl::GetScheme

Call this method to get the URL scheme.

```
inline ATL_URL_SCHEME GetScheme() const throw();
```

Return Value

Returns the [ATL_URL_SCHEME](#) value describing the scheme of the URL.

CUrl::GetSchemeName

Call this method to get the URL scheme name.

```
inline LPCTSTR GetSchemeName() const throw();
```

Return Value

Returns the URL scheme name (such as "http" or "ftp").

CUrl::GetSchemeNameLength

Call this method to get the length of the URL scheme name.

```
inline DWORD GetSchemeNameLength() const throw();
```

Return Value

Returns the URL scheme name length.

CUrl::GetUrlLength

Call this method to get the URL length.

```
inline DWORD GetUrlLength() const throw();
```

Return Value

Returns the URL length.

CUrl::GetUrlPath

Call this method to get the URL path.

```
inline LPCTSTR GetUrlPath() const throw();
```

Return Value

Returns the URL path.

CUrl::GetUrlPathLength

Call this method to get the URL path length.

```
inline DWORD GetUrlPathLength() const throw();
```

Return Value

Returns the URL path length.

CUrl::GetUserName

Call this method to get the user name from the URL.

```
inline LPCTSTR GetUserName() const throw();
```

Return Value

Returns the user name.

CUrl::GetUserNameLength

Call this method to get the length of the user name.

```
inline DWORD GetUserNameLength() const throw();
```

Return Value

Returns the user name length.

CUrl::operator =

Assigns the specified `CUrl` object to the current `CUrl` object.

```
CUrl& operator= (const CUrl& urlThat) throw();
```

Parameters

urlThat

The `CUrl` object to copy into the current object.

Return Value

Returns a reference to the current object.

CUrl::SetExtraInfo

Call this method to set the extra information (such as *text* or *# text*) of the URL.

```
inline BOOL SetExtraInfo(LPCTSTR lpszInfo) throw();
```

Parameters

lpszInfo

The string containing the extra information to include in the URL.

Return Value

Returns TRUE on success, FALSE on failure.

CUrl::SetHostName

Call this method to set the host name.

```
inline BOOL SetHostName(LPCTSTR lpszHost) throw();
```

Parameters

lpszHost

The host name.

Return Value

Returns TRUE on success, FALSE on failure.

CUrl::SetPassword

Call this method to set the password.

```
inline BOOL SetPassword(LPCTSTR lpszPass) throw();
```

Parameters

lpszPass

The password.

Return Value

Returns TRUE on success, FALSE on failure.

CUrl::SetPortNumber

Call this method to set the port number.

```
inline BOOL SetPortNumber(ATL_URL_PORT nPrt) throw();
```

Parameters

nPrt

The port number.

Return Value

Returns TRUE on success, FALSE on failure.

CUrl::SetScheme

Call this method to set the URL scheme.

```
inline BOOL SetScheme(ATL_URL_SCHEME nScheme) throw();
```

Parameters

nScheme

One of the [ATL_URL_SCHEME](#) values for the scheme.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

You can also set the scheme by name (see [CUrl::SetSchemeName](#)).

CUrl::SetSchemeName

Call this method to set the URL scheme name.

```
inline BOOL SetSchemeName(LPCTSTR lpszSchm) throw();
```

Parameters

lpszSchm

The URL scheme name.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

You can also set the scheme by using an [ATL_URL_SCHEME](#) constant (see [CUrl::SetScheme](#)).

CUrl::SetUrlPath

Call this method to set the URL path.

```
inline BOOL SetUrlPath(LPCTSTR lpszPath) throw();
```

Parameters

lpszPath

The URL path.

Return Value

Returns TRUE on success, FALSE on failure.

CUrl::SetUserName

Call this method to set the user name.

```
inline BOOL SetUserName(LPCTSTR lpszUser) throw();
```

Parameters

lpszUser

The user name.

Return Value

Returns TRUE on success, FALSE on failure.

See also

[Classes](#)

CW2AEX Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class is used by the string conversion macros CT2AEX, CW2TEX, CW2CTEX, and CT2CAEX, and the typedef CW2A.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<int t_nBufferLength = 128>
class CW2AEX
```

Parameters

t_nBufferLength

The size of the buffer used in the translation process. The default length is 128 bytes.

Members

Public Constructors

NAME	DESCRIPTION
CW2AEX::CW2AEX	The constructor.
CW2AEX::~~CW2AEX	The destructor.

Public Operators

NAME	DESCRIPTION
CW2AEX::operator LPSTR	Conversion operator.

Public Data Members

NAME	DESCRIPTION
CW2AEX::m_psz	The data member that stores the source string.
CW2AEX::m_szBuffer	The static buffer, used to store the converted string.

Remarks

Unless extra functionality is required, use CT2AEX, CW2TEX, CW2CTEX, CT2CAEX, or CW2A in your code.

This class contains a fixed-size static buffer which is used to store the result of the conversion. If the result is too large to fit into the static buffer, the class allocates memory using **malloc**, freeing the memory when the object

goes out of scope. This ensures that, unlike text conversion macros available in previous versions of ATL, this class is safe to use in loops and that it won't overflow the stack.

If the class tries to allocate memory on the heap and fails, it will call `AtlThrow` with an argument of `E_OUTOFMEMORY`.

By default, the ATL conversion classes and macros use the current thread's ANSI code page for the conversion. If you want to override that behavior for a specific conversion, specify the code page as the second parameter to the constructor for the class.

The following macros are based on this class:

- `CT2AEX`
- `CW2TEX`
- `CW2CTEX`
- `CT2CAEX`

The following typedef is based on this class:

- `CW2A`

For a discussion of these text conversion macros, see [ATL and MFC String Conversion Macros](#).

Example

See [ATL and MFC String Conversion Macros](#) for an example of using these string conversion macros.

Requirements

Header: `atlconv.h`

CW2AEX::CW2AEX

The constructor.

```
CW2AEX(LPCWSTR psz, UINT nCodePage) throw(...);  
CW2AEX(LPCWSTR psz) throw(...);
```

Parameters

psz

The text string to be converted.

nCodePage

The code page used to perform the conversion. See the code page parameter discussion for the Windows SDK function [MultiByteToWideChar](#) for more details.

Remarks

Allocates the buffer used in the translation process.

CW2AEX::~CW2AEX

The destructor.

```
~CW2AEX() throw();
```

Remarks

Frees the allocated buffer.

CW2AEX::m_psz

The data member that stores the source string.

```
LPSTR m_psz;
```

CW2AEX::m_szBuffer

The static buffer, used to store the converted string.

```
char m_szBuffer[t_nBufferLength];
```

CW2AEX::operator LPSTR

Conversion operator.

```
operator LPSTR() const throw();
```

Return Value

Returns the text string as type LPSTR.

See also

[CA2AEX Class](#)

[CA2CAEX Class](#)

[CA2WEX Class](#)

[CW2CWEX Class](#)

[CW2WEX Class](#)

[Class Overview](#)

CW2CWEX Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class is used by the string conversion macros CW2CTEX and CT2CWEX, and the typedef CW2W.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<int t_nBufferLength = 128>
class CW2CWEX
```

Parameters

t_nBufferLength

The size of the buffer used in the translation process. The default length is 128 bytes.

Members

Public Constructors

NAME	DESCRIPTION
CW2CWEX::CW2CWEX	The constructor.
CW2CWEX::~~CW2CWEX	The destructor.

Public Operators

NAME	DESCRIPTION
CW2CWEX::operator LPCWSTR	Conversion operator.

Public Data Members

NAME	DESCRIPTION
CW2CWEX::m_psz	The data member that stores the source string.

Remarks

Unless extra functionality is required, use CW2CTEX, CT2CWEX, or CW2W in your code.

This class is safe to use in loops and won't overflow the stack. By default, the ATL conversion classes and macros use the current thread's ANSI code page for the conversion.

The following macros are based on this class:

- CW2CTEX

- CT2CWEX

The following typedef is based on this class:

- CW2W

For a discussion of these text conversion macros, see [ATL and MFC String Conversion Macros](#).

Example

See [ATL and MFC String Conversion Macros](#) for an example of using these string conversion macros.

Requirements

Header: atlconv.h

CW2CWEX::CW2CWEX

The constructor.

```
CW2CWEX(LPCWSTR psz, UINT nCodePage) throw(...);  
CW2CWEX(LPCWSTR psz) throw(...);
```

Parameters

psz

The text string to be converted.

nCodePage

The code page. Not used in this class.

Remarks

Allocates the buffer used in the translation process.

CW2CWEX::~~CW2CWEX

The destructor.

```
~CW2CWEX() throw();
```

Remarks

Frees the allocated buffer.

CW2CWEX::m_psz

The data member that stores the source string.

```
LPCWSTR m_psz;
```

CW2CWEX::operator LPCWSTR

Conversion operator.

```
operator LPCWSTR() const throw();
```

Return Value

Returns the text string as type LPCWSTR.

See also

[CA2AEX Class](#)

[CA2CAEX Class](#)

[CA2WEX Class](#)

[CW2AEX Class](#)

[CW2WEX Class](#)

[Class Overview](#)

CW2WEX Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class is used by the string conversion macros CW2TEX and CT2WEX, and the typedef CW2W.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <int t_nBufferLength = 128>
class CW2WEX
```

Parameters

t_nBufferLength

The size of the buffer used in the translation process. The default length is 128 bytes.

Members

Public Constructors

NAME	DESCRIPTION
CW2WEX::CW2WEX	The constructor.
CW2WEX::~~CW2WEX	The destructor.

Public Operators

NAME	DESCRIPTION
CW2WEX::operator LPWSTR	Conversion operator.

Public Data Members

NAME	DESCRIPTION
CW2WEX::m_psz	The data member that stores the source string.
CW2WEX::m_szBuffer	The static buffer, used to store the converted string.

Remarks

Unless extra functionality is required, use CW2TEX, CT2WEX, or CW2W in your code.

This class contains a fixed-size static buffer which is used to store the result of the conversion. If the result is too large to fit into the static buffer, the class allocates memory using **malloc**, freeing the memory when the object goes out of scope. This ensures that, unlike text conversion macros available in previous versions of ATL, this

class is safe to use in loops and that it won't overflow the stack.

If the class tries to allocate memory on the heap and fails, it will call `AtlThrow` with an argument of `E_OUTOFMEMORY`.

By default, the ATL conversion classes and macros use the current thread's ANSI code page for the conversion.

The following macros are based on this class:

- `CW2TEX`
- `CT2WEX`

The following typedef is based on this class:

- `CW2W`

For a discussion of these text conversion macros, see [ATL and MFC String Conversion Macros](#).

Example

See [ATL and MFC String Conversion Macros](#) for an example of using these string conversion macros.

Requirements

Header: `atlconv.h`

CW2WEX::CW2WEX

The constructor.

```
CW2WEX(LPCWSTR psz, UINT nCodePage) throw(...);  
CW2WEX( LPCWSTR psz) throw(...);
```

Parameters

psz

The text string to be converted.

nCodePage

The code page. Not used in this class.

Remarks

Creates the buffer required for the translation.

CW2WEX::~~CW2WEX

The destructor..

```
~CW2WEX() throw();
```

Remarks

Frees the allocated buffer.

CW2WEX::m_psz

The data member that stores the source string.

```
LPWSTR m_psz;
```

CW2WEX::m_szBuffer

The static buffer, used to store the converted string.

```
wchar_t m_szBuffer[t_nBufferLength];
```

CW2WEX::operator LPWSTR

Cast operator.

```
operator LPWSTR() const throw();
```

Return Value

Returns the text string as type LPWSTR.

See also

[CA2AEX Class](#)

[CA2CAEX Class](#)

[CA2WEX Class](#)

[CW2AEX Class](#)

[CW2CWEX Class](#)

[Class Overview](#)

CWin32Heap Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

This class implements [IAtlMemMgr](#) using the Win32 heap allocation functions.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CWin32Heap : public IAtlMemMgr
```

Members

Public Constructors

NAME	DESCRIPTION
CWin32Heap::CWin32Heap	The constructor.
CWin32Heap::~~CWin32Heap	The destructor.

Public Methods

NAME	DESCRIPTION
CWin32Heap::Allocate	Allocates a block of memory from the heap object.
CWin32Heap::Attach	Attaches the heap object to an existing heap.
CWin32Heap::Detach	Detaches the heap object from an existing heap.
CWin32Heap::Free	Frees memory previously allocated from the heap.
CWin32Heap::GetSize	Returns the size of a memory block allocated from the heap object.
CWin32Heap::Reallocate	Reallocates a block of memory from the heap object.

Public Data Members

NAME	DESCRIPTION
CWin32Heap::m_bOwnHeap	A flag used to determine current ownership of the heap handle.
CWin32Heap::m_hHeap	Handle to the heap object.

Remarks

`CWin32Heap` implements memory allocation methods using the Win32 heap allocation functions, including [HeapAlloc](#) and [HeapFree](#). Unlike other Heap classes, `CWin32Heap` requires a valid heap handle to be provided before memory is allocated: the other classes default to using the process heap. The handle can be supplied to the constructor or to the [CWin32Heap::Attach](#) method. See the [CWin32Heap::CWin32Heap](#) method for more details.

Example

See the example for [IAtlMemMgr](#).

Inheritance Hierarchy

`IAtlMemMgr`

`CWin32Heap`

Requirements

Header: atlmem.h

CWin32Heap::Allocate

Allocates a block of memory from the heap object.

```
virtual __declspec(allocator) void* Allocate(size_t nBytes) throw();
```

Parameters

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the newly allocated memory block.

Remarks

Call [CWin32Heap::Free](#) or [CWin32Heap::Reallocate](#) to free the memory allocated by this method.

Implemented using [HeapAlloc](#).

CWin32Heap::Attach

Attaches the heap object to an existing heap.

```
void Attach(HANDLE hHeap, bool bTakeOwnership) throw();
```

Parameters

hHeap

An existing heap handle.

bTakeOwnership

A flag indicating if the `CWin32Heap` object is to take ownership over the resources of the heap.

Remarks

If *bTakeOwnership* is TRUE, the `CWin32Heap` object is responsible for deleting the heap handle.

CWin32Heap::CWin32Heap

The constructor.

```
CWin32Heap() throw();
CWin32Heap( HANDLE  hHeap) throw();
CWin32Heap(
    DWORD  dwFlags,
    size_t nInitialSize,
    size_t nMaxSize = 0);
```

Parameters

hHeap

An existing heap object.

dwFlags

Flags used in creating the heap.

nInitialSize

The initial size of the heap.

nMaxSize

The maximum size of the heap.

Remarks

Before allocating memory, it is necessary to provide the `CWin32Heap` object with a valid heap handle. The simplest way to achieve this is to use the process heap:

```
CWin32Heap MyHeap(GetProcessHeap());
```

It is also possible to supply an existing heap handle to the constructor, in which case the new object does not take over ownership of the heap. The original heap handle will still be valid when the `CWin32Heap` object is deleted.

An existing heap can also be attached to the new object, using [CWin32Heap::Attach](#).

If a heap is required where operations are all performed from a single thread, the best way is to create the object as follows:

```
CWin32Heap MyHeap(HEAP_NO_SERIALIZE, SomeInitialSize);
```

The parameter `HEAP_NO_SERIALIZE` specifies that mutual exclusion will not be used when the heap functions allocate and free memory, with an according increase in performance.

The third parameter defaults to 0, which allows the heap to grow as required. See [HeapCreate](#) for an explanation of the memory sizes and flags.

CWin32Heap::~CWin32Heap

The destructor.

```
~CWin32Heap() throw();
```

Remarks

Destroys the heap handle if the `CWin32Heap` object has ownership of the heap.

CWin32Heap::Detach

Detaches the heap object from an existing heap.

```
HANDLE Detach() throw();
```

Return Value

Returns the handle to the heap to which the object was previously attached.

CWin32Heap::Free

Frees memory previously allocated from the heap by [CWin32Heap::Allocate](#) or [CWin32Heap::Reallocate](#).

```
virtual void Free(void* p) throw();
```

Parameters

p

Pointer to the block of memory to free. NULL is a valid value and does nothing.

CWin32Heap::GetSize

Returns the size of a memory block allocated from the heap object.

```
virtual size_t GetSize(void* p) throw();
```

Parameters

p

Pointer to the memory block whose size the method will obtain. This is a pointer returned by [CWin32Heap::Allocate](#) or [CWin32Heap::Reallocate](#).

Return Value

Returns the size, in bytes, of the allocated memory block.

CWin32Heap::m_bOwnHeap

A flag used to determine current ownership of the heap handle stored in [m_hHeap](#).

```
bool m_bOwnHeap;
```

CWin32Heap::m_hHeap

Handle to the heap object.

```
HANDLE m_hHeap;
```

Remarks

A variable used to store a handle to the heap object.

CWin32Heap::Reallocate

Reallocates a block of memory from the heap object.

```
virtual __declspec(allocator) void* Reallocate(void* p, size_t nBytes) throw();
```

Parameters

p

Pointer to the block of memory to reallocate.

nBytes

The new size in bytes of the allocated block. The block can be made larger or smaller.

Return Value

Returns a pointer to the newly allocated memory block.

Remarks

If *p* is NULL, it's assumed that the memory block has not yet been allocated and [CWin32Heap::Allocate](#) is called, with an argument of *nBytes*.

See also

[Class Overview](#)

[IAtlMemMgr Class](#)

[CLocalHeap Class](#)

[CGlobalHeap Class](#)

[CCRTHeap Class](#)

[CComHeap Class](#)

CWindow Class

3/5/2019 • 40 minutes to read • [Edit Online](#)

This class provides methods for manipulating a window.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CWindow
```

Members

Public Constructors

NAME	DESCRIPTION
CWindow::CWindow	Constructor.

Public Methods

NAME	DESCRIPTION
CWindow::ArrangeIconicWindows	Arranges all minimized child windows.
CWindow::Attach	Attaches a window to the <code>CWindow</code> object.
CWindow::BeginPaint	Prepares the window for painting.
CWindow::BringWindowToTop	Brings the window to the top of the Z order.
CWindow::CenterWindow	Centers the window against a given window.
CWindow::ChangeClipboardChain	Removes the window from the chain of Clipboard viewers.
CWindow::CheckDlgButton	Changes the check state of the specified button.
CWindow::CheckRadioButton	Checks the specified radio button.
CWindow::ChildWindowFromPoint	Retrieves the child window containing the specified point.
CWindow::ChildWindowFromPointEx	Retrieves a particular type of child window containing the specified point.
CWindow::ClientToScreen	Converts client coordinates to screen coordinates.

NAME	DESCRIPTION
CWindow::Create	Creates a window.
CWindow::CreateCaret	Creates a new shape for the system caret.
CWindow::CreateGrayCaret	Creates a gray rectangle for the system caret.
CWindow::CreateSolidCaret	Creates a solid rectangle for the system caret.
CWindow::DeferWindowPos	Updates the specified multiple-window-position structure for the specified window.
CWindow::DestroyWindow	Destroys the window associated with the <code>CWindow</code> object.
CWindow::Detach	Detaches the window from the <code>CWindow</code> object.
CWindow::DlgDirList	Fills a list box with the names of all files matching a specified path or file name.
CWindow::DlgDirListComboBox	Fills a combo box with the names of all files matching a specified path or file name.
CWindow::DlgDirSelect	Retrieves the current selection from a list box.
CWindow::DlgDirSelectComboBox	Retrieves the current selection from a combo box.
CWindow::DragAcceptFiles	Registers whether the window accepts dragged files.
CWindow::DrawMenuBar	Redraws the window's menu bar.
CWindow::EnableScrollBar	Enables or disables the scroll bar arrows.
CWindow::EnableWindow	Enables or disables input.
CWindow::EndPoint	Marks the end of painting.
CWindow::FlashWindow	Flashes the window once.
CWindow::GetClientRect	Retrieves the coordinates of the client area.
CWindow::GetDC	Retrieves a device context for the client area.
CWindow::GetDCEX	Retrieves a device context for the client area and allows clipping options.
CWindow::GetDescendantWindow	Retrieves the specified descendant window.
CWindow::GetDlgControl	Retrieves an interface on the specified control.
CWindow::GetDlgCtrlID	Retrieves the window's identifier (for child windows only).

NAME	DESCRIPTION
CWindow::GetDlgHost	Retrieves a pointer to an interface to the ATL Control hosting container.
CWindow::GetDlgItem	Retrieves the specified child window.
CWindow::GetDlgItemInt	Translates a control's text to an integer.
CWindow::GetDlgItemText	Retrieves a control's text.
CWindow::GetExStyle	Retrieves the extended window styles.
CWindow::GetFont	Retrieves the window's current font.
CWindow::GetHotKey	Determines the hot key associated with the window.
CWindow::GetIcon	Retrieves the window's large or small icon.
CWindow::GetLastActivePopup	Retrieves the most recently active pop-up window.
CWindow::GetMenu	Retrieves the window's menu.
CWindow::GetNextDlgGroupItem	Retrieves the previous or next control within a group of controls.
CWindow::GetNextDlgTabItem	Retrieves the previous or next control having the WS_TABSTOP style.
CWindow::GetParent	Retrieves the immediate parent window.
CWindow::GetScrollInfo	Retrieves the parameters of a scroll bar.
CWindow::GetScrollPos	Retrieves the position of the scroll box.
CWindow::GetScrollRange	Retrieves the scroll bar range.
CWindow::GetStyle	Retrieves the window styles.
CWindow::GetSystemMenu	Creates a copy of the system menu for modification.
CWindow::GetTopLevelParent	Retrieves the top-level parent or owner window.
CWindow::GetTopLevelWindow	Retrieves the top-level owner window.
CWindow::GetTopWindow	Retrieves the top-level child window.
CWindow::GetUpdateRect	Retrieves the coordinates of the smallest rectangle that completely encloses the update region.
CWindow::GetUpdateRgn	Retrieves the update region and copies it into a specified region.

NAME	DESCRIPTION
CWindow::GetWindow	Retrieves the specified window.
CWindow::GetWindowContextHelpId	Retrieves the window's help context identifier.
CWindow::GetWindowDC	Retrieves a device context for the entire window.
CWindow::GetWindowLong	Retrieves a 32-bit value at a specified offset into the extra window memory.
CWindow::GetWindowLongPtr	Retrieves information about the specified window, including a value at a specified offset into the extra window memory.
CWindow::GetWindowPlacement	Retrieves the show state and positions.
CWindow::GetWindowProcessID	Retrieves the identifier of the process that created the window.
CWindow::GetWindowRect	Retrieves the window's bounding dimensions.
CWindow::GetWindowRgn	Obtains a copy of the window region of a window.
CWindow::GetWindowText	Retrieves the window's text.
CWindow::GetWindowTextLength	Retrieves the length of the window's text.
CWindow::GetWindowThreadID	Retrieves the identifier of the thread that created the specified window.
CWindow::GetWindowWord	Retrieves a 16-bit value at a specified offset into the extra window memory.
CWindow::GotoDlgCtrl	Sets the keyboard focus to a control in the dialog box.
CWindow::HideCaret	Hides the system caret.
CWindow::HiliteMenuItem	Highlights or removes the highlight from a top-level menu item.
CWindow::Invalidate	Invalidates the entire client area.
CWindow::InvalidateRect	Invalidates the client area within the specified rectangle.
CWindow::InvalidateRgn	Invalidates the client area within the specified region.
CWindow::IsChild	Determines whether the specified window is a child window.
CWindow::IsDialogMessage	Determines whether a message is intended for the specified dialog box.
CWindow::IsDlgButtonChecked	Determines the check state of the button.

NAME	DESCRIPTION
CWindow::IsIconic	Determines whether the window is minimized.
CWindow::IsParentDialog	Determines if the parent window of a control is a dialog window.
CWindow::IsWindow	Determines whether the specified window handle identifies an existing window.
CWindow::IsWindowEnabled	Determines whether the window is enabled for input.
CWindow::IsWindowUnicode	Determines whether the specified window is a native Unicode window.
CWindow::IsWindowVisible	Determines the window's visibility state.
CWindow::IsZoomed	Determines whether the window is maximized.
CWindow::KillTimer	Destroys a timer event.
CWindow::LockWindowUpdate	Disables or enables drawing in the window.
CWindow::MapWindowPoints	Converts a set of points from the window's coordinate space to the coordinate space of another window.
CWindow::MessageBox	Displays a message box.
CWindow::ModifyStyle	Modifies the window styles.
CWindow::ModifyStyleEx	Modifies the extended window styles.
CWindow::MoveWindow	Changes the window's size and position.
CWindow::NextDlgCtrl	Sets the keyboard focus to the next control in the dialog box.
CWindow::OpenClipboard	Opens the Clipboard.
CWindow::PostMessage	Places a message in the message queue associated with the thread that created the window. Returns without waiting for the thread to process the message.
CWindow::PrevDlgCtrl	Sets the keyboard focus to the previous control in the dialog box.
CWindow::Print	Requests that the window be drawn in a specified device context.
CWindow::PrintClient	Requests that the window's client area be drawn in a specified device context.
CWindow::RedrawWindow	Updates a specified rectangle or region in the client area.
CWindow::ReleaseDC	Releases a device context.

NAME	DESCRIPTION
<code>CWindow::ResizeClient</code>	Resizes the window.
<code>CWindow::ScreenToClient</code>	Converts screen coordinates to client coordinates.
<code>CWindow::ScrollWindow</code>	Scrolls the specified client area.
<code>CWindow::ScrollWindowEx</code>	Scrolls the specified client area with additional features.
<code>CWindow::SendDlgItemMessage</code>	Sends a message to a control.
<code>CWindow::SendMessage</code>	Sends a message to the window and does not return until the window procedure has processed the message.
<code>CWindow::SendMessageToDescendants</code>	Sends a message to the specified descendant windows.
<code>CWindow::SendNotifyMessage</code>	Sends a message to the window. If the window was created by the calling thread, <code>SendNotifyMessage</code> does not return until the window procedure has processed the message. Otherwise, it returns immediately.
<code>CWindow::SetActiveWindow</code>	Activates the window.
<code>CWindow::SetCapture</code>	Sends all subsequent mouse input to the window.
<code>CWindow::SetClipboardViewer</code>	Adds the window to the Clipboard viewer chain.
<code>CWindow::SetDlgCtrlID</code>	Changes the window's identifier.
<code>CWindow::SetDlgItemInt</code>	Changes a control's text to the string representation of an integer value.
<code>CWindow::SetDlgItemText</code>	Changes a control's text.
<code>CWindow::SetFocus</code>	Sets the input focus to the window.
<code>CWindow::SetFont</code>	Changes the window's current font.
<code>CWindow::SetHotKey</code>	Associates a hot key with the window.
<code>CWindow::SetIcon</code>	Changes the window's large or small icon.
<code>CWindow::SetMenu</code>	Changes the window's current menu.
<code>CWindow::SetParent</code>	Changes the parent window.
<code>CWindow::SetRedraw</code>	Sets or clears the redraw flag.
<code>CWindow::SetScrollInfo</code>	Sets the parameters of a scroll bar.
<code>CWindow::SetScrollPos</code>	Changes the position of the scroll box.

NAME	DESCRIPTION
CWindow::SetScrollRange	Changes the scroll bar range.
CWindow::SetTimer	Creates a timer event.
CWindow::SetWindowContextHelpId	Sets the window's help context identifier.
CWindow::SetWindowLong	Sets a 32-bit value at a specified offset into the extra window memory.
CWindow::SetWindowLongPtr	Changes an attribute of the specified window, and also sets a value at the specified offset in the extra window memory.
CWindow::SetWindowPlacement	Sets the show state and positions.
CWindow::SetWindowPos	Sets the size, position, and Z order.
CWindow::SetWindowRgn	Sets the window region of a window.
CWindow::SetWindowText	Changes the window's text.
CWindow::SetWindowWord	Sets a 16-bit value at a specified offset into the extra window memory.
CWindow::ShowCaret	Displays the system caret.
CWindow::ShowOwnedPopups	Shows or hides the pop-up windows owned by the window.
CWindow::ShowScrollBar	Shows or hides a scroll bar.
CWindow::ShowWindow	Sets the window's show state.
CWindow::ShowWindowAsync	Sets the show state of a window created by a different thread.
CWindow::UpdateWindow	Updates the client area.
CWindow::ValidateRect	Validates the client area within the specified rectangle.
CWindow::ValidateRgn	Validates the client area within the specified region.
CWindow::WinHelp	Starts Windows Help.

Public Operators

NAME	DESCRIPTION
CWindow::operator HWND	Converts the <code>CWindow</code> object to an HWND.
CWindow::operator =	Assigns an HWND to the <code>CWindow</code> object.

Public Data Members

NAME	DESCRIPTION
CWindow::m_hWnd	The handle to the window associated with the <code>CWindow</code> object.
CWindow::rcDefault	Contains default window dimensions.

Remarks

`CWindow` provides the base functionality for manipulating a window in ATL. Many of the `CWindow` methods simply wrap one of the Win32 API functions. For example, compare the prototypes for `CWindow::ShowWindow` and `ShowWindow`:

CWINDOW METHOD	WIN32 FUNCTION
BOOL ShowWindow(int nCmdShow);	BOOL ShowWindow(HWND hWnd , int nCmdShow);

`CWindow::ShowWindow` calls the Win32 function `ShowWindow` by passing `CWindow::m_hWnd` as the first parameter. Every `CWindow` method that directly wraps a Win32 function passes the `m_hWnd` member; therefore, much of the `CWindow` documentation will refer you to the Windows SDK.

NOTE
Not every window-related Win32 function is wrapped by `CWindow`, and not every `CWindow` method wraps a Win32 function.

`CWindow::m_hWnd` stores the HWND that identifies a window. An HWND is attached to your object when you:

- Specify an HWND in `CWindow`'s constructor.
- Call `CWindow::Attach`.
- Use `CWindow`'s **operator =**.
- Create or subclass a window using one of the following classes derived from `CWindow`:

[CWindowImpl](#) Allows you to create a new window or subclass an existing window.

[CContainedWindow](#) Implements a window contained within another object. You can create a new window or subclass an existing window.

[CDialogImpl](#) Allows you to create a modal or modeless dialog box.

For more information about windows, see [Windows](#) and subsequent topics in the Windows SDK. For more information about using windows in ATL, see the article [ATL Window Classes](#).

Requirements

Header: atlwin.h

CWindow::ArrangeIconicWindows

Arranges all minimized child windows.


```
UINT ArrangeIconicWindows() throw();
```

Remarks

See [ArrangeIconicWindows](#) in the Windows SDK.

CWindow::Attach

Attaches the window identified by *hWndNew* to the `CWindow` object.

```
void Attach(HWND hWndNew) throw();
```

Parameters

hWndNew

[in] The handle to a window.

Example

```
//The following example attaches an HWND to the CWindow object
HWND hWnd = ::GetDesktopWindow();
CWindow myWindow;
myWindow.Attach(hWnd);
```

CWindow::BeginPaint

Prepares the window for painting.

```
HDC BeginPaint(LPPAINTSTRUCT lpPaint) throw();
```

Remarks

See [BeginPaint](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object
//and calls CWindow::BeginPaint() and CWindow::EndPaint() in the
// WM_PAINT handler of a CWindowImpl-derived class
LRESULT CMyCtrl::OnPaint(UINT /*uMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
    BOOL& /*bHandled*/)
{
    CWindow myWindow;
    myWindow.Attach(m_hWnd);
    PAINTSTRUCT ps;
    HDC hDC = myWindow.BeginPaint(&ps);
    //Use the hDC as much as you want
    ::Rectangle(hDC, 0, 0, 50, 50);

    myWindow.EndPaint(&ps);

    return 0;
}
```

CWindow::BringWindowToTop

Brings the window to the top of the Z order.

```
BOOL BringWindowToTop() throw();
```

Remarks

See [BringWindowToTop](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::BringWindowToTop() to bring the window to the top
//of the z-order.

CWindow myWindow;
myWindow.Attach(hWndd);
BOOL bOnTop = myWindow.BringWindowToTop();

//check if we could bring the window on top
if(bOnTop)
{
    //Do something
}
```

CWindow::CenterWindow

Centers the window against a given window.

```
BOOL CenterWindow(HWND hWndCenter = NULL) throw();
```

Parameters

hWndCenter

[in] The handle to the window against which to center. If this parameter is NULL (the default value), the method will set *hWndCenter* to the window's parent window if it is a child window. Otherwise, it will set *hWndCenter* to the window's owner window.

Return Value

TRUE if the window is successfully centered; otherwise, FALSE.

Example

```
//The following example attaches various HWNDs to the CWindow objects
//and calls CWindow::CenterWindow() for each of them

CWindow childWindow, popupWindow, overlappedWindow;

childWindow.Attach(hWndChild); //a window created with WS_CHILD style
childWindow.CenterWindow();    //This will center the child
                                //window against its Parent window

popupWindow.Attach(hWndPopup); //a window created with WS_POPUP style
popupWindow.CenterWindow();    //This will center the popup window
                                //against its Owner window

overlappedWindow.Attach(hWndOverlapped); //a window created with
                                //WS_OVERLAPPED style
overlappedWindow.CenterWindow(::GetDesktopWindow()); //This will center
                                //the overlapped window against the DeskTop window
```

CWindow::ChangeClipboardChain

Removes the window from the chain of Clipboard viewers.

```
BOOL ChangeClipboardChain(HWND hWndNewNext) throw();
```

Remarks

See [ChangeClipboardChain](#) in the Windows SDK.

CWindow::CheckDlgButton

Changes the check state of the specified button.

```
BOOL CheckDlgButton(int nIDButton, UINT nCheck) throw();
```

Remarks

See [CheckDlgButton](#) in the Windows SDK.

CWindow::CheckRadioButton

Checks the specified radio button.

```
BOOL CheckRadioButton(  
    int nIDFirstButton,  
    int nIDLastButton,  
    int nIDCheckButton) throw();
```

Remarks

See [CheckRadioButton](#) in the Windows SDK.

CWindow::ChildWindowFromPoint

Retrieves the child window containing the specified point.

```
HWND ChildWindowFromPoint(POINT point) const throw();
```

Remarks

See [ChildWindowFromPoint](#) in the Windows SDK.

CWindow::ChildWindowFromPointEx

Retrieves a particular type of child window containing the specified point.

```
HWND ChildWindowFromPoint(POINT point, UINT uFlags) const throw();
```

Remarks

See [ChildWindowFromPointEx](#) in the Windows SDK.

CWindow::ClientToScreen

Converts client coordinates to screen coordinates.

```
BOOL ClientToScreen(LPPOINT lpPoint) const throw();  
BOOL ClientToScreen(LPRECT lpRect) const throw();
```

Remarks

See [ClientToScreen](#) in the Windows SDK.

The second version of this method allows you to convert the coordinates of a [RECT](#) structure.

CWindow::Create

Creates a window.

```
HWND Create(  
    LPCTSTR lpstrWndClass,  
    HWND hWndParent,  
    _U_RECT rect = NULL,  
    LPCTSTR szWindowName = NULL,  
    DWORD dwStyle = 0,  
    DWORD dwExStyle = 0,  
    _U_MENUorID MenuOrID = 0U,  
    LPVOID lpCreateParam = NULL) throw();
```

Parameters

lpstrWndClass

[in] A pointer to the window's class.

hWndParent

[in] The handle to the parent or owner window.

rect

[in] A variable of type [_U_RECT](#) specifying the position of the window. The default value is NULL. When this parameter is NULL, the value of `CWindow::rcDefault` is used.

szWindowName

[in] Specifies the name of the window. The default value is NULL.

dwStyle

[in] The style of the window. The default value is 0, meaning no style is specified. For a list of possible values, see [CreateWindow](#) in the Windows SDK.

dwExStyle

[in] The extended window style. The default value is 0, meaning no extended style is specified. For a list of possible values, see [CreateWindowEx](#) in the Windows SDK.

MenuOrID

[in] A variable of type [_U_MENUorID](#) specifying a handle to a menu or a window identifier. The default value is 0U.

lpCreateParam

A pointer to the window-creation data contained in a [CREATESTRUCT](#) structure.

Return Value

If successful, the handle to the newly created window, specified by [m_hWnd](#). Otherwise, NULL.

Remarks

CWindow::rcDefault is defined as

```
__declspec(selectany) RECT CWindow::rcDefault = {CW_USEDEFAULT, CW_USEDEFAULT, 0, 0};
```

See [CreateWindow](#) in the Windows SDK for more information.

Note If 0 is used as the value for the *MenuOrID* parameter, it must be specified as 0U (the default value) to avoid a compiler error.

CWindow::CreateCaret

Creates a new shape for the system caret.

```
BOOL CreateCaret(HBITMAP pBitmap) throw();
```

Remarks

See [CreateCaret](#) in the Windows SDK.

CWindow::CreateGrayCaret

Creates a gray rectangle for the system caret.

```
BOOL CreateGrayCaret(int nWidth, int nHeight) throw();
```

Remarks

See [CreateCaret](#) in the Windows SDK.

Passes (HBITMAP) 1 for the bitmap handle parameter to the Win32 function.

CWindow::CreateSolidCaret

Creates a solid rectangle for the system caret.

```
BOOL CreateSolidCaret(int nWidth, int nHeight) throw();
```

Remarks

See [CreateCaret](#) in the Windows SDK.

Passes (HBITMAP) 0 for the bitmap handle parameter to the Win32 function.

CWindow::CWindow

The constructor.

```
CWindow(HWND hWnd = NULL) throw();
```

Parameters

hWnd

[in] The handle to a window.

Remarks

Initializes the [m_hWnd](#) member to *hWnd*, which by default is NULL.

NOTE

`CWindow::CWindow` does not create a window. Classes [CWindowImpl](#), [CContainedWindow](#), and [CDialogImpl](#) (all of which derive from `CWindow`) provide a method to create a window or dialog box, which is then assigned to `CWindow::m_hWnd`. You can also use the [CreateWindow](#) Win32 function.

CWindow::DeferWindowPos

Updates the specified multiple-window-position structure for the specified window.

```
HDWP DeferWindowPos(  
    HDWP hWinPosInfo,  
    HWND hWndInsertAfter,  
    int x,  
    int y,  
    int cx,  
    int cy,  
    UINT uFlags) throw();
```

Remarks

See [DeferWindowPos](#) in the Windows SDK.

CWindow::DestroyWindow

Destroys the window associated with the `CWindow` object and sets [m_hWnd](#) to NULL.

```
BOOL DestroyWindow() throw();
```

Remarks

See [DestroyWindow](#) in the Windows SDK.

It does not destroy the `CWindow` object itself.

Example

```
//The following example attaches an HWND to the CWindow object and  
//calls CWindow::DestroyWindow() to destroy the window  
  
CWindow myWindow;  
myWindow.Attach(hWndChild);  
//call the CWindow wrappers  
  
myWindow.DestroyWindow();  
hWndChild = NULL;
```

CWindow::Detach

Detaches [m_hWnd](#) from the `CWindow` object and sets `m_hWnd` to NULL.

```
HWND Detach() throw();
```

Return Value

The HWND associated with the `CWindow` object.

Example

```
//The following example attaches an HWND to the CWindow object and
//later detaches the CWindow object from the HWND when no longer needed

CWindow myWindow;
myWindow.Attach(hWnd);

//call CWindow wrappers

//We don't need the C++ object any more, so detach it from the HWND.
myWindow.Detach();
```

CWindow::DlgDirList

Fills a list box with the names of all files matching a specified path or file name.

```
int DlgDirList(
    LPTSTR lpPathSpec,
    int nIDListBox,
    int nIDStaticPath,
    UINT nFileType) throw();
```

Remarks

See [DlgDirList](#) in the Windows SDK.

CWindow::DlgDirListComboBox

Fills a combo box with the names of all files matching a specified path or file name.

```
int DlgDirListComboBox(
    LPTSTR lpPathSpec,
    int nIDComboBox,
    int nIDStaticPath,
    UINT nFileType) throw();
```

Remarks

See [DlgDirListComboBox](#) in the Windows SDK.

CWindow::DlgDirSelect

Retrieves the current selection from a list box.

```
BOOL DlgDirSelect(
    LPTSTR lpString,
    int nCount,
    int nIDListBox) throw();
```

Remarks

See [DlgDirSelectEx](#) in the Windows SDK.

CWindow::DlgDirSelectComboBox

Retrieves the current selection from a combo box.

```
BOOL DlgDirSelectComboBox(  
    LPTSTR lpString,  
    int nCount,  
    int nIDComboBox) throw();
```

Remarks

See [DlgDirSelectComboBoxEx](#) in the Windows SDK.

CWindow::DragAcceptFiles

Registers whether the window accepts dragged files.

```
void DragAcceptFiles(BOOL bAccept = TRUE);
```

Remarks

See [DragAcceptFiles](#) in the Windows SDK.

CWindow::DrawMenuBar

Redraws the window's menu bar.

```
BOOL DrawMenuBar() throw();
```

Remarks

See [DrawMenuBar](#) in the Windows SDK.

CWindow::EnableScrollBar

Enables or disables the scroll bar arrows.

```
BOOL EnableScrollBar(UINT uSBFlags, UINT uArrowFlags = ESB_ENABLE_BOTH) throw();
```

Remarks

See [EnableScrollBar](#) in the Windows SDK.

CWindow::EnableWindow

Enables or disables input.

```
BOOL EnableWindow(BOOL bEnable = TRUE) throw();
```

Remarks

See [EnableWindow](#) in the Windows SDK.

Example


```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::EnableWindow() to enable and disable the window
//wrapped by the CWindow object

CWindow myWindow;
myWindow.Attach(hWnd);

//The following call enables the window
//CWindow::EnableWindow() takes TRUE as the default parameter

myWindow.EnableWindow();

if(myWindow.IsWindowEnabled())
{
    //Do something now that the window is enabled

    //Now it's time to disable the window again
    myWindow.EnableWindow(FALSE);
}
```

CWindow::EndPaint

Marks the end of painting.

```
void EndPaint(LPPAINTSTRUCT lpPaint) throw();
```

Remarks

See [EndPaint](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object
//and calls CWindow::BeginPaint() and CWindow::EndPaint() in the
// WM_PAINT handler of a CWindowImpl-derived class
LRESULT CMyCtrl::OnPaint(UINT /*uMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
    BOOL& /*bHandled*/)
{
    CWindow myWindow;
    myWindow.Attach(m_hWnd);
    PAINTSTRUCT ps;
    HDC hDC = myWindow.BeginPaint(&ps);
    //Use the hDC as much as you want
    ::Rectangle(hDC, 0, 0, 50, 50);

    myWindow.EndPaint(&ps);

    return 0;
}
```

CWindow::FlashWindow

Flashes the window once.

```
BOOL FlashWindow(BOOL bInvert) throw();
```

Remarks

See [FlashWindow](#) in the Windows SDK.

CWindow::GetClientRect

Retrieves the coordinates of the client area.

```
BOOL GetClientRect(LPRECT lpRect) const throw();
```

Remarks

See [GetClientRect](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::GetClientRect() to get the client area rectangle
//of the window

CWindow myWindow;
myWindow.Attach(hWnd);
RECT rc;
myWindow.GetClientRect(&rc);
```

CWindow::GetDC

Retrieves a device context for the client area.

```
HDC GetDC() throw();
```

Remarks

See [GetDC](#) in the Windows SDK.

Example

```
// The following example attaches a HWND to the CWindow object,
// calls CWindow::GetDC to retrieve the DC of the client
// area of the window wrapped by CWindow Object, and calls
// CWindow::ReleaseDC to release the DC.

CWindow myWindow;
myWindow.Attach(hWnd);
HDC hDC = myWindow.GetDC();

// Use the DC

myWindow.ReleaseDC(hDC);
hDC = NULL;
```

CWindow::GetDCEx

Retrieves a device context for the client area and allows clipping options.

```
HDC GetDCEx(HRGN hRgnClip, DWORD flags) throw();
```

Remarks

See [GetDCEx](#) in the Windows SDK.

CWindow::GetDescendantWindow

Finds the descendant window specified by the given identifier.

```
HWND GetDescendantWindow(int nID) const throw();
```

Parameters

nID

[in] The identifier of the descendant window to be retrieved.

Return Value

The handle to a descendant window.

Remarks

`GetDescendantWindow` searches the entire tree of child windows, not only the windows that are immediate children.

CWindow::GetDlgControl

Call this function to get a pointer to an interface of an ActiveX control that is hosted by a composite control or a control-hosting dialog.

```
HRESULT GetDlgControl(  
    int nID,  
    REFIID iid,  
    void** ppCtrl) throw();
```

Parameters

nID

[in] The resource ID of the control being retrieved.

iid

[in] The ID of the interface you would like to get from the control.

ppCtrl

[out] The pointer to the interface.

Return Value

Returns S_OK on success or any valid error HRESULT. For example, the function returns E_FAIL if the control specified by *nID* cannot be found and it returns E_NOINTERFACE if the control can be found, but it doesn't support the interface specified by *iid*.

Remarks

Using this pointer, you can call methods on the interface.

CWindow::GetDlgCtrlID

Retrieves the window's identifier (for child windows only).

```
int GetDlgCtrlID() const throw();
```

Remarks

See [GetDlgCtrlID](#) in the Windows SDK.

CWindow::GetDlgHost

Retrieves a pointer to an interface to the ATL Control hosting container.

```
HRESULT GetDlgHost(  
    int nID,  
    REFIID iid,  
    void** ppHost) throw();
```

Parameters

nID

[in] The resource ID of the control being retrieved.

iid

[in] The ID of the interface you would like to get from the control.

ppHost

[out] The pointer to the interface.

Return Value

Returns S_OK if the window specified by *iid* is a Control Container, and the requested interface could be retrieved. Returns E_FAIL if the window is not a Control Container, or if the interface requested could not be retrieved. If a window with the specified ID could not be found, then the return value is equal to HRESULT_FROM_WIN32(ERROR_CONTROL_ID_NOT_FOUND).

Remarks

Using this pointer, you can call methods on the interface.

CWindow::GetDlgItem

Retrieves the specified child window.

```
HWND GetDlgItem(int nID) const throw();
```

Remarks

See [GetDlgItem](#) in the Windows SDK.

CWindow::GetDlgItemInt

Translates a control's text to an integer.

```
UINT GetDlgItemInt(  
    int nID,  
    BOOL* lpTrans = NULL,  
    BOOL bSigned = TRUE) const throw();
```

Remarks

See [GetDlgItemInt](#) in the Windows SDK.

CWindow::GetDlgItemText

Retrieves a control's text.

```
UINT GetDlgItemText(  
    int nID,  
    LPTSTR lpStr,  
    int nMaxCount) const throw();  
  
BOOL GetDlgItemText(  
    int nID,  
    BSTR& bstrText) const throw();
```

Remarks

See [GetDlgItemText](#) in the Windows SDK.

Remarks

The second version of this method allows you to copy the control's text to a BSTR. This version returns TRUE if the text is successfully copied; otherwise, FALSE.

CWindow::GetExStyle

Retrieves the extended window styles of the window.

```
DWORD GetExStyle() const throw();
```

Return Value

The window's extended styles.

Remarks

To retrieve the regular window styles, call [GetStyle](#).

Example

```
//The following example attaches an HWND to the CWindow object and  
//calls CWindow::GetExStyle() to retrieve the extended styles of  
//the window  
  
CWindow myWindow;  
myWindow.Attach(hWnd);  
DWORD dwExStyles = myWindow.GetExStyle();
```

CWindow::GetFont

Retrieves the window's current font by sending a [WM_GETFONT](#) message to the window.

```
HFONT GetFont() const throw();
```

Return Value

A font handle.

CWindow::GetHotKey

Determines the hot key associated with the window by sending a WM_GETHOTKEY message.

```
DWORD GetHotKey() const throw();
```

Return Value

The virtual key code and modifiers for the hot key associated with the window. For a list of possible modifiers, see [WM_GETHOTKEY](#) in the Windows SDK. For a list of standard virtual key codes, see Winuser.h.

CWindow::GetIcon

Retrieves the handle to the window's large or small icon.

```
HICON GetIcon(BOOL bBigIcon = TRUE) const;
```

Parameters

bBigIcon

[in] If TRUE (the default value) the method returns the large icon. Otherwise, it returns the small icon.

Return Value

An icon handle.

Remarks

`GetIcon` sends a [WM_GETICON](#) message to the window.

CWindow::GetLastActivePopup

Retrieves the most recently active pop-up window.

```
HWND GetLastActivePopup() const throw();
```

Remarks

See [GetLastActivePopup](#) in the Windows SDK.

CWindow::GetMenu

Retrieves the window's menu.

```
HMENU GetMenu() const throw();
```

Remarks

See [GetMenu](#) in the Windows SDK.

CWindow::GetNextDlgGroupItem

Retrieves the previous or next control within a group of controls.

```
HWND GetNextDlgGroupItem(HWND hWndCtl, BOOL bPrevious = FALSE) const throw();
```

Remarks

See [GetNextDlgGroupItem](#) in the Windows SDK.

CWindow::GetNextDlgTabItem

Retrieves the previous or next control having the WS_TABSTOP style.

```
HWND GetNextDlgTabItem(HWND hWndCtl, BOOL bPrevious = FALSE) const throw();
```

Remarks

See [GetNextDlgTabItem](#) in the Windows SDK.

CWindow::GetParent

Retrieves the immediate parent window.

```
HWND GetParent() const throw();
```

Remarks

See [GetParent](#) in the Windows SDK.

Example

```
// The following example attaches a HWND to the CWindow object
// and calls CWindow::GetParent to find out the parent
// window of the window wrapped by CWindow object.

CWindow myWindow;
myWindow.Attach(hWnd);
HWND hWndParent = myWindow.GetParent();
```

CWindow::GetScrollInfo

Retrieves the parameters of a scroll bar.

```
BOOL GetScrollInfo(int nBar, LPSCROLLINFO lpScrollInfo) throw();
```

Remarks

See [GetScrollInfo](#) in the Windows SDK.

CWindow::GetScrollPos

Retrieves the position of the scroll box.

```
int GetScrollPos(int nBar) const throw();
```

Remarks

See [GetScrollPos](#) in the Windows SDK.

CWindow::GetScrollRange

Retrieves the scroll bar range.

```
BOOL GetScrollRange(
    int nBar,
    LPINT lpMinPos,
    LPINT lpMaxPos) const throw();
```

Remarks

See [GetScrollRange](#) in the Windows SDK.

CWindow::GetStyle

Retrieves the window styles of the window.

```
DWORD GetStyle() const throw();
```

Return Value

The window's styles.

Remarks

To retrieve the extended window styles, call [GetExStyle](#).

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::GetStyle() to retrieve the styles of the window

CWindow myWindow;
myWindow.Attach(hWnd);
DWORD dwStyles = myWindow.GetStyle();
```

CWindow::GetSystemMenu

Creates a copy of the system menu for modification.

```
HMENU GetSystemMenu(BOOL bRevert) const throw();
```

Remarks

See [GetSystemMenu](#) in the Windows SDK.

CWindow::GetTopLevelParent

Retrieves the window's top-level parent window.

```
HWND GetTopLevelParent() const throw();
```

Return Value

The handle to the top-level parent window.

CWindow::GetTopLevelWindow

Retrieves the window's top-level parent or owner window.

```
HWND GetTopLevelWindow() const throw();
```

Return Value

The handle to the top-level owner window.

CWindow::GetTopWindow

Retrieves the top-level child window.

```
HWND GetTopWindow() const throw();
```

Remarks

See [GetTopWindow](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::GetTopWindow() to get the top-level child window

CWindow myWindow;
myWindow.Attach(hWnd);
HWND hWndFavoriteChild = myWindow.GetTopWindow();
```

CWindow::GetUpdateRect

Retrieves the coordinates of the smallest rectangle that completely encloses the update region.

```
BOOL GetUpdateRect(LPRECT lpRect, BOOL bErase = FALSE) throw();
```

Remarks

See [GetUpdateRect](#) in the Windows SDK.

CWindow::GetUpdateRgn

Retrieves the update region and copies it into a specified region.

```
int GetUpdateRgn(HRGN hRgn, BOOL bErase = FALSE) throw();
```

Remarks

See [GetUpdateRgn](#) in the Windows SDK.

CWindow::GetWindow

Retrieves the specified window.

```
HWND GetWindow(UINT nCmd) const throw();
```

Remarks

See `GetWindow` in the Windows SDK.

CWindow::GetWindowContextHelpId

Retrieves the window's help context identifier.

```
DWORD GetWindowContextHelpId() const throw();
```

Remarks

See [GetWindowContextHelpId](#) in the Windows SDK.

CWindow::GetWindowDC

Retrieves a device context for the entire window.

```
HDC GetWindowDC() throw();
```

Remarks

See [GetWindowDC](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::GetWindowDC() to retrieve the DC of the entire window

CWindow myWindow;
myWindow.Attach(hWnd);
HDC hDC = myWindow.GetWindowDC();
```

CWindow::GetWindowLong

Retrieves a 32-bit value at a specified offset into the extra window memory.

```
LONG GetWindowLong(int nIndex) const throw();
```

Remarks

See [GetWindowLong](#) in the Windows SDK.

NOTE

To write code that is compatible with both 32-bit and 64-bit versions of Windows, use [CWindow::GetWindowLongPtr](#).

CWindow::GetWindowLongPtr

Retrieves information about the specified window, including a value at a specified offset into the extra window memory.

```
LONG_PTR GetWindowLongPtr(int nIndex) const throw();
```

Remarks

See [GetWindowLongPtr](#) in the Windows SDK.

Remarks

If you are retrieving a pointer or a handle, this function supersedes the `CWindow::GetWindowLong` method.

NOTE

Pointers and handles are 32 bits on 32-bit Windows and 64 bits on 64-bit Windows.

To write code that is compatible with both 32-bit and 64-bit versions of Windows, use

```
CWindow::GetWindowLongPtr .
```

CWindow::GetWindowPlacement

Retrieves the show state and positions.

```
BOOL GetWindowPlacement(WINDOWPLACEMENT FAR* lpwndpl) const throw();
```

Remarks

See [GetWindowPlacement](#) in the Windows SDK.

CWindow::GetWindowProcessID

Retrieves the identifier of the process that created the window.

```
DWORD GetWindowProcessID() throw();
```

Remarks

See [GetWindowThreadProcessID](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and  
//calls CWindow::GetWindowProcessID() to retrieve the id of the  
//process that created the window  
  
CWindow myWindow;  
myWindow.Attach(hWnd);  
DWORD dwID = myWindow.GetWindowProcessID();
```

CWindow::GetWindowRect

Retrieves the window's bounding dimensions.

```
BOOL GetWindowRect(LPRECT lpRect) const throw();
```

Remarks

See [GetWindowRect](#) in the Windows SDK.

CWindow::GetWindowRgn

Obtains a copy of the window region of a window.

```
int GetWindowRgn(HRGN hRgn) throw();
```

Remarks

See [GetWindowRgn](#) in the Windows SDK.

CWindow::GetWindowText

Retrieves the window's text.

```
int GetWindowText(LPTSTR lpszStringBuf, int nMaxCount) const throw();
BOOL GetWindowText(BSTR& bstrText) throw();
int GetWindowText(CSimpleString& strText) const;
```

Parameters

lpszStringBuf

A buffer to which to write the window text.

nMaxCount

The size of the buffer in characters; also the maximum number of characters to write.

bstrText

A BSTR in which to store the window text.

strText

A `CString` in which to store the window text.

Return Value

If the text is successfully copied, the return value is TRUE; otherwise, the return value is FALSE.

Remarks

See [GetWindowText](#) in the Windows SDK.

The second version of this method allows you to store the text in a BSTR; the third version allows you to store the result in a [CString](#), since `CSimpleString` is the base class of `CString`.

CWindow::GetWindowTextLength

Retrieves the length of the window's text.

```
int GetWindowTextLength() const throw();
```

Remarks

See [GetWindowTextLength](#) in the Windows SDK.

CWindow::GetWindowThreadID

Retrieves the identifier of the thread that created the specified window.

```
DWORD GetWindowThreadID() throw();
```

Remarks

See [GetWindowThreadProcessID](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::GetWindowThreadID() to retrieve the id of the thread
//that created the window
```

```
CWindow myWindow;
myWindow.Attach(hWnd);
DWORD dwID = myWindow.GetWindowThreadID();
```

CWindow::GetWindowWord

Retrieves a 16-bit value at a specified offset into the extra window memory.

```
WORD GetWindowWord(int nIndex) const throw();
```

Remarks

See [GetWindowLong](#) in the Windows SDK.

CWindow::GotoDlgCtrl

Sets the keyboard focus to a control in the dialog box.

```
void GotoDlgCtrl(HWND hWndCtrl) const throw();
```

Remarks

See [WM_NEXTDLGCTL](#) in the Windows SDK.

CWindow::HideCaret

Hides the system caret.

```
BOOL HideCaret() throw();
```

Remarks

See [HideCaret](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::HideCaret() to hide the caret of the window owning
//the caret
```

```
CWindow myWindow;
myWindow.Attach(hWndEdit);
myWindow.HideCaret();
```

CWindow::HiliteMenuItem

Highlights or removes the highlight from a top-level menu item.

```
BOOL HiliteMenuItem(  
    HMENU hMenu,  
    UINT uHiliteItem,  
    UINT uHilite) throw();
```

Remarks

See [HiliteMenuItem](#) in the Windows SDK.

CWindow::Invalidate

Invalidates the entire client area.

```
BOOL Invalidate(BOOL bErase = TRUE) throw();
```

Remarks

See [InvalidateRect](#) in the Windows SDK.

Passes NULL for the `RECT` parameter to the `InvalidateRect` Win32 function.

Example

```
//The following example attaches an HWND to the CWindow object and  
//calls CWindow::Invalidate() to invalidate the entire client area  
  
CWindow myWindow;  
myWindow.Attach(hWnd);  
myWindow.Invalidate();
```

CWindow::InvalidateRect

Invalidates the client area within the specified rectangle.

```
BOOL InvalidateRect(LPCRECT lpRect, BOOL bErase = TRUE) throw();
```

Remarks

See [InvalidateRect](#) in the Windows SDK.

CWindow::InvalidateRgn

Invalidates the client area within the specified region.

```
void InvalidateRgn(HRGN hRgn, BOOL bErase = TRUE) throw();
```

Remarks

See [InvalidateRgn](#) in the Windows SDK.

Remarks

Specifies a **void** return type, while the `InvalidateRgn` Win32 function always returns TRUE.

CWindow::IsChild

Determines whether the specified window is a child window.

```
BOOL IsChild(const HWND hWnd) const throw();
```

Remarks

See [IsChild](#) in the Windows SDK.

CWindow::IsDialogMessage

Determines whether a message is intended for the specified dialog box.

```
BOOL IsDialogMessage(LPMSG lpMsg) throw();
```

Remarks

See [IsDialogMessage](#) in the Windows SDK.

CWindow::IsDlgButtonChecked

Determines the check state of the button.

```
UINT IsDlgButtonChecked(int nIDButton) const throw();
```

Remarks

See [IsDlgButtonChecked](#) in the Windows SDK.

CWindow::IsIconic

Determines whether the window is minimized.

```
BOOL IsIconic() const throw();
```

Remarks

See [IsIconic](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and  
//calls CWindow::IsIconic() to determine if the window is minimized
```

```
CWindow myWindow;  
myWindow.Attach(hWnd);  
BOOL bIconic = myWindow.IsIconic();
```

CWindow::IsParentDialog

Determines if the parent window of the control is a dialog window.

```
BOOL IsParentDialog() throw();
```

Return Value

Returns TRUE if the parent window is a dialog, FALSE otherwise.

CWindow::IsWindow

Determines whether the specified window handle identifies an existing window.

```
BOOL IsWindow() throw();
```

Remarks

See [IsWindow](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::IsWindow() to verify if the HWND corresponds
//to an existing window

CWindow myWindow;
myWindow.Attach(hWnd);
BOOL bWindow = myWindow.IsWindow();
```

CWindow::IsWindowEnabled

Determines whether the window is enabled for input.

```
BOOL IsWindowEnabled() const throw();
```

Remarks

See [IsWindowEnabled](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::IsWindowEnabled() to verify if the window is enabled
//for receiving input

CWindow myWindow;
myWindow.Attach(hWnd);
BOOL bEnabled = myWindow.IsWindowEnabled();
```

CWindow::IsWindowVisible

Determines the window's visibility state.

```
BOOL IsWindowVisible() const throw();
```

Remarks

See [IsWindowVisible](#) in the Windows SDK.

Example


```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::IsWindowVisible() to determine the visibility state
//of the window
```

```
CWindow myWindow;
myWindow.Attach(hWnd);
BOOL bVisible = myWindow.IsWindowVisible();
```

CWindow::IsWindowUnicode

Determines whether the specified window is a native Unicode window.

```
BOOL IsWindowUnicode() throw();
```

Remarks

See [IsWindowUnicode](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::IsWindowUnicode() to determine if the window is a
//UNICODE window or an ANSI one.
```

```
CWindow myWindow;
myWindow.Attach(hWnd);
BOOL bUnicode = myWindow.IsWindowUnicode();
```

CWindow::IsZoomed

Determines whether the window is maximized.

```
BOOL IsZoomed() const throw();
```

Remarks

See [IsZoomed](#) in the Windows SDK.

CWindow::KillTimer

Destroys a timer event created by `CWindow::SetTimer`.

```
BOOL KillTimer(UINT nIDEvent) throw();
```

Remarks

See [KillTimer](#) in the Windows SDK.

CWindow::LockWindowUpdate

Disables or enables drawing in the window by calling the [LockWindowUpdate](#) Win32 function.

```
BOOL LockWindowUpdate(BOOL bLock = TRUE) throw();
```

Parameters

bLock

[in] If TRUE (the default value), the window will be locked. Otherwise, it will be unlocked.

Return Value

TRUE if the window is successfully locked; otherwise, FALSE.

Remarks

If *bLock* is TRUE, this method passes [m_hWnd](#) to the Win32 function; otherwise, it passes NULL.

CWindow::m_hWnd

Contains a handle to the window associated with the `CWindow` object.

```
HWND m_hWnd throw() throw();
```

CWindow::MapWindowPoints

Converts a set of points from the window's coordinate space to the coordinate space of another window.

```
int MapWindowPoints(  
    HWND hWndTo,  
    LPPOINT lpPoint,  
    UINT nCount) const throw();  
  
int MapWindowPoints(  
    HWND hWndTo,  
    LPRECT lpRect) const throw();
```

Remarks

See [MapWindowPoints](#) in the Windows SDK.

The second version of this method allows you to convert the coordinates of a [RECT](#) structure.

CWindow::MessageBox

Displays a message box.

```
int MessageBox(  
    LPCTSTR lpszText,  
    LPCTSTR lpszCaption = NULL,  
    UINT nType = MB_OK) throw();
```

Remarks

See [MessageBox](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and  
//calls CWindow::MessageBox() to pop up a Windows message box  
  
CWindow myWindow;  
myWindow.Attach(hWnd);  
myWindow.MessageBox(_T("Hello World"));
```

CWindow::ModifyStyle

Modifies the window styles of the `CWindow` object.

```
BOOL ModifyStyle(  
    DWORD dwRemove,  
    DWORD dwAdd,  
    UINT nFlags = 0) throw();
```

Parameters

dwRemove

[in] Specifies the window styles to be removed during style modification.

dwAdd

[in] Specifies the window styles to be added during style modification.

nFlags

[in] Window-positioning flags. For a list of possible values, see the [SetWindowPos](#) function in the Windows SDK.

Return Value

TRUE if the window styles are modified; otherwise, FALSE.

Remarks

Styles to be added or removed can be combined by using the bitwise OR (`|`) operator. See the [CreateWindow](#) function in the Windows SDK for information about the available window styles.

If *nFlags* is nonzero, `ModifyStyle` calls the Win32 function `SetWindowPos`, and redraws the window by combining *nFlags* with the following four flags:

- `SWP_NOSIZE` Retains the current size.
- `SWP_NOMOVE` Retains the current position.
- `SWP_NOZORDER` Retains the current Z order.
- `SWP_NOACTIVATE` Does not activate the window.

To modify a window's extended styles, call [ModifyStyleEx](#).

Example

```
//The following example attaches an HWND to the CWindow object and  
//calls CWindow::ModifyStyle() to add and remove the window styles  
  
CWindow myWindow;  
myWindow.Attach(hWnd);  
  
//The following line removes the WS_CLIPCHILDREN style from the  
//window and adds the WS_CAPTION style to the window  
myWindow.ModifyStyle(WS_CLIPCHILDREN, WS_CAPTION);
```

CWindow::ModifyStyleEx

Modifies the extended window styles of the `CWindow` object.

```
BOOL ModifyStyleEx(  
    DWORD dwRemove,  
    DWORD dwAdd,  
    UINT nFlags = 0) throw();
```

Parameters

dwRemove

[in] Specifies the extended styles to be removed during style modification.

dwAdd

[in] Specifies the extended styles to be added during style modification.

nFlags

[in] Window-positioning flags. For a list of possible values, see the [SetWindowPos](#) function in the Windows SDK.

Return Value

TRUE if the extended window styles are modified; otherwise, FALSE.

Remarks

Styles to be added or removed can be combined by using the bitwise OR (|) operator. See the [CreateWindowEx](#) function in the Windows SDK for information about the available extended styles.

If *nFlags* is nonzero, `ModifyStyleEx` calls the Win32 function `SetWindowPos`, and redraws the window by combining *nFlags* with the following four flags:

- SWP_NOSIZE Retains the current size.
- SWP_NOMOVE Retains the current position.
- SWP_NOZORDER Retains the current Z order.
- SWP_NOACTIVATE Does not activate the window.

To modify windows using regular window styles, call [ModifyStyle](#).

Example

```
//The following example attaches an HWND to the CWindow object and  
//calls CWindow::ModifyStyleEx() to add and remove the extended  
//window styles  
  
CWindow myWindow;  
myWindow.Attach(hWnd);  
  
//The following line removes WS_EX_CONTEXTHELP extended style from  
//the window and adds WS_EX_TOOLWINDOW extended style to the window  
  
myWindow.ModifyStyleEx(WS_EX_CONTEXTHELP, WS_EX_TOOLWINDOW);
```

CWindow::MoveWindow

Changes the window's size and position.

```

BOOL MoveWindow(
    int x,
    int y,
    int nWidth,
    int nHeight,
    BOOL bRepaint = TRUE) throw();

BOOL MoveWindow(
    LPCRECT lpRect,
    BOOL bRepaint = TRUE) throw();

```

Remarks

For a top-level window object, the x and y parameters are relative to the upper-left corner of the screen. For a child window object, they are relative to the upper-left corner of the parent window's client area.

The second version of this method uses a [RECT](#) structure to determine the window's new position, width, and height.

CWindow::NextDlgCtrl

Sets the keyboard focus to the next control in the dialog box.

```
void NextDlgCtrl() const throw();
```

Remarks

See [WM_NEXTDLGCTL](#) in the Windows SDK.

CWindow::OpenClipboard

Opens the Clipboard.

```
BOOL OpenClipboard() throw();
```

Remarks

See [OpenClipboard](#) in the Windows SDK.

CWindow::operator HWND

Converts a `CWindow` object to an HWND.

```
operator HWND() const throw();
```

CWindow::operator =

Assigns an HWND to the `CWindow` object by setting the `m_hWnd` member to `hWnd`.

```
CWindow& operator= (HWND hWnd) throw();
```

CWindow::PostMessage

Places a message in the message queue associated with the thread that created the window.

```
BOOL PostMessage(  
    UINT message,  
    WPARAM wParam = 0,  
    LPARAM lParam = 0) throw();
```

Remarks

See [PostMessage](#) in the Windows SDK.

Returns without waiting for the thread to process the message.

Example

```
//The following example attaches an HWND to the CWindow object and  
//posts a WM_PAINT message to the Window wrapped by the CWindow object  
//using CWindow::PostMessage() with the default values of WPARAM and  
//LPARAM  
  
CWindow myWindow;  
myWindow.Attach(hWnd);  
myWindow.PostMessage(WM_PAINT);
```

CWindow::PrevDlgCtrl

Sets the keyboard focus to the previous control in the dialog box.

```
void PrevDlgCtrl() const throw();
```

Remarks

See [WM_NEXTDLGCTL](#) in the Windows SDK.

CWindow::Print

Sends a [WM_PRINT](#) message to the window to request that it draw itself in the specified device context.

```
void Print(HDC hDC, DWORD dwFlags) const throw();
```

Parameters

hDC

[in] The handle to a device context.

dwFlags

[in] Specifies the drawing options. You can combine one or more of the following flags:

- [PRF_CHECKVISIBLE](#) Draw the window only if it is visible.
- [PRF_CHILDREN](#) Draw all visible child windows.
- [PRF_CLIENT](#) Draw the client area of the window.
- [PRF_ERASEBKGND](#) Erase the background before drawing the window.
- [PRF_NONCLIENT](#) Draw the non-client area of the window.
- [PRF_OWNED](#) Draw all owned windows.

CWindow::PrintClient

Sends a [WM_PRINTCLIENT](#) message to the window to request that it draw its client area in the specified device context.

```
void PrintClient(HDC hDC, DWORD dwFlags) const throw();
```

Parameters

hDC

[in] The handle to a device context.

dwFlags

[in] Specifies drawing options. You can combine one or more of the following flags:

- PRF_CHECKVISIBLE Draw the window only if it is visible.
- PRF_CHILDREN Draw all visible child windows.
- PRF_CLIENT Draw the client area of the window.
- PRF_ERASEBKGND Erase the background before drawing the window.
- PRF_NONCLIENT Draw the non-client area of the window.
- PRF_OWNED Draw all owned windows.

CWindow::rcDefault

Contains default window dimensions.

```
static RECT rcDefault;
```

CWindow::RedrawWindow

Updates a specified rectangle or region in the client area.

```
BOOL RedrawWindow(  
    LPCRECT lpRectUpdate = NULL,  
    HRGN hRgnUpdate = NULL,  
    UINT flags = RDW_INVALIDATE | RDW_UPDATENOW | RDW_ERASE);  
  
throw();
```

Remarks

See [RedrawWindow](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and  
//calls CWindow::RedrawWindow() to update the entire window using the  
//default arguments  
  
CWindow myWindow;  
myWindow.Attach(hWnd);  
BOOL bRedrawn = myWindow.RedrawWindow();
```

CWindow::ReleaseDC

Releases a device context.

```
int ReleaseDC(HDC hDC);
```

Remarks

See [ReleaseDC](#) in the Windows SDK.

Example

```
// The following example attaches a HWND to the CWindow object,  
// calls CWindow::GetDC to retrieve the DC of the client  
// area of the window wrapped by CWindow Object, and calls  
// CWindow::ReleaseDC to release the DC.  
  
CWindow myWindow;  
myWindow.Attach(hWnd);  
HDC hDC = myWindow.GetDC();  
  
// Use the DC  
  
myWindow.ReleaseDC(hDC);  
hDC = NULL;
```

CWindow::ResizeClient

Resizes the window to the specified client area size.

```
BOOL ResizeClient(  
    int nWidth,  
    int nHeight,  
    BOOL bRedraw = FALSE) throw();
```

Parameters

nWidth

New width of the window in pixels.

nHeight

New height of the window in pixels.

bRedraw

A flag indicating whether to redraw changes. Default is FALSE, indicating the window does not redraw changes.

CWindow::ScreenToClient

Converts screen coordinates to client coordinates.

```
BOOL ScreenToClient(LPPOINT lpPoint) const throw();  
BOOL ScreenToClient(LPRECT lpRect) const throw();
```

Remarks

See [ScreenToClient](#) in the Windows SDK.

The second version of this method allows you to convert the coordinates of a [RECT](#) structure.

CWindow::ScrollWindow

Scrolls the specified client area.

```
BOOL ScrollWindow(  
    int xAmount,  
    int yAmount,  
    LPCRECT lpRect = NULL,  
    LPCRECT lpClipRect = NULL) throw();
```

Remarks

See [ScrollWindow](#) in the Windows SDK.

CWindow::ScrollWindowEx

Scrolls the specified client area with additional features.

```
int ScrollWindowEx(  
    int dx,  
    int dy,  
    LPCRECT lpRectScroll,  
    LPCRECT lpRectClip,  
    HRGN hRgnUpdate,  
    LPRECT lpRectUpdate,  
    UINT flags) throw();
```

Remarks

See [ScrollWindowEx](#) in the Windows SDK.

CWindow::SendDlgItemMessage

Sends a message to a control.

```
LRESULT SendDlgItemMessage(  
    int nID,  
    UINT message,  
    WPARAM wParam = 0,  
    LPARAM lParam = 0) throw();
```

Remarks

See [SendDlgItemMessage](#) in the Windows SDK.

CWindow::SendMessage

Sends a message to the window and does not return until the window procedure has processed the message.

```

LRESULT SendMessage(
    UINT message,
    WPARAM wParam = 0,
    LPARAM lParam = 0) throw();

static LRESULT SendMessage(
    HWND hWnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam) throw();

```

Remarks

See [SendMessage](#) in the Windows SDK.

Example

```

// The following example attaches a HWND to the CWindow
// object and sends a WM_PAINT message to the window
// wrapped by CWindow object using CWindow::SendMessage.

CWindow myWindow;
myWindow.Attach(hWnd);
myWindow.SendMessage(WM_PAINT, 0L, 0L);

```

CWindow::SendMessageToDescendants

Sends the specified message to all immediate children of the `CWindow` object.

```

void SendMessageToDescendants(
    UINT message,
    WPARAM wParam = 0,
    LPARAM lParam = 0,
    BOOL bDeep = TRUE) throw();

```

Parameters

message

[in] The message to be sent.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

bDeep

[in] If TRUE (the default value), the message will be sent to all descendant windows; otherwise, it will be sent only to the immediate child windows.

Remarks

If *bDeep* is TRUE, the message is additionally sent to all other descendant windows.

CWindow::SendMessage

Sends a message to the window.

```
BOOL SendNotifyMessage(  
    UINT message,  
    WPARAM wParam = 0,  
    LPARAM lParam = 0) throw();
```

Remarks

See [SendNotifyMessage](#) in the Windows SDK.

If the window was created by the calling thread, `SendNotifyMessage` does not return until the window procedure has processed the message. Otherwise, it returns immediately.

CWindow::SetActiveWindow

Activates the window.

```
HWND SetActiveWindow() throw();
```

Remarks

See [SetActiveWindow](#) in the Windows SDK.

Example

```
// The following example attaches a HWND to the CWindow object  
// and sets the window as an active window by calling  
// CWindow::SetActiveWindow which returns the HWND of the  
// previously active window.  
  
CWindow myWindow;  
myWindow.Attach(hWnd);  
HWND hWndPrev = myWindow.SetActiveWindow();
```

CWindow::SetCapture

Sends all subsequent mouse input to the window.

```
HWND SetCapture() throw();
```

Remarks

See [SetCapture](#) in the Windows SDK.

CWindow::SetClipboardViewer

Adds the window to the Clipboard viewer chain.

```
HWND SetClipboardViewer() throw();
```

Remarks

See [SetClipboardViewer](#) in the Windows SDK.

CWindow::SetDlgCtrlID

Sets the identifier of the window to the specified value.

```
int SetDlgCtrlID(int nID) throw();
```

Parameters

nID

[in] The new value to set for the window's identifier.

Return Value

If successful, the previous identifier of the window; otherwise 0.

CWindow::SetDlgItemInt

Changes a control's text to the string representation of an integer value.

```
BOOL SetDlgItemInt(  
    int nID,  
    UINT nValue,  
    BOOL bSigned = TRUE) throw();
```

Remarks

See [SetDlgItemInt](#) in the Windows SDK.

CWindow::SetDlgItemText

Changes a control's text.

```
BOOL SetDlgItemText(int nID, LPCTSTR lpszString) throw();
```

Remarks

See [SetDlgItemText](#) in the Windows SDK.

CWindow::SetFocus

Sets the input focus to the window.

```
HWND SetFocus() throw();
```

Remarks

See [SetFocus](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and  
//calls CWindow::SetFocus() to set the input focus  
  
CWindow myWindow;  
myWindow.Attach(hWnd);  
HWND hWndLeftFocus = myWindow.SetFocus();
```

CWindow::SetFont

Changes the window's current font by sending a [WM_SETFONT](#) message to the window.

```
void SetFont(HFONT hFont, BOOL bRedraw = TRUE) throw();
```

Parameters

hFont

[in] The handle to the new font.

bRedraw

[in] If TRUE (the default value), the window is redrawn. Otherwise, it is not.

CWindow::SetHotKey

Associates a hot key with the window by sending a WM_SETHOTKEY message.

```
int SetHotKey(WORD wVirtualKeyCode, WORD wModifiers) throw();
```

Parameters

wVirtualKeyCode

[in] The virtual key code of the hot key. For a list of standard virtual key codes, see Winuser.h.

wModifiers

[in] The modifiers of the hot key. For a list of possible values, see WM_SETHOTKEY in the Windows SDK.

Return Value

For a list of possible return values, see [WM_SETHOTKEY](#) in the Windows SDK.

CWindow::SetIcon

Sets the window's large or small icon to the icon identified by *hIcon*.

```
HICON SetIcon(HICON hIcon, BOOL bBigIcon = TRUE) throw();
```

Parameters

hIcon

[in] The handle to a new icon.

bBigIcon

[in] If TRUE (the default value), the method sets a large icon. Otherwise, it sets a small icon.

Return Value

The handle to the previous icon.

Remarks

`SetIcon` sends a [WM_SETICON](#) message to the window.

CWindow::SetMenu

Changes the window's current menu.

```
BOOL SetMenu(HMENU hMenu) throw();
```

Remarks

See [SetMenu](#) in the Windows SDK.

CWindow::SetParent

Changes the parent window.

```
HWND SetParent(HWND hWndNewParent) throw();
```

Remarks

See [SetParent](#) in the Windows SDK.

Example

```
// The following example attaches a HWND to the CWindow object
// and sets the hWndParent as the parent window of the
// window wrapped by CWindow object using CWindow::SetParent.

CWindow myWindow;
myWindow.Attach(hWndChild);
HWND hWndPrevParent = myWindow.SetParent(hWndParent);
```

CWindow::SetRedraw

Sets or clears the redraw flag by sending a [WM_SETREDRAW](#) message to the window.

```
void SetRedraw(BOOL bRedraw = TRUE) throw();
```

Parameters

bRedraw

[in] Specifies the state of the redraw flag. If TRUE (the default value), the redraw flag is set; if FALSE, the flag is cleared.

Remarks

Call `SetRedraw` to allow changes to be redrawn or to prevent changes from being redrawn.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::SetRedraw() to set and reset the redraw flag

CWindow myWindow;
myWindow.Attach(hWnd);
myWindow.SetRedraw();      //sets the redraw flag to TRUE
//
//
myWindow.SetRedraw(FALSE); //sets the redraw flag to FALSE
```

CWindow::SetScrollInfo

Sets the parameters of a scroll bar.

```
int SetScrollInfo(  
    int nBar,  
    LPSCROLLINFO lpScrollInfo,  
    BOOL bRedraw = TRUE) throw();
```

Remarks

See [SetScrollInfo](#) in the Windows SDK.

CWindow::SetScrollPos

Changes the position of the scroll box.

```
int SetScrollPos(  
    int nBar,  
    int nPos,  
    BOOL bRedraw = TRUE) throw();
```

Remarks

See [SetScrollPos](#) in the Windows SDK.

CWindow::SetScrollRange

Changes the scroll bar range.

```
BOOL SetScrollRange(  
    int nBar,  
    int nMinPos,  
    int nMaxPos,  
    BOOL bRedraw = TRUE) throw();
```

Remarks

See [SetScrollRange](#) in the Windows SDK.

CWindow::SetTimer

Creates a timer event.

```
UINT SetTimer(  
    UINT nIDEvent,  
    UINT nElapse,  
    void (CALLBACK* lpfnTimer)(HWND, UINT, UINT, DWORD) = NULL) throw();
```

Remarks

See [SetTimer](#) in the Windows SDK.

CWindow::SetWindowContextHelpId

Sets the window's help context identifier.

```
BOOL SetWindowContextHelpId(DWORD dwContextHelpId) throw();
```

Remarks

See [SetWindowContextHelpId](#) in the Windows SDK.

CWindow::SetWindowLong

Sets a 32-bit value at a specified offset into the extra window memory.

```
LONG SetWindowLong(int nIndex, LONG dwNewLong) throw();
```

Remarks

See [SetWindowLong](#) in the Windows SDK.

NOTE

To write code that is compatible with both 32-bit and 64-bit versions of Windows, use [CWindow::SetWindowLongPtr](#).

CWindow::SetWindowLongPtr

Changes an attribute of the specified window, and also sets a value at the specified offset in the extra window memory.

```
LONG_PTR SetWindowLongPtr(int nIndex, LONG_PTR dwNewLong) throw();
```

Remarks

See [SetWindowLongPtr](#) in the Windows SDK.

This function supersedes the `CWindow::SetWindowLong` method. To write code that is compatible with both 32-bit and 64-bit versions of Windows, use `CWindow::SetWindowLongPtr`.

CWindow::SetWindowPlacement

Sets the show state and positions.

```
BOOL SetWindowPlacement(const WINDOWPLACEMENT FAR* lpwndpl);
```

Remarks

See [SetWindowPlacement](#) in the Windows SDK.

CWindow::SetWindowPos

Sets the size, position, and Z order.


```

BOOL SetWindowPos(
    HWND hWndInsertAfter,
    int x,
    int y,
    int cx,
    int cy,
    UINT nFlags) throw();

BOOL SetWindowPos(
    HWND hWndInsertAfter,
    LPCRECT lpRect,
    UINT nFlags) throw();

```

Remarks

See [SetWindowPos](#) in the Windows SDK.

The second version of this method uses a [RECT](#) structure to set the window's new position, width, and height.

CWindow::SetWindowRgn

Sets the window region of a window.

```
int SetWindowRgn(HRGN hRgn, BOOL bRedraw = FALSE) throw();
```

Remarks

See [SetWindowRgn](#) in the Windows SDK.

CWindow::SetWindowText

Changes the window's text.

```
BOOL SetWindowText(LPCTSTR lpszString) throw();
```

Remarks

See [SetWindowText](#) in the Windows SDK.

Example

```

//The following example attaches an HWND to the CWindow object and
//calls CWindow::SetWindowText() to set the new title-text of the
//window

CWindow myWindow;
myWindow.Attach(hWnd);
myWindow.SetWindowText(_T("Hello ATL"));

```

CWindow::SetWindowWord

Sets a 16-bit value at a specified offset into the extra window memory.

```
WORD SetWindowWord(int nIndex, WORD wNewWord) throw();
```

Remarks

See [SetWindowLong](#) in the Windows SDK.

CWindow::ShowCaret

Displays the system caret.

```
BOOL ShowCaret() throw();
```

Remarks

See [ShowCaret](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and  
//calls CWindow::ShowCaret() to show the caret  
  
CWindow myWindow;  
myWindow.Attach(hWnd);  
myWindow.ShowCaret();
```

CWindow::ShowOwnedPopups

Shows or hides the pop-up windows owned by the window.

```
BOOL ShowOwnedPopups(BOOL bShow = TRUE) throw();
```

Remarks

See [ShowOwnedPopups](#) in the Windows SDK.

CWindow::ShowScrollBar

Shows or hides a scroll bar.

```
BOOL ShowScrollBar(UINT nBar, BOOL bShow = TRUE) throw();
```

Remarks

See [ShowScrollBar](#) in the Windows SDK.

CWindow::ShowWindow

Sets the window's show state.

```
BOOL ShowWindow(int nCmdShow) throw();
```

Remarks

See [ShowWindow](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::ShowWindow() to show the window in its maximized state

CWindow myWindow;
myWindow.Attach(hWnd);
myWindow.ShowWindow(SW_SHOWMAXIMIZED);
```

CWindow::ShowWindowAsync

Sets the show state of a window created by a different thread.

```
BOOL ShowWindowAsync(int nCmdShow) throw();
```

Remarks

See [ShowWindowAsync](#) in the Windows SDK.

CWindow::UpdateWindow

Updates the client area.

```
BOOL UpdateWindow() throw();
```

Remarks

See [UpdateWindow](#) in the Windows SDK.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::UpdateWindow() to update the window

CWindow myWindow;
myWindow.Attach(hWnd);
BOOL bUpdated = myWindow.UpdateWindow();
```

CWindow::ValidateRect

Validates the client area within the specified rectangle.

```
BOOL ValidateRect(LPCRECT lpRect) throw();
```

Remarks

See [ValidateRect](#) in the Windows SDK.

CWindow::ValidateRgn

Validates the client area within the specified region.

```
BOOL ValidateRgn(HRGN hRgn) throw();
```

Remarks

See [ValidateRgn](#) in the Windows SDK.

CWindow::WinHelp

Starts Windows Help.

```
BOOL WinHelp(  
    LPCTSTR lpszHelp,  
    UINT nCmd = HELP_CONTEXT,  
    DWORD dwData = 0) throw();
```

Remarks

See [WinHelp](#) in the Windows SDK.

See also

[Class Overview](#)

CWindowImpl Class

3/12/2019 • 6 minutes to read • [Edit Online](#)

Provides methods for creating or subclassing a window.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class T, class TBase = CWindow, class TWinTraits = CControlWinTraits>  
class ATL_NO_VTABLE CWindowImpl : public CWindowImplBaseT<TBase, TWinTraits>
```

Parameters

T

Your new class, derived from `CWindowImpl`.

TBase

The base class of your class. By default, the base class is [CWindow](#).

TWinTraits

A [traits class](#) that defines styles for your window. The default is `CControlWinTraits`.

Members

Public Methods

NAME	DESCRIPTION
CWindowImpl::Create	Creates a window.

CWindowImplBaseT Methods

DefWindowProc	Provides default message processing.
GetCurrentMessage	Returns the current message.
GetWindowProc	Returns the current window procedure.
OnFinalMessage	Called after the last message is received (typically WM_NCDESTROY).
SubclassWindow	Subclasses a window.
UnsubclassWindow	Restores a previously subclassed window.

Static Methods

GetWndClassInfo	Returns a static instance of CWndClassInfo , which manages the window class information.
WindowProc	Processes messages sent to the window.

Data Members

m_pfnSuperWindowProc	Points to the window class's original window procedure.

Remarks

You can use [CWindowImpl](#) to create a window or subclass an existing window. the [CWindowImpl](#) window procedure uses a message map to direct messages to the appropriate handlers.

[CWindowImpl::Create](#) creates a window based on the window class information that's managed by [CWndClassInfo](#). [CWindowImpl](#) contains the [DECLARE_WND_CLASS](#) macro, which means [CWndClassInfo](#) registers a new window class. If you want to superclass an existing window class, derive your class from [CWindowImpl](#) and include the [DECLARE_WND_SUPERCLASS](#) macro. In this case, [CWndClassInfo](#) registers a window class that's based on an existing class but uses [CWindowImpl::WindowProc](#). For example:

```
class ATL_NO_VTABLE CMyWindow :
    OtherInheritedClasses
    public CComControl<CMyWindow>
        // CComControl derives from CWindowImpl
{
public:
    // 1. The NULL parameter means ATL will generate a
    //    name for the superclass
    // 2. The "EDIT" parameter means the superclass is
    //    based on the standard Windows Edit box
    DECLARE_WND_SUPERCLASS(NULL, _T("EDIT"))

    // Remainder of class declaration omitted
```

NOTE

Because [CWndClassInfo](#) manages the information for just one window class, each window created through an instance of [CWindowImpl](#) is based on the same window class.

[CWindowImpl](#) also supports window subclassing. The [SubclassWindow](#) method attaches an existing window to the [CWindowImpl](#) object and changes the window procedure to [CWindowImpl::WindowProc](#). Each instance of [CWindowImpl](#) can subclass a different window.

NOTE

For any given [CWindowImpl](#) object, call either [Create](#) or [SubclassWindow](#). Don't invoke both methods on the same object.

In addition to [CWindowImpl](#), ATL provides [CContainedWindow](#) to create a window that's contained in another object.

The base class destructor (~ CWindowImplRoot) ensures that the window is gone before the object is destroyed.

CWindowImpl derives from CWindowImplBaseT, which derives from CWindowImplRoot, which derives from TBase and CMessageMap.

FOR MORE INFORMATION ABOUT	SEE
Creating controls	ATL Tutorial
Using windows in ATL	ATL Window Classes
ATL Project Wizard	Creating an ATL Project

Inheritance Hierarchy

[CMessageMap](#)



Requirements

Header: atlwin.h

CWindowImpl::Create

Creates a window based on a new window class.

```
HWND Create(  
    HWND hWndParent,  
    _U_RECT rect = NULL,  
    LPCTSTR szWindowName = NULL,  
    DWORD dwStyle = 0,  
    DWORD dwExStyle = 0,  
    _U_MENUorID MenuOrID = 0U,  
    LPVOID lpCreateParam = NULL);
```

Parameters

hWndParent

[in] The handle to the parent or owner window.

rect

[in] A [RECT](#) structure specifying the position of the window. The `RECT` can be passed by pointer or by reference.

szWindowName

[in] Specifies the name of the window. The default value is NULL.

dwStyle

[in] The style of the window. This value is combined with the style provided by the traits class for the window. The default value gives the traits class full control over the style. For a list of possible values, see [CreateWindow](#) in the Windows SDK.

dwExStyle

[in] The extended window style. This value is combined with the style provided by the traits class for the window. The default value gives the traits class full control over the style. For a list of possible values, see [CreateWindowEx](#) in the Windows SDK.

MenuOrID

[in] For a child window, the window identifier. For a top-level window, a menu handle for the window. The default value is **0U**.

lpCreateParam

[in] A pointer to window-creation data. For a full description, see the description for the final parameter to [CreateWindowEx](#).

Return Value

If successful, the handle to the newly created window. Otherwise, NULL.

Remarks

`Create` first registers the window class if it has not yet been registered. The newly created window is automatically attached to the `CWindowImpl` object.

NOTE

Do not call `Create` if you have already called [SubclassWindow](#).

To use a window class that is based on an existing window class, derive your class from `CWindowImpl` and include the `DECLARE_WND_SUPERCLASS` macro. The existing window class's window procedure is saved in `m_pfnSuperWindowProc`. For more information, see the [CWindowImpl](#) overview.

NOTE

If 0 is used as the value for the *MenuOrID* parameter, it must be specified as 0U (the default value) to avoid a compiler error.

CWindowImpl::DefWindowProc

Called by [WindowProc](#) to process messages not handled by the message map.

```
LRESULT DefWindowProc(  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam);  
  
LRESULT DefWindowProc();
```

Parameters

uMsg

[in] The message sent to the window.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

Return Value

The result of the message processing.

Remarks

By default, `DefWindowProc` calls the [CallWindowProc](#) Win32 function to send the message information to the window procedure specified in `m_pfnSuperWindowProc`.

The function with no parameters automatically retrieves the needed parameters from the current message.

CWindowImpl::GetCurrentMessage

Returns the current message, packaged in the `MSG` structure.

```
const MSG* GetCurrentMessage();
```

Return Value

The current message.

CWindowImpl::GetWindowProc

Returns `WindowProc`, the current window procedure.

```
virtual WNDPROC GetWindowProc();
```

Return Value

The current window procedure.

Remarks

Override this method to replace the window procedure with your own.

CWindowImpl::GetWndClassInfo

Called by [Create](#) to access the window class information.

```
static CWndClassInfo& GetWndClassInfo();
```

Return Value

A static instance of [CWndClassInfo](#).

Remarks

By default, `CWindowImpl` obtains this method through the [DECLARE_WND_CLASS](#) macro, which specifies a new window class.

To superclass an existing window class, derive your class from `CWindowImpl` and include the [DECLARE_WND_SUPERCLASS](#) macro to override `GetWndClassInfo`. For more information, see the [CWindowImpl](#) overview.

Besides using the `DECLARE_WND_CLASS` and `DECLARE_WND_SUPERCLASS` macros, you can override `GetWndClassInfo` with your own implementation.

CWindowImpl::m_pfnSuperWindowProc

Depending on the window, points to one of the following window procedures.

```
WNDPROC m_pfnSuperWindowProc;
```

Remarks

TYPE OF WINDOW	WINDOW PROCEDURE
A window based on a new window class, specified through the DECLARE_WND_CLASS macro.	The DefWindowProc Win32 function.
A window based on a window class that modifies an existing class, specified through the DECLARE_WND_SUPERCLASS macro.	The existing window class's window procedure.
A subclassed window.	The subclassed window's original window procedure.

[CWindowImpl::DefWindowProc](#) sends message information to the window procedure saved in `m_pfnSuperWindowProc`.

CWindowImpl::OnFinalMessage

Called after receiving the last message (typically WM_NCDESTROY).

```
virtual void OnFinalMessage(HWND hWnd);
```

Parameters

hWnd

[in] A handle to the window being destroyed.

Remarks

The default implementation of `OnFinalMessage` does nothing, but you can override this function to handle cleanup before destroying a window. If you want to automatically delete your object upon the window destruction, you can call **delete this**; in this function.

CWindowImpl::SubclassWindow

Subclasses the window identified by *hWnd* and attaches it to the `CWindowImpl` object.

```
BOOL SubclassWindow(HWND hWnd);
```

Parameters

hWnd

[in] The handle to the window being subclassed.

Return Value

TRUE if the window is successfully subclassed; otherwise, FALSE.

Remarks

The subclassed window now uses [CWindowImpl::WindowProc](#). The original window procedure is saved in [m_pfnSuperWindowProc](#).

NOTE

Do not call `SubclassWindow` if you have already called [Create](#).

CWindowImpl::UnsubclassWindow

Detaches the subclassed window from the `CWindowImpl` object and restores the original window procedure, saved in `m_pfnSuperWindowProc`.

```
HWND UnsubclassWindow();
```

Return Value

The handle to the window previously subclassed.

CWindowImpl::WindowProc

This static function implements the window procedure.

```
static LRESULT CALLBACK WindowProc(  
    HWND hWnd,  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam);
```

Parameters

hWnd

[in] The handle to the window.

uMsg

[in] The message sent to the window.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

Return Value

The result of the message processing.

Remarks

`WindowProc` uses the default message map (declared with [BEGIN_MSG_MAP](#)) to direct messages to the appropriate handlers. If necessary, `WindowProc` calls [DefWindowProc](#) for additional message processing. If the final message is not handled, `WindowProc` does the following:

- Performs unsubclassing if the window was unsubclassed.
- Clears `m_hWnd`.
- Calls [OnFinalMessage](#) before the window is destroyed.

You can override `WindowProc` to provide a different mechanism for handling messages.

See also

BEGIN_MSG_MAP

CComControl Class

Class Overview

CWinTraits Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides a method for standardizing the styles used when creating a window object.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <DWORD t_dwStyle = 0, DWORD t_dwExStyle = 0> class CWinTraits
```

Parameters

t_dwStyle

Default standard window styles.

t_dwExStyle

Default extended window styles.

Members

Public Methods

NAME	DESCRIPTION
CWinTraits::GetWndExStyle	(Static) Retrieves the extended styles for the <code>CWinTraits</code> object.
CWinTraits::GetWndStyle	(Static) Retrieves the standard styles for the <code>CWinTraits</code> object.

Remarks

This [window traits](#) class provides a simple method for standardizing the styles used for the creation of an ATL window object. Use a specialization of this class as a template parameter to [CWindowImpl](#) or another of ATL's window classes to specify the default standard and extended styles used for instances of that window class.

Use this template when you want to provide default window styles that will be used only when no other styles are specified in the call to [CWindowImpl::Create](#).

ATL provides three predefined specializations of this template for commonly used combinations of window styles:

- `CControlWinTraits`

Designed for a standard control window. The following standard styles are used: `WS_CHILD`, `WS_VISIBLE`, `WS_CLIPCHILDREN`, and `WS_CLIPSIBLINGS`. There are no extended styles.

- `CFrameWinTraits`

Designed for a standard frame window. The standard styles used include: `WS_OVERLAPPEDWINDOW`,

WS_CLIPCHILDREN, and WS_CLIPSIBLINGS. The extended styles used include: WS_EX_APPWINDOW and WS_EX_WINDOWEDGE.

- `CMDIChildWinTraits`

Designed for a standard MDI child window. The standard styles used include:

WS_OVERLAPPEDWINDOW, WS_CHILD, WS_VISIBLE, WS_CLIPCHILDREN, and WS_CLIPSIBLINGS.

The extended styles used include: WS_EX_MDICHILD.

If you want to ensure that certain styles are set for all instances of the window class while permitting other styles to be set on a per-instance basis, use [CWinTraitsOR](#) instead.

Requirements

Header: `atlwin.h`

CWinTraits::GetWndStyle

Call this function to retrieve the standard styles of the `CWinTraits` object.

```
static DWORD GetWndStyle(DWORD dwStyle);
```

Parameters

dwStyle

Standard styles used for creation of a window. If *dwStyle* is 0, the template style values (`t_dwStyle`) are returned. If *dwStyle* is nonzero, *dwStyle* is returned.

Return Value

The standard window styles of the object.

CWinTraits::GetWndExStyle

Call this function to retrieve the extended styles of the `CWinTraits` object.

```
static DWORD GetWndExStyle(DWORD dwExStyle);
```

Parameters

dwExStyle

Extended styles used for creation of a window. If *dwExStyle* is 0, the template style values (`t_dwExStyle`) are returned. If *dwExStyle* is nonzero, *dwExStyle* is returned.

Return Value

The extended window styles of the object.

See also

[Class Overview](#)

[Understanding Window Traits](#)

CWinTraitsOR Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides a method for standardizing the styles used when creating a window object.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <DWORD t_dwStyle = 0,
         DWORD t_dwExStyle = 0,
         class TWinTraits = CControlWinTraits>
class CWinTraitsOR
```

Parameters

t_dwStyle

Default window styles.

t_dwExStyle

Default extended window styles.

Members

Public Methods

NAME	DESCRIPTION
CWinTraitsOR::GetWndExStyle	Retrieves the extended styles for the <code>CWinTraitsOR</code> object.
CWinTraitsOR::GetWndStyle	Retrieves the standard styles for the <code>CWinTraitsOR</code> object.

Remarks

This [window traits](#) class provides a simple method for standardizing the styles used for the creation of an ATL window object. Use a specialization of this class as a template parameter to [CWindowImpl](#) or another of ATL's window classes to specify the minimum set of standard and extended styles to be used for instances of that window class.

Use a specialization of this template if you want to ensure that certain styles are set for all instances of the window class while permitting other styles to be set on a per-instance basis in the call to [CWindowImpl::Create](#).

If you want to provide default window styles that will be used only when no other styles are specified in the call to `CWindowImpl::Create`, use [CWinTraits](#) instead.

Requirements

Header: atlwin.h

CWinTraitsOR::GetWndStyle

Call this function to retrieve a combination (using the logical OR operator) of the standard styles of the `CWinTraits` object and the default styles specified by `t_dwStyle`.

```
static DWORD GetWndStyle(DWORD dwStyle);
```

Parameters

dwStyle

Styles used for creation of a window.

Return Value

A combination of styles that are passed in *dwStyle* and the default ones specified by `t_dwStyle`, using the logical OR operator.

CWinTraitsOR::GetWndExStyle

Call this function to retrieve a combination (using the logical OR operator) of the extended styles of the `CWinTraits` object and the default styles specified by `t_dwStyle`.

```
static DWORD GetWndExStyle(DWORD dwExStyle);
```

Parameters

dwExStyle

Extended styles used for creation of a window.

Return Value

A combination of extended styles that are passed in *dwExStyle* and default ones specified by `t_dwExStyle`, using the logical OR operator

See also

[Class Overview](#)

[Understanding Window Traits](#)

CWndClassInfo Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

This class provides methods for registering information for a window class.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class CWndClassInfo
```

Members

Public Methods

Register	Registers the window class.

Data Members

m_atom	Uniquely identifies the registered window class.
m_bSystemCursor	Specifies whether the cursor resource refers to a system cursor or to a cursor contained in a module resource.
m_lpszCursorID	Specifies the name of the cursor resource.
m_lpszOrigName	Contains the name of an existing window class.
m_szAutoName	Holds an ATL-generated name of the window class.
m_wc	Maintains window class information in a <code>WNDCLASSEX</code> structure.
pWndProc	Points to the window procedure of an existing window class.

Remarks

`CWndClassInfo` manages the information of a window class. You typically use `CWndClassInfo` through one of three macros, `DECLARE_WND_CLASS`, `DECLARE_WND_CLASS_EX`, or `DECLARE_WND_SUPERCLASS`, as described in the following table:

MACRO	DESCRIPTION
DECLARE_WND_CLASS	<code>CWndClassInfo</code> registers information for a new window class.
DECLARE_WND_CLASS_EX	<code>CWndClassInfo</code> registers information for a new window class, including the class parameters.
DECLARE_WND_SUPERCLASS	<code>CWndClassInfo</code> registers information for a window class that is based on an existing class but uses a different window procedure. This technique is called superclassing.

By default, [CWindowImpl](#) includes the `DECLARE_WND_CLASS` macro to create a window based on a new window class. `DECLARE_WND_CLASS` provides default styles and background color for the control. If you want to specify the style and background color yourself, derive your class from `CWindowImpl` and include the `DECLARE_WND_CLASS_EX` macro in your class definition.

If you want to create a window based on an existing window class, derive your class from `CWindowImpl` and include the `DECLARE_WND_SUPERCLASS` macro in your class definition. For example:

```
class ATL_NO_VTABLE CMyWindow :
    OtherInheritedClasses
public CComControl<CMyWindow>
    // CComControl derives from CWindowImpl
{
public:
    // 1. The NULL parameter means ATL will generate a
    //    name for the superclass
    // 2. The "EDIT" parameter means the superclass is
    //    based on the standard Windows Edit box
    DECLARE_WND_SUPERCLASS(NULL, _T("EDIT"))

    // Remainder of class declaration omitted
```

For more information about window classes, see [Window Classes](#) in the Windows SDK.

For more information about using windows in ATL, see the article [ATL Window Classes](#).

Requirements

Header: `atlwin.h`

`CWndClassInfo::m_atom`

Contains the unique identifier for the registered window class.

```
ATOM m_atom;
```

`CWndClassInfo::m_bSystemCursor`

If `TRUE`, the system cursor resource will be loaded when the window class is registered.

```
BOOL m_bSystemCursor;
```

Remarks

Otherwise, the cursor resource contained in your module will be loaded.

`CWndClassInfo` uses `m_bSystemCursor` only when the `DECLARE_WND_CLASS` (the default in `CWindowImpl`) or the `DECLARE_WND_CLASS_EX` macro is specified. In this case, `m_bSystemCursor` is initialized to `TRUE`. For more information, see the [CWndClassInfo](#) overview.

CWndClassInfo::m_lpszCursorID

Specifies either the name of the cursor resource or the resource identifier in the low-order word and zero in the high-order word.

```
LPCTSTR m_lpszCursorID;
```

Remarks

When the window class is registered, the handle to the cursor identified by `m_lpszCursorID` is retrieved and stored by `m_wc`.

`CWndClassInfo` uses `m_lpszCursorID` only when the `DECLARE_WND_CLASS` (the default in `CWindowImpl`) or the `DECLARE_WND_CLASS_EX` macro is specified. In this case, `m_lpszCursorID` is initialized to `IDC_ARROW`. For more information, see the [CWndClassInfo](#) overview.

CWndClassInfo::m_lpszOrigName

Contains the name of an existing window class.

```
LPCTSTR m_lpszOrigName;
```

Remarks

`CWndClassInfo` uses `m_lpszOrigName` only when you include the `DECLARE_WND_SUPERCLASS` macro in your class definition. In this case, `CWndClassInfo` registers a window class based on the class named by `m_lpszOrigName`. For more information, see the [CWndClassInfo](#) overview.

CWndClassInfo::m_szAutoName

Holds the name of the window class.

```
TCHAR m_szAutoName[13];
```

Remarks

`CWndClassInfo` uses `m_szAutoName` only if `NULL` is passed for the `WndClassName` parameter to `DECLARE_WND_CLASS`, the `DECLARE_WND_CLASS_EX` or `DECLARE_WND_SUPERCLASS`. ATL will construct a name when the window class is registered.

CWndClassInfo::m_wc

Maintains the window class information in a `WNDCLASSEX` structure.

```
WNDCLASSEX m_wc;
```

Remarks

If you have specified the [DECLARE_WND_CLASS](#) (the default in [CWindowImpl](#)) or the [DECLARE_WND_CLASS_EX](#) macro, `m_wc` contains information about a new window class.

If you have specified the [DECLARE_WND_SUPERCLASS](#) macro, `m_wc` contains information about a superclass — a window class that is based on an existing class but uses a different window procedure. [m_lpszOrigName](#) and [pWndProc](#) save the existing window class's name and window procedure, respectively.

CWndClassInfo::pWndProc

Points to the window procedure of an existing window class.

```
WNDPROC pWndProc;
```

Remarks

`CWndClassInfo` uses `pWndProc` only when you include the [DECLARE_WND_SUPERCLASS](#) macro in your class definition. In this case, `CWndClassInfo` registers a window class that is based on an existing class but uses a different window procedure. The existing window class's window procedure is saved in `pWndProc`. For more information, see the [CWndClassInfo](#) overview.

CWndClassInfo::Register

Called by [CWindowImpl::Create](#) to register the window class if it has not yet been registered.

```
ATOM Register(WNDPROC* pProc);
```

Parameters

pProc

[out] Specifies the original window procedure of an existing window class.

Return Value

If successful, an atom that uniquely identifies the window class being registered. Otherwise, 0.

Remarks

If you have specified the [DECLARE_WND_CLASS](#) (the default in [CWindowImpl](#)) or the [DECLARE_WND_CLASS_EX](#) macro, `Register` registers a new window class. In this case, the *pProc* parameter is not used.

If you have specified the [DECLARE_WND_SUPERCLASS](#) macro, `Register` registers a superclass — a window class that is based on an existing class but uses a different window procedure. The existing window class's window procedure is returned in *pProc*.

See also

[CComControl Class](#)
[Class Overview](#)

CWorkerThread Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

This class creates a worker thread or uses an existing one, waits on one or more kernel object handles, and executes a specified client function when one of the handles is signaled.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class ThreadTraits = DefaultThreadTraits>
class CWorkerThread
```

Parameters

ThreadTraits

The class providing the thread creation function, such as [CRTThreadTraits](#) or [Win32ThreadTraits](#).

Members

Protected Structures

NAME	DESCRIPTION
<code>WorkerClientEntry</code>	

Public Constructors

NAME	DESCRIPTION
CWorkerThread::CWorkerThread	The constructor for the worker thread.
CWorkerThread::~~CWorkerThread	The destructor for the worker thread.

Public Methods

NAME	DESCRIPTION
CWorkerThread::AddHandle	Call this method to add a waitable object's handle to the list maintained by the worker thread.
CWorkerThread::AddTimer	Call this method to add a periodic waitable timer to the list maintained by the worker thread.
CWorkerThread::GetThreadHandle	Call this method to get the thread handle of the worker thread.
CWorkerThread::GetThreadId	Call this method to get the thread ID of the worker thread.

NAME	DESCRIPTION
CWorkerThread::Initialize	Call this method to initialize the worker thread.
CWorkerThread::RemoveHandle	Call this method to remove a handle from the list of waitable objects.
CWorkerThread::Shutdown	Call this method to shut down the worker thread.

Remarks

To use CWorkerThread

1. Create an instance of this class.
2. Call [CWorkerThread::Initialize](#).
3. Call [CWorkerThread::AddHandle](#) with the handle of a kernel object and a pointer to an implementation of [IWorkerThreadClient](#).

- or -

 Call [CWorkerThread::AddTimer](#) with a pointer to an implementation of [IWorkerThreadClient](#).
4. Implement [IWorkerThreadClient::Execute](#) to take some action when the handle or timer is signaled.
5. To remove an object from the list of waitable objects, call [CWorkerThread::RemoveHandle](#).
6. To terminate the thread, call [CWorkerThread::Shutdown](#).

Requirements

Header: atlutil.h

CWorkerThread::AddHandle

Call this method to add a waitable object's handle to the list maintained by the worker thread.

```
HRESULT AddHandle(
    HANDLE hObject,
    IWorkerThreadClient* pClient,
    DWORD_PTR dwParam) throw();
```

Parameters

hObject

The handle to a waitable object.

pClient

The pointer to the [IWorkerThreadClient](#) interface on the object to be called when the handle is signaled.

dwParam

The parameter to be passed to [IWorkerThreadClient::Execute](#) when the handle is signaled.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

[IWorkerThreadClient::Execute](#) will be called through *pClient* when the handle, *hObject*, is signaled.

CWorkerThread::AddTimer

Call this method to add a periodic waitable timer to the list maintained by the worker thread.

```
HRESULT AddTimer(  
    DWORD dwInterval,  
    IWorkerThreadClient* pClient,  
    DWORD_PTR dwParam,  
    HANDLE* phTimer) throw();
```

Parameters

dwInterval

Specifies the period of the timer in milliseconds.

pClient

The pointer to the [IWorkerThreadClient](#) interface on the object to be called when the handle is signaled.

dwParam

The parameter to be passed to [IWorkerThreadClient::Execute](#) when the handle is signaled.

phTimer

[out] Address of the HANDLE variable that, on success, receives the handle to the newly created timer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

[IWorkerThreadClient::Execute](#) will be called through *pClient* when the timer is signaled.

Pass the timer handle from *phTimer* to [CWorkerThread::RemoveHandle](#) to close the timer.

CWorkerThread::CWorkerThread

The constructor.

```
CWorkerThread() throw();
```

CWorkerThread::~~CWorkerThread

The destructor.

```
~CWorkerThread() throw();
```

Remarks

Calls [CWorkerThread::Shutdown](#).

CWorkerThread::GetThreadHandle

Call this method to get the thread handle of the worker thread.

```
HANDLE GetThreadHandle() throw();
```

Return Value

Returns the thread handle or NULL if the worker thread has not been initialized.

CWorkerThread::GetThreadId

Call this method to get the thread ID of the worker thread.

```
DWORD GetThreadId() throw();
```

Return Value

Returns the thread ID or NULL if the worker thread has not been initialized.

CWorkerThread::Initialize

Call this method to initialize the worker thread.

```
HRESULT Initialize() throw();  
  
HRESULT Initialize(CWorkerThread<ThreadTraits>* pThread) throw();
```

Parameters

pThread

An existing worker thread.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method should be called to initialize the object after creation or after a call to [CWorkerThread::Shutdown](#).

To have two or more `CWorkerThread` objects use the same worker thread, initialize one of them without passing any arguments then pass a pointer to that object to the `Initialize` methods of the others. The objects initialized using the pointer should be shut down before the object used to initialize them.

See [CWorkerThread::Shutdown](#) for information on how that method's behavior changes when initialized using a pointer to an existing object.

CWorkerThread::RemoveHandle

Call this method to remove a handle from the list of waitable objects.

```
HRESULT RemoveHandle(HANDLE hObject) throw();
```

Parameters

hObject

The handle to remove.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

When the handle is removed [IWorkerThreadClient::CloseHandle](#) will be called on the associated object that was

passed to [AddHandle](#). If this call fails, `CWorkerThread` will call the Windows [CloseHandle](#) function on the handle.

CWorkerThread::Shutdown

Call this method to shut down the worker thread.

```
HRESULT Shutdown(DWORD dwWait = ATL_WORKER_THREAD_WAIT) throw();
```

Parameters

dwWait

The time in milliseconds to wait for the worker thread to shut down. `ATL_WORKER_THREAD_WAIT` defaults to 10 seconds. If necessary, you can define your own value for this symbol before including `atlutil.h`.

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure, such as if the timeout value, *dwWait*, is exceeded.

Remarks

To reuse the object, call [CWorkerThread::Initialize](#) after calling this method.

Note that calling `Shutdown` on an object initialized with a pointer to another `CWorkerThread` object has no effect and always returns `S_OK`.

See also

[DefaultThreadTraits](#)

[Classes](#)

[Multithreading: Creating Worker Threads](#)

[IWorkerThreadClient Interface](#)

IAtlAutoThreadModule Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class represents an interface to a `CreateInstance` method.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
__interface IAtlAutoThreadModule
```

Remarks

The class `CAtlAutoThreadModuleT` derives from `IAtlAutoThreadModule`, using it to provide code for creating an object and retrieving an interface pointer.

Requirements

Header: atabase.h

See also

[Class Overview](#)

IAtlMemMgr Class

3/4/2019 • 13 minutes to read • [Edit Online](#)

This class represents the interface to a memory manager.

Syntax

```
__interface __declspec(uuid("654F7EF5-CFDF-4df9-A450-6C6A13C622C0")) IAtlMemMgr
```

Members

Methods

Allocate	Call this method to allocate a block of memory.
Free	Call this method to free a block of memory.
GetSize	Call this method to retrieve the size of an allocated memory block.
Reallocate	Call this method to reallocate a block of memory.

Remarks

This interface is implemented by [CComHeap](#), [CCRTHeap](#), [CLocalHeap](#), [CGlobalHeap](#), or [CWin32Heap](#).

NOTE

The local and global heap functions are slower than other memory management functions, and do not provide as many features. Therefore, new applications should use the [heap functions](#). These are available in the [CWin32Heap](#) class.

Example

```

// Demonstrate IAtlMemMgr using the five possible
// memory function implementation classes.

HRESULT MemoryManagerDemonstration(IAtlMemMgr& MemoryManager) throw()
{
    // The IAtlMemMgr interface guarantees not to throw exceptions
    // so we can make the same guarantee for this function
    // without adding exception handling code.

    // A variable which will point to some allocated memory.
    void* pMemory = NULL;

    const size_t BytesInChunk = 1024;

    // Allocate a chunk of memory
    pMemory = MemoryManager.Allocate(BytesInChunk);

    // Confirm the validity of the allocated memory
    if (pMemory == NULL)
        return E_OUTOFMEMORY;

    // Confirm the size of the allocated memory
    ATLASSERT(MemoryManager.GetSize(pMemory) == BytesInChunk);

    // Increase the size of the allocated memory
    pMemory = MemoryManager.Reallocate(pMemory, BytesInChunk * 2);

    // Confirm the validity of the allocated memory
    if (pMemory == NULL)
        return E_OUTOFMEMORY;

    // Confirm the size of the reallocated memory
    ATLASSERT(MemoryManager.GetSize(pMemory) == BytesInChunk * 2);

    // Free the allocated memory
    MemoryManager.Free(pMemory);

    return S_OK;
}

int DoMemoryManagerDemonstration()
{
    CComHeap heapCom;
    CCRTHeap heapCrt;
    CLocalHeap heapLocal;
    CGlobalHeap heapGlobal;
    // It is necessary to provide extra information
    // to the constructor when using CWin32Heap
    CWin32Heap heapWin32(NULL, 4096);

    ATLASSERT(S_OK==MemoryManagerDemonstration(heapCom));
    ATLASSERT(S_OK==MemoryManagerDemonstration(heapCrt));
    ATLASSERT(S_OK==MemoryManagerDemonstration(heapLocal));
    ATLASSERT(S_OK==MemoryManagerDemonstration(heapGlobal));
    ATLASSERT(S_OK==MemoryManagerDemonstration(heapWin32));

    return 0;
}

```

Requirements

Header: atlmem.h

IAtlMemMgr::Allocate

Call this method to allocate a block of memory.

```
void* Allocate(size_t nBytes) throw();
```

Parameters

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call [IAtlMemMgr::Free](#) or [IAtlMemMgr::Reallocate](#) to free the memory allocated by this method.

Example

For an example, see the [IAtlMemMgr Overview](#).

IAtlMemMgr::Free

Call this method to free a block of memory.

```
void Free(void* p) throw();
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

Remarks

Use this method to free memory obtained by [IAtlMemMgr::Allocate](#) or [IAtlMemMgr::Reallocate](#).

Example

For an example, see the [IAtlMemMgr Overview](#).

IAtlMemMgr::GetSize

Call this method to retrieve the size of an allocated memory block.

```
size_t GetSize(void* p) throw();
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

Return Value

Returns the size of the memory block in bytes.

Example

For an example, see the [IAtlMemMgr Overview](#).

IAtlMemMgr::Reallocate

Call this method to reallocate memory allocated by this memory manager.

```
void* Reallocate(void* p, size_t nBytes) throw();
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call [IAtlMemMgr::Free](#) or [IAtlMemMgr::Reallocate](#) to free the memory allocated by this method.

Conceptually this method frees the existing memory and allocates a new memory block. In reality, the existing memory may be extended or otherwise reused.

Example

For an example, see the [IAtlMemMgr Overview](#).

IAxWinAmbientDispatch::get_AllowContextMenu

The `AllowContextMenu` property specifies whether the hosted control is allowed to display its own context menu.

```
STDMETHOD(get_AllowContextMenu)(VARIANT_BOOL* pbAllowContextMenu);
```

Parameters

pbAllowContextMenu

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

IAxWinAmbientDispatch::get_AllowShowUI

The `AllowShowUI` property specifies whether the hosted control is allowed to display its own user interface.

```
STDMETHOD(get_AllowShowUI)(VARIANT_BOOL* pbAllowShowUI);
```

Parameters

pbAllowShowUI

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_FALSE as the default value of this property.

IAxWinAmbientDispatch::get_AllowWindowlessActivation

The `AllowWindowlessActivation` property specifies whether the container will allow windowless activation.

```
STDMETHOD(get_AllowWindowlessActivation)(VARIANT_BOOL* pbAllowWindowless);
```

Parameters

pbAllowWindowless

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

IAxWinAmbientDispatch::get_BackColor

The `BackColor` property specifies the ambient background color of the container.

```
STDMETHOD(get_BackColor)(OLE_COLOR* pclrBackground);
```

Parameters

pclrBackground

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses COLOR_BTNFACE or COLOR_WINDOW as the default value of this property (depending on whether the parent of the host window is a dialog or not).

IAxWinAmbientDispatch::get_DisplayAsDefault

`DisplayAsDefault` is an ambient property that allows a control to find out if it is the default control.

```
STDMETHOD(get_DisplayAsDefault)(VARIANT_BOOL* pbDisplayAsDefault);
```

Parameters

pbDisplayAsDefault

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_FALSE as the default value of this property.

IAxWinAmbientDispatch::get_DocHostDoubleClickFlags

The `DocHostDoubleClickFlags` property specifies the operation that should take place in response to a double-click.

```
STDMETHOD(get_DocHostDoubleClickFlags)(DWORD* pdwDocHostDoubleClickFlags);
```

Parameters

pdwDocHostDoubleClickFlags

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses DOCHOSTUIDBLCLK_DEFAULT as the default value of this property.

IAxWinAmbientDispatch::get_DocHostFlags

The `DocHostFlags` property specifies the user interface capabilities of the host object.

```
STDMETHOD(get_DocHostFlags)(DWORD* pdwDocHostFlags);
```

Parameters

pdwDocHostFlags

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses DOCHOSTUIFLAG_NO3DBORDER as the default value of this property.

IAxWinAmbientDispatch::get_Font

The `Font` property specifies the ambient font of the container.

```
STDMETHOD(get_Font)(IFontDisp** pFont);
```

Parameters

pFont

[out] The address of an `IFontDisp` interface pointer used to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses the default GUI font or the system font as the default value of this property.

IAxWinAmbientDispatch::get_ForeColor

The `ForeColor` property specifies the ambient foreground color of the container.

```
STDMETHOD(get_ForeColor)(OLE_COLOR* pClrForeground);
```

Parameters

pClrForeground

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses the system window text color as the default value of this property.

IAxWinAmbientDispatch::get_LocaleID

The `LocaleID` property specifies the ambient locale ID of the container.

```
STDMETHOD(get_LocaleID)(LCID* plcidLocaleID);
```

Parameters

plcidLocaleID

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses the user's default locale as the default value of this property.

With this method you can discover the Ambient LocalID, that is, the LocaleID of the program your control is being used in. Once you know the LocaleID, you can call code to load locale-specific captions, error message text, and so forth from a resource file or satellite DLL.

IAxWinAmbientDispatch::get_MessageReflect

The `MessageReflect` ambient property specifies whether the container will reflect messages to the hosted control.

```
STDMETHOD(get_MessageReflect)(VARIANT_BOOL* pbMessageReflect);
```

Parameters

pbMessageReflect

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

IAxWinAmbientDispatch::get_OptionKeyPath

The `OptionKeyPath` property specifies the registry key path to user settings.

```
STDMETHOD(get_OptionKeyPath)(BSTR* pbstrOptionKeyPath);
```

Parameters

pbstrOptionKeyPath

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

IAxWinAmbientDispatch::get_ShowGrabHandles

The `ShowGrabHandles` ambient property allows the control to find out if it should draw itself with grab handles.

```
STDMETHOD(get_ShowGrabHandles)(VARIANT_BOOL* pbShowGrabHandles);
```

Parameters

pbShowGrabHandles

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation always returns VARIANT_FALSE as the value of this property.

IAxWinAmbientDispatch::get_ShowHatching

The `ShowHatching` ambient property allows the control to find out if it should draw itself hatched.

```
STDMETHOD(get_ShowHatching)(VARIANT_BOOL* pbShowHatching);
```

Parameters

pbShowHatching

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation always returns VARIANT_FALSE as the value of this property.

IAxWinAmbientDispatch::get_UserMode

The `UserMode` property specifies the ambient user mode of the container.

```
STDMETHOD(get_UserMode)(VARIANT_BOOL* pbUserMode);
```

Parameters

pbUserMode

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

IAxWinAmbientDispatch::put_AllowContextMenu

The `AllowContextMenu` property specifies whether the hosted control is allowed to display its own context menu.

```
STDMETHOD(put_AllowContextMenu)(VARIANT_BOOL bAllowContextMenu);
```

Parameters

bAllowContextMenu

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

IAxWinAmbientDispatch::put_AllowShowUI

The `AllowShowUI` property specifies whether the hosted control is allowed to display its own user interface.

```
STDMETHOD(put_AllowShowUI)(VARIANT_BOOL bAllowShowUI);
```

Parameters

bAllowShowUI

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_FALSE as the default value of this property.

IAxWinAmbientDispatch::put_AllowWindowlessActivation

The `AllowWindowlessActivation` property specifies whether the container will allow windowless activation.

```
STDMETHOD(put_AllowWindowlessActivation)(VARIANT_BOOL bAllowWindowless);
```

Parameters

bAllowWindowless

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

IAxWinAmbientDispatch::put_BackColor

The `BackColor` property specifies the ambient background color of the container.

```
STDMETHOD(put_BackColor)(OLE_COLOR clrBackground);
```

Parameters

clrBackground

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses COLOR_BTNFACE or COLOR_WINDOW as the default value of this property (depending on whether the parent of the host window is a dialog or not).

IAxWinAmbientDispatch::put_DisplayAsDefault

`DisplayAsDefault` is an ambient property that allows a control to find out if it is the default control.

```
STDMETHOD(put_DisplayAsDefault)(VARIANT_BOOL bDisplayAsDefault);
```

Parameters

bDisplayAsDefault

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_FALSE as the default value of this property.

IAxWinAmbientDispatch::put_DocHostDoubleClickFlags

The `DocHostDoubleClickFlags` property specifies the operation that should take place in response to a double-click.

```
STDMETHOD(put_DocHostDoubleClickFlags)(DWORD dwDocHostDoubleClickFlags);
```

Parameters

dwDocHostDoubleClickFlags

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses DOCHOSTUIDBLCLK_DEFAULT as the default value of this property.

IAxWinAmbientDispatch::put_DocHostFlags

The `DocHostFlags` property specifies the user interface capabilities of the host object.

```
STDMETHOD(put_DocHostFlags)(DWORD dwDocHostFlags);
```

Parameters

dwDocHostFlags

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses DOCHOSTUIFLAG_NO3DBORDER as the default value of this property.

IAxWinAmbientDispatch::put_Font

The `Font` property specifies the ambient font of the container.

```
STDMETHOD(put_Font)(IFontDisp* pFont);
```

Parameters

pFont

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses the default GUI font or the system font as the default value of this property.

IAxWinAmbientDispatch::put_ForeColor

The `ForeColor` property specifies the ambient foreground color of the container.

```
STDMETHOD(put_ForeColor)(OLE_COLOR clrForeground);
```

Parameters

clrForeground

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses the system window text color as the default value of this property.

IAxWinAmbientDispatch::put_LocaleID

The `LocaleID` property specifies the ambient locale ID of the container.

```
STDMETHOD(put_LocaleID)(LCID lcidLocaleID);
```

Parameters

lcidLocaleID

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses the user's default locale as the default value of this property.

IAxWinAmbientDispatch::put_MessageReflect

The `MessageReflect` ambient property specifies whether the container will reflect messages to the hosted control.

```
STDMETHOD(put_MessageReflect)(VARIANT_BOOL bMessageReflect);
```

Parameters

bMessageReflect

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

IAxWinAmbientDispatch::put_OptionKeyPath

The `OptionKeyPath` property specifies the registry key path to user settings.

```
STDMETHOD(put_OptionKeyPath)(BSTR bstrOptionKeyPath);
```

Parameters

bstrOptionKeyPath

[in] The new value of this property.

Return Value

A standard HRESULT value.

IAxWinAmbientDispatch::put_UserMode

The `UserMode` property specifies the ambient user mode of the container.

```
STDMETHOD(put_UserMode)(VARIANT_BOOL bUserMode);
```

Parameters

bUserMode

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

IAxWinAmbientDispatchEx::SetAmbientDispatch

This method is called to supplement the default ambient property interface with a user-defined interface.

```
virtual HRESULT STDMETHODCALLTYPE SetAmbientDispatch(IDispatch* pDispatch) = 0;
```

Parameters

pDispatch

Pointer to the new interface.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

When `SetAmbientDispatch` is called with a pointer to a new interface, this new interface will be used to invoke any properties or methods asked for by the hosted control — if those properties are not already provided by [IAxWinAmbientDispatch](#).

IAxWinHostWindow::AttachControl

Attaches an existing (and previously initialized) control to the host object using the window identified by *hWnd*.

```
STDMETHOD(AttachControl)(IUnknown* pUnkControl, HWND hWnd);
```

Parameters

pUnkControl

[in] A pointer to the `IUnknown` interface of the control to be attached to the host object.

hWnd

[in] A handle to the window to be used for hosting.

Return Value

A standard HRESULT value.

IAxWinHostWindow::CreateControl

Creates a control, initializes it, and hosts it in the window identified by *hWnd*.

```
STDMETHOD(CreateControl)(
    LPCOLESTR lpTricsData,
    HWND hWnd,
    IStream* pStream);
```

Parameters

lpTricsData

[in] A string identifying the control to create. Can be a CLSID (must include the braces), ProgID, URL, or raw HTML (prefixed by **MSHTML:**).

hWnd

[in] A handle to the window to be used for hosting.

pStream

[in] An interface pointer for a stream containing initialization data for the control. Can be NULL.

Return Value

A standard HRESULT value.

Remarks

This window will be subclassed by the host object exposing this interface so that messages can be reflected to the control and other container features will work.

Calling this method is equivalent to calling [IAxWinHostWindow::CreateControlEx](#).

To create a licensed ActiveX control, see [IAxWinHostWindowLic::CreateControlLic](#).

IAxWinHostWindow::CreateControlEx

Creates an ActiveX control, initializes it, and hosts it in the specified window, similar to [IAxWinHostWindow::CreateControl](#).

```
STDMETHOD(CreateControlEx)(
    LPCOLESTR lpszTricsData,
    HWND hWnd,
    IStream* pStream,
    IUnknown** ppUnk,
    REFIID riidAdvise,
    IUnknown* punkAdvise);
```

Parameters

lpTricsData

[in] A string identifying the control to create. Can be a CLSID (must include the braces), ProgID, URL, or raw HTML (prefixed with **MSHTML:**).

hWnd

[in] A handle to the window to be used for hosting.

pStream

[in] An interface pointer for a stream containing initialization data for the control. Can be NULL.

ppUnk

[out] The address of a pointer that will receive the `IUnknown` interface of the created control. Can be NULL.

riidAdvise

[in] The interface identifier of an outgoing interface on the contained object. Can be IID_NULL.

punkAdvise

[in] A pointer to the `IUnknown` interface of the sink object to be connected to the connection point on the contained object specified by `iidSink`.

Return Value

A standard HRESULT value.

Remarks

Unlike the `CreateControl` method, `CreateControlEx` also allows you to receive an interface pointer to the newly created control and set up an event sink to receive events fired by the control.

To create a licensed ActiveX control, see [IAxWinHostWindowLic::CreateControlLicEx](#).

IAxWinHostWindow::QueryControl

Returns the specified interface pointer provided by the hosted control.

```
STDMETHOD(QueryControl)(REFIID riid, void** ppvObject);
```

Parameters

riid

[in] The ID of an interface on the control being requested.

ppvObject

[out] The address of a pointer that will receive the specified interface of the created control.

Return Value

A standard HRESULT value.

IAxWinHostWindow::SetExternalDispatch

Sets the external dispinterface, which is available to contained controls through the [IDocHostUIHandlerDispatch::GetExternal](#) method.

```
STDMETHOD(SetExternalDispatch)(IDispatch* pDisp);
```

Parameters

pDisp

[in] A pointer to an `IDispatch` interface.

Return Value

A standard HRESULT value.

IAxWinHostWindow::SetExternalUIHandler

Call this function to set the external [IDocHostUIHandlerDispatch](#) interface for the `CAXWindow` object.

```
STDMETHOD(SetExternalUIHandler)(IDocHostUIHandlerDispatch* pDisp);
```

Parameters

pDisp

[in] A pointer to an `IDocHostUIHandlerDispatch` interface.

Return Value

A standard HRESULT value.

Remarks

This function is used by controls (such as the Web browser control) that query the host's site for the `IDocHostUIHandlerDispatch` interface.

IAxWinHostWindowLic::CreateControlLic

Creates a licensed control, initializes it, and hosts it in the window identified by `hWnd`.

```
STDMETHOD(CreateControlLic)(
    LPCOLESTR lpTricsData,
    HWND hWnd,
    IStream* pStream,
    BSTR bstrLic);
```

Parameters

bstrLic

[in] The BSTR that contains the license key for the control.

Remarks

See [IAxWinHostWindow::CreateControl](#) for a description of the remaining parameters and return value.

Calling this method is equivalent to calling [IAxWinHostWindowLic::CreateControlLicEx](#).

Example

See [Hosting ActiveX Controls Using ATL AXHost](#) for a sample that uses

```
IAxWinHostWindowLic::CreateControlLic .
```

IAxWinHostWindowLic::CreateControlLicEx

Creates a licensed ActiveX control, initializes it, and hosts it in the specified window, similar to [IAxWinHostWindow::CreateControl](#).

```
STDMETHOD(CreateControlLicEx)(
    LPCOLESTR lpszTricsData,
    HWND hWnd,
    IStream* pStream,
    IUnknown** ppUnk,
    REFIID riidAdvise,
    IUnknown* punkAdvise,
    BSTR bstrLic);
```

Parameters

bstrLic

[in] The BSTR that contains the license key for the control.

Remarks

See [IAxWinHostWindow::CreateControlEx](#) for a description of the remaining parameters and return value.

Example

See [Hosting ActiveX Controls Using ATL AXHost](#) for a sample that uses

```
IAxWinHostWindowLic::CreateControlLicEx .
```

See also

[Class Overview](#)

IAxWinAmbientDispatch Interface

3/5/2019 • 10 minutes to read • [Edit Online](#)

This interface provides methods for specifying characteristics of the hosted control or container.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
interface IAxWinAmbientDispatch : IDispatch
```

Members

Methods

get_AllowContextMenu	The AllowContextMenu property specifies whether the hosted control is allowed to display its own context menu.
get_AllowShowUI	The AllowShowUI property specifies whether the hosted control is allowed to display its own user interface.
get_AllowWindowlessActivation	The AllowWindowlessActivation property specifies whether the container will allow windowless activation.
get_BackColor	The BackColor property specifies the ambient background color of the container.
get_DisplayAsDefault	DisplayAsDefault is an ambient property that allows a control to find out if it is the default control.
get_DocHostDoubleClickFlags	The DocHostDoubleClickFlags property specifies the operation that should take place in response to a double-click.
get_DocHostFlags	The DocHostFlags property specifies the user interface capabilities of the host object.
get_Font	The Font property specifies the ambient font of the container.
get_ForeColor	The ForeColor property specifies the ambient foreground color of the container.
get_LocaleID	The LocaleID property specifies the ambient locale ID of the container.

get_MessageReflect	The <code>MessageReflect</code> ambient property specifies whether the container will reflect messages to the hosted control.
get_OptionKeyPath	The <code>OptionKeyPath</code> property specifies the registry key path to user settings.
get_ShowGrabHandles	The <code>ShowGrabHandles</code> ambient property allows the control to find out if it should draw itself with grab handles.
get_ShowHatching	The <code>ShowHatching</code> ambient property allows the control to find out if it should draw itself hatched.
get_UserMode	The <code>UserMode</code> property specifies the ambient user mode of the container.
put_AllowContextMenu	The <code>AllowContextMenu</code> property specifies whether the hosted control is allowed to display its own context menu.
put_AllowShowUI	The <code>AllowShowUI</code> property specifies whether the hosted control is allowed to display its own user interface.
put_AllowWindowlessActivation	The <code>AllowWindowlessActivation</code> property specifies whether the container will allow windowless activation.
put_BackColor	The <code>BackColor</code> property specifies the ambient background color of the container.
put_DisplayAsDefault	<code>DisplayAsDefault</code> is an ambient property that allows a control to find out if it is the default control.
put_DocHostDoubleClickFlags	The <code>DocHostDoubleClickFlags</code> property specifies the operation that should take place in response to a double-click.
put_DocHostFlags	The <code>DocHostFlags</code> property specifies the user interface capabilities of the host object.
put_Font	The <code>Font</code> property specifies the ambient font of the container.
put_ForeColor	The <code>ForeColor</code> property specifies the ambient foreground color of the container.
put_LocaleID	The <code>LocaleID</code> property specifies the ambient locale ID of the container.
put_MessageReflect	The <code>MessageReflect</code> ambient property specifies whether the container will reflect messages to the hosted control.
put_OptionKeyPath	The <code>OptionKeyPath</code> property specifies the registry key path to user settings.

<code>put_UserMode</code>	The <code>UserMode</code> property specifies the ambient user mode of the container.

Remarks

This interface is exposed by ATL's ActiveX control hosting objects. Call the methods on this interface to set the ambient properties available to the hosted control or to specify other aspects of the container's behavior. To supplement the properties provided by `IAxWinAmbientDispatch`, use [IAxWinAmbientDispatchEx](#).

`AxHost` will try to load type information about `IAxWinAmbientDispatch` and `IAxWinAmbientDispatchEx` from the typelib that contains the code.

If you are linking to ATL90.dll, **AXHost** will load the type information from the typelib in the DLL.

See [Hosting ActiveX Controls Using ATL AXHost](#) for more details.

Requirements

The definition of this interface is available in a number of forms, as shown in the table below.

DEFINITION TYPE	FILE
IDL	atliface.idl
Type Library	ATL.dll
C++	atliface.h (also included in ATLBase.h)

IAxWinAmbientDispatch::get_AllowContextMenu

The `AllowContextMenu` property specifies whether the hosted control is allowed to display its own context menu.

```
STDMETHOD(get_AllowContextMenu)(VARIANT_BOOL* pbAllowContextMenu);
```

Parameters

pbAllowContextMenu

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

IAxWinAmbientDispatch::get_AllowShowUI

The `AllowShowUI` property specifies whether the hosted control is allowed to display its own user interface.

```
STDMETHOD(get_AllowShowUI)(VARIANT_BOOL* pbAllowShowUI);
```

Parameters

pbAllowShowUI

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_FALSE as the default value of this property.

IAxWinAmbientDispatch::get_AllowWindowlessActivation

The `AllowWindowlessActivation` property specifies whether the container will allow windowless activation.

```
STDMETHOD(get_AllowWindowlessActivation)(VARIANT_BOOL* pbAllowWindowless);
```

Parameters

pbAllowWindowless

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

IAxWinAmbientDispatch::get_BackColor

The `BackColor` property specifies the ambient background color of the container.

```
STDMETHOD(get_BackColor)(OLE_COLOR* pClrBackground);
```

Parameters

pClrBackground

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses COLOR_BTNFACE or COLOR_WINDOW as the default value of this property (depending on whether the parent of the host window is a dialog or not).

IAxWinAmbientDispatch::get_DisplayAsDefault

`DisplayAsDefault` is an ambient property that allows a control to find out if it is the default control.

```
STDMETHOD(get_DisplayAsDefault)(VARIANT_BOOL* pbDisplayAsDefault);
```

Parameters

pbDisplayAsDefault

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_FALSE as the default value of this property.

IAxWinAmbientDispatch::get_DocHostDoubleClickFlags

The `DocHostDoubleClickFlags` property specifies the operation that should take place in response to a double-click.

```
STDMETHOD(get_DocHostDoubleClickFlags)(DWORD* pdwDocHostDoubleClickFlags);
```

Parameters

pdwDocHostDoubleClickFlags

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses DOCHOSTUIDBLCLK_DEFAULT as the default value of this property.

IAxWinAmbientDispatch::get_DocHostFlags

The `DocHostFlags` property specifies the user interface capabilities of the host object.

```
STDMETHOD(get_DocHostFlags)(DWORD* pdwDocHostFlags);
```

Parameters

pdwDocHostFlags

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses DOCHOSTUIFLAG_NO3DBORDER as the default value of this property.

IAxWinAmbientDispatch::get_Font

The `Font` property specifies the ambient font of the container.

```
STDMETHOD(get_Font)(IFontDisp** pFont);
```

Parameters

pFont

[out] The address of an `IFontDisp` interface pointer used to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses the default GUI font or the system font as the default value of this property.

IAxWinAmbientDispatch::get_ForeColor

The `ForeColor` property specifies the ambient foreground color of the container.

```
STDMETHOD(get_ForeColor)(OLE_COLOR* pClrForeground);
```

Parameters

pClrForeground

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses the system window text color as the default value of this property.

IAxWinAmbientDispatch::get_LocaleID

The `LocaleID` property specifies the ambient locale ID of the container.

```
STDMETHOD(get_LocaleID)(LCID* plcidLocaleID);
```

Parameters

plcidLocaleID

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses the user's default locale as the default value of this property.

With this method you can discover the Ambient LocalID, that is, the LocaleID of the program your control is being used in. Once you know the LocaleID, you can call code to load locale-specific captions, error message text, and so forth from a resource file or satellite DLL.

IAxWinAmbientDispatch::get_MessageReflect

The `MessageReflect` ambient property specifies whether the container will reflect messages to the hosted control.

```
STDMETHOD(get_MessageReflect)(VARIANT_BOOL* pbMessageReflect);
```

Parameters

pbMessageReflect

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

IAxWinAmbientDispatch::get_OptionKeyPath

The `OptionKeyPath` property specifies the registry key path to user settings.

```
STDMETHOD(get_OptionKeyPath)(BSTR* pbstrOptionKeyPath);
```

Parameters

pbstrOptionKeyPath

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

IAxWinAmbientDispatch::get_ShowGrabHandles

The `ShowGrabHandles` ambient property allows the control to find out if it should draw itself with grab handles.

```
STDMETHOD(get_ShowGrabHandles)(VARIANT_BOOL* pbShowGrabHandles);
```

Parameters

pbShowGrabHandles

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation always returns VARIANT_FALSE as the value of this property.

IAxWinAmbientDispatch::get_ShowHatching

The `ShowHatching` ambient property allows the control to find out if it should draw itself hatched.

```
STDMETHOD(get_ShowHatching)(VARIANT_BOOL* pbShowHatching);
```

Parameters

pbShowHatching

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation always returns VARIANT_FALSE as the value of this property.

IAxWinAmbientDispatch::get_UserMode

The `UserMode` property specifies the ambient user mode of the container.

```
STDMETHOD(get_UserMode)(VARIANT_BOOL* pbUserMode);
```

Parameters

pbUserMode

[out] The address of a variable to receive the current value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

IAxWinAmbientDispatch::put_AllowContextMenu

The `AllowContextMenu` property specifies whether the hosted control is allowed to display its own context menu.

```
STDMETHOD(put_AllowContextMenu)(VARIANT_BOOL bAllowContextMenu);
```

Parameters

bAllowContextMenu

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

IAxWinAmbientDispatch::put_AllowShowUI

The `AllowShowUI` property specifies whether the hosted control is allowed to display its own user interface.

```
STDMETHOD(put_AllowShowUI)(VARIANT_BOOL bAllowShowUI);
```

Parameters

bAllowShowUI

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_FALSE as the default value of this property.

IAxWinAmbientDispatch::put_AllowWindowlessActivation

The `AllowWindowlessActivation` property specifies whether the container will allow windowless activation.

```
STDMETHOD(put_AllowWindowlessActivation)(VARIANT_BOOL bAllowWindowless);
```

Parameters

bAllowWindowless

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

IAxWinAmbientDispatch::put_BackColor

The `BackColor` property specifies the ambient background color of the container.

```
STDMETHOD(put_BackColor)(OLE_COLOR clrBackground);
```

Parameters

clrBackground

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses COLOR_BTNFACE or COLOR_WINDOW as the default value of this property (depending on whether the parent of the host window is a dialog or not).

IAxWinAmbientDispatch::put_DisplayAsDefault

`DisplayAsDefault` is an ambient property that allows a control to find out if it is the default control.

```
STDMETHOD(put_DisplayAsDefault)(VARIANT_BOOL bDisplayAsDefault);
```

Parameters

bDisplayAsDefault

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_FALSE as the default value of this property.

IAxWinAmbientDispatch::put_DocHostDoubleClickFlags

The `DocHostDoubleClickFlags` property specifies the operation that should take place in response to a double-click.

```
STDMETHOD(put_DocHostDoubleClickFlags)(DWORD dwDocHostDoubleClickFlags);
```

Parameters

dwDocHostDoubleClickFlags

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses DOCHOSTUIDBLCLK_DEFAULT as the default value of this property.

IAxWinAmbientDispatch::put_DocHostFlags

The `DocHostFlags` property specifies the user interface capabilities of the host object.

```
STDMETHOD(put_DocHostFlags)(DWORD dwDocHostFlags);
```

Parameters

dwDocHostFlags

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses DOCHOSTUIFLAG_NO3DBORDER as the default value of this property.

IAxWinAmbientDispatch::put_Font

The `Font` property specifies the ambient font of the container.

```
STDMETHOD(put_Font)(IFontDisp* pFont);
```

Parameters

pFont

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses the default GUI font or the system font as the default value of this property.

IAxWinAmbientDispatch::put_ForeColor

The `ForeColor` property specifies the ambient foreground color of the container.

```
STDMETHOD(put_ForeColor)(OLE_COLOR clrForeground);
```

Parameters

clrForeground

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses the system window text color as the default value of this property.

IAxWinAmbientDispatch::put_LocaleID

The `LocaleID` property specifies the ambient locale ID of the container.

```
STDMETHOD(put_LocaleID)(LCID lcidLocaleID);
```

Parameters

lcidLocaleID

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses the user's default locale as the default value of this property.

IAxWinAmbientDispatch::put_MessageReflect

The `MessageReflect` ambient property specifies whether the container will reflect messages to the hosted control.

```
STDMETHOD(put_MessageReflect)(VARIANT_BOOL bMessageReflect);
```

Parameters

bMessageReflect

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

IAxWinAmbientDispatch::put_OptionKeyPath

The `OptionKeyPath` property specifies the registry key path to user settings.

```
STDMETHOD(put_OptionKeyPath)(BSTR bstrOptionKeyPath);
```

Parameters

bstrOptionKeyPath

[in] The new value of this property.

Return Value

A standard HRESULT value.

IAxWinAmbientDispatch::put_UserMode

The `UserMode` property specifies the ambient user mode of the container.

```
STDMETHOD(put_UserMode)(VARIANT_BOOL bUserMode);
```

Parameters

bUserMode

[in] The new value of this property.

Return Value

A standard HRESULT value.

Remarks

The ATL host object implementation uses VARIANT_TRUE as the default value of this property.

See also

[IAxWinAmbientDispatchEx Interface](#)

[IAxWinHostWindow Interface](#)

[CAxWindow::QueryHost](#)

[AtlAxGetHost](#)

IAxWinAmbientDispatchEx Interface

3/5/2019 • 2 minutes to read • [Edit Online](#)

This interface implements supplemental ambient properties for a hosted control.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
MIDL_INTERFACE("B2D0778B - AC99 - 4c58 - A5C8 - E7724E5316B5") IAxWinAmbientDispatchEx : public  
IAxWinAmbientDispatch
```

Members

Methods

SetAmbientDispatch	This method is called to supplement the default ambient property interface with a user-defined interface.

Remarks

Include this interface in ATL applications that are statically linked to ATL and host ActiveX Controls, especially ActiveX Controls that have Ambient Properties. Not including this interface will generate this assertion: "Did you forget to pass the LIBID to CComModule::Init"

This interface is exposed by ATL's ActiveX control hosting objects. Derived from [IAxWinAmbientDispatch](#), `IAxWinAmbientDispatchEx` adds a method that allows you to supplement the ambient property interface provided by ATL with one of your own.

[AxHost](#) will try to load type information about `IAxWinAmbientDispatch` and `IAxWinAmbientDispatchEx` from the type library that contains the code.

If you are linking to ATL90.dll, **AXHost** will load the type information from the type library in the DLL.

See [Hosting ActiveX Controls Using ATL AXHost](#) for more details.

Requirements

The definition of this interface is available in a number of forms, as shown in the following table.

DEFINITION TYPE	FILE
IDL	atliface.idl
Type Library	ATL.dll

DEFINITION TYPE	FILE
C++	atliface.h (also included in ATLBase.h)

IAxWinAmbientDispatchEx::SetAmbientDispatch

This method is called to supplement the default ambient property interface with a user-defined interface.

```
virtual HRESULT STDMETHODCALLTYPE SetAmbientDispatch(IDispatch* pDispatch) = 0;
```

Parameters

pDispatch

Pointer to the new interface.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

When `SetAmbientDispatch` is called with a pointer to a new interface, this new interface will be used to invoke any properties or methods asked for by the hosted control, if those properties are not already provided by [IAxWinAmbientDispatch](#).

See also

[IAxWinAmbientDispatch Interface](#)

IAxWinHostWindow Interface

3/4/2019 • 3 minutes to read • [Edit Online](#)

This interface provides methods for manipulating a control and its host object.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
interface IAxWinHostWindow : IUnknown
```

Members

Methods

AttachControl	Attaches an existing control to the host object.
CreateControl	Creates a control and attaches it to the host object.
CreateControlEx	Creates a control, attaches it to the host object, and optionally sets up an event handler.
QueryControl	Returns an interface pointer to the hosted control.
SetExternalDispatch	Sets the external <code>IDispatch</code> interface.
SetExternalUIHandler	Sets the external <code>IDocHostUIHandlerDispatch</code> interface.

Remarks

This interface is exposed by ATL's ActiveX control hosting objects. Call the methods on this interface to create and/or attach a control to the host object, to get an interface from a hosted control, or to set the external dispinterface or UI handler for use when hosting the Web browser.

Requirements

The definition of this interface is available as IDL or C++, as shown below.

DEFINITION TYPE	FILE
IDL	ATLIFace.idl
C++	ATLIFace.h (also included in ATLBase.h)

IAxWinHostWindow::AttachControl

Attaches an existing (and previously initialized) control to the host object using the window identified by *hWnd*.

```
STDMETHOD(AttachControl)(IUnknown* pUnkControl, HWND hWnd);
```

Parameters

pUnkControl

[in] A pointer to the `IUnknown` interface of the control to be attached to the host object.

hWnd

[in] A handle to the window to be used for hosting.

Return Value

A standard HRESULT value.

IAxWinHostWindow::CreateControl

Creates a control, initializes it, and hosts it in the window identified by *hWnd*.

```
STDMETHOD(CreateControl)(  
    LPCOLESTR lpTricsData,  
    HWND hWnd,  
    IStream* pStream);
```

Parameters

lpTricsData

[in] A string identifying the control to create. Can be a CLSID (must include the braces), ProgID, URL, or raw HTML (prefixed by **MSHTML:**).

hWnd

[in] A handle to the window to be used for hosting.

pStream

[in] An interface pointer for a stream containing initialization data for the control. Can be NULL.

Return Value

A standard HRESULT value.

Remarks

This window will be subclassed by the host object exposing this interface so that messages can be reflected to the control and other container features will work.

Calling this method is equivalent to calling [IAxWinHostWindow::CreateControlEx](#).

To create a licensed ActiveX control, see [IAxWinHostWindowLic::CreateControlLic](#).

IAxWinHostWindow::CreateControlEx

Creates an ActiveX control, initializes it, and hosts it in the specified window, similar to

[IAxWinHostWindow::CreateControl](#).

```

STDMETHOD(CreateControlEx)(
    LPCOLESTR lpszTricsData,
    HWND hWnd,
    IStream* pStream,
    IUnknown** ppUnk,
    REFIID riidAdvise,
    IUnknown* punkAdvise);

```

Parameters

lpTricsData

[in] A string identifying the control to create. Can be a CLSID (must include the braces), ProgID, URL, or raw HTML (prefixed with **MSHTML:**).

hWnd

[in] A handle to the window to be used for hosting.

pStream

[in] An interface pointer for a stream containing initialization data for the control. Can be NULL.

ppUnk

[out] The address of a pointer that will receive the `IUnknown` interface of the created control. Can be NULL.

riidAdvise

[in] The interface identifier of an outgoing interface on the contained object. Can be IID_NULL.

punkAdvise

[in] A pointer to the `IUnknown` interface of the sink object to be connected to the connection point on the contained object specified by `iidSink`.

Return Value

A standard HRESULT value.

Remarks

Unlike the `CreateControl` method, `CreateControlEx` also allows you to receive an interface pointer to the newly created control and set up an event sink to receive events fired by the control.

To create a licensed ActiveX control, see [IAxWinHostWindowLic::CreateControlLicEx](#).

IAxWinHostWindow::QueryControl

Returns the specified interface pointer provided by the hosted control.

```

STDMETHOD(QueryControl)(
    REFIID riid,
    void** ppvObject);

```

Parameters

riid

[in] The ID of an interface on the control being requested.

ppvObject

[out] The address of a pointer that will receive the specified interface of the created control.

Return Value

A standard HRESULT value.

IAxWinHostWindow::SetExternalDispatch

Sets the external dispinterface, which is available to contained controls through the [IDocHostUIHandlerDispatch::GetExternal](#) method.

```
STDMETHOD(SetExternalDispatch)(IDispatch* pDisp);
```

Parameters

pDisp

[in] A pointer to an `IDispatch` interface.

Return Value

A standard HRESULT value.

IAxWinHostWindow::SetExternalUIHandler

Call this function to set the external [IDocHostUIHandlerDispatch](#) interface for the `CAXWindow` object.

```
STDMETHOD(SetExternalUIHandler)(IDocHostUIHandlerDispatch* pDisp);
```

Parameters

pDisp

[in] A pointer to an `IDocHostUIHandlerDispatch` interface.

Return Value

A standard HRESULT value.

Remarks

This function is used by controls (such as the Web browser control) that query the host's site for the `IDocHostUIHandlerDispatch` interface.

See also

[IAxWinAmbientDispatch Interface](#)

[CAXWindow::QueryHost](#)

[AtlAxGetHost](#)

IAxWinHostWindowLic Interface

3/4/2019 • 2 minutes to read • [Edit Online](#)

This interface provides methods for manipulating a licensed control and its host object.

Syntax

```
interface IAxWinHostWindowLic : IAxWinHostWindow
```

Members

Methods

CreateControlLic	Creates a licensed control and attaches it to the host object.
CreateControlLicEx	Creates a licensed control, attaches it to the host object, and optionally sets up an event handler.

Remarks

`IAxWinHostWindowLic` inherits from [IAxWinHostWindow](#) and adds methods that support the creation of licensed controls.

See [Hosting ActiveX Controls Using ATL AXHost](#) for a sample that uses the members of this interface.

Requirements

The definition of this interface is available as IDL or C++, as shown below.

DEFINITION TYPE	FILE
IDL	ATLIFace.idl
C++	ATLIFace.h (also included in ATLBase.h)

IAxWinHostWindowLic::CreateControlLic

Creates a licensed control, initializes it, and hosts it in the window identified by `hWnd`.

```
STDMETHOD(CreateControlLic)(  
    LPCOLESTR lpTricsData,  
    HWND hWnd,  
    IStream* pStream,  
    BSTR bstrLic);
```

Parameters

bstrLic

[in] The BSTR that contains the license key for the control.

Remarks

See [IAxWinHostWindow::CreateControl](#) for a description of the remaining parameters and return value.

Calling this method is equivalent to calling [IAxWinHostWindowLic::CreateControlLicEx](#)

Example

See [Hosting ActiveX Controls Using ATL AXHost](#) for a sample that uses `IAxWinHostWindowLic::CreateControlLic`.

IAxWinHostWindowLic::CreateControlLicEx

Creates a licensed ActiveX control, initializes it, and hosts it in the specified window, similar to [IAxWinHostWindow::CreateControl](#).

```
STDMETHOD(CreateControlLicEx)(  
    LPCOLESTR lpszTricsData,  
    HWND hWnd,  
    IStream* pStream,  
    IUnknown** ppUnk,  
    REFIID riidAdvise,  
    IUnknown* punkAdvise,  
    BSTR bstrLic);
```

Parameters

bstrLic

[in] The BSTR that contains the license key for the control.

Remarks

See [IAxWinHostWindow::CreateControlEx](#) for a description of the remaining parameters and return value.

Example

See [Hosting ActiveX Controls Using ATL AXHost](#) for a sample that uses `IAxWinHostWindowLic::CreateControlLicEx`.

ICollectionOnSTLImpl Class

4/1/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods used by a collection class.

Syntax

```
template <class T, class CollType, class ItemType, class CopyItem, class EnumType>
class ICollectionOnSTLImpl : public T
```

Parameters

T

A COM collection interface.

CollType

A C++ Standard Library container class.

ItemType

The type of item exposed by the container interface.

CopyItem

A [copy policy class](#).

EnumType

A [CComEnumOnSTL](#)-compatible enumerator class.

Members

Public Methods

NAME	DESCRIPTION
ICollectionOnSTLImpl::get__NewEnum	Returns an enumerator object for the collection.
ICollectionOnSTLImpl::getcount	Returns the number of elements in the collection.
ICollectionOnSTLImpl::get_Item	Returns the requested item from the collection.

Public Data Members

NAME	DESCRIPTION
ICollectionOnSTLImpl::m_coll	The collection.

Remarks

This class provides the implementation for three methods of a collection interface: [getcount](#), [get_Item](#), and [get__NewEnum](#).

To use this class:

- Define (or borrow) a collection interface that you wish to implement.
- Derive your class from a specialization of `ICollectionOnSTLImpl` based on this collection interface.
- Use your derived class to implement any methods from the collection interface not handled by `ICollectionOnSTLImpl`.

NOTE

If the collection interface is a dual interface, derive your class from `IDispatchImpl`, passing the `ICollectionOnSTLImpl` specialization as the first template parameter if you want ATL to provide the implementation of the `IDispatch` methods.

- Add items to the `m_coll` member to populate the collection.

For more information and examples, see [ATL Collections and Enumerators](#).

Inheritance Hierarchy

T

`ICollectionOnSTLImpl`

Requirements

Header: `atlcom.h`

ICollectionOnSTLImpl::getcount

This method returns the number of items in the collection.

```
STDMETHOD(getcount)(long* pcount);
```

Parameters

pcount

[out] The number of elements in the collection.

Return Value

A standard HRESULT value.

ICollectionOnSTLImpl::get_Item

This method returns the specified item from the collection.

```
STDMETHOD(get_Item)(long Index, ItemType* pvar);
```

Parameters

Index

[in] The 1-based index of an item in the collection.

pvar

[out] The item corresponding to *Index*.

Return Value

A standard HRESULT value.

Remarks

The item is obtained by copying the data at the specified position in `m_coll` using the copy method of the [copy policy class](#) passed as a template argument in the `ICollectionOnSTLImpl` specialization.

ICollectionOnSTLImpl::get__NewEnum

Returns an enumerator object for the collection.

```
STDMETHOD(get__NewEnum)(IUnknown** ppUnk);
```

Parameters

ppUnk

[out] The **IUnknown** pointer of a newly created enumerator object.

Return Value

A standard HRESULT value.

Remarks

The newly created enumerator maintains an iterator on the original collection, `m_coll`, (so no copy is made) and holds a COM reference on the collection object to ensure that the collection remains alive while there are outstanding enumerators.

ICollectionOnSTLImpl::m_coll

This member holds the items represented by the collection.

```
CollType m_coll;
```

See also

[ATLCollections Sample](#)
[Class Overview](#)

IConnectionPointContainerImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements a connection point container to manage a collection of [IConnectionPointImpl](#) objects.

Syntax

```
template<class T>
class ATL_NO_VTABLE IConnectionPointContainerImpl
: public IConnectionPointContainer
```

Parameters

T

Your class, derived from `IConnectionPointContainerImpl`.

Members

Public Methods

NAME	DESCRIPTION
IConnectionPointContainerImpl::EnumConnectionPoints	Creates an enumerator to iterate through the connection points supported in the connectable object.
IConnectionPointContainerImpl::FindConnectionPoint	Retrieves an interface pointer to the connection point that supports the specified IID.

Remarks

`IConnectionPointContainerImpl` implements a connection point container to manage a collection of [IConnectionPointImpl](#) objects. `IConnectionPointContainerImpl` provides two methods that a client can call to retrieve more information about a connectable object:

- `EnumConnectionPoints` allows the client to determine which outgoing interfaces the object supports.
- `FindConnectionPoint` allows the client to determine whether the object supports a specific outgoing interface.

For information about using connection points in ATL, see the article [Connection Points](#).

Inheritance Hierarchy

`IConnectionPointContainer`

`IConnectionPointContainerImpl`

Requirements

Header: atlcom.h

IConnectionPointContainerImpl::EnumConnectionPoints

Creates an enumerator to iterate through the connection points supported in the connectable object.

```
STDMETHOD(EnumConnectionPoints)(IEnumConnectionPoints** ppEnum);
```

Remarks

See [IConnectionPointContainer::EnumConnectionPoints](#) in the Windows SDK.

IConnectionPointContainerImpl::FindConnectionPoint

Retrieves an interface pointer to the connection point that supports the specified IID.

```
STDMETHOD(FindConnectionPoint)(REFIID riid, IConnectionPoint** ppCP);
```

Remarks

See [IConnectionPointContainer::FindConnectionPoint](#) in the Windows SDK.

See also

[IConnectionPointContainer](#)

[Class Overview](#)

IConnectionPointImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements a connection point.

Syntax

```
template<class T, const IID* piid, class CDV = CComDynamicUnkArray>
class ATL_NO_VTABLE IConnectionPointImpl : public _ICPLocator<piid>
```

Parameters

T

Your class, derived from `IConnectionPointImpl`.

piid

A pointer to the IID of the interface represented by the connection point object.

CDV

A class that manages the connections. The default value is `CComDynamicUnkArray`, which allows unlimited connections. You can also use `CComUnkArray`, which specifies a fixed number of connections.

Members

Public Methods

NAME	DESCRIPTION
<code>IConnectionPointImpl::Advise</code>	Establishes a connection between the connection point and a sink.
<code>IConnectionPointImpl::EnumConnections</code>	Creates an enumerator to iterate through the connections for the connection point.
<code>IConnectionPointImpl::GetConnectionInterface</code>	Retrieves the IID of the interface represented by the connection point.
<code>IConnectionPointImpl::GetConnectionPointContainer</code>	Retrieves an interface pointer to the connectable object.
<code>IConnectionPointImpl::Unadvise</code>	Terminates a connection previously established through <code>Advise</code> .

Public Data Members

NAME	DESCRIPTION
<code>IConnectionPointImpl::m_vec</code>	Manages the connections for the connection point.

Remarks

`IConnectionPointImpl` implements a connection point, which allows an object to expose an outgoing interface

to the client. The client implements this interface on an object called a sink.

ATL uses [IConnectionPointContainerImpl](#) to implement the connectable object. Each connection point within the connectable object represents an outgoing interface, identified by *piid*. Class *CDV* manages the connections between the connection point and a sink. Each connection is uniquely identified by a "cookie."

For more information about using connection points in ATL, see the article [Connection Points](#).

Inheritance Hierarchy

_ICPLocator

IConnectionPointImpl

Requirements

Header: atlcom.h

IConnectionPointImpl::Advise

Establishes a connection between the connection point and a sink.

```
STDMETHOD(Advise)(
    IUnknown* pUnkSink,
    DWORD* pdwCookie);
```

Remarks

Use [Unadvise](#) to terminate the connection call.

See [IConnectionPoint::Advise](#) in the Windows SDK.

IConnectionPointImpl::EnumConnections

Creates an enumerator to iterate through the connections for the connection point.

```
STDMETHOD(EnumConnections)(IEnumConnections** ppEnum);
```

Remarks

See [IConnectionPoint::EnumConnections](#) in the Windows SDK.

IConnectionPointImpl::GetConnectionInterface

Retrieves the IID of the interface represented by the connection point.

```
STDMETHOD(GetConnectionInterface)(IID* piid2);
```

Remarks

See [IConnectionPoint::GetConnectionInterface](#) in the Windows SDK.

IConnectionPointImpl::GetConnectionPointContainer

Retrieves an interface pointer to the connectable object.

```
STDMETHOD(GetConnectionPointContainer)(IConnectionPointContainer** ppCPC);
```

Remarks

See [IConnectionPoint::GetConnectionPointContainer](#) in the Windows SDK.

IConnectionPointImpl::m_vec

Manages the connections between the connection point object and a sink.

```
CDV m_vec;
```

Remarks

By default, `m_vec` is of type [CComDynamicUnkArray](#).

IConnectionPointImpl::Unadvise

Terminates a connection previously established through [Advise](#).

```
STDMETHOD(Unadvise)(DWORD dwCookie);
```

Remarks

See [IConnectionPoint::Unadvise](#) in the Windows SDK.

See also

[IConnectionPoint](#)
[Class Overview](#)

IDataObjectImpl Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

This class provides methods for supporting Uniform Data Transfer and managing connections.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class T>
class IDataObjectImpl
```

Parameters

T

Your class, derived from `IDataObjectImpl`.

Members

Public Methods

NAME	DESCRIPTION
IDataObjectImpl::DAdvise	Establishes a connection between the data object and an advise sink. This enables the advise sink to receive notifications of changes in the object.
IDataObjectImpl::DUnadvise	Terminates a connection previously established through <code>DAdvise</code> .
IDataObjectImpl::EnumDAdvise	Creates an enumerator to iterate through the current advisory connections.
IDataObjectImpl::EnumFormatEtc	Creates an enumerator to iterate through the <code>FORMATETC</code> structures supported by the data object. The ATL implementation returns <code>E_NOTIMPL</code> .
IDataObjectImpl::FireDataChange	Sends a change notification back to each advise sink.
IDataObjectImpl::GetCanonicalFormatEtc	Retrieves a logically equivalent <code>FORMATETC</code> structure to one that is more complex. The ATL implementation returns <code>E_NOTIMPL</code> .
IDataObjectImpl::GetData	Transfers data from the data object to the client. The data is described in a <code>FORMATETC</code> structure and is transferred through a <code>STGMEDIUM</code> structure.

NAME	DESCRIPTION
IDataObjectImpl::GetDataHere	Similar to <code>GetData</code> , except the client must allocate the <code>STGMEDIUM</code> structure. The ATL implementation returns <code>E_NOTIMPL</code> .
IDataObjectImpl::QueryGetData	Determines whether the data object supports a particular <code>FORMATETC</code> structure for transferring data. The ATL implementation returns <code>E_NOTIMPL</code> .
IDataObjectImpl::SetData	Transfers data from the client to the data object. The ATL implementation returns <code>E_NOTIMPL</code> .

Remarks

The [IDataObject](#) interface provides methods to support Uniform Data Transfer. `IDataObject` uses the standard format structures [FORMATETC](#) and [STGMEDIUM](#) to retrieve and store data.

`IDataObject` also manages connections to advise sinks to handle data change notifications. In order for the client to receive data change notifications from the data object, the client must implement the [IAdviseSink](#) interface on an object called an advise sink. When the client then calls `IDataObject::DAdvise`, a connection is established between the data object and the advise sink.

Class `IDataObjectImpl` provides a default implementation of `IDataObject` and implements `IUnknown` by sending information to the dump device in debug builds.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy

`IDataObject`

`IDataObjectImpl`

Requirements

Header: atlctl.h

IDataObjectImpl::DAdvise

Establishes a connection between the data object and an advise sink.

```
HRESULT DAdvise(  
    FORMATETC* pformatetc,  
    DWORD advf,  
    IAdviseSink* pAdvSink,  
    DWORD* pdwConnection);
```

Remarks

This enables the advise sink to receive notifications of changes in the object.

To terminate the connection, call [DUnadvise](#).

See [IDataObject::DAdvise](#) in the Windows SDK.

IDataObjectImpl::DUnadvise

Terminates a connection previously established through [DAdvise](#).

```
HRESULT DUnadvise(DWORD dwConnection);
```

Remarks

See [IDataObject::DUnadvise](#) in the Windows SDK.

IDataObjectImpl::EnumDAdvise

Creates an enumerator to iterate through the current advisory connections.

```
HRESULT DAdvise(  
    FORMATETC* pformatetc,  
    DWORD advf,  
    IAdviseSink* pAdvSink,  
    DWORD* pdwConnection);
```

Remarks

See [IDataObject::EnumDAdvise](#) in the Windows SDK.

IDataObjectImpl::EnumFormatEtc

Creates an enumerator to iterate through the `FORMATETC` structures supported by the data object.

```
HRESULT EnumFormatEtc(  
    DWORD dwDirection,  
    IEnumFORMATETC** ppenumFormatEtc);
```

Remarks

See [IDataObject::EnumFormatEtc](#) in the Windows SDK.

Return Value

Returns E_NOTIMPL.

IDataObjectImpl::FireDataChange

Sends a change notification back to each advise sink that is currently being managed.

```
HRESULT FireDataChange();
```

Return Value

A standard HRESULT value.

IDataObjectImpl::GetCanonicalFormatEtc

Retrieves a logically equivalent `FORMATETC` structure to one that is more complex.

```
HRESULT GetCanonicalFormatEtc(FORMATETC* pformatetcIn, FORMATETC* pformatetcOut);
```

Return Value

Returns E_NOTIMPL.

Remarks

See [IDataObject::GetCanonicalFormatEtc](#) in the Windows SDK.

IDataObjectImpl::GetData

Transfers data from the data object to the client.

```
HRESULT GetData(  
    FORMATETC* pformatetcIn,  
    STGMEDIUM* pmedium);
```

Remarks

The *pformatetcIn* parameter must specify a storage medium type of TYMED_MFPICT.

See [IDataObject::GetData](#) in the Windows SDK.

IDataObjectImpl::GetDataHere

Similar to `GetData`, except the client must allocate the `STGMEDIUM` structure.

```
HRESULT GetDataHere(  
    FORMATETC* pformatetc,  
    STGMEDIUM* pmedium);
```

Return Value

Returns E_NOTIMPL.

Remarks

See [IDataObject::GetDataHere](#) in the Windows SDK.

IDataObjectImpl::QueryGetData

Determines whether the data object supports a particular `FORMATETC` structure for transferring data.

```
HRESULT QueryGetData(FORMATETC* pformatetc);
```

Return Value

Returns E_NOTIMPL.

Remarks

See [IDataObject::QueryGetData](#) in the Windows SDK.

IDataObjectImpl::SetData

Transfers data from the client to the data object.

```
HRESULT SetData(  
    FORMATETC* pformatetc,  
    STGMEDIUM* pmedium,  
    BOOL fRelease);
```

Return Value

Returns E_NOTIMPL.

Remarks

See [IDataObject::SetData](#) in the Windows SDK.

See also

[Class Overview](#)

IDispatchImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Provides a default implementation for the `IDispatch` part of a dual interface.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class T,
        const IID* piid= &__uuidof(T),
        const GUID* plibid = &CAtlModule::m_libid,
        WORD wMajor = 1,
        WORD wMinor = 0,
        class tihclass = CComTypeInfoHolder>
class ATL_NO_VTABLE IDispatchImpl : public T
```

Parameters

T

[in] A dual interface.

piid

[in] A pointer to the IID of *T*.

plibid

[in] A pointer to the LIBID of the type library that contains information about the interface. By default, the server-level type library is passed.

wMajor

[in] The major version of the type library. By default, the value is 1.

wMinor

[in] The minor version of the type library. By default, the value is 0.

tihclass

[in] The class used to manage the type information for *T*. By default, the value is `CComTypeInfoHolder`.

Members

Public Constructors

NAME	DESCRIPTION
IDispatchImpl::IDispatchImpl	The constructor. Calls <code>AddRef</code> on the protected member variable that manages the type information for the dual interface. The destructor calls <code>Release</code> .

Public Methods

NAME	DESCRIPTION
IDispatchImpl::GetIDsOfNames	Maps a set of names to a corresponding set of dispatch identifiers.
IDispatchImpl::GetTypeInfo	Retrieves the type information for the dual interface.
IDispatchImpl::GetTypeInfoCount	Determines whether there is type information available for the dual interface.
IDispatchImpl::Invoke	Provides access to the methods and properties exposed by the dual interface.

Remarks

`IDispatchImpl` provides a default implementation for the `IDispatch` part of any dual interface on an object. A dual interface derives from `IDispatch` and uses only Automation-compatible types. Like a dispinterface, a dual interface supports early binding and late binding; however, a dual interface also supports vtable binding.

The following example shows a typical implementation of `IDispatchImpl`.

```
class ATL_NO_VTABLE CBeeper :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CBeeper, &CLSID_Beeper>,
    public IDispatchImpl<IBeeper, &IID_IBeeper, &LIBID_NVC_ATL_COMLib, /*wMajor =*/ 1, /*wMinor =*/ 0>
```

By default, the `IDispatchImpl` class looks up the type information for *T* in the registry. To implement an unregistered interface, you can use the `IDispatchImpl` class without accessing the registry by using a predefined version number. If you create an `IDispatchImpl` object that has 0xFFFF as the value for *wMajor* and 0xFFFF as the value for *wMinor*, the `IDispatchImpl` class retrieves the type library from the .dll file instead of the registry.

`IDispatchImpl` contains a static member of type `CComTypeInfoHolder` that manages the type information for the dual interface. If you have multiple objects that implement the same dual interface, only one instance of `CComTypeInfoHolder` is used.

Inheritance Hierarchy

`T`

`IDispatchImpl`

Requirements

Header: atlcom.h

IDispatchImpl::GetIDsOfNames

Maps a set of names to a corresponding set of dispatch identifiers.

```
STDMETHOD(GetIDsOfNames)(
    REFIID riid,
    LPOLESTR* rgszNames,
    UINT cNames,
    LCID lcid,
    DISPID* rgdispid);
```

Remarks

See [IDispatch::GetIDsOfNames](#) in the Windows SDK.

IDispatchImpl::GetTypeInfo

Retrieves the type information for the dual interface.

```
STDMETHOD(GetTypeInfo)(
    UINT itinfo,
    LCID lcid,
    ITypeInfo** pptinfo);
```

Remarks

See [IDispatch::GetTypeInfo](#) in the Windows SDK.

IDispatchImpl::GetTypeInfoCount

Determines whether there is type information available for the dual interface.

```
STDMETHOD(GetTypeInfoCount)(UINT* pctinfo);
```

Remarks

See [IDispatch::GetTypeInfoCount](#) in the Windows SDK.

IDispatchImpl::IDispatchImpl

The constructor. Calls [AddRef](#) on the protected member variable that manages the type information for the dual interface. The destructor calls [Release](#).

```
IDispatchImpl();
```

IDispatchImpl::Invoke

Provides access to the methods and properties exposed by the dual interface.

```
STDMETHOD(Invoke)(
    DISPID dispidMember,
    REFIID riid,
    LCID lcid,
    WORD wFlags,
    DISPPARAMS* pdispparams,
    VARIANT* pvarResult,
    EXCEPINFO* pexcepinfo,
    UINT* puArgErr);
```

Remarks

See [IDispatch::Invoke](#) in the Windows SDK.

See also

[Class Overview](#)

IDisEventImpl Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

This class provides implementations of the `IDispatch` methods.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <UINT nID, class T,
    const IID* pdiid = &IID_NULL,
    const GUID* plibid = &GUID_NULL,
    WORD wMajor = 0,
    WORD wMinor = 0,
    class tihclass = CComTypeInfoHolder>
class ATL_NO_VTABLE IDisEventImpl : public IDisEventSimpleImpl<nID, T, pdiid>
```

Parameters

nID

A unique identifier for the source object. When `IDisEventImpl` is the base class for a composite control, use the resource ID of the desired contained control for this parameter. In other cases, use an arbitrary positive integer.

T

The user's class, which is derived from `IDisEventImpl`.

pdIID

The pointer to the IID of the event dispatch interface implemented by this class. This interface must be defined in the type library denoted by *plibid*, *wMajor*, and *wMinor*.

plibid

A pointer to the type library that defines the dispatch interface pointed to by *pdIID*. If **&GUID_NULL**, the type library will be loaded from the object sourcing the events.

wMajor

The major version of the type library. The default value is 0.

wMinor

The minor version of the type library. The default value is 0.

tihclass

The class used to manage the type information for *T*. The default value is a class of type `CComTypeInfoHolder`; however, you can override this template parameter by providing a class of a type other than

`CComTypeInfoHolder`.

Members

Public Typedefs

NAME	DESCRIPTION
IDispEventImpl::_tiiclass	The class used to manage the type information. By default, <code>CComTypeInfoHolder</code> .

Public Constructors

NAME	DESCRIPTION
IDispEventImpl::IDispEventImpl	The constructor.

Public Methods

NAME	DESCRIPTION
IDispEventImpl::GetFuncInfoFromId	Locates the function index for the specified dispatch identifier.
IDispEventImpl::GetIDsOfNames	Maps a single member and an optional set of argument names to a corresponding set of integer DISPIDs.
IDispEventImpl::GetTypeInfo	Retrieves the type information for an object.
IDispEventImpl::GetTypeInfoCount	Retrieves the number of type information interfaces.
IDispEventImpl::GetUserDefinedType	Retrieves the basic type of a user-defined type.

Remarks

`IDispEventImpl` provides a way of implementing an event dispinterface without requiring you to supply implementation code for every method/event on that interface. `IDispEventImpl` provides implementations of the `IDispatch` methods. You only need to supply implementations for the events that you are interested in handling.

`IDispEventImpl` works in conjunction with the event sink map in your class to route events to the appropriate handler function. To use this class:

Add a [SINK_ENTRY](#) or [SINK_ENTRY_EX](#) macro to the event sink map for each event on each object that you want to handle. When using `IDispEventImpl` as a base class of a composite control, you can call [AtlAdviseSinkMap](#) to establish and break the connection with the event sources for all entries in the event sink map. In other cases, or for greater control, call [DispEventAdvise](#) to establish the connection between the source object and the base class. Call [DispEventUnadvise](#) to break the connection.

You must derive from `IDispEventImpl` (using a unique value for *nID*) for each object for which you need to handle events. You can reuse the base class by unadvising against one source object then advising against a different source object, but the maximum number of source objects that can be handled by a single object at one time is limited by the number of `IDispEventImpl` base classes.

`IDispEventImpl` provides the same functionality as [IDispEventSimpleImpl](#), except it gets type information about the interface from a type library rather than having it supplied as a pointer to an `_ATL_FUNC_INFO` structure. Use `IDispEventSimpleImpl` when you do not have a type library describing the event interface or want to avoid the overhead associated with using the type library.

NOTE

`IDisEventImpl` and `IDisEventSimpleImpl` provide their own implementation of `IUnknown::QueryInterface` enabling each `IDisEventImpl` and `IDisEventSimpleImpl` base class to act as a separate COM identity while still allowing direct access to class members in your main COM object.

CE ATL implementation of ActiveX event sinks only supports return values of type HRESULT or void from your event handler methods; any other return value is unsupported and its behavior is undefined.

For more information, see [Supporting IDisEventImpl](#).

Inheritance Hierarchy

`_IDisEvent`

`_IDisEventLocator`

[IDisEventSimpleImpl](#)

`IDisEventImpl`

Requirements

Header: atlcom.h

IDisEventImpl::GetFuncInfoFromId

Locates the function index for the specified dispatch identifier.

```
HRESULT GetFuncInfoFromId(  
    const IID& iid,  
    DISPID dispidMember,  
    LCID lcid,  
    _ATL_FUNC_INFO& info);
```

Parameters

iid

[in] A reference to the ID of the function.

dispidMember

[in] The dispatch ID of the function.

lcid

[in] The locale context of the function ID.

info

[in] The structure indicating how the function is called.

Return Value

A standard HRESULT value.

IDisEventImpl::GetIDsOfNames

Maps a single member and an optional set of argument names to a corresponding set of integer DISPIDs, which can be used on subsequent calls to [IDispatch::Invoke](#).

```
STDMETHOD(GetIDsOfNames)(
    REFIID riid,
    LPOLESTR* rgpszNames,
    UINT cNames,
    LCID lcid,
    DISPID* rgdispid);
```

Remarks

See [IDispatch::GetIDsOfNames](#) in the Windows SDK.

IDisEventImpl::GetTypeInfo

Retrieves the type information for an object, which can then be used to get the type information for an interface.

```
STDMETHOD(GetTypeInfo)(
    UINT itinfo,
    LCID lcid,
    ITypeInfo** pptinfo);
```

Remarks

IDisEventImpl::GetTypeInfoCount

Retrieves the number of type information interfaces that an object provides (either 0 or 1).

```
STDMETHOD(GetTypeInfoCount)(UINT* pctinfo);
```

Remarks

See [IDispatch::GetTypeInfoCount](#) in the Windows SDK.

IDisEventImpl::GetUserDefinedType

Retrieves the basic type of a user-defined type.

```
VARTYPE GetUserDefinedType(
    ITypeInfo* pTI,
    HREFTYPE hrt);
```

Parameters

pTI

[in] A pointer to the [ITypeInfo](#) interface containing the user-defined type.

hrt

[in] A handle to the type description to be retrieved.

Return Value

The type of variant.

Remarks

See [ITypeInfo::GetRefTypeInfo](#).

IDisEventImpl::IDisEventImpl

The constructor. Stores the values of the class template parameters *plibid*, *pdiid*, *wMajor*, and *wMinor*.

```
IDispEventImpl();
```

IDispEventImpl::tihclass

This typedef is an instance of the class template parameter *tihclass*.

```
typedef tihclass _tihclass;
```

Remarks

By default, the class is `CComTypeInfoHolder`. `CComTypeInfoHolder` manages the type information for the class.

See also

[_ATL_FUNC_INFO Structure](#)

[IDispatchImpl Class](#)

[IDispEventSimpleImpl Class](#)

[SINK_ENTRY](#)

[SINK_ENTRY_EX](#)

[SINK_ENTRY_INFO](#)

[Class Overview](#)

IDispEventSimpleImpl Class

3/4/2019 • 5 minutes to read • [Edit Online](#)

This class provides implementations of the `IDispatch` methods, without getting type information from a type library.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <UINT nID, class T, const IID* pdiid>
class ATL_NO_VTABLE IDispEventSimpleImpl : public _IDispEventLocator<nID, pdiid>
```

Parameters

nID

A unique identifier for the source object. When `IDispEventSimpleImpl` is the base class for a composite control, use the resource ID of the desired contained control for this parameter. In other cases, use an arbitrary positive integer.

T

The user's class, which is derived from `IDispEventSimpleImpl`.

pdIID

The pointer to the IID of the event dispinterface implemented by this class.

Members

Public Methods

NAME	DESCRIPTION
IDispEventSimpleImpl::Advise	Establishes a connection with the default event source.
IDispEventSimpleImpl::DispEventAdvise	Establishes a connection with the event source.
IDispEventSimpleImpl::DispEventUnadvise	Breaks the connection with the event source.
IDispEventSimpleImpl::GetIDsOfNames	Returns E_NOTIMPL.
IDispEventSimpleImpl::GetTypeInfo	Returns E_NOTIMPL.
IDispEventSimpleImpl::GetTypeInfoCount	Returns E_NOTIMPL.
IDispEventSimpleImpl::Invoke	Calls the event handlers listed in the event sink map.
IDispEventSimpleImpl::Unadvise	Breaks the connection with the default event source.

Remarks

`IDispEventSimpleImpl` provides a way of implementing an event dispinterface without requiring you to supply implementation code for every method/event on that interface. `IDispEventSimpleImpl` provides implementations of the `IDispatch` methods. You only need to supply implementations for the events that you are interested in handling.

`IDispEventSimpleImpl` works in conjunction with the event sink map in your class to route events to the appropriate handler function. To use this class:

- Add a [SINK_ENTRY_INFO](#) macro to the event sink map for each event on each object that you want to handle.
- Supply type information for each event by passing a pointer to an [_ATL_FUNC_INFO](#) structure as a parameter to each entry. On the x86 platform, the `_ATL_FUNC_INFO.cc` value must be `CC_CDECL` with the callback function calling method of `__stdcall`.
- Call [DispEventAdvise](#) to establish the connection between the source object and the base class.
- Call [DispEventUnadvise](#) to break the connection.

You must derive from `IDispEventSimpleImpl` (using a unique value for *nID*) for each object for which you need to handle events. You can reuse the base class by unadvising against one source object then advising against a different source object, but the maximum number of source objects that can be handled by a single object at one time is limited by the number of `IDispEventSimpleImpl` base classes.

`IDispEventSimpleImpl` provides the same functionality as [IDispEventImpl](#), except it does not get type information about the interface from a type library. The wizards generate code based only on `IDispEventImpl`, but you can use `IDispEventSimpleImpl` by adding the code by hand. Use `IDispEventSimpleImpl` when you don't have a type library describing the event interface or want to avoid the overhead associated with using the type library.

NOTE

`IDispEventImpl` and `IDispEventSimpleImpl` provide their own implementation of `IUnknown::QueryInterface` enabling each `IDispEventImpl` or `IDispEventSimpleImpl` base class to act as a separate COM identity while still allowing direct access to class members in your main COM object.

CE ATL implementation of ActiveX event sinks only supports return values of type `HRESULT` or void from your event handler methods; any other return value is unsupported and its behavior is undefined.

For more information, see [Supporting IDispEventImpl](#).

Inheritance Hierarchy

`_IDispEvent`

`_IDispEventLocator`

`IDispEventSimpleImpl`

Requirements

Header: `atlcom.h`

`IDispEventSimpleImpl::Advise`

Call this method to establish a connection with the event source represented by *pUnk*.

```
HRESULT Advise(IUnknown* pUnk);
```

Parameters

pUnk

[in] A pointer to the `IUnknown` interface of the event source object.

Return Value

S_OK or any failure HRESULT value.

Remarks

Once the connection is established, events fired from *pUnk* will be routed to handlers in your class by way of the event sink map.

NOTE

If your class derives from multiple `IDispEventSimpleImpl` classes, you will need to disambiguate calls to this method by scoping the call with the particular base class you are interested in.

`Advise` establishes a connection with the default event source, it gets the IID of the default event source of the object as determined by [AtlGetObjectSourceInterface](#).

IDispEventSimpleImpl::DispEventAdvise

Call this method to establish a connection with the event source represented by *pUnk*.

```
HRESULT DispEventAdvise(IUnknown* pUnk const IID* piid);
```

Parameters

pUnk

[in] A pointer to the `IUnknown` interface of the event source object.

piid

A pointer to the IID of the event source object.

Return Value

S_OK or any failure HRESULT value.

Remarks

Subsequently, events fired from *pUnk* will be routed to handlers in your class by way of the event sink map.

NOTE

If your class derives from multiple `IDispEventSimpleImpl` classes, you will need to disambiguate calls to this method by scoping the call with the particular base class you are interested in.

`DispEventAdvise` establishes a connection with the event source specified in `piid`.

IDispEventSimpleImpl::DispEventUnadvise

Breaks the connection with the event source represented by *pUnk*.


```
HRESULT DisEventUnadvise(IUnknown* pUnk  const IID* piid);
```

Parameters

pUnk

[in] A pointer to the `IUnknown` interface of the event source object.

piid

A pointer to the IID of the event source object.

Return Value

S_OK or any failure HRESULT value.

Remarks

Once the connection is broken, events will no longer be routed to the handler functions listed in the event sink map.

NOTE

If your class derives from multiple `IDispEventSimpleImpl` classes, you will need to disambiguate calls to this method by scoping the call with the particular base class you are interested in.

`DisEventAdvise` breaks a connection that was established with the event source specified in `pdiid`.

IDispEventSimpleImpl::GetIDsOfNames

This implementation of `IDispatch::GetIDsOfNames` returns E_NOTIMPL.

```
STDMETHOD(GetIDsOfNames)(
    REFIID /* riid */,
    LPOLESTR* /* rgpszNames */,
    UINT /* cNames */,
    LCID /* lcid */,
    DISPID* /* rgdispid */);
```

Remarks

See [IDispatch::GetIDsOfNames](#) in the Windows SDK.

IDispEventSimpleImpl::GetTypeInfo

This implementation of `IDispatch::GetTypeInfo` returns E_NOTIMPL.

```
STDMETHOD(GetTypeInfo)(
    UINT /* itinfo */,
    LCID /* lcid */,
    ITypeInfo** /* pptinfo */);
```

Remarks

See [IDispatch::GetTypeInfo](#) in the Windows SDK.

IDispEventSimpleImpl::GetTypeInfoCount

This implementation of `IDispatch::GetTypeInfoCount` returns E_NOTIMPL.

```
STDMETHOD(GetTypeInfoCount)(UINT* /* pctinfo */);
```

Remarks

See [IDispatch::GetTypeInfoCount](#) in the Windows SDK.

IDispEventSimpleImpl::Invoke

This implementation of `IDispatch::Invoke` calls the event handlers listed in the event sink map.

```
STDMETHOD(Invoke)(
    DISPID dispidMember,
    REFIID /* riid */,
    LCID lcid,
    WORD /* wFlags */,
    DISPPARAMS* pdispparams,
    VARIANT* pvarResult,
    EXCEPINFO* /* pexcepthinfo */,
    UINT* /* puArgErr */);
```

Remarks

See [IDispatch::Invoke](#).

IDispEventSimpleImpl::Unadvise

Breaks the connection with the event source represented by *pUnk*.

```
HRESULT Unadvise(IUnknown* pUnk);
```

Parameters

pUnk

[in] A pointer to the `IUnknown` interface of the event source object.

Return Value

S_OK or any failure HRESULT value.

Remarks

Once the connection is broken, events will no longer be routed to the handler functions listed in the event sink map.

NOTE

If your class derives from multiple `IDispEventSimpleImpl` classes, you will need to disambiguate calls to this method by scoping the call with the particular base class you are interested in.

`Unadvise` breaks a connection that was established with the default event source specified in `pdiid`.

`Unadvise` breaks a connection with the default event source, it gets the IID of the default event source of the object as determined by [AtlGetObjectSourceInterface](#).

See also

[_ATL_FUNC_INFO](#) Structure

[IDispatchImpl](#) Class

[IDispEventImpl Class](#)

[SINK_ENTRY_INFO](#)

[Class Overview](#)

IDocHostUIHandlerDispatch Interface

3/5/2019 • 2 minutes to read • [Edit Online](#)

An interface to the Microsoft HTML parsing and rendering engine.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
interface IDocHostUIHandlerDispatch : IDispatch
```

Members

Public Methods

NOTE

The links in the following table are to the INet SDK reference topics for the members of the [IDocUIHostHandler](#) interface.

[IDocHostUIHandlerDispatch](#) has the same functionality as [IDocUIHostHandler](#), with the difference being that [IDocHostUIHandlerDispatch](#) is a dispinterface whereas [IDocUIHostHandler](#) is a custom interface.

EnableModeless	Called from MSHTML implementation of IOleInPlaceActiveObject::EnableModeless . Also called when MSHTML displays modal UI.
FilterDataObject	Called on the host by MSHTML to allow the host to replace MSHTML's data object.
GetDropTarget	Called by MSHTML when it is being used as a drop target to allow the host to supply an alternative IDropTarget .
GetExternal	Called by MSHTML to obtain the host's IDispatch interface.
GetHostInfo	Retrieves the UI capabilities of MSHTML host.
GetOptionKeyPath	Returns the registry key under which MSHTML stores user preferences.
HideUI	Called when MSHTML removes its menus and toolbars.
OnDocWindowActivate	Called from MSHTML implementation of IOleInPlaceActiveObject::OnDocWindowActivate .
OnFrameWindowActivate	Called from MSHTML implementation of IOleInPlaceActiveObject::OnFrameWindowActivate .

ResizeBorder	Called from MSHTML implementation of IOleInPlaceActiveObject::ResizeBorder .
ShowContextMenu	Called from MSHTML to display a context menu.
ShowUI	Allows the host to replace MSHTML menus and toolbars.
TranslateAccelerator	Called by MSHTML when IOleInPlaceActiveObject::TranslateAccelerator or IOleControlSite::TranslateAccelerator is called.
TranslateUrl	Called by MSHTML to allow the host an opportunity to modify the URL to be loaded.
UpdateUI	Notifies the host that the command state has changed.

Remarks

A host can replace the menus, toolbars, and context menus used by the Microsoft HTML parsing and rendering engine (MSHTML) by implementing this interface.

Requirements

The definition of this interface is available as IDL or C++, as shown below.

DEFINITION TYPE	FILE
IDL	ATLIFace.idl
C++	ATLIFace.h (also included in ATLBase.h)

See also

[IDocUIHostHandler](#)

IEnumOnSTLImpl Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

This class defines an enumerator interface based on a C++ Standard Library collection.

Syntax

```
template <class Base,  
          const IID* piid, class T, class Copy, class CollType>  
class ATL_NO_VTABLE IEnumOnSTLImpl : public Base
```

Parameters

Base

A COM enumerator. See [IEnumString](#) for an example.

piid

A pointer to the interface ID of the enumerator interface.

T

The type of item exposed by the enumerator interface.

Copy

A [copy policy class](#).

CollType

A C++ Standard Library container class.

Members

Public Methods

NAME	DESCRIPTION
IEnumOnSTLImpl::Clone	The implementation of Clone .
IEnumOnSTLImpl::Init	Initializes the enumerator.
IEnumOnSTLImpl::Next	The implementation of Next .
IEnumOnSTLImpl::Reset	The implementation of Reset .
IEnumOnSTLImpl::Skip	The implementation of Skip .

Public Data Members

NAME	DESCRIPTION
IEnumOnSTLImpl::m_iter	The iterator that represents the enumerator's current position within the collection.

NAME	DESCRIPTION
IEnumOnSTLImpl::m_pcollection	A pointer to the C++ Standard Library container holding the items to be enumerated.
IEnumOnSTLImpl::m_spUnk	The <code>IUnknown</code> pointer of the object supplying the collection.

Remarks

`IEnumOnSTLImpl` provides the implementation for a COM enumerator interface where the items being enumerated are stored in a C++ Standard Library-compatible container. This class is analogous to the `CComEnumImpl` class, which provides an implementation for an enumerator interface based on an array.

NOTE

See `CComEnumImpl::Init` for details on further differences between `CComEnumImpl` and `IEnumOnSTLImpl`.

Typically, you will *not* need to create your own enumerator class by deriving from this interface implementation. If you want to use an ATL-supplied enumerator based on a C++ Standard Library container, it is more common to create an instance of `CComEnumOnSTL`, or to create a collection class that returns an enumerator by deriving from `ICollectionOnSTLImpl`.

However, if you do need to provide a custom enumerator (for example, one that exposes interfaces in addition to the enumerator interface), you can derive from this class. In this situation it is likely that you'll need to override the `Clone` method to provide your own implementation.

Inheritance Hierarchy

Base

`IEnumOnSTLImpl`

Requirements

Header: `atlcom.h`

IEnumOnSTLImpl::Init

Initializes the enumerator.

```
HRESULT Init(
    IUnknown* pUnkForRelease,
    CollType& collection);
```

Parameters

pUnkForRelease

[in] The `IUnknown` pointer of an object that must be kept alive during the lifetime of the enumerator. Pass NULL if no such object exists.

collection

A reference to the C++ Standard Library container that holds the items to be enumerated.

Return Value

A standard HRESULT value.

Remarks

If you pass `Init` a reference to a collection held in another object, you can use the *pUnkForRelease* parameter to ensure that the object, and the collection it holds, is available for as long as the enumerator needs it.

You must call this method before passing a pointer to the enumerator interface back to any clients.

IEnumOnSTLImpl::Clone

This method provides the implementation of the **Clone** method by creating an object of type `CComEnumOnSTL`, initializing it with the same collection and iterator used by the current object, and returning the interface on the newly created object.

```
STDMETHOD(Clone)(Base** ppEnum);
```

Parameters

ppEnum

[out] The enumerator interface on a newly created object cloned from the current enumerator.

Return Value

A standard HRESULT value.

IEnumOnSTLImpl::m_spUnk

The `IUnknown` pointer of the object supplying the collection.

```
CComPtr<IUnknown> m_spUnk;
```

Remarks

This smart pointer maintains a reference on the object passed to [IEnumOnSTLImpl::Init](#), ensuring that it remains alive during the lifetime of the enumerator.

IEnumOnSTLImpl::m_pcollection

This member points to the collection that provides the data driving the implementation of the enumerator interface.

```
CollType* m_pcollection;
```

Remarks

This member is initialized by a call to [IEnumOnSTLImpl::Init](#).

IEnumOnSTLImpl::m_iter

This member holds the iterator used to mark the current position within the collection and navigate to subsequent elements.

```
CollType::iterator m_iter;
```


IEnumOnSTLImpl::Next

This method provides the implementation of the **Next** method.

```
STDMETHOD(Next)(
    ULONG celt,
    T* rgelt,
    ULONG* pceltFetched);
```

Parameters

celt

[in] The number of elements requested.

rgelt

[out] The array to be filled in with the elements.

pceltFetched

[out] The number of elements actually returned in *rgelt*. This can be less than *celt* if fewer than *celt* elements remain in the list.

Return Value

A standard HRESULT value.

IEnumOnSTLImpl::Reset

This method provides the implementation of the **Reset** method.

```
STDMETHOD(Reset)(void);
```

Return Value

A standard HRESULT value.

IEnumOnSTLImpl::Skip

This method provides the implementation of the **Skip** method.

```
STDMETHOD(Skip)(ULONG celt);
```

Parameters

celt

[in] The number of elements to skip.

Return Value

A standard HRESULT value.

See also

[Class Overview](#)

IObjectSafetyImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides a default implementation of the `IObjectSafety` interface to allow a client to retrieve and set an object's safety levels.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class T,DWORD dwSupportedSafety>
class IObjectSafetyImpl
```

Parameters

T

Your class, derived from `IObjectSafetyImpl`.

dwSupportedSafety

Specifies the supported safety options for the control. Can be one of the following values:

- `INTERFACESAFE_FOR_UNTRUSTED_CALLER` The interface identified by the [SetInterfaceSafetyOptions](#) parameter `riid` should be made safe for scripting.
- `INTERFACESAFE_FOR_UNTRUSTED_DATA` The interface identified by the `SetInterfaceSafetyOptions` parameter `riid` should be made safe for untrusted data during initialization.

Members

Public Methods

NAME	DESCRIPTION
IObjectSafetyImpl::GetInterfaceSafetyOptions	Retrieves the safety options supported by the object, as well as the safety options currently set for the object.
IObjectSafetyImpl::SetInterfaceSafetyOptions	Makes the object safe for initialization or scripting.

Public Data Members

NAME	DESCRIPTION
IObjectSafetyImpl::m_dwCurrentSafety	Stores the object's current safety level.

Remarks

Class `IObjectSafetyImpl` provides a default implementation of `IObjectSafety`. The `IObjectSafety` interface allows a client to retrieve and set an object's safety levels. For example, a web browser can call

`IObjectSafety::SetInterfaceSafetyOptions` to make a control safe for initialization or safe for scripting.

Note that using the `IMPLEMENTED_CATEGORY` macro with the `CATID_SafeForScripting` and `CATID_SafeForInitializing` component categories provides an alternative way of specifying that a component is safe.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy

`IObjectSafety`

`IObjectSafetyImpl`

Requirements

Header: atlctl.h

IObjectSafetyImpl::GetInterfaceSafetyOptions

Retrieves the safety options supported by the object, as well as the safety options currently set for the object.

```
HRESULT GetInterfaceSafetyOptions(  
    REFIID riid,  
    DWORD* pdwSupportedOptions,  
    DWORD* pdwEnabledOptions);
```

Remarks

The implementation returns the appropriate values for any interface supported by the object's implementation of

`IUnknown::QueryInterface`.

IMPORTANT

Any object that supports `IObjectSafety` is responsible for its own security, and that of any object it delegates. The programmer must take into account issues arising from running code in the user's context, cross-site scripting and perform suitable zone checking.

See [IObjectSafety::GetInterfaceSafetyOptions](#) in the Windows SDK.

IObjectSafetyImpl::m_dwCurrentSafety

Stores the object's current safety level.

```
DWORD m_dwCurrentSafety;
```

IObjectSafetyImpl::SetInterfaceSafetyOptions

Makes the object safe for initialization or scripting by setting the [m_dwCurrentSafety](#) member to the appropriate value.

```
HRESULT SetInterfaceSafetyOptions(  
    REFIID riid,  
    DWORD dwOptionsSetMask,  
    DWORD dwEnabledOptions);
```

Remarks

The implementation returns E_NOINTERFACE for any interface not supported by the object's implementation of

`IUnknown::QueryInterface`.

IMPORTANT

Any object that supports `IObjectSafety` is responsible for its own security, and that of any object it delegates. The programmer must take into account issues arising from running code in the user's context, cross-site scripting and perform suitable zone checking.

See [IObjectSafety::SetInterfaceSafetyOptions](#) in the Windows SDK.

See also

[IObjectSafety Interface](#)

[Class Overview](#)

IObjectWithSiteImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods allowing an object to communicate with its site.

Syntax

```
template <class T>
    class ATL_NO_VTABLE IObjectWithSiteImpl :
    public IObjectWithSite
```

Parameters

T

Your class, derived from `IObjectWithSiteImpl`.

Members

Public Methods

NAME	DESCRIPTION
IObjectWithSiteImpl::GetSite	Queries the site for an interface pointer.
IObjectWithSiteImpl::SetChildSite	Provides the object with the site's <code>IUnknown</code> pointer.
IObjectWithSiteImpl::SetSite	Provides the object with the site's <code>IUnknown</code> pointer.

Public Data Members

NAME	DESCRIPTION
IObjectWithSiteImpl::m_spUnkSite	Manages the site's <code>IUnknown</code> pointer.

Remarks

The [IObjectWithSite](#) interface allows an object to communicate with its site. Class `IObjectWithSiteImpl` provides a default implementation of this interface and implements `IUnknown` by sending information to the dump device in debug builds.

`IObjectWithSiteImpl` specifies two methods. The client first calls `SetSite`, passing the site's `IUnknown` pointer. This pointer is stored within the object, and can later be retrieved through a call to `GetSite`.

Typically, you derive your class from `IObjectWithSiteImpl` when you are creating an object that is not a control. For controls, derive your class from [IOleObjectImpl](#), which also provides a site pointer. Do not derive your class from both `IObjectWithSiteImpl` and `IOleObjectImpl`.

Inheritance Hierarchy

`IObjectWithSite`

Requirements

Header: atlcom.h

IObjectWithSiteImpl::GetSite

Queries the site for a pointer to the interface identified by `riid`.

```
STDMETHOD(GetSite)(  
    REFIID riid,  
    void** ppvSite);
```

Remarks

If the site supports this interface, the pointer is returned via `ppvSite`. Otherwise, `ppvSite` is set to NULL.

See [IObjectWithSite::GetSite](#) in the Windows SDK.

IObjectWithSiteImpl::m_spUnkSite

Manages the site's `IUnknown` pointer.

```
CComPtr<IUnknown> m_spUnkSite;
```

Remarks

`m_spUnkSite` initially receives this pointer through a call to [SetSite](#).

IObjectWithSiteImpl::SetChildSite

Provides the object with the site's `IUnknown` pointer.

```
HRESULT SetChildSite(IUnknown* pUnkSite);
```

Parameters

pUnkSite

[in] Pointer to the `IUnknown` interface pointer of the site managing this object. If NULL, the object should call

`IUnknown::Release` on any existing site at which point the object no longer knows its site.

Return Value

Returns S_OK.

IObjectWithSiteImpl::SetSite

Provides the object with the site's `IUnknown` pointer.

```
STDMETHOD(SetSite)(IUnknown* pUnkSite);
```

Remarks

See [IObjectWithSite::SetSite](#) in the Windows SDK.

See also

[Class Overview](#)

IOleControlImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides a default implementation of the `IOleControl` interface and implements `IUnknown`.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class T>
class IOleControlImpl
```

Parameters

T

Your class, derived from `IOleControlImpl`.

Members

Public Methods

NAME	DESCRIPTION
IOleControlImpl::FreezeEvents	Indicates whether or not the container ignores or accepts events from the control.
IOleControlImpl::GetControlInfo	Fills in information about the control's keyboard behavior. The ATL implementation returns E_NOTIMPL.
IOleControlImpl::OnAmbientPropertyChange	Informs a control that one or more of the container's ambient properties has changed. The ATL implementation returns S_OK.
IOleControlImpl::OnMnemonic	Informs the control that a user has pressed a specified keystroke. The ATL implementation returns E_NOTIMPL.

Remarks

Class `IOleControlImpl` provides a default implementation of the `IOleControl` interface and implements `IUnknown` by sending information to the dump device in debug builds.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy

`IOleControl`

`IOleControlImpl`

Requirements

Header: atlctl.h

IOleControlImpl::FreezeEvents

In ATL's implementation, `FreezeEvents` increments the control class's `m_nFreezeEvents` data member if `bFreeze` is TRUE, and decrements `m_nFreezeEvents` if `bFreeze` is FALSE.

```
HRESULT FreezeEvents(BOOL bFreeze);
```

Remarks

`FreezeEvents` then returns S_OK.

See [IOleControl::FreezeEvents](#) in the Windows SDK.

IOleControlImpl::GetControlInfo

Fills in information about the control's keyboard behavior.

```
HRESULT GetControlInfo(LPCONTROLINFO pCI);
```

Remarks

See [IOleControl::GetControlInfo](#) in the Windows SDK.

Return Value

Returns E_NOTIMPL.

IOleControlImpl::OnAmbientPropertyChange

Informs a control that one or more of the container's ambient properties has changed.

```
HRESULT OnAmbientPropertyChange(DISPID dispid);
```

Return Value

Returns S_OK.

Remarks

See [IOleControl::OnAmbientPropertyChange](#) in the Windows SDK.

IOleControlImpl::OnMnemonic

Informs the control that a user has pressed a specified keystroke.

```
HRESULT OnMnemonic(LPMSG pMsg);
```

Return Value

Returns E_NOTIMPL.

Remarks

See [IOleControl::OnMnemonic](#) in the Windows SDK.

See also

[IOleObjectImpl Class](#)

[ActiveX Controls Interfaces](#)

[Class Overview](#)

IOleInPlaceActiveObjectImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides methods for assisting communication between an in-place control and its container.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class T>
class IOleInPlaceActiveObjectImpl
```

Parameters

T

Your class, derived from `IOleInPlaceActiveObjectImpl`.

Members

Public Methods

NAME	DESCRIPTION
IOleInPlaceActiveObjectImpl::ContextSensitiveHelp	Enables context-sensitive help. The ATL implementation returns E_NOTIMPL.
IOleInPlaceActiveObjectImpl::EnableModeless	Enables modeless dialog boxes. The ATL implementation returns S_OK.
IOleInPlaceActiveObjectImpl::GetWindow	Gets a window handle.
IOleInPlaceActiveObjectImpl::OnDocWindowActivate	Notifies the control when the container's document window is activated or deactivated. The ATL implementation returns S_OK.
IOleInPlaceActiveObjectImpl::OnFrameWindowActivate	Notifies the control when the container's top-level frame window is activated or deactivated. The ATL implementation returns
IOleInPlaceActiveObjectImpl::ResizeBorder	Informs the control it needs to resize its borders. The ATL implementation returns S_OK.
IOleInPlaceActiveObjectImpl::TranslateAccelerator	Processes menu accelerator-key messages from the container. The ATL implementation returns E_NOTIMPL.

Remarks

The [IOleInPlaceActiveObject](#) interface assists communication between an in-place control and its container; for example, communicating the active state of the control and container, and informing the control it needs to resize

itself. Class `IInPlaceActiveObjectImpl` provides a default implementation of `IInPlaceActiveObject` and supports `IUnknown` by sending information to the dump device in debug builds.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy

`IInPlaceActiveObject`

`IInPlaceActiveObjectImpl`

Requirements

Header: `atlctl.h`

IInPlaceActiveObjectImpl::ContextSensitiveHelp

Enables context-sensitive help.

```
HRESULT ContextSensitiveHelp(BOOL fEnterMode);
```

Return Value

Returns `E_NOTIMPL`.

Remarks

See [IInWindow::ContextSensitiveHelp](#) in the Windows SDK.

IInPlaceActiveObjectImpl::EnableModeless

Enables modeless dialog boxes.

```
HRESULT EnableModeless(BOOL fEnable);
```

Return Value

Returns `S_OK`.

Remarks

See [IInPlaceActiveObject::EnableModeless](#) in the Windows SDK.

IInPlaceActiveObjectImpl::GetWindow

The container calls this function to get the window handle of the control.

```
HRESULT GetWindow(HWND* phwnd);
```

Remarks

Some containers will not work with a control that has been windowless, even if it is currently windowed. In ATL's implementation, if the `CComControl::m_bWasOnceWindowless` data member is `TRUE`, the function returns `E_FAIL`. Otherwise, if `*phwnd` is not `NULL`, `GetWindow` assigns `phwnd` to the control class's data member `m_hwnd` and returns `S_OK`.

See [IInWindow::GetWindow](#) in the Windows SDK.

IOleInPlaceActiveObjectImpl::OnDocWindowActivate

Notifies the control when the container's document window is activated or deactivated.

```
HRESULT OnDocWindowActivate(BOOL fActivate);
```

Return Value

Returns S_OK.

Remarks

See [IOleInPlaceActiveObject::OnDocWindowActivate](#) in the Windows SDK.

IOleInPlaceActiveObjectImpl::OnFrameWindowActivate

Notifies the control when the container's top-level frame window is activated or deactivated.

```
HRESULT OnFrameWindowActivate(BOOL fActivate);
```

Return Value

Returns S_OK.

Remarks

See [IOleInPlaceActiveObject::OnFrameWindowActivate](#) in the Windows SDK.

IOleInPlaceActiveObjectImpl::ResizeBorder

Informs the control it needs to resize its borders.

```
HRESULT ResizeBorder(  
    LPRECT prcBorder,  
    IOleInPlaceUIWindow* pUIWindow,  
    BOOL fFrameWindow);
```

Return Value

Returns S_OK.

Remarks

See [IOleInPlaceActiveObject::ResizeBorder](#) in the Windows SDK.

IOleInPlaceActiveObjectImpl::TranslateAccelerator

Processes menu accelerator-key messages from the container.

```
HRESULT TranslateAccelerator(LPMSG lpmsg);
```

Return Value

This method supports the following return values:

S_OK if the message was translated successfully.

S_FALSE if the message was not translated.

Remarks

See [IOleInPlaceActiveObject::TranslateAccelerator](#) in the Windows SDK.

See also

[CComControl Class](#)

[ActiveX Controls Interfaces](#)

[Class Overview](#)

IOleInPlaceObjectWindowlessImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements `IUnknown` and provides methods that enable a windowless control to receive window messages and to participate in drag-and-drop operations.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class T>  
class IOleInPlaceObjectWindowlessImpl
```

Parameters

T

Your class, derived from `IOleInPlaceObjectWindowlessImpl`.

Members

Public Methods

NAME	DESCRIPTION
IOleInPlaceObjectWindowlessImpl::ContextSensitiveHelp	Enables context-sensitive help. The ATL implementation returns <code>E_NOTIMPL</code> .
IOleInPlaceObjectWindowlessImpl::GetDropTarget	Supplies the <code>IDropTarget</code> interface for an in-place active, windowless object that supports drag and drop. The ATL implementation returns <code>E_NOTIMPL</code> .
IOleInPlaceObjectWindowlessImpl::GetWindow	Gets a window handle.
IOleInPlaceObjectWindowlessImpl::InPlaceDeactivate	Deactivates an active in-place control.
IOleInPlaceObjectWindowlessImpl::OnWindowMessage	Dispatches a message from the container to a windowless control that is in-place active.
IOleInPlaceObjectWindowlessImpl::ReactivateAndUndo	Reactivates a previously deactivated control. The ATL implementation returns <code>E_NOTIMPL</code> .
IOleInPlaceObjectWindowlessImpl::SetObjectRects	Indicates what part of the in-place control is visible.
IOleInPlaceObjectWindowlessImpl::UIDeactivate	Deactivates and removes the user interface that supports in-place activation.

Remarks

The [IOleInPlaceObject](#) interface manages the reactivation and deactivation of in-place controls and determines how much of the control should be visible. The [IOleInPlaceObjectWindowless](#) interface enables a windowless control to receive window messages and to participate in drag-and-drop operations. Class `IOleInPlaceObjectWindowlessImpl` provides a default implementation of `IOleInPlaceObject` and `IOleInPlaceObjectWindowless` and implements `IUnknown` by sending information to the dump device in debug builds.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy

`IOleInPlaceObjectWindowless`

`IOleInPlaceObjectWindowlessImpl`

Requirements

Header: atlctl.h

IOleInPlaceObjectWindowlessImpl::ContextSensitiveHelp

Returns E_NOTIMPL.

```
HRESULT ContextSensitiveHelp(BOOL fEnterMode);
```

Remarks

See [IOleWindow::ContextSensitiveHelp](#) in the Windows SDK.

IOleInPlaceObjectWindowlessImpl::GetDropTarget

Returns E_NOTIMPL.

```
HRESULT GetDropTarget(IDropTarget** ppDropTarget);
```

Remarks

See [IOleInPlaceObjectWindowless::GetDropTarget](#) in the Windows SDK.

IOleInPlaceObjectWindowlessImpl::GetWindow

The container calls this function to get the window handle of the control.

```
HRESULT GetWindow(HWND* phwnd);
```

Remarks

Some containers will not work with a control that has been windowless, even if it is currently windowed. In ATL's implementation, if the control class's data member `m_bWasOnceWindowless` is TRUE, the function returns E_FAIL. Otherwise, if *phwnd* is not NULL, `GetWindow` sets **phwnd* to the control class's data member `m_hWnd` and returns S_OK.

See [IOleWindow::GetWindow](#) in the Windows SDK.

IOleInPlaceObjectWindowlessImpl::InPlaceDeactivate

Called by the container to deactivate an in-place active control.

```
HRESULT InPlaceDeactivate(HWND* phwnd);
```

Remarks

This method performs a full or partial deactivation depending on the state of the control. If necessary, the control's user interface is deactivated, and the control's window, if any, is destroyed. The container is notified that the control is no longer active in place. The `IOleInPlaceUIWindow` interface used by the container to negotiate menus and border space is released.

See [IOleInPlaceObject::InPlaceDeactivate](#) in the Windows SDK.

IOleInPlaceObjectWindowlessImpl::OnWindowMessage

Dispatches a message from a container to a windowless control that is in-place active.

```
HRESULT OnWindowMessage(  
    UINT msg,  
    WPARAM wParam,  
    LPARAM lParam,  
    LRESULT* plResultParam);
```

Remarks

See [IOleInPlaceObjectWindowless::OnWindowMessage](#) in the Windows SDK.

IOleInPlaceObjectWindowlessImpl::ReactivateAndUndo

Returns E_NOTIMPL.

```
HRESULT ReactivateAndUndo();
```

Remarks

See [IOleInPlaceObject::ReactivateAndUndo](#) in the Windows SDK.

IOleInPlaceObjectWindowlessImpl::SetObjectRects

Called by the container to inform the control that its size and/or position has changed.

```
HRESULT SetObjectRects(LPCRECT prcPos, LPCRECT prcClip);
```

Remarks

Updates the control's `m_rcPos` data member and the control display. Only the part of the control that intersects the clip region is displayed. If a control's display was previously clipped but the clipping has been removed, this function can be called to redraw a full view of the control.

See [IOleInPlaceObject::SetObjectRects](#) in the Windows SDK.

IOleInPlaceObjectWindowlessImpl::UIDeactivate

Deactivates and removes the control's user interface that supports in-place activation.

```
HRESULT UIDeactivate();
```

Remarks

Sets the control class's data member `m_bUIActive` to FALSE. The ATL implementation of this function always returns S_OK.

See [IOleInPlaceObject::UIDeactivate](#) in the Windows SDK.

See also

[CComControl Class](#)

[Class Overview](#)

IOleObjectImpl Class

4/1/2019 • 11 minutes to read • [Edit Online](#)

This class implements `IUnknown` and is the principal interface through which a container communicates with a control.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class T>
class ATL_NO_VTABLE IOleObjectImpl : public IOleObject
```

Parameters

T

Your class, derived from `IOleObjectImpl`.

Members

Public Methods

NAME	DESCRIPTION
IOleObjectImpl::Advise	Establishes an advisory connection with the control.
IOleObjectImpl::Close	Changes the control state from running to loaded.
IOleObjectImpl::DoVerb	Tells the control to perform one of its enumerated actions.
IOleObjectImpl::DoVerbDiscardUndo	Tells the control to discard any undo state it is maintaining.
IOleObjectImpl::DoVerbHide	Tells the control to remove its user interface from view.
IOleObjectImpl::DoVerbInPlaceActivate	Runs the control and installs its window, but does not install the control's user interface.
IOleObjectImpl::DoVerbOpen	Causes the control to be open-edited in a separate window.
IOleObjectImpl::DoVerbPrimary	Performs the specified action when the user double-clicks the control. The control defines the action, usually to activate the control in-place.
IOleObjectImpl::DoVerbShow	Shows a newly inserted control to the user.
IOleObjectImpl::DoVerbUIActivate	Activates the control in-place and shows the control's user interface, such as menus and toolbars.

NAME	DESCRIPTION
IOleObjectImpl::EnumAdvise	Enumerates the control's advisory connections.
IOleObjectImpl::EnumVerbs	Enumerates actions for the control.
IOleObjectImpl::GetClientSite	Retrieves the control's client site.
IOleObjectImpl::GetClipboardData	Retrieves data from the Clipboard. The ATL implementation returns E_NOTIMPL.
IOleObjectImpl::GetExtent	Retrieves the extent of the control's display area.
IOleObjectImpl::GetMiscStatus	Retrieves the status of the control.
IOleObjectImpl::GetMoniker	Retrieves the control's moniker. The ATL implementation returns E_NOTIMPL.
IOleObjectImpl::GetUserClassID	Retrieves the control's class identifier.
IOleObjectImpl::GetUserType	Retrieves the control's user-type name.
IOleObjectImpl::InitFromData	Initializes the control from selected data. The ATL implementation returns E_NOTIMPL.
IOleObjectImpl::IsUpToDate	Checks if the control is up to date. The ATL implementation returns S_OK.
IOleObjectImpl::OnPostVerbDiscardUndo	Called by DoVerbDiscardUndo after the undo state is discarded.
IOleObjectImpl::OnPostVerbHide	Called by DoVerbHide after the control is hidden.
IOleObjectImpl::OnPostVerbInPlaceActivate	Called by DoVerbInPlaceActivate after the control is activated in place.
IOleObjectImpl::OnPostVerbOpen	Called by DoVerbOpen after the control has been opened for editing in a separate window.
IOleObjectImpl::OnPostVerbShow	Called by DoVerbShow after the control has been made visible.
IOleObjectImpl::OnPostVerbUIActivate	Called by DoVerbUIActivate after the control's user interface has been activated.
IOleObjectImpl::OnPreVerbDiscardUndo	Called by DoVerbDiscardUndo before the undo state is discarded.
IOleObjectImpl::OnPreVerbHide	Called by DoVerbHide before the control is hidden.
IOleObjectImpl::OnPreVerbInPlaceActivate	Called by DoVerbInPlaceActivate before the control is activated in place.

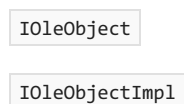
NAME	DESCRIPTION
IOleObjectImpl::OnPreVerbOpen	Called by DoVerbOpen before the control has been opened for editing in a separate window.
IOleObjectImpl::OnPreVerbShow	Called by DoVerbShow before the control has been made visible.
IOleObjectImpl::OnPreVerbUIActivate	Called by DoVerbUIActivate before the control's user interface has been activated.
IOleObjectImpl::SetClientSite	Tells the control about its client site in the container.
IOleObjectImpl::SetColorScheme	Recommends a color scheme to the control's application, if any. The ATL implementation returns E_NOTIMPL.
IOleObjectImpl::SetExtent	Sets the extent of the control's display area.
IOleObjectImpl::SetHostNames	Tells the control the names of the container application and container document.
IOleObjectImpl::SetMoniker	Tells the control what its moniker is. The ATL implementation returns E_NOTIMPL.
IOleObjectImpl::Unadvise	Deletes an advisory connection with the control.
IOleObjectImpl::Update	Updates the control. The ATL implementation returns S_OK.

Remarks

The [IOleObject](#) interface is the principal interface through which a container communicates with a control. Class `IOleObjectImpl` provides a default implementation of this interface and implements `IUnknown` by sending information to the dump device in debug builds.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy



Requirements

Header: atlctl.h

IOleObjectImpl::Advise

Establishes an advisory connection with the control.

```

STDMETHOD(Advise)(
    IAdviseSink* pAdvSink,
    DWORD* pdwConnection);
  
```

Remarks

See [IOleObject::Advise](#) in the Windows SDK.

IOleObjectImpl::Close

Changes the control state from running to loaded.

```
STDMETHOD(Close)(DWORD dwSaveOption);
```

Remarks

Deactivates the control and destroys the control window if it exists. If the control class data member [CComControlBase::m_bRequiresSave](#) is TRUE and the *dwSaveOption* parameter is either OLECLOSE_SAVEIFDIRTY or OLECLOSE_PROMPTSAVE, the control properties are saved before closing.

The pointers held in the control class data members [CComControlBase::m_spInPlaceSite](#) and [CComControlBase::m_spAdviseSink](#) are released, and the data members [CComControlBase::m_bNegotiatedWnd](#), [CComControlBase::m_bWndless](#), and [CComControlBase::m_bInPlaceSiteEx](#) are set to FALSE.

See [IOleObject::Close](#) in the Windows SDK.

IOleObjectImpl::DoVerb

Tells the control to perform one of its enumerated actions.

```
STDMETHOD(DoVerb)(
    LONG iVerb,
    LPMSG /* pMsg */,
    IOleClientSite* pActiveSite,
    LONG /* lindex */,
    HWND hwndParent,
    LPCRECT lprcPosRect);
```

Remarks

Depending on the value of `iVerb`, one of the ATL `DoVerb` helper functions is called as follows:

IVERB VALUE	DOVERB HELPER FUNCTION CALLED
OLEIVERB_DISCARDUNDOSTATE	DoVerbDiscardUndo
OLEIVERB_HIDE	DoVerbHide
OLEIVERB_INPLACEACTIVATE	DoVerbInPlaceActivate
OLEIVERB_OPEN	DoVerbOpen
OLEIVERB_PRIMARY	DoVerbPrimary
OLEIVERB_PROPERTIES	CComControlBase::DoVerbProperties
OLEIVERB_SHOW	DoVerbShow
OLEIVERB_UIACTIVATE	DoVerbUIActivate

See [IOleObject::DoVerb](#) in the Windows SDK.

IOleObjectImpl::DoVerbDiscardUndo

Tells the control to discard any undo state it is maintaining.

```
HRESULT DoVerbDiscardUndo(LPCRECT /* prcPosRect */, HWND /* hwndParent */);
```

Parameters

prcPosRect

[in] Pointer to the rectangle the container wants the control to draw into.

hwndParent

[in] Handle of the window containing the control.

Return Value

Returns S_OK.

IOleObjectImpl::DoVerbHide

Deactivates and removes the control's user interface, and hides the control.

```
HRESULT DoVerbHide(LPCRECT /* prcPosRect */, HWND /* hwndParent */);
```

Parameters

prcPosRect

[in] Pointer to the rectangle the container wants the control to draw into.

hwndParent

[in] Handle of the window containing the control. Not used in the ATL implementation.

Return Value

Returns S_OK.

IOleObjectImpl::DoVerbInPlaceActivate

Runs the control and installs its window, but does not install the control's user interface.

```
HRESULT DoVerbInPlaceActivate(LPCRECT prcPosRect, HWND /* hwndParent */);
```

Parameters

prcPosRect

[in] Pointer to the rectangle the container wants the control to draw into.

hwndParent

[in] Handle of the window containing the control. Not used in the ATL implementation.

Return Value

One of the standard HRESULT values.

Remarks

Activates the control in place by calling [CComControlBase::InPlaceActivate](#). Unless the control class's data member `m_bWindowOnly` is TRUE, `DoVerbInPlaceActivate` first attempts to activate the control as a windowless

control (possible only if the container supports [IOleInPlaceSiteWindowless](#)). If that fails, the function attempts to activate the control with extended features (possible only if the container supports [IOleInPlaceSiteEx](#)). If that fails, the function attempts to activate the control with no extended features (possible only if the container supports [IOleInPlaceSite](#)). If activation succeeds, the function notifies the container the control has been activated.

IOleObjectImpl::DoVerbOpen

Causes the control to be open-edited in a separate window.

```
HRESULT DoVerbOpen(LPCRECT /* prcPosRect */, HWND /* hwndParent */);
```

Parameters

prcPosRec

[in] Pointer to the rectangle the container wants the control to draw into.

hwndParent

[in] Handle of the window containing the control.

Return Value

Returns S_OK.

IOleObjectImpl::DoVerbPrimary

Defines the action taken when the user double-clicks the control.

```
HRESULT DoVerbPrimary(LPCRECT prcPosRect, HWND hwndParent);
```

Parameters

prcPosRec

[in] Pointer to the rectangle the container wants the control to draw into.

hwndParent

[in] Handle of the window containing the control.

Return Value

One of the standard HRESULT values.

Remarks

By default, set to display the property pages. You can override this in your control class to invoke a different behavior on double-click; for example, play a video or go in-place active.

IOleObjectImpl::DoVerbShow

Tells the container to make the control visible.

```
HRESULT DoVerbShow(LPCRECT prcPosRect, HWND /* hwndParent */);
```

Parameters

prcPosRec

[in] Pointer to the rectangle the container wants the control to draw into.

hwndParent

[in] Handle of the window containing the control. Not used in the ATL implementation.

Return Value

One of the standard HRESULT values.

IOleObjectImpl::DoVerbUIActivate

Activates the control's user interface and notifies the container that its menus are being replaced by composite menus.

```
HRESULT DoVerbUIActivate(LPCRECT prcPosRect, HWND /* hwndParent */);
```

Parameters

prcPosRect

[in] Pointer to the rectangle the container wants the control to draw into.

hwndParent

[in] Handle of the window containing the control. Not used in the ATL implementation.

Return Value

One of the standard HRESULT values.

IOleObjectImpl::EnumAdvise

Supplies an enumeration of registered advisory connections for this control.

```
STDMETHOD(EnumAdvise)(IEnumSTATDATA** ppenumAdvise);
```

Remarks

See [IOleObject::EnumAdvise](#) in the Windows SDK.

IOleObjectImpl::EnumVerbs

Supplies an enumeration of registered actions (verbs) for this control by calling `OleRegEnumVerbs`.

```
STDMETHOD(EnumVerbs)(IEnumOLEVERB** ppEnumOleVerb);
```

Remarks

You can add verbs to your project's .rgs file. For example, see CIRCCTL.RGS in the [CIRC](#) sample.

See [IOleObject::EnumVerbs](#) in the Windows SDK.

IOleObjectImpl::GetClientSite

Puts the pointer in the control class data member `CComControlBase::m_spClientSite` into *ppClientSite* and increments the reference count on the pointer.

```
STDMETHOD(GetClientSite)(IOleClientSite** ppClientSite);
```

Remarks

See [IOleObject::GetClientSite](#) in the Windows SDK.

IOleObjectImpl::GetClipboardData

Retrieves data from the Clipboard.

```
STDMETHOD(GetClipboardData)(  
    DWORD /* dwReserved */,  
    IDataObject** /* ppDataObject */);
```

Return Value

Returns E_NOTIMPL.

Remarks

See [IOleObject::GetClipboardData](#) in the Windows SDK.

IOleObjectImpl::GetExtent

Retrieves a running control's display size in HIMETRIC units (0.01 millimeter per unit).

```
STDMETHOD(GetExtent)(  
    DWORD dwDrawAspect,  
    SIZEL* pszel);
```

Remarks

The size is stored in the control class data member [CComControlBase::m_sizeExtent](#).

See [IOleObject::GetExtent](#) in the Windows SDK.

IOleObjectImpl::GetMiscStatus

Returns a pointer to registered status information for the control by calling `OleRegGetMiscStatus` .

```
STDMETHOD(GetMiscStatus)(  
    DWORD dwAspect,  
    DWORD* pdwStatus);
```

Remarks

The status information includes behaviors supported by the control and presentation data. You can add status information to your project's .rgs file.

See [IOleObject::GetMiscStatus](#) in the Windows SDK.

IOleObjectImpl::GetMoniker

Retrieves the control's moniker.

```
STDMETHOD(GetMoniker)(  
    DWORD /* dwAssign */,  
    DWORD /* dwWhichMoniker */,  
    IMoniker** /* ppmk */);
```

Return Value

Returns E_NOTIMPL.

Remarks

See [IOleObject::GetMoniker](#) in the Windows SDK.

IOleObjectImpl::GetUserClassID

Returns the control's class identifier.

```
STDMETHOD(GetUserClassID)(CLSID* pClsid);
```

Remarks

See [IOleObject::GetUserClassID](#) in the Windows SDK.

IOleObjectImpl::GetUserType

Returns the control's user-type name by calling `OleRegGetUserType`.

```
STDMETHOD(GetUserType)(  
    DWORD dwFormOfType,  
    LPOLESTR* pszUserType);
```

Remarks

The user-type name is used for display in user-interfaces elements such as menus and dialog boxes. You can change the user-type name in your project's .rgs file.

See [IOleObject::GetUserType](#) in the Windows SDK.

IOleObjectImpl::InitFromData

Initializes the control from selected data.

```
STDMETHOD(InitFromData)(  
    IDataObject* /* pDataObject */,  
    BOOL /* fCreation */,  
    DWORD /* dwReserved */);
```

Return Value

Returns E_NOTIMPL.

Remarks

See [IOleObject::InitFromData](#) in the Windows SDK.

IOleObjectImpl::IsUpToDate

Checks if the control is up to date.

```
STDMETHOD(IsUpToDate)(void);
```

Return Value

Returns S_OK.

Remarks

See [IOleObject::IsUpToDate](#) in the Windows SDK.

IOleObjectImpl::OnPostVerbDiscardUndo

Called by [DoVerbDiscardUndo](#) after the undo state is discarded.

```
HRESULT OnPostVerbDiscardUndo();
```

Return Value

Returns S_OK.

Remarks

Override this method with code you want executed after the undo state is discarded.

IOleObjectImpl::OnPostVerbHide

Called by [DoVerbHide](#) after the control is hidden.

```
HRESULT OnPostVerbHide();
```

Return Value

Returns S_OK.

Remarks

Override this method with code you want executed after the control is hidden.

IOleObjectImpl::OnPostVerbInPlaceActivate

Called by [DoVerbInPlaceActivate](#) after the control is activated in place.

```
HRESULT OnPostVerbInPlaceActivate();
```

Return Value

Returns S_OK.

Remarks

Override this method with code you want executed after the control is activated in place.

IOleObjectImpl::OnPostVerbOpen

Called by [DoVerbOpen](#) after the control has been opened for editing in a separate window.

```
HRESULT OnPostVerbOpen();
```

Return Value

Returns S_OK.

Remarks

Override this method with code you want executed after the control has been opened for editing in a separate window.

IOleObjectImpl::OnPostVerbShow

Called by [DoVerbShow](#) after the control has been made visible.

```
HRESULT OnPostVerbShow();
```

Return Value

Returns S_OK.

Remarks

Override this method with code you want executed after the control has been made visible.

IOleObjectImpl::OnPostVerbUIActivate

Called by [DoVerbUIActivate](#) after the control's user interface has been activated.

```
HRESULT OnPostVerbUIActivate();
```

Return Value

Returns S_OK.

Remarks

Override this method with code you want executed after the control's user interface has been activated.

IOleObjectImpl::OnPreVerbDiscardUndo

Called by [DoVerbDiscardUndo](#) before the undo state is discarded.

```
HRESULT OnPreVerbDiscardUndo();
```

Return Value

Returns S_OK.

Remarks

To prevent the undo state from being discarded, override this method to return an error HRESULT.

IOleObjectImpl::OnPreVerbHide

Called by [DoVerbHide](#) before the control is hidden.

```
HRESULT OnPreVerbHide();
```

Return Value

Returns S_OK.

Remarks

To prevent the control from being hidden, override this method to return an error HRESULT.

IOleObjectImpl::OnPreVerbInPlaceActivate

Called by [DoVerbInPlaceActivate](#) before the control is activated in place.

```
HRESULT OnPreVerbInPlaceActivate();
```

Return Value

Returns S_OK.

Remarks

To prevent the control from being activated in place, override this method to return an error HRESULT.

IOleObjectImpl::OnPreVerbOpen

Called by [DoVerbOpen](#) before the control has been opened for editing in a separate window.

```
HRESULT OnPreVerbOpen();
```

Return Value

Returns S_OK.

Remarks

To prevent the control from being opened for editing in a separate window, override this method to return an error HRESULT.

IOleObjectImpl::OnPreVerbShow

Called by [DoVerbShow](#) before the control has been made visible.

```
HRESULT OnPreVerbShow();
```

Return Value

Returns S_OK.

Remarks

To prevent the control from being made visible, override this method to return an error HRESULT.

IOleObjectImpl::OnPreVerbUIActivate

Called by [DoVerbUIActivate](#) before the control's user interface has been activated.

```
HRESULT OnPreVerbUIActivate();
```

Return Value

Returns S_OK.

Remarks

To prevent the control's user interface from being activated, override this method to return an error HRESULT.

IOleObjectImpl::SetClientSite

Tells the control about its client site in the container.

```
STDMETHOD(SetClientSite)(IOleClientSite* pClientSite);
```

Remarks

The method then returns S_OK.

See [IOleObject::SetClientSite](#) in the Windows SDK.

IOleObjectImpl::SetColorScheme

Recommends a color scheme to the control's application, if any.

```
STDMETHOD(SetColorScheme)(LOGPALETTE* /* pLogPal */);
```

Return Value

Returns E_NOTIMPL.

Remarks

See [IOleObject::SetColorScheme](#) in the Windows SDK.

IOleObjectImpl::SetExtent

Sets the extent of the control's display area.

```
STDMETHOD(SetExtent)(  
    DWORD dwDrawAspect,  
    SIZEL* psize1);
```

Remarks

Otherwise, `SetExtent` stores the value pointed to by `psize1` in the control class data member [CComControlBase::m_sizeExtent](#). This value is in HIMETRIC units (0.01 millimeter per unit).

If the control class data member [CComControlBase::m_bResizeNatural](#) is TRUE, `SetExtent` also stores the value pointed to by `psize1` in the control class data member [CComControlBase::m_sizeNatural](#).

If the control class data member [CComControlBase::m_bRecomposeOnResize](#) is TRUE, `SetExtent` calls `SendOnDataChange` and `SendOnViewChange` to notify all advisory sinks registered with the advise holder that the control size has changed.

See [IOleObject::SetExtent](#) in the Windows SDK.

IOleObjectImpl::SetHostNames

Tells the control the names of the container application and container document.

```
STDMETHOD(SetHostNames)(LPCOLESTR /* szContainerApp */, LPCOLESTR /* szContainerObj */);
```

Return Value

Returns S_OK.

Remarks

See [IOleObject::SetHostNames](#) in the Windows SDK.

IOleObjectImpl::SetMoniker

Tells the control what its moniker is.

```
STDMETHOD(SetMoniker)(
    DWORD /* dwWhichMoniker */,
    IMoniker** /* pmk */);
```

Return Value

Returns E_NOTIMPL.

Remarks

See [IOleObject::SetMoniker](#) in the Windows SDK.

IOleObjectImpl::Unadvise

Deletes the advisory connection stored in the control class's `m_spOleAdviseHolder` data member.

```
STDMETHOD(Unadvise)(DWORD dwConnection);
```

Remarks

See [IOleObject::Unadvise](#) in the Windows SDK.

IOleObjectImpl::Update

Updates the control.

```
STDMETHOD(Update)(void);
```

Return Value

Returns S_OK.

Remarks

See [IOleObject::Update](#) in the Windows SDK.

See also

[CComControl Class](#)

[ActiveX Controls Interfaces](#)

[Class Overview](#)

IPropertyBrowsingImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements `IUnknown` and allows a client to access the information in an object's property pages.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class T>
class ATL_NO_VTABLE IPropertyBrowsingImpl :
    public IPropertyBrowsing
```

Parameters

T

Your class, derived from `IPropertyBrowsingImpl`.

Members

Public Methods

NAME	DESCRIPTION
IPropertyBrowsingImpl::GetDisplayString	Retrieves a string describing a given property.
IPropertyBrowsingImpl::GetPredefinedStrings	Retrieves an array of strings corresponding to the values that a given property can accept.
IPropertyBrowsingImpl::GetPredefinedValue	Retrieves a VARIANT containing the value of a property identified by a given DISPID. The DISPID is associated with the string name retrieved from <code>GetPredefinedStrings</code> . The ATL implementation returns E_NOTIMPL.
IPropertyBrowsingImpl::MapPropertyToPage	Retrieves the CLSID of the property page associated with a given property.

Remarks

The [IPropertyBrowsing](#) interface allows a client to access the information in an object's property pages. Class `IPropertyBrowsingImpl` provides a default implementation of this interface and implements `IUnknown` by sending information to the dump device in debug builds.

NOTE

If you are using Microsoft Access as the container application, you must derive your class from `IPropertyBrowsingImpl`. Otherwise, Access will not load your control.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy

IPropertyBrowsing

IPropertyBrowsingImpl

Requirements

Header: atlctl.h

IPropertyBrowsingImpl::GetDisplayString

Retrieves a string describing a given property.

```
STDMETHOD(GetDisplayString)(  
    DISPID dispID,  
    BSTR* pBstr);
```

Remarks

See [IPropertyBrowsing::GetDisplayString](#) in the Windows SDK.

IPropertyBrowsingImpl::GetPredefinedStrings

Fills each array with zero items.

```
STDMETHOD(GetPredefinedStrings)(  
    DISPID dispID,  
    CALPOLESTR* pCaStringsOut,  
    CADWORD* pCaCookiesOut);
```

Return Value

ATL's implementation of [GetPredefinedValue](#) returns E_NOTIMPL.

Remarks

See [IPropertyBrowsing::GetPredefinedStrings](#) in the Windows SDK.

IPropertyBrowsingImpl::GetPredefinedValue

Retrieves a VARIANT containing the value of a property identified by a given DISPID. The DISPID is associated with the string name retrieved from [GetPredefinedStrings](#).

```
STDMETHOD(GetPredefinedValue)(  
    DISPID dispID,  
    DWORD dwCookie,  
    VARIANT* pVarOut);
```

Return Value

Returns E_NOTIMPL.

Remarks

ATL's implementation of [GetPredefinedStrings](#) retrieves no corresponding strings.

See [IPerPropertyBrowsing::GetPredefinedValue](#) in the Windows SDK.

IPerPropertyBrowsingImpl::MapPropertyToPage

Retrieves the CLSID of the property page associated with the specified property.

```
STDMETHOD(MapPropertyToPage)(  
    DISPID dispID,  
    CLSID* pClsid);
```

Remarks

ATL uses the object's property map to obtain this information.

See [IPerPropertyBrowsing::MapPropertyToPage](#) in the Windows SDK.

See also

[IPropertyPageImpl Class](#)

[ISpecifyPropertyPagesImpl Class](#)

[Class Overview](#)

IPersistPropertyBagImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements `IUnknown` and allows an object to save its properties to a client-supplied property bag.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class T>
class ATL_NO_VTABLE IPersistPropertyBagImpl : public IPersistPropertyBag
```

Parameters

T

Your class, derived from `IPersistPropertyBagImpl`.

Members

Public Methods

NAME	DESCRIPTION
IPersistPropertyBagImpl::GetClassID	Retrieves the object's CLSID.
IPersistPropertyBagImpl::InitNew	Initializes a newly created object. The ATL implementation returns S_OK.
IPersistPropertyBagImpl::Load	Loads the object's properties from a client-supplied property bag.
IPersistPropertyBagImpl::Save	Saves the object's properties into a client-supplied property bag.

Remarks

The [IPersistPropertyBag](#) interface allows an object to save its properties to a client-supplied property bag. Class `IPersistPropertyBagImpl` provides a default implementation of this interface and implements `IUnknown` by sending information to the dump device in debug builds.

`IPersistPropertyBag` works in conjunction with [IPropertyBag](#) and [IErrorLog](#). These latter two interfaces must be implemented by the client. Through `IPropertyBag`, the client saves and loads the object's individual properties. Through `IErrorLog`, both the object and the client can report any errors encountered.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy

IPersistPropertyBag

IPersistPropertyBagImpl

Requirements

Header: atlcom.h

IPersistPropertyBagImpl::GetClassID

Retrieves the object's CLSID.

```
STDMETHOD(GetClassID)(CLSID* pClassID);
```

Remarks

See [IPersist::GetClassID](#) in the Windows SDK.

IPersistPropertyBagImpl::InitNew

Initializes a newly created object.

```
STDMETHOD(InitNew)();
```

Return Value

Returns S_OK.

Remarks

See [IPersistPropertyBag::InitNew](#) in the Windows SDK.

IPersistPropertyBagImpl::Load

Loads the object's properties from a client-supplied property bag.

```
STDMETHOD(Load)(LPPROPERTYBAG pPropBag, LPERRORLOG pErrorLog);
```

Remarks

ATL uses the object's property map to retrieve this information.

See [IPersistPropertyBag::Load](#) in the Windows SDK.

IPersistPropertyBagImpl::Save

Saves the object's properties into a client-supplied property bag.

```
STDMETHOD(Save)(  
    LPPROPERTYBAG pPropBag,  
    BOOL fClearDirty,  
    BOOL fSaveAllProperties);
```

Remarks

ATL uses the object's property map to store this information. By default, this method saves all properties, regardless of the value of *fSaveAllProperties*.

See [IPersistPropertyBag::Save](#) in the Windows SDK.

See also

[BEGIN_PROP_MAP](#)

[Class Overview](#)

IPersistStorageImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements the [IPersistStorage](#) interface.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class T>
class ATL_NO_VTABLE IPersistStorageImpl : public IPersistStorage
```

Parameters

T

Your class, derived from `IPersistStorageImpl`.

Members

Public Methods

NAME	DESCRIPTION
IPersistStorageImpl::GetClassID	Retrieves the object's CLSID.
IPersistStorageImpl::HandsOffStorage	Instructs the object to release all storage objects and enter HandsOff mode. The ATL implementation returns S_OK.
IPersistStorageImpl::InitNew	Initializes a new storage.
IPersistStorageImpl::IsDirty	Checks whether the object's data has changed since it was last saved.
IPersistStorageImpl::Load	Loads the object's properties from the specified storage.
IPersistStorageImpl::Save	Saves the object's properties to the specified storage.
IPersistStorageImpl::SaveCompleted	Notifies an object that it can return to Normal mode to write to its storage object. The ATL implementation returns S_OK.

Remarks

`IPersistStorageImpl` implements the [IPersistStorage](#) interface, which allows a client to request that your object load and save its persistent data using a storage.

The implementation of this class requires class `T` to make an implementation of the `IPersistStreamInit` interface available via `QueryInterface`. Typically this means that class `T` should derive from [IPersistStreamInitImpl](#), provide an entry for `IPersistStreamInit` in the [COM map](#), and use a [property map](#) to describe the class's persistent data.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy

IPersistStorage

IPersistStorageImpl

Requirements

Header: atlcom.h

IPersistStorageImpl::GetClassID

Retrieves the object's CLSID.

```
STDMETHOD(GetClassID)(CLSID* pClassID);
```

Remarks

See [IPersist::GetClassID](#) in the Windows SDK.

IPersistStorageImpl::HandsOffStorage

Instructs the object to release all storage objects and enter HandsOff mode.

```
STDMETHOD(HandsOffStorage)(void);
```

Return Value

Returns S_OK.

Remarks

See [IPersistStorage::HandsOffStorage](#) in the Windows SDK.

IPersistStorageImpl::InitNew

Initializes a new storage.

```
STDMETHOD(InitNew)(IStorage*);
```

Remarks

The ATL implementation delegates to the [IPersistStreamInit](#) interface.

See [IPersistStorage::InitNew](#) in the Windows SDK.

IPersistStorageImpl::IsDirty

Checks whether the object's data has changed since it was last saved.

```
STDMETHOD(IsDirty)(void);
```

Remarks

The ATL implementation delegates to the [IPersistStreamInit](#) interface.

See [IPersistStorage:IsDirty](#) in the Windows SDK.

IPersistStorageImpl::Load

Loads the object's properties from the specified storage.

```
STDMETHOD(Load)(IStorage* pStorage);
```

Remarks

The ATL implementation delegates to the [IPersistStreamInit](#) interface. `Load` uses a stream named "Contents" to retrieve the object's data. The [Save](#) method originally creates this stream.

See [IPersistStorage:Load](#) in the Windows SDK.

IPersistStorageImpl::Save

Saves the object's properties to the specified storage.

```
STDMETHOD(Save)(IStorage* pStorage, BOOL fSameAsLoad);
```

Remarks

The ATL implementation delegates to the [IPersistStreamInit](#) interface. When `Save` is first called, it creates a stream named "Contents" on the specified storage. This stream is then used in later calls to `Save` and in calls to [Load](#).

See [IPersistStorage:Save](#) in the Windows SDK.

IPersistStorageImpl::SaveCompleted

Notifies an object that it can return to Normal mode to write to its storage object.

```
STDMETHOD(SaveCompleted)(IStorage*);
```

Return Value

Returns S_OK.

Remarks

See [IPersistStorage:SaveCompleted](#) in the Windows SDK.

See also

[Storages and Streams](#)

[IPersistStreamInitImpl Class](#)

[IPersistPropertyBagImpl Class](#)

[Class Overview](#)

IPersistStreamInitImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements `IUnknown` and provides a default implementation of the [IPersistStreamInit](#) interface.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class T>
class ATL_NO_VTABLE IPersistStreamInitImpl
: public IPersistStreamInit
```

Parameters

T

Your class, derived from `IPersistStreamInitImpl`.

Members

Public Methods

NAME	DESCRIPTION
IPersistStreamInitImpl::GetClassID	Retrieves the object's CLSID.
IPersistStreamInitImpl::GetSizeMax	Retrieves the size of the stream needed to save the object's data. The ATL implementation returns E_NOTIMPL.
IPersistStreamInitImpl::InitNew	Initializes a newly created object.
IPersistStreamInitImpl::IsDirty	Checks whether the object's data has changed since it was last saved.
IPersistStreamInitImpl::Load	Loads the object's properties from the specified stream.
IPersistStreamInitImpl::Save	Saves the object's properties to the specified stream.

Remarks

The [IPersistStreamInit](#) interface allows a client to request that your object loads and saves its persistent data to a single stream. Class `IPersistStreamInitImpl` provides a default implementation of this interface and implements `IUnknown` by sending information to the dump device in debug builds.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy

IPersistStreamInit

IPersistStreamInitImpl

Requirements

Header: atlcom.h

IPersistStreamInitImpl::GetClassID

Retrieves the object's CLSID.

```
STDMETHOD(GetClassID)(CLSID* pClassID);
```

Remarks

See [IPersist::GetClassID](#) in the Windows SDK.

IPersistStreamInitImpl::GetSizeMax

Retrieves the size of the stream needed to save the object's data.

```
STDMETHOD(GetSizeMax)(ULARGE_INTEGER FAR* pcbSize);
```

Return Value

Returns E_NOTIMPL.

Remarks

See [IPersistStreamInit::GetSizeMax](#) in the Windows SDK.

IPersistStreamInitImpl::InitNew

Initializes a newly created object.

```
STDMETHOD(InitNew)();
```

Remarks

See [IPersistStreamInit::InitNew](#) in the Windows SDK.

IPersistStreamInitImpl::IsDirty

Checks whether the object's data has changed since it was last saved.

```
STDMETHOD(IsDirty)();
```

Remarks

See [IPersistStreamInit::IsDirty](#) in the Windows SDK.

IPersistStreamInitImpl::Load

Loads the object's properties from the specified stream.

```
STDMETHOD(Load)(LPSTREAM pStm);
```

Remarks

ATL uses the object's property map to retrieve this information.

See [IPersistStreamInit::Load](#) in the Windows SDK.

IPersistStreamInitImpl::Save

Saves the object's properties to the specified stream.

```
STDMETHOD(Save)(LPSTREAM pStm, BOOL fClearDirty);
```

Remarks

ATL uses the object's property map to store this information.

See [IPersistStreamInit::Save](#) in the Windows SDK.

See also

[Storages and Streams](#)

[Class Overview](#)

IPointerInactiveImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements `IUnknown` and the `IPointerInactive` interface methods.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class T>
class IPointerInactiveImpl
```

Parameters

T

Your class, derived from `IPointerInactiveImpl`.

Members

Public Methods

NAME	DESCRIPTION
IPointerInactiveImpl::GetActivationPolicy	Retrieves the current activation policy for the object. The ATL implementation returns E_NOTIMPL.
IPointerInactiveImpl::OnInactiveMouseMove	Notifies the object that the mouse pointer has moved over it, indicating the object can fire mouse events. The ATL implementation returns E_NOTIMPL.
IPointerInactiveImpl::OnInactiveSetCursor	Sets the mouse pointer for the inactive object. The ATL implementation returns E_NOTIMPL.

Remarks

An inactive object is one that is simply loaded or running. Unlike an active object, an inactive object cannot receive Windows mouse and keyboard messages. Thus, inactive objects use fewer resources and are typically more efficient.

The `IPointerInactive` interface allows an object to support a minimal level of mouse interaction while remaining inactive. This functionality is particularly useful for controls.

Class `IPointerInactiveImpl` implements the `IPointerInactive` methods by simply returning E_NOTIMPL. However, it implements `IUnknown` by sending information to the dump device in debug builds.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy

IPoInterInactive

IPoInterInactiveImpl

Requirements

Header: atlctl.h

IPoInterInactiveImpl::GetActivationPolicy

Retrieves the current activation policy for the object.

```
HRESULT GetActivationPolicy(DWORD* pdwPolicy);
```

Return Value

Returns E_NOTIMPL.

Remarks

See [IPoInterInactive::GetActivationPolicy](#) in the Windows SDK.

IPoInterInactiveImpl::OnInactiveMouseMove

Notifies the object that the mouse pointer has moved over it, indicating the object can fire mouse events.

```
HRESULT OnInactiveMouseMove(  
    LPCRECT pRectBounds,  
    long x,  
    long y,  
    DWORD dwMouseMsg);
```

Return Value

Returns E_NOTIMPL.

Remarks

See [IPoInterInactive::OnInactiveMouseMove](#) in the Windows SDK.

IPoInterInactiveImpl::OnInactiveSetCursor

Sets the mouse pointer for the inactive object.

```
HRESULT OnInactiveSetCursor(  
    LPCRECT pRectBounds,  
    long x,  
    long y,  
    DWORD dwMouseMsg,  
    BOOL fSetAlways);
```

Return Value

Returns E_NOTIMPL.

Remarks

See [IPoInterInactive::OnInactiveSetCursor](#) in the Windows SDK.

See also

[Class Overview](#)

IPROPERTYNotifySinkCP Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class exposes [IPROPERTYNotifySink](#) interface as an outgoing interface on a connectable object.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class T, class CDV = CComDynamicUnkArray>
class IPROPERTYNotifySinkCP
: public IConnectionPointImpl<T, &IID_IPROPERTYNotifySink, CDV>
```

Parameters

T

Your class, derived from `IPROPERTYNotifySinkCP`.

CDV

A class that manages the connections between a connection point and its sinks. The default value is [CComDynamicUnkArray](#), which allows unlimited connections. You can also use [CComUnkArray](#), which specifies a fixed number of connections.

Remarks

`IPROPERTYNotifySinkCP` inherits all methods through its base class, [IConnectionPointImpl](#).

The [IPROPERTYNotifySink](#) interface allows a sink object to receive notifications about property changes. Class `IPROPERTYNotifySinkCP` exposes this interface as an outgoing interface on a connectable object. The client must implement the `IPROPERTYNotifySink` methods on the sink.

Derive your class from `IPROPERTYNotifySinkCP` when you want to create a connection point that represents the `IPROPERTYNotifySink` interface.

For more information about using connection points in ATL, see the article [Connection Points](#).

Requirements

Header: atlctl.h

See also

[IConnectionPointImpl Class](#)

[IConnectionPointContainerImpl Class](#)

[Class Overview](#)

IPropertyPage2Impl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements `IUnknown` and inherits the default implementation of [IPropertyPageImpl](#).

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class T>
class IPropertyPage2Impl : public IPropertyPageImpl<T>
```

Parameters

T

Your class, derived from `IPropertyPage2Impl`.

Members

Public Methods

NAME	DESCRIPTION
IPropertyPage2Impl::EditProperty	Specifies which property control will receive the focus when the property page is activated. The ATL implementation returns E_NOTIMPL.

Remarks

The [IPropertyPage2](#) interface extends [IPropertyPage](#) by adding the `EditProperty` method. This method allows a client to select a specific property in a property page object.

Class `IPropertyPage2Impl` simply returns E_NOTIMPL for `IPropertyPage2::EditProperty`. However, it inherits the default implementation of [IPropertyPageImpl](#) and implements `IUnknown` by sending information to the dump device in debug builds.

When you create a property page, your class is typically derived from `IPropertyPageImpl`. To provide the extra support of `IPropertyPage2`, modify your class definition and override the `EditProperty` method.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy

`IPropertyPage`

[IPropertyPageImpl](#)

`IPropertyPage2Impl`

Requirements

Header: atlctl.h

IPropertyPage2Impl::EditProperty

Specifies which property control will receive the focus when the property page is activated.

```
HRESULT EditProperty(DISPID dispID);
```

Return Value

Returns E_NOTIMPL.

Remarks

See [IPropertyPage2::EditProperty](#) in the Windows SDK.

See also

[IPropertyPage2Impl Class](#)

[IPropertyPage2Impl Class](#)

[Class Overview](#)

IPropertyPageImpl Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

This class implements `IUnknown` and provides a default implementation of the [IPropertyPage](#) interface.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class T>
class IPropertyPageImpl
```

Parameters

T

Your class, derived from `IPropertyPageImpl`.

Members

Public Constructors

NAME	DESCRIPTION
IPropertyPageImpl::IPropertyPageImpl	Constructor.

Public Methods

NAME	DESCRIPTION
IPropertyPageImpl::Activate	Creates the dialog box window for the property page.
IPropertyPageImpl::Apply	Applies current property page values to the underlying objects specified through <code>SetObjects</code> . The ATL implementation returns S_OK.
IPropertyPageImpl::Deactivate	Destroys the window created with <code>Activate</code> .
IPropertyPageImpl::GetPageInfo	Retrieves information about the property page.
IPropertyPageImpl::Help	Invokes Windows help for the property page.
IPropertyPageImpl::IsPageDirty	Indicates whether the property page has changed since it was activated.
IPropertyPageImpl::Move	Positions and resizes the property page dialog box.
IPropertyPageImpl::SetDirty	Flags the property page's state as changed or unchanged.

NAME	DESCRIPTION
IPropertyPageImpl::SetObjects	Provides an array of <code>IUnknown</code> pointers for the objects associated with the property page. These objects receive the current property page values through a call to <code>Apply</code> .
IPropertyPageImpl::SetPageSite	Provides the property page with an <code>IPropertyPageSite</code> pointer, through which the property page communicates with the property frame.
IPropertyPageImpl::Show	Makes the property page dialog box visible or invisible.
IPropertyPageImpl::TranslateAccelerator	Processes a specified keystroke.

Public Data Members

NAME	DESCRIPTION
IPropertyPageImpl::m_bDirty	Specifies whether the property page's state has changed.
IPropertyPageImpl::m_dwDocString	Stores the resource identifier associated with the text string describing the property page.
IPropertyPageImpl::m_dwHelpContext	Stores the context identifier for the help topic associated with the property page.
IPropertyPageImpl::m_dwHelpFile	Stores the resource identifier associated with the name of the help file describing the property page.
IPropertyPageImpl::m_dwTitle	Stores the resource identifier associated with the text string that appears in the tab for the property page.
IPropertyPageImpl::m_nObjects	Stores the number of objects associated with the property page.
IPropertyPageImpl::m_pPageSite	Points to the <code>IPropertyPageSite</code> interface through which the property page communicates with the property frame.
IPropertyPageImpl::m_ppUnk	Points to an array of <code>IUnknown</code> pointers to the objects associated with the property page.
IPropertyPageImpl::m_size	Stores the height and width of the property page's dialog box, in pixels.

Remarks

The [IPropertyPage](#) interface allows an object to manage a particular property page within a property sheet. Class `IPropertyPageImpl` provides a default implementation of this interface and implements `IUnknown` by sending information to the dump device in debug builds.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy

`IPropertyPage`

Requirements

Header: atlctl.h

IPropertyPageImpl::Activate

Creates the dialog box window for the property page.

```
HRESULT Activate(  
    HWND hWndParent,  
    LPCRECT pRect,  
    BOOL bModal);
```

Remarks

By default, the dialog box is always modeless, regardless of the value of the *bModal* parameter.

See [IPropertyPage::Activate](#) in the Windows SDK.

IPropertyPageImpl::Apply

Applies current property page values to the underlying objects specified through `SetObjects`.

```
HRESULT Apply();
```

Return Value

Returns S_OK.

Remarks

See [IPropertyPage::Apply](#) in the Windows SDK.

IPropertyPageImpl::Deactivate

Destroys the dialog box window created with [Activate](#).

```
HRESULT Deactivate();
```

Remarks

See [IPropertyPage::Deactivate](#) in the Windows SDK.

IPropertyPageImpl::GetPageInfo

Fills the *pPageInfo* structure with information contained in the data members.

```
HRESULT GetPageInfo(PROPPAGEINFO* pPageInfo);
```

Remarks

`GetPageInfo` loads the string resources associated with `m_dwDocString`, `m_dwHelpFile`, and `m_dwTitle`.

See [IPropertyPage::GetPageInfo](#) in the Windows SDK.

IPropertyPageImpl::Help

Invokes Windows help for the property page.

```
HRESULT Help(PROPPAGEINFO* pPageInfo);
```

Remarks

See [IPropertyPage::Help](#) in the Windows SDK.

IPropertyPageImpl::IPropertyPageImpl

The constructor.

```
IPropertyPageImpl();
```

Remarks

Initializes all data members.

IPropertyPageImpl::IsPageDirty

Indicates whether the property page has changed since it was activated.

```
HRESULT IsPageDirty(void);
```

Remarks

`IsPageDirty` returns `S_OK` if the page has changed since it was activated.

IPropertyPageImpl::m_bDirty

Specifies whether the property page's state has changed.

```
BOOL m_bDirty;
```

IPropertyPageImpl::m_nObjects

Stores the number of objects associated with the property page.

```
ULONG m_nObjects;
```

IPropertyPageImpl::m_dwHelpContext

Stores the context identifier for the help topic associated with the property page.

```
DWORD m_dwHelpContext;
```

IPropertyPageImpl::m_dwDocString

Stores the resource identifier associated with the text string describing the property page.

```
UINT m_dwDocString;
```

IPropertyPageImpl::m_dwHelpFile

Stores the resource identifier associated with the name of the help file describing the property page.

```
UINT m_dwHelpFile;
```

IPropertyPageImpl::m_dwTitle

Stores the resource identifier associated with the text string that appears in the tab for the property page.

```
UINT m_dwTitle;
```

IPropertyPageImpl::m_pPageSite

Points to the [IPropertyPageSite](#) interface through which the property page communicates with the property frame.

```
IPropertyPageSite* m_pPageSite;
```

IPropertyPageImpl::m_ppUnk

Points to an array of [IUnknown](#) pointers to the objects associated with the property page.

```
IUnknown** m_ppUnk;
```

IPropertyPageImpl::m_size

Stores the height and width of the property page's dialog box, in pixels.

```
SIZE m_size;
```

IPropertyPageImpl::Move

Positions and resizes the property page dialog box.

```
HRESULT Move(LPCRECT pRect);
```

Remarks

See [IPropertyPage::Move](#) in the Windows SDK.

IPropertyPageImpl::SetDirty

Flags the property page's state as changed or unchanged, depending on the value of *bDirty*.

```
void SetDirty(BOOL bDirty);
```

Parameters

bDirty

[in] If TRUE, the property page's state is marked as changed. Otherwise, it is marked as unchanged.

Remarks

If necessary, `SetDirty` informs the frame that the property page has changed.

IPropertyPageImpl::SetObjects

Provides an array of `IUnknown` pointers for the objects associated with the property page.

```
HRESULT SetObjects(ULONG nObjects, IUnknown** ppUnk);
```

Remarks

See [IPropertyPage::SetObjects](#) in the Windows SDK.

IPropertyPageImpl::SetPageSite

Provides the property page with an [IPropertyPageSite](#) pointer, through which the property page communicates with the property frame.

```
HRESULT SetPageSite(IPropertyPageSite* pPageSite);
```

Remarks

See [IPropertyPage::SetPageSite](#) in the Windows SDK.

IPropertyPageImpl::Show

Makes the property page dialog box visible or invisible.

```
HRESULT Show(UINT nCmdShow);
```

Remarks

See [IPropertyPage::Show](#) in the Windows SDK.

IPropertyPageImpl::TranslateAccelerator

Processes the keystroke specified in `pMsg`.

```
HRESULT TranslateAccelerator(MSG* pMsg);
```

Remarks

See [IPropertyPage::TranslateAccelerator](#) in the Windows SDK.

See also

[IPropertyPage2Impl Class](#)

[IPerPropertyBrowsingImpl Class](#)

[ISpecifyPropertyPagesImpl Class](#)

[Class Overview](#)

IProvideClassInfo2Impl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides a default implementation of the [IProvideClassInfo](#) and [IProvideClassInfo2](#) methods.

Syntax

```
template <const CLSID* pcoclsid,
    const IID* psrcid,
    const GUID* plibid = &CAtlModule::m_libid,
    WORD wMajor = 1,
    WORD wMinor = 0, class tihclass = CComTypeInfoHolder>
class ATL_NO_VTABLE IProvideClassInfo2Impl : public IProvideClassInfo2
```

Parameters

pcoclsid

A pointer to the coclass' identifier.

psrcid

A pointer to the identifier for the coclass' default outgoing dispinterface.

plibid

A pointer to the LIBID of the type library that contains information about the interface. By default, the server-level type library is passed.

wMajor

The major version of the type library. The default value is 1.

wMinor

The minor version of the type library. The default value is 0.

tihclass

The class used to manage the coclass' type information. The default value is `CComTypeInfoHolder`.

Members

Constructors

NAME	DESCRIPTION
IProvideClassInfo2Impl::IProvideClassInfo2Impl	Constructor.

Public Methods

NAME	DESCRIPTION
IProvideClassInfo2Impl::GetClassInfo	Retrieves an <code>ITypeInfo</code> pointer to the coclass' type information.
IProvideClassInfo2Impl::GetGUID	Retrieves the GUID for the object's outgoing dispinterface.

Protected Data Members

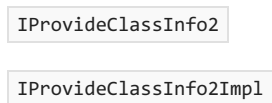
NAME	DESCRIPTION
IProvideClassInfo2Impl::_tih	Manages the type information for the coclass.

Remarks

The [IProvideClassInfo2](#) interface extends [IProvideClassInfo](#) by adding the `GetGUID` method. This method allows a client to retrieve an object's outgoing interface IID for its default event set. Class `IProvideClassInfo2Impl` provides a default implementation of the `IProvideClassInfo` and `IProvideClassInfo2` methods.

`IProvideClassInfo2Impl` contains a static member of type `CComTypeInfoHolder` that manages the type information for the coclass.

Inheritance Hierarchy



Requirements

Header: atlcom.h

IProvideClassInfo2Impl::GetClassInfo

Retrieves an `ITypeInfo` pointer to the coclass' type information.

```
STDMETHOD(GetClassInfo)(ITypeInfo** pptinfo);
```

Remarks

See [IProvideClassInfo::GetClassInfo](#) in the Windows SDK.

IProvideClassInfo2Impl::GetGUID

Retrieves the GUID for the object's outgoing dispinterface.

```
STDMETHOD(GetGUID)(
    DWORD dwGuidKind,
    GUID* pGUID);
```

Remarks

See [IProvideClassInfo2::GetGUID](#) in the Windows SDK.

IProvideClassInfo2Impl::IProvideClassInfo2Impl

The constructor.

```
IProvideClassInfo2Impl();
```

Remarks

Calls `AddRef` on the `_tih` member. The destructor calls `Release`.

IProvideClassInfo2Impl::_tih

This static data member is an instance of the class template parameter, *tihclass*, which by default is

`CComTypeInfoHolder` .

```
static tihclass  
    _tih;
```

Remarks

`_tih` manages the type information for the coclass.

See also

[Class Overview](#)

IQuickActivateImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class combines containers' control initialization into a single call.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template <class T>
class ATL_NO_VTABLE IQuickActivateImpl : public IQuickActivate
```

Parameters

T

Your class, derived from `IQuickActivateImpl`.

Members

Public Methods

NAME	DESCRIPTION
IQuickActivateImpl::GetContentExtent	Retrieves the current display size for a running control.
IQuickActivateImpl::QuickActivate	Performs quick initialization of controls being loaded.
IQuickActivateImpl::SetContentExtent	Informs the control of how much display space the container has assigned to it.

Remarks

The [IQuickActivate](#) interface helps containers avoid delays when loading controls by combining initialization in a single call. The `QuickActivate` method allows the container to pass a pointer to a [QACONTAINER](#) structure that holds pointers to all the interfaces the control needs. On return, the control passes back a pointer to a [QACONTROL](#) structure that holds pointers to its own interfaces, which are used by the container. Class `IQuickActivateImpl` provides a default implementation of `IQuickActivate` and implements `IUnknown` by sending information to the dump device in debug builds.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy

`IQuickActivate`

`IQuickActivateImpl`

Requirements

Header: atlctl.h

IQuickActivateImpl::GetContentExtent

Retrieves the current display size for a running control.

```
STDMETHOD(GetContentExtent)(LPSIZEL pSize);
```

Remarks

The size is for a full rendering of the control and is specified in HIMETRIC units.

See [IQuickActivate::GetContentExtent](#) in the Windows SDK.

IQuickActivateImpl::QuickActivate

Performs quick initialization of controls being loaded.

```
STDMETHOD(QuickActivate)(  
    QACONTAINER* pQACont,  
    QACONTROL* pQACtrl);
```

Remarks

The structure contains pointers to interfaces needed by the control and the values of some ambient properties. Upon return, the control passes a pointer to a [QACONTROL](#) structure that contains pointers to its own interfaces that the container requires, and additional status information.

See [IQuickActivate::QuickActivate](#) in the Windows SDK.

IQuickActivateImpl::SetContentExtent

Informs the control of how much display space the container has assigned to it.

```
STDMETHOD(SetContentExtent)(LPSIZEL pSize);
```

Remarks

The size is specified in HIMETRIC units.

See [IQuickActivate::SetContentExtent](#) in the Windows SDK.

See also

[CComControl Class](#)

[Class Overview](#)

IRegistrar Interface

3/4/2019 • 2 minutes to read • [Edit Online](#)

This interface is defined in `atliface.h` and is used internally by `CAtlModule` member functions such as [UpdateRegistryFromResourceD](#).

Syntax

```
typedef interface IRegistrar IRegistrar;
```

Remarks

See the topic [Using Replaceable Parameters \(The Registrar's Preprocessor\)](#) for more details.

Members

Public Methods

NAME	DESCRIPTION
IRegistrar::ResourceRegisterSz	Registers the resource.
IRegistrar::ResourceUnregisterSz	Unregisters the resource.
IRegistrar::FileRegister	Registers the file.
IRegistrar::FileUnregister	Unregisters the file.
IRegistrar::StringRegister	Registers the string.
IRegistrar::StringUnregister	Unregisters the string
IRegistrar::ResourceRegister	Registers the resource.
IRegistrar::ResourceUnregister	Unregisters the resource.

Requirements

Header: `atliface.h`

IRegistrar::ResourceRegisterSz

Registers the resource.

```
virtual HRESULT STDMETHODCALLTYPE ResourceRegisterSz(  
    /* [in] */ _In_z_ LPCOLESTR resFileName,  
    /* [in] */ _In_z_ LPCOLESTR szID,  
    /* [in] */ _In_z_ LPCOLESTR szType) = 0;
```

IRegistrar::ResourceUnregisterSz

Unregisters the resource.

```
virtual HRESULT STDMETHODCALLTYPE ResourceUnregisterSz(  
    /* [in] */ _In_z_ LPCOLESTR resFileName,  
    /* [in] */ _In_z_ LPCOLESTR szID,  
    /* [in] */ _In_z_ LPCOLESTR szType) = 0;
```

IRegistrar::FileRegister

Registers the file.

```
virtual HRESULT STDMETHODCALLTYPE FileRegister(  
    /* [in] */ _In_z_ LPCOLESTR fileName) = 0;
```

IRegistrar::FileUnregister

Unregisters the file.

```
virtual HRESULT STDMETHODCALLTYPE FileUnregister(  
    /* [in] */ _In_z_ LPCOLESTR fileName) = 0;
```

IRegistrar::StringRegister

Registers the specified string data.

```
virtual HRESULT STDMETHODCALLTYPE StringRegister(  
    /* [in] */ _In_z_ LPCOLESTR data) = 0;
```

IRegistrar::StringUnregister

Unregisters the specified string data.

```
virtual HRESULT STDMETHODCALLTYPE StringUnregister(  
    /* [in] */ _In_z_ LPCOLESTR data) = 0;
```

IRegistrar::ResourceRegister

Registers the resource.

```
virtual HRESULT STDMETHODCALLTYPE ResourceRegister(  
    /* [in] */ _In_z_ LPCOLESTR resFileName,  
    /* [in] */ _In_ UINT nID,  
    /* [in] */ _In_z_ LPCOLESTR szType) = 0;
```

IRegistrar::ResourceUnregister

Unregisters the resource.


```
virtualHRESULT STDMETHODCALLTYPE ResourceUnregister(  
    /* [in] */ _In_z_ LPCOLESTR resFileName,  
    /* [in] */ _In_ UINT nID,  
    /* [in] */ _In_z_ LPCOLESTR szType) = 0;
```

See also

[Using Replaceable Parameters \(The Registrar's Preprocessor\)](#)

[Class Overview](#)

[Module Classes](#)

[Registry Component \(Registrar\)](#)

IRunnableObjectImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements `IUnknown` and provides a default implementation of the `IRunnableObject` interface.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class T>
class IRunnableObjectImpl
```

Parameters

T

Your class, derived from `IRunnableObjectImpl`.

Members

Public Methods

NAME	DESCRIPTION
IRunnableObjectImpl::GetRunningClass	Returns the CLSID of the running control. The ATL implementation sets the CLSID to GUID_NULL and returns E_UNEXPECTED.
IRunnableObjectImpl::IsRunning	Determines if the control is running. The ATL implementation returns TRUE.
IRunnableObjectImpl::LockRunning	Locks the control into the running state. The ATL implementation returns S_OK.
IRunnableObjectImpl::Run	Forces the control to run. The ATL implementation returns S_OK.
IRunnableObjectImpl::SetContainedObject	Indicates that the control is embedded. The ATL implementation returns S_OK.

Remarks

The `IRunnableObject` interface enables a container to determine if a control is running, force it to run, or lock it into the running state. Class `IRunnableObjectImpl` provides a default implementation of this interface and implements `IUnknown` by sending information to the dump device in debug builds.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy

IRunnableObject

IRunnableObjectImpl

Requirements

Header: atlctl.h

IRunnableObjectImpl::GetRunningClass

Returns the CLSID of the running control.

```
HRESULT GetRunningClass(LPCLSID lpClsid);
```

Return Value

The ATL implementation sets * *lpClsid* to GUID_NULL and returns E_UNEXPECTED.

Remarks

See [IRunnableObject::GetRunningClass](#) in the Windows SDK.

IRunnableObjectImpl::IsRunning

Determines if the control is running.

```
virtual BOOL IsRunning();
```

Return Value

The ATL implementation returns TRUE.

Remarks

See [IRunnableObject::IsRunning](#) in the Windows SDK.

IRunnableObjectImpl::LockRunning

Locks the control into the running state.

```
HRESULT LockRunning(BOOL fLock, BOOL fLastUnlockCloses);
```

Return Value

The ATL implementation returns S_OK.

Remarks

See [IRunnableObject::LockRunning](#) in the Windows SDK.

IRunnableObjectImpl::Run

Forces the control to run.

```
HRESULT Run(LPBINDCTX lpbc);
```

Return Value

The ATL implementation returns S_OK.

Remarks

See [IRunnableObject::Run](#) in the Windows SDK.

IRunnableObjectImpl::SetContainedObject

Indicates that the control is embedded.

```
HRESULT SetContainedObject(BOOL fContained);
```

Return Value

The ATL implementation returns S_OK.

Remarks

See [IRunnableObject::SetContainedObject](#) in the Windows SDK.

See also

[CComControl Class](#)

[Class Overview](#)

IServiceProviderImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides a default implementation of the `IServiceProvider` interface.

Syntax

```
template <class T>
class ATL_NO_VTABLE IServiceProviderImpl : public IServiceProvider
```

Parameters

T

Your class, derived from `IServiceProviderImpl`.

Members

Public Methods

NAME	DESCRIPTION
IServiceProviderImpl::QueryService	Creates or accesses the specified service and returns an interface pointer to the specified interface for the service.

Remarks

The `IServiceProvider` interface locates a service specified by its GUID and returns the interface pointer for the requested interface on the service. Class `IServiceProviderImpl` provides a default implementation of this interface.

`IServiceProviderImpl` specifies one method: [QueryService](#), which creates or accesses the specified service and returns an interface pointer to the specified interface for the service.

`IServiceProviderImpl` uses a service map, starting with [BEGIN_SERVICE_MAP](#) and ending with [END_SERVICE_MAP](#).

The service map contains two entries: [SERVICE_ENTRY](#), which indicates a specified service id (SID) supported by the object, and [SERVICE_ENTRY_CHAIN](#), which calls `QueryService` to chain to another object.

Inheritance Hierarchy

`IServiceProvider`

`IServiceProviderImpl`

Requirements

Header: atlcom.h

IServiceProviderImpl::QueryService

Creates or accesses the specified service and returns an interface pointer to the specified interface for the service.

```
STDMETHOD(QueryService)(
    REFGUID guidService,
    REFIID riid,
    void** ppvObject);
```

Parameters

guidService

[in] Pointer to a service identifier (SID).

riid

[in] Identifier of the interface to which the caller is to gain access.

ppvObj

[out] Indirect pointer to the requested interface.

Return Value

The returned HRESULT value is one of the following:

RETURN VALUE	MEANING
S_OK	The service was successfully created or retrieved.
E_INVALIDARG	One or more of the arguments is invalid.
E_OUTOFMEMORY	Memory is insufficient to create the service.
E_UNEXPECTED	An unknown error occurred.
E_NOINTERFACE	The requested interface is not part of this service, or the service is unknown.

Remarks

`QueryService` returns an indirect pointer to the requested interface in the specified service. The caller is responsible for releasing this pointer when it is no longer required.

When you call `QueryService`, you pass both a service identifier (*guidService*) and an interface identifier (*riid*). The *guidService* specifies the service to which you want access, and the *riid* identifies an interface that is part of the service. In return, you receive an indirect pointer to the interface.

The object that implements the interface might also implement interfaces that are part of other services. Consider the following:

- Some of these interfaces might be optional. Not all interfaces defined in the service description are necessarily present on every implementation of the service or on every returned object.
- Unlike calls to `QueryInterface`, passing a different service identifier does not necessarily mean that a different Component Object Model (COM) object is returned.
- The returned object might have additional interfaces that are not part of the definition of the service.

Two different services, such as `SID_SMyService` and `SID_SYourService`, can both specify the use of the same interface, even though the implementation of the interface might have nothing in common between the two services. This works, because a call to `QueryService` (`SID_SMyService`, `IID_IDispatch`) can return a different object than `QueryService` (`SID_SYourService`, `IID_IDispatch`). Object identity is not assumed when you specify a different service identifier.

See also

[Class Overview](#)

ISpecifyPropertyPagesImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class implements `IUnknown` and provides a default implementation of the [ISpecifyPropertyPages](#) interface.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class T>
class ATL_NO_VTABLE ISpecifyPropertyPagesImpl
: public ISpecifyPropertyPages
```

Parameters

T

Your class, derived from `ISpecifyPropertyPagesImpl`.

Members

Public Methods

NAME	DESCRIPTION
ISpecifyPropertyPagesImpl::GetPages	Fills a Counted Array of UUID values. Each UUID corresponds to the CLSID for one of the property pages that can be displayed in the object's property sheet.

Remarks

The [ISpecifyPropertyPages](#) interface allows a client to obtain a list of CLSIDs for the property pages supported by an object. Class `ISpecifyPropertyPagesImpl` provides a default implementation of this interface and implements `IUnknown` by sending information to the dump device in debug builds.

NOTE

Do not expose the `ISpecifyPropertyPages` interface if your object does not support property pages.

Related Articles [ATL Tutorial](#), [Creating an ATL Project](#)

Inheritance Hierarchy

`ISpecifyPropertyPages`

`ISpecifyPropertyPagesImpl`

Requirements

Header: atlcom.h

ISpecifyPropertyPagesImpl::GetPages

Fills the array in the [CAUUIID](#) structure with the CLSIDs for the property pages that can be displayed in the object's property sheet.

```
STDMETHOD(GetPages)(CAUUIID* pPages);
```

Remarks

ATL uses the object's property map to retrieve each CLSID.

See [ISpecifyPropertyPages::GetPages](#) in the Windows SDK.

See also

[IPropertyPageImpl Class](#)

[IPerPropertyBrowsingImpl Class](#)

[Class Overview](#)

ISupportErrorInfoImpl Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides a default implementation of the [ISupportErrorInfo Interface](#) and can be used when only a single interface generates errors on an object.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<const IID* piid>
class ATL_NO_VTABLE ISupportErrorInfoImpl
: public ISupportErrorInfo
```

Parameters

piid

A pointer to the IID of an interface that supports [IErrorInfo](#).

Members

Public Methods

NAME	DESCRIPTION
ISupportErrorInfoImpl::InterfaceSupportsErrorInfo	Indicates whether the interface identified by <code>riid</code> supports the IErrorInfo interface.

Remarks

The [ISupportErrorInfo Interface](#) ensures that error information can be returned to the client. Objects that use `IErrorInfo` must implement `ISupportErrorInfo`.

Class `ISupportErrorInfoImpl` provides a default implementation of `ISupportErrorInfo` and can be used when only a single interface generates errors on an object. For example:

```
class ATL_NO_VTABLE CMySuppErrClass :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CMySuppErrClass, &CLSID_MySuppErrClass>,
public ISupportErrorInfoImpl<&IID_IMySuppErrClass>,
public IDispatchImpl<IMySuppErrClass, &IID_IMySuppErrClass, &LIBID_NVC_ATL_COMLib, /*wMajor =*/ 1,
/*wMinor =*/ 0>
```

Inheritance Hierarchy

`ISupportErrorInfo`

`ISupportErrorInfoImpl`

Requirements

Header: atlcom.h

ISupportErrorInfoImpl::InterfaceSupportsErrorInfo

Indicates whether the interface identified by `riid` supports the [IErrorInfo](#) interface.

```
STDMETHOD(InterfaceSupportsErrorInfo)(REFIID riid);
```

Remarks

See [ISupportErrorInfo::InterfaceSupportsErrorInfo](#) in the Windows SDK.

IThreadPoolConfig::GetSize

Call this method to get the number of threads in the pool.

```
STDMETHOD(GetSize)(int* pnNumThreads);
```

Parameters

pnNumThreads

[out] Address of the variable that, on success, receives the number of threads in the pool.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

```

HRESULT DoPoolOperations(IThreadPoolConfig* pPool, int nSize)
{
    int nCurrSize = 0;
    HRESULT hr = pPool->GetSize(&nCurrSize);
    if (SUCCEEDED(hr))
    {
        printf_s("Current pool size: %d\n", nCurrSize);
        hr = pPool->SetSize(nSize);
        if (SUCCEEDED(hr))
        {
            printf_s("New pool size : %d\n", nSize);
            DWORD dwTimeout = 0;
            hr = pPool->GetTimeout(&dwTimeout);
            if (SUCCEEDED(hr))
            {
                printf_s("Current pool timeout: %u\n", dwTimeout);
                // Increase timeout by 10 seconds.
                dwTimeout += 10 * 1000;
                hr = pPool->SetTimeout(dwTimeout);
                if (SUCCEEDED(hr))
                {
                    printf_s("New pool timeout: %u\n", dwTimeout);
                }
                else
                {
                    printf_s("Failed to set pool timeout: 0x%08X\n", hr);
                }
            }
            else
            {
                printf_s("Failed to get pool timeout: 0x%08X\n", hr);
            }
        }
        else
        {
            printf_s("Failed to resize pool: 0x%08X\n", hr);
        }
    }
    else
    {
        printf_s("Failed to get pool size: 0x%08x\n", hr);
    }

    return hr;
}

```

IThreadPoolConfig::GetTimeout

Call this method to get the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

```
STDMETHOD(GetTimeout)(DWORD* pdwMaxWait);
```

Parameters

pdwMaxWait

[out] Address of the variable that, on success, receives the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See [IThreadPoolConfig::GetSize](#).

IThreadPoolConfig::SetSize

Call this method to set the number of threads in the pool.

```
STDMETHOD(SetSize)(int nNumThreads);
```

Parameters

nNumThreads

The requested number of threads in the pool.

If *nNumThreads* is negative, its absolute value will be multiplied by the number of processors in the machine to get the total number of threads.

If *nNumThreads* is zero, `ATLS_DEFAULT_THREADSPERPROC` will be multiplied by the number of processors in the machine to get the total number of threads.

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure.

Example

See [IThreadPoolConfig::GetSize](#).

IThreadPoolConfig::SetTimeout

Call this method to set the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

```
STDMETHOD(SetTimeout)(DWORD dwMaxWait);
```

Parameters

dwMaxWait

The requested maximum time in milliseconds that the thread pool will wait for a thread to shut down.

Return Value

Returns `S_OK` on success, or an error `HRESULT` on failure.

Example

See [IThreadPoolConfig::GetSize](#).

See also

[Class Overview](#)

IThreadPoolConfig Interface

3/4/2019 • 2 minutes to read • [Edit Online](#)

This interface provides methods for configuring a thread pool.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
__interface  
__declspec(uuid("B1F64757-6E88-4fa2-8886-7848B0D7E660")) IThreadPoolConfig : public IUnknown
```

Members

Methods

GetSize	Call this method to get the number of threads in the pool.
GetTimeout	Call this method to get the maximum time in milliseconds that the thread pool will wait for a thread to shut down.
SetSize	Call this method to set the number of threads in the pool.
SetTimeout	Call this method to set the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

Remarks

This interface is implemented by [CThreadPool](#).

Requirements

Header: atlutil.h

IThreadPoolConfig::GetSize

Call this method to get the number of threads in the pool.

```
STDMETHOD(GetSize)(int* pnNumThreads);
```

Parameters

pnNumThreads

[out] Address of the variable that, on success, receives the number of threads in the pool.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

```
HRESULT DoPoolOperations(IThreadPoolConfig* pPool, int nSize)
{
    int nCurrSize = 0;
    HRESULT hr = pPool->GetSize(&nCurrSize);
    if (SUCCEEDED(hr))
    {
        printf_s("Current pool size: %d\n", nCurrSize);
        hr = pPool->SetSize(nSize);
        if (SUCCEEDED(hr))
        {
            printf_s("New pool size : %d\n", nSize);
            DWORD dwTimeout = 0;
            hr = pPool->GetTimeout(&dwTimeout);
            if (SUCCEEDED(hr))
            {
                printf_s("Current pool timeout: %u\n", dwTimeout);
                // Increase timeout by 10 seconds.
                dwTimeout += 10 * 1000;
                hr = pPool->SetTimeout(dwTimeout);
                if (SUCCEEDED(hr))
                {
                    printf_s("New pool timeout: %u\n", dwTimeout);
                }
                else
                {
                    printf_s("Failed to set pool timeout: 0x%08X\n", hr);
                }
            }
            else
            {
                printf_s("Failed to get pool timeout: 0x%08X\n", hr);
            }
        }
        else
        {
            printf_s("Failed to resize pool: 0x%08X\n", hr);
        }
    }
    else
    {
        printf_s("Failed to get pool size: 0x%08x\n", hr);
    }

    return hr;
}
```

IThreadPoolConfig::GetTimeout

Call this method to get the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

```
STDMETHOD(GetTimeout)(DWORD* pdwMaxWait);
```

Parameters

pdwMaxWait

[out] Address of the variable that, on success, receives the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See [IThreadPoolConfig::GetSize](#).

IThreadPoolConfig::SetSize

Call this method to set the number of threads in the pool.

```
STDMETHOD(SetSize)(int nNumThreads);
```

Parameters

nNumThreads

The requested number of threads in the pool.

If *nNumThreads* is negative, its absolute value will be multiplied by the number of processors in the machine to get the total number of threads.

If *nNumThreads* is zero, ATLS_DEFAULT_THREADSPERPROC will be multiplied by the number of processors in the machine to get the total number of threads.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See [IThreadPoolConfig::GetSize](#).

IThreadPoolConfig::SetTimeout

Call this method to set the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

```
STDMETHOD(SetTimeout)(DWORD dwMaxWait);
```

Parameters

dwMaxWait

The requested maximum time in milliseconds that the thread pool will wait for a thread to shut down.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See [IThreadPoolConfig::GetSize](#).

See also

[Classes](#)

[CThreadPool Class](#)

IViewObjectExImpl Class

3/4/2019 • 5 minutes to read • [Edit Online](#)

This class implements `IUnknown` and provides default implementations of the [IViewObject](#), [IViewObject2](#), and [IViewObjectEx](#) interfaces.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
template<class T>
class ATL_NO_VTABLE IViewObjectExImpl
: public IViewObjectEx
```

Parameters

T

Your class, derived from `IViewObjectExImpl`.

Members

Public Methods

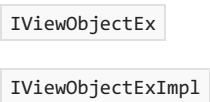
NAME	DESCRIPTION
IViewObjectExImpl::Draw	Draws a representation of the control onto a device context.
IViewObjectExImpl::Freeze	Freezes the drawn representation of a control so it won't change until an <code>Unfreeze</code> . The ATL implementation returns E_NOTIMPL.
IViewObjectExImpl::GetAdvise	Retrieves an existing advisory sink connection on the control, if there is one.
IViewObjectExImpl::GetColorSet	Returns the logical palette used by the control for drawing. The ATL implementation returns E_NOTIMPL.
IViewObjectExImpl::GetExtent	Retrieves the control's display size in HIMETRIC units (0.01 millimeter per unit) from the control class data member CComControlBase::m_sizeExtent .
IViewObjectExImpl::GetNaturalExtent	Provides sizing hints from the container for the object to use as the user resizes it.
IViewObjectExImpl::GetRect	Returns a rectangle describing a requested drawing aspect. The ATL implementation returns E_NOTIMPL.
IViewObjectExImpl::GetViewStatus	Returns information about the opacity of the object and what drawing aspects are supported.

NAME	DESCRIPTION
IViewObjectExImpl::QueryHitPoint	Checks if the specified point is in the specified rectangle and returns a HITRESULT value in <code>pHitResult</code> .
IViewObjectExImpl::QueryHitRect	Checks whether the control's display rectangle overlaps any point in the specified location rectangle and returns a HITRESULT value in <code>pHitResult</code> .
IViewObjectExImpl::SetAdvise	Sets up a connection between the control and an advise sink so the sink can be notified about changes in the control's view.
IViewObjectExImpl::Unfreeze	Unfreezes the drawn representation of the control. The ATL implementation returns <code>E_NOTIMPL</code> .

Remarks

The [IViewObject](#), [IViewObject2](#), and [IViewObjectEx](#) interfaces enable a control to display itself directly, and to create and manage an advise sink to notify the container of changes in the control display. The `IViewObjectEx` interface provides support for extended control features such as flicker-free drawing, non-rectangular and transparent controls, and hit-testing (for example, how close a mouse click must be to be considered on the control). Class `IViewObjectExImpl` provides a default implementation of these interfaces and implements `IUnknown` by sending information to the dump device in debug builds.

Inheritance Hierarchy



Requirements

Header: atlctl.h

IViewObjectExImpl::Draw

Draws a representation of the control onto a device context.

```
STDMETHOD(Draw)(
    DWORD dwDrawAspect,
    LONG lindex,
    void* pvAspect,
    DVTARGETDEVICE* ptd,
    HDC hicTargetDev,
    LPCRECTL prcBounds,
    LPCRECTL prcWBounds,
    BOOL(_stdcall* /* pfnContinue*/) (DWORD_PTR dwContinue),
    DWORD_PTR /* dwContinue */);
```

Remarks

This method calls `CComControl1::OnDrawAdvanced` which in turn calls your control class's `OnDraw` method. An `OnDraw` method is automatically added to your control class when you create your control with the ATL Control Wizard. The Wizard's default `OnDraw` draws a rectangle with the label "ATL 3.0".

See [IViewObject::Draw](#) in the Windows SDK.

IViewObjectExImpl::Freeze

Freezes the drawn representation of a control so it won't change until an `Unfreeze`. The ATL implementation returns `E_NOTIMPL`.

```
STDMETHOD(Freeze)(
    DWORD /* dwAspect */,
    LONG /* lindex */,
    void* /* pvAspect */,
    DWORD* /* pdwFreeze */);
```

Remarks

See [IViewObject::Freeze](#) in the Windows SDK.

IViewObjectExImpl::GetAdvise

Retrieves an existing advisory sink connection on the control, if there is one.

```
STDMETHOD(GetAdvise)(
    DWORD* /* pAspects */,
    DWORD* /* pAdvf */,
    IAdviseSink** /* ppAdvSink */);
```

Remarks

The advisory sink is stored in the control class data member [CComControlBase::m_spAdviseSink](#).

See [IViewObject::GetAdvise](#) in the Windows SDK.

IViewObjectExImpl::GetColorSet

Returns the logical palette used by the control for drawing. The ATL implementation returns `E_NOTIMPL`.

```
STDMETHOD(GetColorSet)(
    DWORD /* dwAspect */,
    LONG /* lindex */,
    void* /* pvAspect */,
    DVTARGETDEVICE* /* ptd */,
    HDC /* hicTargetDevice */,
    LOGPALETTE** /* ppColorSet */);
```

Remarks

See [IViewObject::GetColorSet](#) in the Windows SDK.

IViewObjectExImpl::GetExtent

Retrieves the control's display size in HIMETRIC units (0.01 millimeter per unit) from the control class data member [CComControlBase::m_sizeExtent](#).

```
STDMETHOD(GetExtent)(
    DWORD /* dwDrawAspect */,
    LONG /* lindex */,
    DVTARGETDEVICE* /* ptd */,
    LPSIZEL* lpSize);
```

Remarks

See [IViewObject2::GetExtent](#) in the Windows SDK.

IViewObjectExImpl::GetNaturalExtent

Provides sizing hints from the container for the object to use as the user resizes it.

```
STDMETHOD(GetNaturalExtent)(
    DWORD dwAspect,
    LONG /* lindex */,
    DVTARGETDEVICE* /* ptd */,
    HDC /* hicTargetDevice */,
    DVEXTENTINFO* pExtentInfo,
    LPSIZEL psize1);
```

Remarks

If `dwAspect` is DVASPECT_CONTENT and `pExtentInfo->dwExtentMode` is DVEXTENT_CONTENT, sets * `psize1` to the control class's data member [CComControlBase::m_sizeNatural](#). Otherwise, returns an error HRESULT.

See [IViewObjectEx::GetNaturalExtent](#) in the Windows SDK.

IViewObjectExImpl::GetRect

Returns a rectangle describing a requested drawing aspect. The ATL implementation returns E_NOTIMPL.

```
STDMETHOD(GetRect)(DWORD /* dwAspect */, LPRECT /* pRect */);
```

Remarks

See [IViewObjectEx::GetRect](#) in the Windows SDK.

IViewObjectExImpl::GetViewStatus

Returns information about the opacity of the object and what drawing aspects are supported.

```
STDMETHOD(GetViewStatus)(DWORD* pdwStatus);
```

Remarks

By default, ATL sets `pdwStatus` to indicate that the control supports VIEWSTATUS_OPAQUE (possible values are in the [VIEWSTATUS](#) enumeration).

See [IViewObjectEx::GetViewStatus](#) in the Windows SDK.

IViewObjectExImpl::QueryHitPoint

Checks if the specified point is in the specified rectangle and returns a [HITRESULT](#) value in `pHitResult`.

```
STDMETHOD(QueryHitPoint)(
    DWORD dwAspect,
    LPCRECT pRectBounds,
    POINT ptLoc,
    LONG /* lCloseHit */,
    DWORD* /* pHitResult */);
```

Remarks

The value can be either `HITRESULT_HIT` or `HITRESULT_OUTSIDE`.

If `dwAspect` equals `DVASPECT_CONTENT`, the method returns `S_OK`. Otherwise, the method returns `E_FAIL`.

See [IViewObjectEx::QueryHitPoint](#) in the Windows SDK.

IViewObjectExImpl::QueryHitRect

Checks whether the control's display rectangle overlaps any point in the specified location rectangle and returns a `HITRESULT` value in `pHitResult`.

```
STDMETHOD(QueryHitRect)(
    DWORD dwAspect,
    LPCRECT pRectBounds,
    LPRECT prcLoc,
    LONG /* lCloseHit */,
    DWORD* /* pHitResult */);
```

Remarks

The value can be either `HITRESULT_HIT` or `HITRESULT_OUTSIDE`.

If `dwAspect` equals `DVASPECT_CONTENT`, the method returns `S_OK`. Otherwise, the method returns `E_FAIL`.

See [IViewObjectEx::QueryHitRect](#) in the Windows SDK.

IViewObjectExImpl::SetAdvise

Sets up a connection between the control and an advise sink so the sink can be notified about changes in the control's view.

```
STDMETHOD(SetAdvise)(
    DWORD /* aspects */,
    DWORD /* advf */,
    IAdviseSink* pAdvSink);
```

Remarks

The pointer to the [IAdviseSink](#) interface on the advise sink is stored in the control class data member [CComControlBase::m_spAdviseSink](#).

See [IViewObject::SetAdvise](#) in the Windows SDK.

IViewObjectExImpl::Unfreeze

Unfreezes the drawn representation of the control. The ATL implementation returns `E_NOTIMPL`.

```
STDMETHOD(Unfreeze)(DWORD /* dwFreeze */);
```

Remarks

See [IViewObject::Unfreeze](#) in the Windows SDK.

IWorkerThreadClient::CloseHandle

Implement this method to close the handle associated with this object.

```
HRESULT CloseHandle(HANDLE hHandle);
```

Parameters

hHandle

The handle to be closed.

Return Value

Return S_OK on success, or an error HRESULT on failure.

Remarks

The handle passed to this method was previously associated with this object by a call to [CWorkerThread::AddHandle](#).

Example

The following code shows a simple implementation of `IWorkerThreadClient::CloseHandle`.

```
HRESULT CloseHandle(HANDLE hObject)
{
    // Users should do any shutdown operation required here.
    // Generally, this means just closing the handle.

    if (!::CloseHandle(hObject))
    {
        // Closing the handle failed for some reason.
        return AtlHresultFromLastError();
    }

    return S_OK;
}
```

IWorkerThreadClient::Execute

Implement this method to execute code when the handle associated with this object becomes signaled.

```
HRESULT Execute(DWORD_PTR dwParam, HANDLE hObject);
```

Parameters

dwParam

The user parameter.

hObject

The handle that has become signaled.

Return Value

Return S_OK on success, or an error HRESULT on failure.

Remarks

The handle and DWORD/pointer passed to this method were previously associated with this object by a call to [CWorkerThread::AddHandle](#).

Example

The following code shows a simple implementation of `IWorkerThreadClient::Execute`.

```
HRESULT Execute(DWORD_PTR dwParam, HANDLE hObject)
{
    // Cast the parameter to its known type.
    LONG* pn = reinterpret_cast<LONG*>(dwParam);

    // Increment the LONG.
    LONG n = InterlockedIncrement(pn);

    // Log the results.
    printf_s("Handle 0x%08X incremented value to : %d\n", (DWORD_PTR)hObject, n);

    return S_OK;
}
```

See also

[CComControl Class](#)

[ActiveX Controls Interfaces](#)

[Tutorial](#)

[Creating an ATL Project](#)

[Class Overview](#)

IWorkerThreadClient Interface

3/4/2019 • 2 minutes to read • [Edit Online](#)

`IWorkerThreadClient` is the interface implemented by clients of the `CWorkerThread` class.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
__interface IWorkerThreadClient
```

Members

Methods

CloseHandle	Implement this method to close the handle associated with this object.
Execute	Implement this method to execute code when the handle associated with this object becomes signaled.

Remarks

Implement this interface when you have code that needs to execute on a worker thread in response to a handle becoming signaled.

Requirements

Header: atlutil.h

IWorkerThreadClient::CloseHandle

Implement this method to close the handle associated with this object.

```
HRESULT CloseHandle(HANDLE hHandle);
```

Parameters

hHandle

The handle to be closed.

Return Value

Return `S_OK` on success, or an error `HRESULT` on failure.

Remarks

The handle passed to this method was previously associated with this object by a call to [CWorkerThread::AddHandle](#).

Example

The following code shows a simple implementation of `IWorkerThreadClient::CloseHandle`.

```
HRESULT CloseHandle(HANDLE hObject)
{
    // Users should do any shutdown operation required here.
    // Generally, this means just closing the handle.

    if (!::CloseHandle(hObject))
    {
        // Closing the handle failed for some reason.
        return AtlHresultFromLastError();
    }

    return S_OK;
}
```

IWorkerThreadClient::Execute

Implement this method to execute code when the handle associated with this object becomes signaled.

```
HRESULT Execute(DWORD_PTR dwParam, HANDLE hObject);
```

Parameters

dwParam

The user parameter.

hObject

The handle that has become signaled.

Return Value

Return S_OK on success, or an error HRESULT on failure.

Remarks

The handle and DWORD/pointer passed to this method were previously associated with this object by a call to [CWorkerThread::AddHandle](#).

Example

The following code shows a simple implementation of `IWorkerThreadClient::Execute`.

```
HRESULT Execute(DWORD_PTR dwParam, HANDLE hObject)
{
    // Cast the parameter to its known type.
    LONG* pn = reinterpret_cast<LONG*>(dwParam);

    // Increment the LONG.
    LONG n = InterlockedIncrement(pn);

    // Log the results.
    printf_s("Handle 0x%08X incremented value to : %d\n", (DWORD_PTR)hObject, n);

    return S_OK;
}
```

See also

[Classes](#)

[CWorkerThread Class](#)

Win32ThreadTraits Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class provides the creation function for a Windows thread. Use this class if the thread will not use CRT functions.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

Syntax

```
class Win32ThreadTraits
```

Members

Public Methods

NAME	DESCRIPTION
Win32ThreadTraits::CreateThread	(Static) Call this function to create a thread that should not use CRT functions.

Remarks

Thread traits are classes that provide a creation function for a particular type of thread. The creation function has the same signature and semantics as the Windows [CreateThread](#) function.

Thread traits are used by the following classes:

- [CThreadPool](#)
- [CWorkerThread](#)

If the thread will be using CRT functions, use [CRTThreadTraits](#) instead.

Requirements

Header: `atlbase.h`

Win32ThreadTraits::CreateThread

Call this function to create a thread that should not use CRT functions.

```
static HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES lpsa,  
    DWORD dwStackSize,  
    LPTHREAD_START_ROUTINE pfnThreadProc,  
    void* pvParam,  
    DWORD dwCreationFlags,  
    DWORD* pdwThreadId) throw();
```

Parameters

lpsa

The security attributes for the new thread.

dwStackSize

The stack size for the new thread.

pfnThreadProc

The thread procedure of the new thread.

pvParam

The parameter to be passed to the thread procedure.

dwCreationFlags

The creation flags (0 or CREATE_SUSPENDED).

pdwThreadId

[out] Address of the DWORD variable that, on success, receives the thread ID of the newly created thread.

Return Value

Returns the handle to the newly created thread or NULL on failure. Call [GetLastError](#) to get extended error information.

Remarks

See [CreateThread](#) for further information on the parameters to this function.

This function calls `CreateThread` to create the thread.

See also

[Class Overview](#)

Worker Archetype

3/4/2019 • 2 minutes to read • [Edit Online](#)

Classes that conform to the *worker* archetype provide the code to process work items queued on a thread pool.

Implementation

To implement a class conforming to this archetype, the class must provide the following features:

METHOD	DESCRIPTION
Initialize	Called to initialize the worker object before any requests are passed to Execute .
Execute	Called to process a work item.
Terminate	Called to uninitialize the worker object after all requests have been passed to Execute .

TYPEDEF	DESCRIPTION
RequestType	A typedef for the type of work item that can be processed by the worker class.

A typical *worker* class looks like this:

```
class CMyWorker
{
public:
    typedef MyRequestType RequestType;

    BOOL Initialize(void* pvWorkerParam);

    void Execute(MyRequestType request, void* pvWorkerParam, OVERLAPPED* pOverlapped);

    void Terminate(void* pvWorkerParam);
};
```

Existing Implementations

These classes conform to this archetype:

CLASS	DESCRIPTION
CNonStatelessWorker	Receives requests from the thread pool and passes them on to a worker object that is created and destroyed for each request.

Use

These template parameters expect the class to conform to this archetype:

PARAMETER NAME	USED BY
<i>Worker</i>	CThreadPool
<i>Worker</i>	CNonStatelessWorker

Requirements

Header: atlutil.h

WorkerArchetype::Execute

Called to process a work item.

```
void Execute(
    RequestType request,
    void* pvWorkerParam,
    OVERLAPPED* pOverlapped);
```

Parameters

request

The work item to be processed. The work item is of the same type as `RequestType`.

pvWorkerParam

A custom parameter understood by the worker class. Also passed to `WorkerArchetype::Initialize` and `Terminate`.

pOverlapped

A pointer to the `OVERLAPPED` structure used to create the queue on which work items were queued.

WorkerArchetype::Initialize

Called to initialize the worker object before any requests are passed to `WorkerArchetype::Execute`.

```
BOOL Initialize(void* pvParam) throw();
```

Parameters

pvParam

A custom parameter understood by the worker class. Also passed to `WorkerArchetype::Terminate` and `WorkerArchetype::Execute`.

Return Value

Return TRUE on success, FALSE on failure.

WorkerArchetype::RequestType

A typedef for the type of work item that can be processed by the worker class.

```
typedef MyRequestType RequestType;
```

Remarks

This type must be used as the first parameter of `WorkerArchetype::Execute` and must be capable of being cast to and from a `ULONG_PTR`.

WorkerArchetype::Terminate

Called to uninitialize the worker object after all requests have been passed to `WorkerArchetype::Execute`).

```
void Terminate(void* pvParam) throw();
```

Parameters

pvParam

A custom parameter understood by the worker class. Also passed to `WorkerArchetype::Initialize` and `WorkerArchetype::Execute` .

See also

[Concepts](#)

[ATL COM Desktop Components](#)

ATL_URL_SCHEME

3/4/2019 • 2 minutes to read • [Edit Online](#)

The members of this enumeration provide constants for the schemes understood by [CUrl](#).

Syntax

```
enum ATL_URL_SCHEME{
    ATL_URL_SCHEME_UNKNOWN = -1,
    ATL_URL_SCHEME_FTP      = 0,
    ATL_URL_SCHEME_GOPHER   = 1,
    ATL_URL_SCHEME_HTTP     = 2,
    ATL_URL_SCHEME_HTTPS    = 3,
    ATL_URL_SCHEME_FILE     = 4,
    ATL_URL_SCHEME_NEWS     = 5,
    ATL_URL_SCHEME_MAILTO   = 6,
    ATL_URL_SCHEME_SOCKS    = 7
};
```

Requirements

Header: atlutil.h

See also

[Concepts](#)

[CUrl::SetScheme](#)

[CUrl::GetScheme](#)

ATL Functions

3/4/2019 • 2 minutes to read • [Edit Online](#)

To find an ATL function by category, see the following topics.

[ATL Path Functions](#)

Provide support for manipulating file paths.

[COM Map Global Functions](#)

Provide support for COM map `IUnknown` implementations.

[Composite Control Global Functions](#)

Provide support for creating dialog boxes, and for creating, hosting and licensing ActiveX controls.

[Connection Point Global Functions](#)

Provide support for connection points and sink maps.

[Debugging and Error Reporting Global Functions](#)

Provide useful debugging and trace facilities.

[Device Context Global Functions](#)

Creates a device context for a given device.

[Event Handling Global Functions](#)

Provides an event handler.

[Marshaling Global Functions](#)

Provide support for marshaling and converting marshaling data into interface pointers.

[Pixel/HIMETRIC Conversion Global Functions](#)

Provide support for converting to and from pixel and HIMETRIC units.

[Registry and TypeLib Global Functions](#)

Provide support for loading and registering a type library.

[Security Global Functions](#)

Provide support for modifying SID and ACL objects.

[Security Identifier Global Functions](#)

Return common well-known SID objects.

[Server Registration Global Functions](#)

Provide support for registering and unregistering server objects in the object map.

[WinModule Global Functions](#)

Provide support for `_AtlCreateWndData` structure operations.

See also

[ATL COM Desktop Components](#)

[Macros](#)

[Global Variables](#)

[Typedefs](#)

[Classes and structs](#)

ATL HTTP Utility Functions

3/4/2019 • 5 minutes to read • [Edit Online](#)

These functions support manipulation of URLs.

AtlCanonicalizeUrl	Canonicalizes a URL, which includes converting unsafe characters and spaces into escape sequences.
AtlCombineUrl	Combines a base URL and a relative URL into a single, canonical URL.
AtlEscapeUrl	Converts all unsafe characters to escape sequences.
AtlGetDefaultUrlPort	Gets the default port number associated with a particular Internet protocol or scheme.
AtlIsUnsafeUrlChar	Determines whether a character is safe for use in a URL.
AtlUnescapeUrl	Converts escaped characters back to their original values.
RGBToHtml	Converts a COLORREF value to the HTML text corresponding to that color value.
SystemTimeToHttpDate	Call this function to convert a system time to a string in a format suitable for using in HTTP headers.

Requirements

Header: `atlutil.h`

AtlCanonicalizeUrl

Call this function to canonicalize a URL, which includes converting unsafe characters and spaces into escape sequences.

```
inline BOOL AtlCanonicalizeUrl(  
    LPCTSTR szUrl,  
    LPTSTR szCanonicalized,  
    DWORD* pdwMaxLength,  
    DWORD dwFlags = 0) throw();
```

Parameters

szUrl

The URL to be canonicalized.

szCanonicalized

Caller-allocated buffer to receive the canonicalized URL.

pdwMaxLength

Pointer to a variable that contains the length in characters of *szCanonicalized*. If the function succeeds, the variable

receives the number of characters written to the buffer including the terminating null character. If the function fails, the variable receives the required length in bytes of the buffer including space for the terminating null character.

dwFlags

ATL_URL flags controlling the behavior of this function.

- ATL_URL_BROWSER_MODE Does not encode or decode characters after "#" or "?", and does not remove trailing white space after "?". If this value is not specified, the entire URL is encoded and trailing white space is removed.
- ATL_URL_DECODE Converts all %XX sequences to characters, including escape sequences, before the URL is parsed.
- ATL_URL_ENCODE_PERCENT Encodes any percent signs encountered. By default, percent signs are not encoded.
- ATL_URL_ENCODE_SPACES_ONLY Encodes spaces only.
- ATL_URL_ESCAPE Converts all escape sequences (%XX) to their corresponding characters.
- ATL_URL_NO_ENCODE Does not convert unsafe characters to escape sequences.
- ATL_URL_NO_META Does not remove meta sequences (such as "." and "..") from the URL.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Behaves like the current version of [InternetCanonicalizeUrl](#) but does not require WinInet or Internet Explorer to be installed.

AtlCombineUrl

Call this function to combine a base URL and a relative URL into a single, canonical URL.

```
inline BOOL AtlCombineUrl(  
    LPCTSTR szBaseUrl,  
    LPCTSTR szRelativeUrl,  
    LPTSTR szBuffer,  
    DWORD* pdwMaxLength,  
    DWORD dwFlags = 0) throw();
```

Parameters

szBaseUrl

The base URL.

szRelativeUrl

The URL relative to the base URL.

szBuffer

Caller-allocated buffer to receive the canonicalized URL.

pdwMaxLength

Pointer to a variable that contains the length in characters of *szBuffer*. If the function succeeds, the variable receives the number of characters written to the buffer including the terminating null character. If the function fails, the variable receives the required length in bytes of the buffer including space for the terminating null character.

dwFlags

Flags controlling the behavior of this function. See [AtlCanonicalizeUrl](#).

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Behaves like the current version of [InternetCombineUrl](#) but does not require WinInet or Internet Explorer to be installed.

AtlEscapeUrl

Call this function to convert all unsafe characters to escape sequences.

```
inline BOOL AtlEscapeUrl(  
    LPCSTR szStringIn,  
    LPSTR szStringOut,  
    DWORD* pdwStrLen,  
    DWORD dwMaxLength,  
    DWORD dwFlags = 0) throw();  
  
inline BOOL AtlEscapeUrl(  
    LPCWSTR szStringIn,  
    LPWSTR szStringOut,  
    DWORD* pdwStrLen,  
    DWORD dwMaxLength,  
    DWORD dwFlags = 0) throw();
```

Parameters

lpszStringIn

The URL to be converted.

lpszStringOut

Caller-allocated buffer to which the converted URL will be written.

pdwStrLen

Pointer to a DWORD variable. If the function succeeds, *pdwStrLen* receives the number of characters written to the buffer including the terminating null character. If the function fails, the variable receives the required length in bytes of the buffer including space for the terminating null character. When using the wide character version of this method, *pdwStrLen* receives the number of characters required, not the number of bytes.

dwMaxLength

The size of the buffer *lpszStringOut*.

dwFlags

ATL_URL flags controlling the behavior of this function. See [ATLCanonicalizeUrl](#) for possible values.

Return Value

Returns TRUE on success, FALSE on failure.

AtlGetDefaultUrlPort

Call this function to get the default port number associated with a particular Internet protocol or scheme.

```
inline ATL_URL_PORT AtlGetDefaultUrlPort(ATL_URL_SCHEME m_nScheme) throw();
```

Parameters

m_nScheme

The [ATL_URL_SCHEME](#) value identifying the scheme for which you want to obtain the port number.

Return Value

The [ATL_URL_PORT](#) associated with the specified scheme or ATL_URL_INVALID_PORT_NUMBER if the scheme is not recognized.

AtlIsUnsafeUrlChar

Call this function to find out whether a character is safe for use in a URL.

```
inline BOOL AtlIsUnsafeUrlChar(char chIn) throw();
```

Parameters

chIn

The character to be tested for safety.

Return Value

Returns TRUE if the input character is unsafe, FALSE otherwise.

Remarks

Characters that should not be used in URLs can be tested using this function and converted using [AtlCanonicalizeUrl](#).

AtlUnescapeUrl

Call this function to convert escaped characters back to their original values.

```
inline BOOL AtlUnescapeUrl(
    LPCSTR szStringIn,
    LPSTR szStringOut,
    LPDWORD pdwStrLen,
    DWORD dwMaxLength) throw();

inline BOOL AtlUnescapeUrl(
    LPCWSTR szStringIn,
    LPWSTR szStringOut,
    LPDWORD pdwStrLen,
    DWORD dwMaxLength) throw();
```

Parameters

lpzStringIn

The URL to be converted.

lpzStringOut

Caller-allocated buffer to which the converted URL will be written.

pdwStrLen

Pointer to a DWORD variable. If the function succeeds, the variable receives the number of characters written to the buffer including the terminating null character. If the function fails, the variable receives the required length in bytes of the buffer including space for the terminating null character.

dwMaxLength

The size of the buffer *lpzStringOut*.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Reverses the conversion process applied by [AtlEscapeUrl](#).

RGBToHtml

Converts a [COLORREF](#) value to the HTML text corresponding to that color value.

```
bool inline RGBToHtml(  
    COLORREF color,  
    LPTSTR pbOut,  
    long nBuffer);
```

Parameters

color

An RGB color value.

pbOut

Caller-allocated buffer to receive the text for the HTML color value. The buffer must have space for at least 8 characters including space for the null terminator).

nBuffer

The size in bytes of the buffer (including space for the null terminator).

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

An HTML color value is a pound sign followed by a 6-digit hexadecimal value using 2 digits for each of the red, green, and blue components of the color (for example, #FFFFFF is white).

SystemTimeToHttpDate

Call this function to convert a system time to a string in a format suitable for using in HTTP headers.

```
inline void SystemTimeToHttpDate(  
    const SYSTEMTIME& st,  
    CStringA& strTime);
```

Parameters

st

The system time to be obtained as an HTTP format string.

strTime

A reference to a string variable to receive the HTTP date time as defined in RFC 2616 (<http://www.ietf.org/rfc/rfc2616.txt>) and RFC 1123 (<http://www.ietf.org/rfc/rfc1123.txt>).

See also

[Concepts](#)

[ATL COM Desktop Components](#)

[InternetCanonicalizeUrl](#)

ATL Text Encoding Functions

3/4/2019 • 13 minutes to read • [Edit Online](#)

These functions support text encoding and decoding.

AtlGetHexValue	Call this function to get the numeric value of a hexadecimal digit.
AtlGetVersion	Call this function to get the version of the ATL library that you are using.
AtlHexDecode	Decodes a string of data that has been encoded as hexadecimal text such as by a previous call to AtlHexEncode .
AtlHexDecodeGetRequiredLength	Call this function to get the size in bytes of a buffer that could contain data decoded from a hex-encoded string of the specified length.
AtlHexEncode	Call this function to encode some data as a string of hexadecimal text.
AtlHexEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
AtlHexValue	Call this function to get the numeric value of a hexadecimal digit.
AtlUnicodeToUTF8	Call this function to convert a Unicode string to UTF-8.
BEncode	Call this function to convert some data using the "B" encoding.
BEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
EscapeXML	Call this function to convert characters that are unsafe for use in XML to their safe equivalents.
GetExtendedChars	Call this function to get the number of extended characters in a string.
IsExtendedChar	Call this function to find out if a given character is an extended character (less than 32, greater than 126, and not a tab, linefeed or carriage return)
QEncode	Call this function to convert some data using the "Q" encoding.
QEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.

QPDecode	Decodes a string of data that has been encoded in quoted-printable format such as by a previous call to QPEncode .
QPDecodeGetRequiredLength	Call this function to get the size in bytes of a buffer that could contain data decoded from quoted-printable-encoded string of the specified length.
QPEncode	Call this function to encode some data in quoted-printable format.
QPEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
UUDecode	Decodes a string of data that has been uuencoded such as by a previous call to UUEncode .
UUDecodeGetRequiredLength	Call this function to get the size in bytes of a buffer that could contain data decoded from a uuencoded string of the specified length.
UUEncode	Call this function to uuencode some data.
UUEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.

Requirements

Header: atlenc.h

AtlGetHexValue

Call this function to get the numeric value of a hexadecimal digit.

```
inline char AtlGetHexValue(char chIn) throw();
```

Parameters

chIn

The hexadecimal character '0'-'9', 'A'-'F', or 'a'-'f'.

Return Value

The numeric value of the input character interpreted as a hexadecimal digit. For example, an input of '0' returns a value of 0 and an input of 'A' returns a value of 10. If the input character is not a hexadecimal digit, this function returns -1.

AtlGetVersion

Call this function to get the version of the ATL library that you are using.

```
ATLAPI_(DWORD) AtlGetVersion(void* pReserved);
```

Parameters

pReserved

A reserved pointer.

Return Value

Returns a DWORD integer value of the version of the ATL library that you are compiling or running.

Example

The function should be called as follows.

```
DWORD ver;  
ver = AtlGetVersion(NULL);
```

Requirements

Header: atabase.h

AtlHexDecode

Decodes a string of data that has been encoded as hexadecimal text such as by a previous call to [AtlHexEncode](#).

```
inline BOOL AtlHexDecode(  
    LPCSTR pSrcData,  
    int nSrcLen,  
    LPBYTE pbDest,  
    int* pnDestLen) throw();
```

Parameters

pSrcData

The string containing the data to be decoded.

nSrcLen

The length in characters of *pSrcData*.

pbDest

Caller-allocated buffer to receive the decoded data.

pnDestLen

Pointer to a variable that contains the length in bytes of *pbDest*. If the function succeeds, the variable receives the number of bytes written to the buffer. If the function fails, the variable receives the required length in bytes of the buffer.

Return Value

Returns TRUE on success, FALSE on failure.

AtlHexDecodeGetRequiredLength

Call this function to get the size in bytes of a buffer that could contain data decoded from a hex-encoded string of the specified length.

```
inline int AtlHexDecodeGetRequiredLength(int nSrcLen) throw();
```

Parameters

nSrcLen

The number of characters in the encoded string.

Return Value

The number of bytes required for a buffer that could hold a decoded string of *nSrcLen* characters.

AtlHexEncode

Call this function to encode some data as a string of hexadecimal text.

```
inline BOOL AtlHexEncode(  
    const BYTE * pbSrcData,  
    int nSrcLen,  
    LPSTR szDest,  
    int * pnDestLen) throw();
```

Parameters

pbSrcData

The buffer containing the data to be encoded.

nSrcLen

The length in bytes of the data to be encoded.

szDest

Caller-allocated buffer to receive the encoded data.

pnDestLen

Pointer to a variable that contains the length in characters of *szDest*. If the function succeeds, the variable receives the number of characters written to the buffer. If the function fails, the variable receives the required length in characters of the buffer.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Each byte of source data is encoded as 2 hexadecimal characters.

AtlHexEncodeGetRequiredLength

Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.

```
inline int AtlHexEncodeGetRequiredLength(int nSrcLen) throw();
```

Parameters

nSrcLen

The number of bytes of data to be encoded.

Return Value

The number of characters required for a buffer that could hold encoded data of *nSrcLen* bytes.

AtlHexValue

Call this function to get the numeric value of a hexadecimal digit.

```
inline short AtlHexValue(char chIn) throw();
```

Parameters

chIn

The hexadecimal character '0'-'9', 'A'-'F', or 'a'-'f'.

Return Value

The numeric value of the input character interpreted as a hexadecimal digit. For example, an input of '0' returns a value of 0 and an input of 'A' returns a value of 10. If the input character is not a hexadecimal digit, this function returns -1.

AtlUnicodeToUTF8

Call this function to convert a Unicode string to UTF-8.

```
ATL_NOINLINE inline int AtlUnicodeToUTF8(  
    LPCWSTR wszSrc,  
    int nSrc,  
    LPSTR szDest,  
    int nDest) throw();
```

Parameters

wszSrc

The Unicode string to be converted

nSrc

The length in characters of the Unicode string.

szDest

Caller-allocated buffer to receive the converted string.

nDest

The length in bytes of the buffer.

Return Value

Returns the number of characters for the converted string.

Remarks

To determine the size of the buffer required for the converted string, call this function passing 0 for *szDest* and *nDest*.

BEncode

Call this function to convert some data using the "B" encoding.

```
inline BOOL BEncode(  
    BYTE* pbSrcData,  
    int nSrcLen,  
    LPSTR szDest,  
    int* pnDestLen,  
    LPCSTR pszCharSet) throw();
```

Parameters

pbSrcData

The buffer containing the data to be encoded.

nSrcLen

The length in bytes of the data to be encoded.

szDest

Caller-allocated buffer to receive the encoded data.

pnDestLen

Pointer to a variable that contains the length in characters of *szDest*. If the function succeeds, the variable receives the number of characters written to the buffer. If the function fails, the variable receives the required length in characters of the buffer.

pszCharSet

The character set to use for the conversion.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The "B" encoding scheme is described in RFC 2047 (<http://www.ietf.org/rfc/rfc2047.txt>).

BEncodeGetRequiredLength

Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.

```
inline int BEncodeGetRequiredLength(int nSrcLen, int nCharsetLen) throw();
```

Parameters

nSrcLen

The number of bytes of data to be encoded.

nCharsetLen

The length in characters of the character set to use for the conversion.

Return Value

The number of characters required for a buffer that could hold encoded data of *nSrcLen* bytes.

Remarks

The "B" encoding scheme is described in RFC 2047 (<http://www.ietf.org/rfc/rfc2047.txt>).

EscapeXML

Call this function to convert characters that are unsafe for use in XML to their safe equivalents.

```
inline int EscapeXML(  
    const wchar_t * szIn,  
    int nSrcLen,  
    wchar_t * szEsc,  
    int nDestLen,  
    DWORD dwFlags = ATL_ESC_FLAG_NONE) throw();
```

Parameters

sz/n

The string to be converted.

nSrcLen

The length in characters of the string to be converted.

szEsc

Caller-allocated buffer to receive the converted string.

nDestLen

The length in characters of the caller-allocated buffer.

dwFlags

ATL_ESC Flags describing how the conversion is to be performed.

- ATL_ESC_FLAG_NONE Default behavior. Quote marks and apostrophes are not converted.
- ATL_ESC_FLAG_ATTR Quote marks and apostrophes are converted to `"` and `'` respectively.

Return Value

The length in characters of the converted string.

Remarks

Possible conversions performed by this function are shown in the table:

SOURCE	DESTINATION
<	<
>	>
&	&
'	'
"	"

GetExtendedChars

Call this function to get the number of extended characters in a string.

```
inline int GetExtendedChars(LPCSTR szSrc, int nSrcLen) throw();
```

Parameters

szSrc

The string to be analyzed.

nSrcLen

The length of the string in characters.

Return Value

Returns the number of extended characters found within the string as determined by [IsExtendedChar](#).

IsExtendedChar

Call this function to find out if a given character is an extended character (less than 32, greater than 126, and not a tab, linefeed or carriage return)

```
inline int IsExtendedChar(char ch) throw();
```

Parameters

ch

The character to be tested

Return Value

TRUE if the character is extended, FALSE otherwise.

QEncode

Call this function to convert some data using the "Q" encoding.

```
inline BOOL QEncode(  
    BYTE* pbSrcData,  
    int nSrcLen,  
    LPSTR szDest,  
    int* pnDestLen,  
    LPCSTR pszCharSet,  
    int* pnNumEncoded = NULL) throw();
```

Parameters

pbSrcData

The buffer containing the data to be encoded.

nSrcLen

The length in bytes of the data to be encoded.

szDest

Caller-allocated buffer to receive the encoded data.

pnDestLen

Pointer to a variable that contains the length in characters of *szDest*. If the function succeeds, the variable receives the number of characters written to the buffer. If the function fails, the variable receives the required length in characters of the buffer.

pszCharSet

The character set to use for the conversion.

pnNumEncoded

A pointer to a variable that on return contains the number of unsafe characters that had to be converted.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The "Q" encoding scheme is described in RFC 2047 (<http://www.ietf.org/rfc/rfc2047.txt>).

QEncodeGetRequiredLength

Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.

```
inline int QEncodeGetRequiredLength(int nSrcLen, int nCharsetLen) throw();
```

Parameters

nSrcLen

The number of bytes of data to be encoded.

nCharsetLen

The length in characters of the character set to use for the conversion.

Return Value

The number of characters required for a buffer that could hold encoded data of *nSrcLen* bytes.

Remarks

The "Q" encoding scheme is described in RFC 2047 (<http://www.ietf.org/rfc/rfc2047.txt>).

QPDecode

Decodes a string of data that has been encoded in quoted-printable format such as by a previous call to [QPEncode](#).

```
inline BOOL QPDecode(  
    BYTE* pbSrcData,  
    int nSrcLen,  
    LPSTR szDest,  
    int* pnDestLen,  
    DWORD dwFlags = 0) throw();
```

Parameters

pbSrcData

[in] The buffer containing the data to be decoded.

nSrcLen

[in] The length in bytes of *pbSrcData*.

szDest

[out] Caller-allocated buffer to receive the decoded data.

pnDestLen

[out] Pointer to a variable that contains the length in bytes of *szDest*. If the function succeeds, the variable receives the number of bytes written to the buffer. If the function fails, the variable receives the required length in bytes of the buffer.

dwFlags

[in] ATLSMTP_QPENCODE flags describing how the conversion is to be performed.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The quoted-printable encoding scheme is described in RFC 2045 (<http://www.ietf.org/rfc/rfc2045.txt>).

QPDecodeGetRequiredLength

Call this function to get the size in bytes of a buffer that could contain data decoded from quoted-printable-encoded string of the specified length.

```
inline int QPDecodeGetRequiredLength(int nSrcLen) throw();
```

Parameters

nSrcLen

The number of characters in the encoded string.

Return Value

The number of bytes required for a buffer that could hold a decoded string of *nSrcLen* characters.

Remarks

The quoted-printable encoding scheme is described in RFC 2045 (<http://www.ietf.org/rfc/rfc2045.txt>).

QPEncode

Call this function to encode some data in quoted-printable format.

```
inline BOOL QPEncode(  
    BYTE* pbSrcData,  
    int nSrcLen,  
    LPSTR szDest,  
    int* pnDestLen,  
    DWORD dwFlags = 0) throw ();
```

Parameters

pbSrcData

The buffer containing the data to be encoded.

nSrcLen

The length in bytes of the data to be encoded.

szDest

Caller-allocated buffer to receive the encoded data.

pnDestLen

Pointer to a variable that contains the length in characters of *szDest*. If the function succeeds, the variable receives the number of characters written to the buffer. If the function fails, the variable receives the required length in characters of the buffer.

dwFlags

ATLSMTP_QPENCODER flags describing how the conversion is to be performed.

- ATLSMTP_QPENCODER_DOT If a period appears at the start of a line, it is added to the output as well as encoded.
- ATLSMTP_QPENCODER_TRAILING_SOFT Appends `=\r\n` to the encoded string.

The quoted-printable encoding scheme is described in [RFC 2045](#).

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The quoted-printable encoding scheme is described in RFC 2045 (<http://www.ietf.org/rfc/rfc2045.txt>).

QPEncodeGetRequiredLength

Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.


```
inline int QPEncodeGetRequiredLength(int nSrcLen) throw ();
```

Parameters

nSrcLen

The number of bytes of data to be encoded.

Return Value

The number of characters required for a buffer that could hold encoded data of *nSrcLen* bytes.

Remarks

The quoted-printable encoding scheme is described in RFC 2045 (<http://www.ietf.org/rfc/rfc2045.txt>).

UUDecode

Decodes a string of data that has been uuencoded such as by a previous call to [UUEncode](#).

```
inline BOOL UUDecode(  
    BYTE* pbSrcData,  
    int nSrcLen,  
    BYTE* pbDest,  
    int* pnDestLen) throw ();
```

Parameters

pbSrcData

The string containing the data to be decoded.

nSrcLen

The length in bytes of *pbSrcData*.

pbDest

Caller-allocated buffer to receive the decoded data.

pnDestLen

Pointer to a variable that contains the length in bytes of *pbDest*. If the function succeeds, the variable receives the number of bytes written to the buffer. If the function fails, the variable receives the required length in bytes of the buffer.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This uuencoding implementation follows the POSIX P1003.2b/D11 specification.

UUDecodeGetRequiredLength

Call this function to get the size in bytes of a buffer that could contain data decoded from a uuencoded string of the specified length.

```
inline int UUDecodeGetRequiredLength(int nSrcLen) throw ();
```

Parameters

nSrcLen

The number of characters in the encoded string.

Return Value

The number of bytes required for a buffer that could hold a decoded string of *nSrcLen* characters.

Remarks

This uuencoding implementation follows the POSIX P1003.2b/D11 specification.

UUEncode

Call this function to uuencode some data.

```
inline BOOL UUEncode(  
    const BYTE* pbSrcData,  
    int nSrcLen,  
    LPSTR szDest,  
    int* pnDestLen,  
    LPCTSTR lpszFile = _T("file"),  
    DWORD dwFlags = 0) throw ();
```

Parameters

pbSrcData

The buffer containing the data to be encoded.

nSrcLen

The length in bytes of the data to be encoded.

szDest

Caller-allocated buffer to receive the encoded data.

pnDestLen

Pointer to a variable that contains the length in characters of *szDest*. If the function succeeds, the variable receives the number of characters written to the buffer. If the function fails, the variable receives the required length in characters of the buffer.

lpszFile

The file to be added to the header when ATLSMTP_UUENCODE_HEADER is specified in *dwFlags*.

dwFlags

Flags controlling the behavior of this function.

- ATLSMTP_UUENCODE_HEADE The header will be encoded.
- ATLSMTP_UUENCODE_END The end will be encoded.
- ATLSMTP_UUENCODE_DOT Data stuffing will be performed.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This uuencoding implementation follows the POSIX P1003.2b/D11 specification.

UUEncodeGetRequiredLength

Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.

```
inline int UUEncodeGetRequiredLength(int nSrcLen) throw ();
```

Parameters

nSrcLen

The number of bytes of data to be encoded.

Return Value

The number of characters required for a buffer that could hold encoded data of *nSrcLen* bytes.

Remarks

This uuencoding implementation follows the POSIX P1003.2b/D11 specification.

See also

[Concepts](#)

[ATL COM Desktop Components](#)

ATL Path functions

3/4/2019 • 6 minutes to read • [Edit Online](#)

ATL provides the `ATLPath` class for manipulating paths in the form of `CPathT`. This code can be found in `atlpch.h`.

Related Classes

<code>CPathT</code> Class	This class represents a path.

Related Typedefs

<code>CPath</code>	A specialization of <code>CPathT</code> using <code>CString</code> .
<code>CPathA</code>	A specialization of <code>CPathT</code> using <code>CStringA</code> .
<code>CPathW</code>	A specialization of <code>CPathT</code> using <code>CStringW</code> .

Functions

<code>ATLPath::AddBackslash</code>	This function is an overloaded wrapper for <code>PathAddBackslash</code> .
<code>ATLPath::AddExtension</code>	This function is an overloaded wrapper for <code>PathAddExtension</code> .
<code>ATLPath::Append</code>	This function is an overloaded wrapper for <code>PathAppend</code> .
<code>ATLPath::BuildRoot</code>	This function is an overloaded wrapper for <code>PathBuildRoot</code> .
<code>ATLPath::Canonicalize</code>	This function is an overloaded wrapper for <code>PathCanonicalize</code> .
<code>ATLPath::Combine</code>	This function is an overloaded wrapper for <code>PathCombine</code> .
<code>ATLPath::CommonPrefix</code>	This function is an overloaded wrapper for <code>PathCommonPrefix</code> .
<code>ATLPath::CompactPath</code>	This function is an overloaded wrapper for <code>PathCompactPath</code> .
<code>ATLPath::CompactPathEx</code>	This function is an overloaded wrapper for <code>PathCompactPathEx</code> .
<code>ATLPath::FileExists</code>	This function is an overloaded wrapper for <code>PathFileExists</code> .
<code>ATLPath::FindExtension</code>	This function is an overloaded wrapper for <code>PathFindExtension</code> .
<code>ATLPath::FindFileName</code>	This function is an overloaded wrapper for <code>PathFindFileName</code> .
<code>ATLPath::GetDriveNumber</code>	This function is an overloaded wrapper for <code>PathGetDriveNumber</code> .

ATLPath::IsDirectory	This function is an overloaded wrapper for PathIsDirectory .
ATLPath::IsFileSpec	This function is an overloaded wrapper for PathIsFileSpec .
ATLPath::IsPrefix	This function is an overloaded wrapper for PathIsPrefix .
ATLPath::IsRelative	This function is an overloaded wrapper for PathIsRelative .
ATLPath::IsRoot	This function is an overloaded wrapper for PathIsRoot .
ATLPath::IsSameRoot	This function is an overloaded wrapper for PathIsSameRoot .
ATLPath::IsUNC	This function is an overloaded wrapper for PathIsUNC .
ATLPath::IsUNCServer	This function is an overloaded wrapper for PathIsUNCServer .
ATLPath::IsUNCServerShare	This function is an overloaded wrapper for PathIsUNCServerShare .
ATLPath::MakePretty	This function is an overloaded wrapper for PathMakePretty .
ATLPath::MatchSpec	This function is an overloaded wrapper for PathMatchSpec .
ATLPath::QuoteSpaces	This function is an overloaded wrapper for PathQuoteSpaces .
ATLPath::RelativePathTo	This function is an overloaded wrapper for PathRelativePathTo .
ATLPath::RemoveArgs	This function is an overloaded wrapper for PathRemoveArgs .
ATLPath::RemoveBackslash	This function is an overloaded wrapper for PathRemoveBackslash .
ATLPath::RemoveBlanks	This function is an overloaded wrapper for PathRemoveBlanks .
ATLPath::RemoveExtension	This function is an overloaded wrapper for PathRemoveExtension .
ATLPath::RemoveFileSpec	This function is an overloaded wrapper for PathRemoveFileSpec .
ATLPath::RenameExtension	This function is an overloaded wrapper for PathRenameExtension .
ATLPath::SkipRoot	This function is an overloaded wrapper for PathSkipRoot .
ATLPath::StripPath	This function is an overloaded wrapper for PathStripPath .
ATLPath::StripToRoot	This function is an overloaded wrapper for PathStripToRoot .
ATLPath::UnquoteSpaces	This function is an overloaded wrapper for PathUnquoteSpaces .

Requirements

Header: atlp.h

ATLPath::AddBackSlash

This function is an overloaded wrapper for [PathAddBackslash](#).

Syntax

```
inline char* AddBackslash(char* pszPath);  
inline wchar_t* AddBackslash(wchar_t* pszPath);
```

Remarks

See [PathAddBackslash](#) for details.

ATLPath::AddExtension

This function is an overloaded wrapper for [PathAddExtension](#).

Syntax

```
inline BOOL AddExtension(char* pszPath, const char* pszExtension);  
inline BOOL AddExtension(wchar_t* pszPath, const wchar_t* pszExtension);
```

Remarks

See [PathAddExtension](#) for details.

ATLPath::Append

This function is an overloaded wrapper for [PathAppend](#).

Syntax

```
inline BOOL Append(char* pszPath, const char* pszMore);  
inline BOOL Append(wchar_t* pszPath, const wchar_t* pszMore);
```

Remarks

See [PathAppend](#) for details.

ATLPath::BuildRoot

This function is an overloaded wrapper for [PathBuildRoot](#).

Syntax

```
inline char* BuildRoot(char* pszPath, int iDrive);  
inline wchar_t* BuildRoot(wchar_t* pszPath, int iDrive);
```

Remarks

See [PathBuildRoot](#) for details.

ATLPath::Canonicalize

This function is an overloaded wrapper for [PathCanonicalize](#).

Syntax

```
inline BOOL Canonicalize(char* pszDest, const char* pszSrc);  
inline BOOL Canonicalize(wchar_t* pszDest, const wchar_t* pszSrc);
```

Remarks

See [PathCanonicalize](#) for details.

ATLPath::Combine

This function is an overloaded wrapper for [PathCombine](#).

Syntax

```
inline char* Combine(  
    char* pszDest,  
    const char* pszDir,  
    const char* pszFile  
);  
  
inline wchar_t* Combine(  
    wchar_t* pszDest,  
    const wchar_t* pszDir,  
    const wchar_t* pszFile);
```

Remarks

See [PathCombine](#) for details.

ATLPath::CommonPrefix

This function is an overloaded wrapper for [PathCommonPrefix](#).

Syntax

```
inline int CommonPrefix(  
    const char* pszFile1,  
    const char* pszFile2,  
    char* pszDest);  
  
inline int CommonPrefix(  
    const wchar_t* pszFile1,  
    const wchar_t* pszFile2,  
    wchar_t* pszDest);
```

Remarks

See [PathCommonPrefix](#) for details.

ATLPath::CompactPath

This function is an overloaded wrapper for [PathCompactPath](#).

Syntax

```

inline BOOL CompactPath(
    HDC hDC,
    char* pszPath,
    UINT dx);

inline BOOL CompactPath(
    HDC hDC,
    wchar_t* pszPath,
    UINT dx);

```

Remarks

See [PathCompactPath](#) for details.

ATLPath::CompactPathEx

This function is an overloaded wrapper for [PathCompactPathEx](#).

Syntax

```

inline BOOL CompactPathEx(
    char* pszDest,
    const char* pszSrc,
    UINT nMaxChars,
    DWORD dwFlags);

inline BOOL CompactPathEx(
    wchar_t* pszDest,
    const wchar_t* pszSrc,
    UINT nMaxChars,
    DWORD dwFlags);

```

Remarks

See [PathCompactPathEx](#) for details.

ATLPath::FileExists

This function is an overloaded wrapper for [PathFileExists](#).

Syntax

```

inline BOOL FileExists(const char* pszPath);
inline BOOL FileExists(const wchar_t* pszPath);

```

Remarks

See [PathFileExists](#) for details.

ATLPath::FindExtension

This function is an overloaded wrapper for [PathFindExtension](#).

Syntax

```

inline char* FindExtension(const char* pszPath);
inline wchar_t* FindExtension(const wchar_t* pszPath);

```

Remarks

See [PathFindExtension](#) for details.

ATLPath::FindFileName

This function is an overloaded wrapper for [PathFindFileName](#).

Syntax

```
inline char* FindFileName(const char* pszPath);  
inline wchar_t* FindFileName(const wchar_t* pszPath);
```

Remarks

See [PathFindFileName](#) for details.

ATLPath::GetDriveNumber

This function is an overloaded wrapper for [PathGetDriveNumber](#).

Syntax

```
inline int GetDriveNumber(const char* pszPath);  
inline int GetDriveNumber(const wchar_t* pszPath);
```

Remarks

See [PathGetDriveNumber](#) for details.

ATLPath::IsDirectory

This function is an overloaded wrapper for [PathIsDirectory](#).

```
inline BOOL IsDirectory(const char* pszPath);  
inline BOOL IsDirectory(const wchar_t* pszPath);
```

Remarks

See [PathIsDirectory](#) for details.

ATLPath::IsFileSpec

This function is an overloaded wrapper for [PathIsFileSpec](#).

Syntax

```
inline BOOL IsFileSpec(const char* pszPath);  
inline BOOL IsFileSpec(const wchar_t* pszPath);
```

Remarks

See [PathIsFileSpec](#) for details.

ATLPath::IsPrefix

This function is an overloaded wrapper for [PathIsPrefix](#).

Syntax

```
inline BOOL IsPrefix(const char* pszPrefix, const char* pszPath);  
inline BOOL IsPrefix(const wchar_t* pszPrefix, const wchar_t* pszPath);
```

Remarks

See [PathIsPrefix](#) for details.

ATLPath::IsRelative

This function is an overloaded wrapper for [PathIsRelative](#).

Syntax

```
inline BOOL IsRelative(const char* pszPath);  
inline BOOL IsRelative(const wchar_t* pszPath);
```

Remarks

See [PathIsRelative](#) for details.

ATLPath::IsRoot

This function is an overloaded wrapper for [PathIsRoot](#).

Syntax

```
inline BOOL IsRoot(const char* pszPath);  
inline BOOL IsRoot(const wchar_t* pszPath);
```

Remarks

See [PathIsRoot](#) for details.

ATLPath::IsSameRoot

This function is an overloaded wrapper for [PathIsSameRoot](#).

Syntax

```
inline BOOL IsSameRoot(const char* pszPath1, const char* pszPath2);  
inline BOOL IsSameRoot(const wchar_t* pszPath1, const wchar_t* pszPath2);
```

Remarks

See [PathIsSameRoot](#) for details.

ATLPath::IsUNC

This function is an overloaded wrapper for [PathIsUNC](#).

Syntax

```
inline BOOL IsUNC(const char* pszPath);  
inline BOOL IsUNC(const wchar_t* pszPath);
```

Remarks

See [PathIsUNC](#) for details.

ATLPath::IsUNCServer

This function is an overloaded wrapper for [PathIsUNCServer](#).

Syntax

```
inline BOOL IsUNCServer(const char* pszPath);  
inline BOOL IsUNCServer(const wchar_t* pszPath);
```

Remarks

See [PathIsUNCServer](#) for details.

ATLPath::IsUNCServerShare

This function is an overloaded wrapper for [PathIsUNCServerShare](#).

Syntax

```
inline BOOL IsUNCServerShare(const char* pszPath);  
inline BOOL IsUNCServerShare(const wchar_t* pszPath);
```

Remarks

See [PathIsUNCServerShare](#) for details.

ATLPath::MakePretty

This function is an overloaded wrapper for [PathMakePretty](#).

Syntax

```
inline BOOL MakePretty(char* pszPath);  
inline BOOL MakePretty(wchar_t* pszPath);
```

Remarks

See [PathMakePretty](#) for details.

ATLPath::MatchSpec

This function is an overloaded wrapper for [PathMatchSpec](#).

Syntax

```
inline BOOL MatchSpec(const char* pszPath, const char* pszSpec);  
inline BOOL MatchSpec(const wchar_t* pszPath, const wchar_t* pszSpec);
```

Remarks

See [PathMatchSpec](#) for details.

ATLPath::QuoteSpaces

This function is an overloaded wrapper for [PathQuoteSpaces](#).

Syntax

```
inline void QuoteSpaces(char* pszPath);  
inline void QuoteSpaces(wchar_t* pszPath);
```

Remarks

See [PathQuoteSpaces](#) for details.

ATLPath::RelativePathTo

This function is an overloaded wrapper for [PathRelativePathTo](#).

Syntax

```
inline BOOL RelativePathTo(  
    char* pszPath,  
    const char* pszFrom,  
    DWORD dwAttrFrom,  
    const char* pszTo,  
    DWORD dwAttrTo);  
  
inline BOOL RelativePathTo(  
    wchar_t* pszPath,  
    const wchar_t* pszFrom,  
    DWORD dwAttrFrom,  
    const wchar_t* pszTo,  
    DWORD dwAttrTo);
```

Remarks

See [PathRelativePathTo](#) for details.

ATLPath::RemoveArgs

This function is an overloaded wrapper for [PathRemoveArgs](#).

Syntax

```
inline void RemoveArgs(char* pszPath);  
inline void RemoveArgs(wchar_t* pszPath);
```

Remarks

See [PathRemoveArgs](#) for details.

ATLPath::RemoveBackslash

This function is an overloaded wrapper for [PathRemoveBackslash](#).

Syntax

```
inline char* RemoveBackslash(char* pszPath);  
inline wchar_t* RemoveBackslash(wchar_t* pszPath);
```

Remarks

See [PathRemoveBackslash](#) for details.

ATLPath::RemoveBlanks

This function is an overloaded wrapper for [PathRemoveBlanks](#).

Syntax

```
inline void RemoveBlanks(char* pszPath);  
inline void RemoveBlanks(wchar_t* pszPath);
```

Remarks

See [PathRemoveBlanks](#) for details.

ATLPath::RemoveExtension

This function is an overloaded wrapper for [PathRemoveExtension](#).

Syntax

```
inline void RemoveExtension(char* pszPath);  
inline void RemoveExtension(wchar_t* pszPath);
```

Remarks

See [PathRemoveExtension](#) for details.

ATLPath::RemoveFileSpec

This function is an overloaded wrapper for [PathRemoveFileSpec](#).

Syntax

```
inline BOOL RemoveFileSpec(char* pszPath);  
inline BOOL RemoveFileSpec(wchar_t* pszPath);
```

Remarks

See [PathRemoveFileSpec](#) for details.

ATLPath::RenameExtension

This function is an overloaded wrapper for [PathRenameExtension](#).

Syntax

```
inline BOOL RenameExtension(char* pszPath, const char* pszExt);  
inline BOOL RenameExtension(wchar_t* pszPath, const wchar_t* pszExt);
```

Remarks

See [PathRenameExtension](#) for details.

ATLPath::SkipRoot

This function is an overloaded wrapper for [PathSkipRoot](#).

Syntax

```
inline char* SkipRoot(const char* pszPath);  
inline wchar_t* SkipRoot(const wchar_t* pszPath);
```

Remarks

See [PathSkipRoot](#) for details.

ATLPath::StripPath

This function is an overloaded wrapper for [PathStripPath](#).

Syntax

```
inline void StripPath(char* pszPath);  
inline void StripPath(wchar_t* pszPath);
```

Remarks

See [PathStripPath](#) for details.

ATLPath::StripToRoot

This function is an overloaded wrapper for [PathStripToRoot](#).

Syntax

```
inline BOOL StripToRoot(char* pszPath);  
inline BOOL StripToRoot(wchar_t* pszPath);
```

Remarks

See [PathStripToRoot](#) for details.

ATLPath::UnquoteSpaces

This function is an overloaded wrapper for [PathUnquoteSpaces](#).

Syntax

```
inline void UnquoteSpaces(char* pszPath);  
inline void UnquoteSpaces(wchar_t* pszPath);
```

Remarks

See [PathUnquoteSpaces](#) for details.

COM Map Global Functions

3/4/2019 • 2 minutes to read • [Edit Online](#)

These functions provide support for COM Map `IUnknown` implementations.

AtlInternalQueryInterface	Delegates to the <code>IUnknown</code> of a nonaggregated object.
InlinelsEqualIUnknown	Generates efficient code for comparing interfaces against <code>IUnknown</code> .

Requirements

Header: `atlbase.h`

AtlInternalQueryInterface

Retrieves a pointer to the requested interface.

```
HRESULT AtlInternalQueryInterface(  
    void* pThis,  
    const _ATL_INTMAP_ENTRY* pEntries,  
    REFIID iid,  
    void** ppvObject);
```

Parameters

pThis

[in] A pointer to the object that contains the COM map of interfaces exposed to `QueryInterface`.

pEntries

[in] An array of `_ATL_INTMAP_ENTRY` structures that access a map of available interfaces.

iid

[in] The GUID of the interface being requested.

ppvObject

[out] A pointer to the interface pointer specified in *iid*, or NULL if the interface is not found.

Return Value

One of the standard HRESULT values.

Remarks

`AtlInternalQueryInterface` only handles interfaces in the COM map table. If your object is aggregated, `AtlInternalQueryInterface` does not delegate to the outer unknown. You can enter interfaces into the COM map table with the macro `COM_INTERFACE_ENTRY` or one of its variants.

Example

```
// MyTimerProc is a callback function passed to SetTimer()
VOID CALLBACK MyTimerProc(HWND /*hwnd*/, UINT /*uMsg*/, UINT /*idEvent*/,
    DWORD /*dwTime*/)
{
    LPDISPATCH pDisp = NULL;
    // gpMyCtrl is a global variable of type CMyCtrl*
    // _GetEntries() is a static function you get with BEGIN_COM_MAP()
    AtlInternalQueryInterface (gpMyCtrl, CMyCtrl::_GetEntries(), IID_IDispatch,
        (LPVOID*)&pDisp);
    //...
    pDisp->Release ();
}
```

InlineIsEqualUnknown

Call this function, for the special case of testing for `IUnknown`.

```
BOOL InlineIsEqualUnknown(REFGUID rguid1);
```

Parameters

rguid1

[in] The GUID to compare to `IID_IUnknown`.

See also

[Functions](#)

[COM Map Macros](#)

Composite Control Global Functions

3/4/2019 • 11 minutes to read • [Edit Online](#)

These functions provide support for creating dialog boxes, and for creating, hosting and licensing ActiveX controls.

IMPORTANT

The functions listed in the following table cannot be used in applications that execute in the Windows Runtime.

AtlAxDialogBox	Creates a modal dialog box from a dialog template provided by the user. The resulting dialog box can contain ActiveX controls.
AtlAxCreateDialog	Creates a modeless dialog box from a dialog template provided by the user. The resulting dialog box can contain ActiveX controls.
AtlAxCreateControl	Creates an ActiveX control, initializes it, and hosts it in the specified window.
AtlAxCreateControlEx	Creates an ActiveX control, initializes it, hosts it in the specified window, and retrieves an interface pointer (or pointers) from the control.
AtlAxCreateControlLic	Creates a licensed ActiveX control, initializes it, and hosts it in the specified window.
AtlAxCreateControlLicEx	Creates a licensed ActiveX control, initializes it, hosts it in the specified window, and retrieves an interface pointer (or pointers) from the control.
AtlAxAttachControl	Attaches a previously created control to the specified window.
AtlAxGetHost	Used to obtain a direct interface pointer to the container for a specified window (if any), given its handle.
AtlAxGetControl	Used to obtain a direct interface pointer to the control contained inside a specified window (if any), given its handle.
AtlSetChildSite	Initializes the <code>IUnknown</code> of the child site.
AtlAxWinInit	Initializes the hosting code for AxWin objects.
AtlAxWinTerm	Uninitializes the hosting code for AxWin objects.
AtlGetObjectSourceInterface	Returns information about the default source interface of an object.

Requirements

Header: atlhost.h

AtlAxDialogBox

Creates a modal dialog box from a dialog template provided by the user.

```
ATLAPI_(int) AtlAxDialogBox(  
    HINSTANCE hInstance,  
    LPCWSTR lpTemplateName,  
    HWND hWndParent,  
    DLGPROC lpDialogProc,  
    LPARAM dwInitParam);
```

Parameters

hInstance

[in] Identifies an instance of the module whose executable file contains the dialog box template.

lpTemplateName

[in] Identifies the dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and its low-order word must contain the identifier. You can use the [MAKEINTRESOURCE](#) macro to create this value.

hWndParent

[in] Identifies the window that owns the dialog box.

lpDialogProc

[in] Points to the dialog box procedure. For more information about the dialog box procedure, see [DialogProc](#).

dwInitParam

[in] Specifies the value to pass to the dialog box in the *lParam* parameter of the WM_INITDIALOG message.

Return Value

One of the standard HRESULT values.

Remarks

To use `AtlAxDialogBox` with a dialog template that contains an ActiveX control, specify a valid CLSID, APPID or URL string as the *text* field of the **CONTROL** section of the dialog resource, along with "AtlAxWin80" as the *class name* field under the same section. The following demonstrates what a valid **CONTROL** section might look like:

```
CONTROL    "{04FE35E9-ADBC-4f1d-83FE-8FA4D1F71C7F}", IDC_TEST,  
           "AtlAxWin80", WS_GROUP | WS_TABSTOP, 0, 0, 100, 100
```

For more information on editing resource scripts, see [How to: Open a Resource Script File in Text Format](#). For more information on control resource-definition statements, see [Common Control Parameters](#) under Windows SDK: SDK Tools.

For more information on dialog boxes in general, refer to [DialogBox](#) and [CreateDialogParam](#) in the Windows SDK.

AtlAxCreateDialog

Creates a modeless dialog box from a dialog template provided by the user.

```
ATLAPI_(HWND) AtlAxCreateDialog(  
    HINSTANCE hInstance,  
    LPCWSTR lpTemplateName,  
    HWND hWndParent,  
    DLGPROC lpDialogProc,  
    LPARAM dwInitParam);
```

Parameters

hInstance

[in] Identifies an instance of the module whose executable file contains the dialog box template.

lpTemplateName

[in] Identifies the dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and its low-order word must contain the identifier. You can use the [MAKEINTRESOURCE](#) macro to create this value.

hWndParent

[in] Identifies the window that owns the dialog box.

lpDialogProc

[in] Points to the dialog box procedure. For more information about the dialog box procedure, see [DialogProc](#).

dwInitParam

[in] Specifies the value to pass to the dialog box in the *lParam* parameter of the WM_INITDIALOG message.

Return Value

One of the standard HRESULT values.

Remarks

The resulting dialog box can contain ActiveX controls.

See [CreateDialog](#) and [CreateDialogParam](#) in the Windows SDK.

AtlAxCreateControl

Creates an ActiveX control, initializes it, and hosts it in the specified window.

```
ATLAPI AtlAxCreateControl(  
    LPCOLESTR lpszName,  
    HWND hWnd,  
    IStream* pStream,  
    IUnknown** ppUnkContainer);
```

Parameters

lpszName

A pointer to a string to be passed to the control. Must be formatted in one of the following ways:

- A ProgID such as "MSCAL.Calendar.7"
- A CLSID such as "{8E27C92B-1264-101C-8A2F-040224009C02}"
- A URL such as "<http://www.microsoft.com>"
- A reference to an Active document such as "file:///\\Documents\\MyDoc.doc"
- A fragment of HTML such as "MSHTML:<HTML><BODY>This is a line of text</BODY></HTML>"

NOTE

"MSHTML:" must precede the HTML fragment so that it is designated as being an MSHTML stream.

hWnd

[in] Handle to the window that the control will be attached to.

pStream

[in] A pointer to a stream that is used to initialize the properties of the control. Can be NULL.

ppUnkContainer

[out] The address of a pointer that will receive the `IUnknown` of the container. Can be NULL.

Return Value

One of the standard HRESULT values.

Remarks

This global function gives you the same result as calling `AtlAxCreateControlEx(lpszName, hWnd, pStream, NULL, NULL, NULL, NULL)`;

To create a licensed ActiveX control, see [AtlAxCreateControlLic](#).

AtlAxCreateControlEx

Creates an ActiveX control, initializes it, and hosts it in the specified window. An interface pointer and event sink for the new control can also be created.

```
ATLAPI AtlAxCreateControlEx(  
    LPCOLESTR lpszName,  
    HWND hWnd,  
    IStream* pStream,  
    IUnknown** ppUnkContainer,  
    IUnknown** ppUnkControl,  
    REFIID iidSink = IID_NULL,  
    IUnknown* punkSink = NULL);
```

Parameters

lpszName

A pointer to a string to be passed to the control. Must be formatted in one of the following ways:

- A ProgID such as "MSCAL.Calendar.7"
- A CLSID such as "{8E27C92B-1264-101C-8A2F-040224009C02}"
- A URL such as "<http://www.microsoft.com>"
- A reference to an Active document such as "file://\\Documents\\MyDoc.doc"
- A fragment of HTML such as "MSHTML:<HTML><BODY>This is a line of text</BODY></HTML>"

NOTE

"MSHTML:" must precede the HTML fragment so that it is designated as being an MSHTML stream.

hWnd

[in] Handle to the window that the control will be attached to.

pStream

[in] A pointer to a stream that is used to initialize the properties of the control. Can be NULL.

ppUnkContainer

[out] The address of a pointer that will receive the `IUnknown` of the container. Can be NULL.

ppUnkControl

[out] The address of a pointer that will receive the `IUnknown` of the created control. Can be NULL.

iidSink

The interface identifier of an outgoing interface on the contained object.

punkSink

A pointer to the `IUnknown` interface of the sink object to be connected to the connection point specified by *iidSink* on the contained object after the contained object has been successfully created.

Return Value

One of the standard HRESULT values.

Remarks

`AtlAxCreateControlEx` is similar to [AtlAxCreateControl](#) but also allows you to receive an interface pointer to the newly created control and set up an event sink to receive events fired by the control.

To create a licensed ActiveX control, see [AtlAxCreateControlLicEx](#).

AtlAxCreateControlLic

Creates a licensed ActiveX control, initializes it, and hosts it in the specified window.

```
ATLAPI AtlAxCreateControlLic(  
    LPCOLESTR lpszName,  
    HWND hWnd,  
    IStream* pStream,  
    IUnknown** ppUnkContainer,  
    BSTR bstrLic = NULL);
```

Parameters

lpszName

A pointer to a string to be passed to the control. Must be formatted in one of the following ways:

- A ProgID such as "MSCAL.Calendar.7"
- A CLSID such as "{8E27C92B-1264-101C-8A2F-040224009C02}"
- A URL such as "<http://www.microsoft.com>"
- A reference to an Active document such as "file:///\\Documents\\MyDoc.doc"
- A fragment of HTML such as "MSHTML:<HTML><BODY>This is a line of text</BODY></HTML>"

NOTE

"MSHTML:" must precede the HTML fragment so that it is designated as being an MSHTML stream.

hWnd

Handle to the window that the control will be attached to.

pStream

A pointer to a stream that is used to initialize the properties of the control. Can be NULL.

ppUnkContainer

The address of a pointer that will receive the `IUnknown` of the container. Can be NULL.

bstrLic

The BSTR containing the license for the control.

Return Value

One of the standard HRESULT values.

Example

See [Hosting ActiveX Controls Using ATL AXHost](#) for a sample of how to use `AtlAxCreateControlLic`.

AtlAxCreateControlLicEx

Creates a licensed ActiveX control, initializes it, and hosts it in the specified window. An interface pointer and event sink for the new control can also be created.

```
ATLAPI AtlAxCreateControlLicEx(  
    LPCOLESTR lpszName,  
    HWND hWnd,  
    IStream* pStream,  
    IUnknown** ppUnkContainer,  
    IUnknown** ppUnkControl,  
    REFIID iidSink = IID_NULL,  
    IUnknown* punkSink = NULL,  
    BSTR bstrLic = NULL);
```

Parameters

lpszName

A pointer to a string to be passed to the control. Must be formatted in one of the following ways:

- A ProgID such as "MSCAL.Calendar.7"
- A CLSID such as "{8E27C92B-1264-101C-8A2F-040224009C02}"
- A URL such as "<http://www.microsoft.com>"
- A reference to an Active document such as "file://\\Documents\\MyDoc.doc"
- A fragment of HTML such as "MSHTML:<HTML><BODY>This is a line of text</BODY></HTML>"

NOTE

"MSHTML:" must precede the HTML fragment so that it is designated as being an MSHTML stream.

hWnd

Handle to the window that the control will be attached to.

pStream

A pointer to a stream that is used to initialize the properties of the control. Can be NULL.

ppUnkContainer

The address of a pointer that will receive the `IUnknown` of the container. Can be NULL.

ppUnkControl

[out] The address of a pointer that will receive the `IUnknown` of the created control. Can be NULL.

iidSink

The interface identifier of an outgoing interface on the contained object.

punkSink

A pointer to the `IUnknown` interface of the sink object to be connected to the connection point specified by *iidSink* on the contained object after the contained object has been successfully created.

bstrLic

The BSTR containing the license for the control.

Return Value

One of the standard HRESULT values.

Remarks

`AtlAxCreateControlLicEx` is similar to [AtlAxCreateControlLic](#) but also allows you to receive an interface pointer to the newly created control and set up an event sink to receive events fired by the control.

Example

See [Hosting ActiveX Controls Using ATL AXHost](#) for a sample of how to use `AtlAxCreateControlLicEx`.

AtlAxAttachControl

Attaches a previously created control to the specified window.

```
ATLAPI AtlAxAttachControl(  
    IUnknown* pControl,  
    HWND hWnd,  
    IUnknown** ppUnkContainer);
```

Parameters

pControl

[in] A pointer to the `IUnknown` of the control.

hWnd

[in] Handle to the window that will host the control.

ppUnkContainer

[out] A pointer to a pointer to the `IUnknown` of the container object.

Return Value

One of the standard HRESULT values.

Remarks

Use [AtlAxCreateControlEx](#) and [AtlAxCreateControl](#) to simultaneously create and attach a control.

NOTE

The control object being attached must be correctly initialized before calling `AtlAxAttachControl`.

AtlAxGetHost

Obtains a direct interface pointer to the container for a specified window (if any), given its handle.

```
ATLAPI AtlAxGetHost(HWND h, IUnknown** pp);
```

Parameters

h

[in] A handle to the window that is hosting the control.

pp

[out] The `IUnknown` of the container of the control.

Return Value

One of the standard HRESULT values.

AtlAxGetControl

Obtains a direct interface pointer to the control contained inside a specified window given its handle.

```
ATLAPI AtlAxGetControl(HWND h, IUnknown** pp);
```

Parameters

h

[in] A handle to the window that is hosting the control.

pp

[out] The `IUnknown` of the control being hosted.

Return Value

One of the standard HRESULT values.

AtlSetChildSite

Call this function to set the site of the child object to the `IUnknown` of the parent object.

```
HRESULT AtlSetChildSite(IUnknown* punkChild, IUnknown* punkParent);
```

Parameters

punkChild

[in] A pointer to the `IUnknown` interface of the child.

punkParent

[in] A pointer to the `IUnknown` interface of the parent.

Return Value

A standard HRESULT value.

AtlAxWinInit

This function initializes ATL's control hosting code by registering the "**AtlAxWin80**" and "**AtlAxWinLic80**" window classes plus a couple of custom window messages.

```
ATLAPI_(BOOL) AtlAxWinInit();
```

Return Value

Nonzero if the initialization of the control hosting code was successful; otherwise FALSE.

Remarks

This function must be called before using the ATL control hosting API. Following a call to this function, the **"AtlAxWin"** window class can be used in calls to [CreateWindow](#) or [CreateWindowEx](#), as described in the Windows SDK.

AtlAxWinTerm

This function uninitializes ATL's control hosting code by unregistering the **"AtlAxWin80"** and **"AtlAxWinLic80"** window classes.

```
inline BOOL AtlAxWinTerm();
```

Return Value

Always returns TRUE.

Remarks

This function simply calls [UnregisterClass](#) as described in the Windows SDK.

Call this function to clean up after all existing host windows have been destroyed if you called [AtlAxWinInit](#) and you no longer need to create host windows. If you don't call this function, the window class will be unregistered automatically when the process terminates.

AtlGetObjectSourceInterface

Call this function to retrieve information about the default source interface of an object.

```
ATLAPI AtlGetObjectSourceInterface(  
    IUnknown* punkObj,  
    GUID* plibid,  
    IID* piid,  
    unsigned short* pdwMajor,  
    unsigned short* pdwMinor);
```

Parameters

punkObj

[in] A pointer to the object for which information is to be returned.

plibid

[out] A pointer to the LIBID of the type library containing the definition of the source interface.

piid

[out] A pointer to the interface ID of the object's default source interface.

pdwMajor

[out] A pointer to the major version number of the type library containing the definition of the source interface.

pdwMinor

[out] A pointer to the minor version number of the type library containing the definition of the source interface.

Return Value

A standard HRESULT value.

Remarks

`AtlGetObjectSourceInterface` can provide you with the interface ID of the default source interface, along with the LIBID and major and minor version numbers of the type library describing that interface.

NOTE

For this function to successfully retrieve the requested information, the object represented by *punkObj* must implement `IDispatch` (and return type information through `IDispatch::GetTypeInfo`) plus it must also implement either `IProvideClassInfo2` or `IPersist`. The type information for the source interface must be in the same type library as the type information for `IDispatch`.

Example

The example below shows how you might define an event sink class, `CEasySink`, that reduces the number of template arguments that you can pass to `IDispEventImpl` to the bare essentials. `EasyAdvise` and `EasyUnadvise` use `AtlGetObjectSourceInterface` to initialize the `IDispEventImpl` members before calling `DispEventAdvise` or `DispEventUnadvise`.

```
template <UINT nID, class T>
class CEasySink : public IDispEventImpl<nID, T>
{
public:
    HRESULT EasyAdvise(IUnknown* pUnk)
    {
        AtlGetObjectSourceInterface(pUnk,
            &m_libid, &m_iid, &m_wMajorVerNum, &m_wMinorVerNum);
        return DispEventAdvise(pUnk, &m_iid);
    }
    HRESULT EasyUnadvise(IUnknown* pUnk)
    {
        AtlGetObjectSourceInterface(pUnk,
            &m_libid, &m_iid, &m_wMajorVerNum, &m_wMinorVerNum);
        return DispEventUnadvise(pUnk, &m_iid);
    }
};
```

See also

[Functions](#)

[Composite Control Macros](#)

Connection Point Global Functions

3/4/2019 • 2 minutes to read • [Edit Online](#)

These functions provide support for connection points and sink maps.

IMPORTANT

The functions listed in the following table cannot be used in applications that execute in the Windows Runtime.

AtlAdvise	Creates a connection between an object's connection point and a client's sink.
AtlUnadvise	Terminates the connection established through <code>AtlAdvise</code> .
AtlAdviseSinkMap	Advisees or unadvisees entries in an event sink map.

Requirements

Header: `atlbase.h`

AtlAdvise

Creates a connection between an object's connection point and a client's sink.

IMPORTANT

This function cannot be used in applications that execute in the Windows Runtime.

```
HRESULT AtlAdvise(  
    IUnknown* pUnkCP,  
    IUnknown* pUnk,  
    const IID& iid,  
    LPDWORD pdw);
```

Parameters

pUnkCP

[in] A pointer to the `IUnknown` of the object the client wants to connect with.

pUnk

[in] A pointer to the client's `IUnknown`.

iid

[in] The GUID of the connection point. Typically, this is the same as the outgoing interface managed by the connection point.

pdw

[out] A pointer to the cookie that uniquely identifies the connection.

Return Value

A standard HRESULT value.

Remarks

The sink implements the outgoing interface supported by the connection point. The client uses the *pdw* cookie to remove the connection by passing it to [AtlUnadvise](#).

Example

```
LPUNKNOWN m_pSourceUnk;
LPUNKNOWN m_pSinkUnk;
DWORD m_dwCustCookie;

// create source object
HRESULT hr = CoCreateInstance (CLSID_MyComponent, NULL, CLSCTX_ALL,
    IID_IUnknown, (LPVOID*)&m_pSourceUnk);
ATLASSERT(SUCCEEDED(hr));

// Create sink object. CMySink is a CComObjectRootEx-derived class
// that implements the event interface methods.
CComObject<CMySink>* pSinkClass;
CComObject<CMySink>::CreateInstance(&pSinkClass);
hr = pSinkClass->QueryInterface (IID_IUnknown, (LPVOID*)&m_pSinkUnk);
ATLASSERT(SUCCEEDED(hr));

hr = AtlAdvise (m_pSourceUnk, m_pSinkUnk, __uuidof(_IMyComponentEvents), &m_dwCustCookie);
ATLASSERT(SUCCEEDED(hr));
```

AtlUnadvise

Terminates the connection established through [AtlAdvise](#).

IMPORTANT

This function cannot be used in applications that execute in the Windows Runtime.

```
HRESULT AtlUnadvise(
    IUnknown* pUnkCP,
    const IID& iid,
    DWORD dw);
```

Parameters

pUnkCP

[in] A pointer to the `IUnknown` of the object that the client is connected with.

iid

[in] The GUID of the connection point. Typically, this is the same as the outgoing interface managed by the connection point.

dw

[in] The cookie that uniquely identifies the connection.

Return Value

A standard HRESULT value.

Example

```

LPUNKNOWN m_pSourceUnk;
LPUNKNOWN m_pSinkUnk;
DWORD m_dwCustCookie;

// create source object
HRESULT hr = CoCreateInstance (CLSID_MyComponent, NULL, CLSCTX_ALL,
    IID_IUnknown, (LPVOID*)&m_pSourceUnk);
ATLASSERT(SUCCEEDED(hr));

// Create sink object. CMySink is a CComObjectRootEx-derived class
// that implements the event interface methods.
CComObject<CMySink>* pSinkClass;
CComObject<CMySink>::CreateInstance(&pSinkClass);
hr = pSinkClass->QueryInterface (IID_IUnknown, (LPVOID*)&m_pSinkUnk);
ATLASSERT(SUCCEEDED(hr));

hr = AtlAdvise (m_pSourceUnk, m_pSinkUnk, __uuidof(_IMyComponentEvents), &m_dwCustCookie);
ATLASSERT(SUCCEEDED(hr));

// do something
CComBSTR bstrMsg(L"Hi there!");
((CMyComponent*)m_pSourceUnk)->Fire_ShowMyMsg(bstrMsg);

hr = AtlUnadvise (m_pSourceUnk, __uuidof(_IMyComponentEvents), m_dwCustCookie);
ATLASSERT(SUCCEEDED(hr));

```

AtlAdviseSinkMap

Call this function to advise or unadvise all entries in the object's sink event map.

IMPORTANT

This function cannot be used in applications that execute in the Windows Runtime.

```

HRESULT AtlAdviseSinkMap(T* pT, bool bAdvise);

```

Parameters

pT

[in] A pointer to the object containing the sink map.

bAdvise

[in] TRUE if all sink entries are to be advised; FALSE if all sink entries are to be unadvised.

Return Value

A standard HRESULT value.

Example

```

class CMyDlg :
    public CxDialogImpl<CMyDlg>
{
public:
    BEGIN_MSG_MAP(CMyDlg)
        MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
        COMMAND_HANDLER(IDOK, BN_CLICKED, OnClickedOK)
        COMMAND_HANDLER(IDCANCEL, BN_CLICKED, OnClickedCancel)
        CHAIN_MSG_MAP(CxDialogImpl<CMyDlg>)
    END_MSG_MAP()

    LRESULT OnInitDialog(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
    {
        CxDialogImpl<CMyDlg>::OnInitDialog(uMsg, wParam, lParam, bHandled);

        AtlAdviseSinkMap(this, TRUE);

        bHandled = TRUE;
        return 1; // Let the system set the focus
    }

    // Remainder of class declaration omitted.

```

See also

[Functions](#)

[Connection Point Macros](#)

Debugging and Error Reporting Global Functions

3/4/2019 • 4 minutes to read • [Edit Online](#)

These functions provide useful debugging and trace facilities.

AtlHresultFromLastError	Returns a <code>GetLastError</code> error code in the form of an HRESULT.
AtlHresultFromWin32	Converts a Win32 error code into an HRESULT.
AtlReportError	Sets up <code>IErrorInfo</code> to provide error details to a client.
AtlThrow	Throws a <code>CAtlException</code> .
AtlThrowLastWin32	Call this function to signal an error based on the result of the Windows function <code>GetLastError</code> .

AtlHresultFromLastError

Returns the calling thread's last-error code value in the form of an HRESULT.

```
HRESULT AtlHresultFromLastError();
```

Remarks

`AtlHresultFromLastError` calls `GetLastError` to obtain the last error and returns the error after converting it to an HRESULT using the `HRESULT_FROM_WIN32` macro.

Requirements

Header: `atlcomcli.h`

AtlHresultFromWin32

Converts a Win32 error code into an HRESULT.

```
AtlHresultFromWin32(DWORD error);
```

Parameters

error

The error value to convert.

Remarks

Converts a Win32 error code into an HRESULT, using the macro `HRESULT_FROM_WIN32`.

NOTE

Instead of using `HRESULT_FROM_WIN32(GetLastError())`, use the function [AtlHresultFromLastError](#).

Requirements

Header: atlcomcli.h

AtlReportError

Sets up the `IErrorInfo` interface to provide error information to clients of the object.

```
HRESULT WINAPI AtlReportError(
    const CLSID& clsid,
    LPCOLESTR lpszDesc,
    const IID& iid = GUID_NULL,
    HRESULT hRes = 0);

HRESULT WINAPI AtlReportError(
    const CLSID& clsid,
    LPCOLESTR lpszDesc,
    DWORD dwHelpID,
    LPCOLESTR lpszHelpFile,
    const IID& iid = GUID_NULL,
    HRESULT hRes = 0);

HRESULT WINAPI AtlReportError(
    const CLSID& clsid,
    LPCSTR lpszDesc,
    const IID& iid = GUID_NULL,
    HRESULT hRes = 0);

HRESULT WINAPI AtlReportError(
    const CLSID& clsid,
    LPCSTR lpszDesc,
    DWORD dwHelpID,
    LPCSTR lpszHelpFile,
    const IID& iid = GUID_NULL,
    HRESULT hRes = 0);

HRESULT WINAPI AtlReportError(
    const CLSID& clsid,
    UINT nID,
    const IID& iid = GUID_NULL,
    HRESULT hRes = 0,
    HINSTANCE hInst = _AtlBaseModule.GetResourceInstance());

HRESULT WINAPI AtlReportError(
    const CLSID& clsid,
    UINT nID,
    DWORD dwHelpID,
    LPCOLESTR lpszHelpFile,
    const IID& iid = GUID_NULL,
    HRESULT hRes = 0,
    HINSTANCE hInst = _AtlBaseModule.GetResourceInstance());
```

Parameters

clsid

[in] The CLSID of the object reporting the error.

lpszDesc

[in] The string describing the error. The Unicode versions specify that *lpszDesc* is of type LPCOLESTR; the ANSI version specifies a type of LPCSTR.

iid

[in] The IID of the interface defining the error or GUID_NULL if the error is defined by the operating system.

hRes

[in] The HRESULT you want returned to the caller.

nID

[in] The resource identifier where the error description string is stored. This value should lie between 0x0200 and 0xFFFF, inclusively. In debug builds, an **ASSERT** will result if *nID* does not index a valid string. In release builds, the error description string will be set to "Unknown Error."

dwHelpID

[in] The help context identifier for the error.

lpzHelpFile

[in] The path and name of the help file describing the error.

hInst

[in] The handle to the resource. By default, this parameter is `__AtlBaseModuleModule::GetResourceInstance`, where `__AtlBaseModuleModule` is the global instance of `CAtlBaseModule` or a class derived from it.

Return Value

If the *hRes* parameter is nonzero, returns the value of *hRes*. If *hRes* is zero, then the first four versions of `AtlReportError` return `DISP_E_EXCEPTION`. The last two versions return the result of the macro **MAKE_HRESULT(1, FACILITY_ITF, *nID*)**.

Remarks

The string *lpzDesc* is used as the text description of the error. When the client receives the *hRes* you return from `AtlReportError`, the client can access the `IErrorInfo` structure for details about the error.

Example

```
STDMETHODIMP CMyControl::MyErrorProneMethod()
{
    BOOL bSucceeded = ErrorProneFunc();
    if (bSucceeded)
        return S_OK;
    else
        // hRes is set to DISP_E_EXCEPTION
        return AtlReportError(GetObjectCLSID(), L"My error message");
}
```

Caution

Do not use `AtlReportError` in C++ catch handlers. Some overrides of these functions use the ATL string conversion macros internally, which in turn use the `_alloca` function internally. Using `AtlReportError` in a C++ catch handler can cause exceptions in C++ catch handlers.

Requirements

Header: `atlcom.h`

AtlThrow

Call this function to signal an error based on a HRESULT status code.

```
__declspec(noreturn) inline void AtlThrow(HRESULT hr);
```

Parameters

hr

Standard HRESULT value.

Remarks

This function is used by ATL and MFC code in the event of an error condition. It can also be called from your own code. The default implementation of this function depends on the definition of the symbol `_ATL_NO_EXCEPTIONS` and on the type of project, MFC or ATL.

In all cases, this function traces the HRESULT to the debugger.

In Visual Studio 2015 Update 3 and later, this function is attributed `__declspec(noreturn)` to avoid spurious SAL warnings.

If `_ATL_NO_EXCEPTIONS` is not defined in an MFC project, this function throws a [CMemoryException](#) or a [COleException](#) based on the value of the HRESULT.

If `_ATL_NO_EXCEPTIONS` is not defined in an ATL project, the function throws a [CAtlException](#).

If `_ATL_NO_EXCEPTIONS` is defined, the function causes an assertion failure instead of throwing an exception.

For ATL projects, it is possible to provide your own implementation of this function to be used by ATL in the event of a failure. To do this, define your own function with the same signature as `AtlThrow` and `#define AtlThrow` to be the name of your function. This must be done before including `atlexcept.h` (which means that it must be done prior to including any ATL headers since `atlbase.h` includes `atlexcept.h`). Attribute your function `__declspec(noreturn)` to avoid spurious SAL warnings.

Example

```
// Constructors and operators cannot return error codes, and
// so they are the place where exceptions are generally used.
class CMyClass
{
private:
    CComPtr<IBuddy> m_spBuddy;
public:
    CMyClass()
    {
        HRESULT hr = m_spBuddy.CoCreateInstance(CLSID_Buddy);
        if (FAILED(hr))
            AtlThrow(hr);
    }
    // methods ..
};
```

Requirements

Header: `atldef.h`

AtlThrowLastWin32

Call this function to signal an error based on the result of the Windows function `GetLastError`.

```
inline void AtlThrowLastWin32();
```

Remarks

This function traces the result of `GetLastError` to the debugger.

If `_ATL_NO_EXCEPTIONS` is not defined in an MFC project, this function throws a [CMemoryException](#) or a [COleException](#) based on the value returned by `GetLastError`.

If `_ATL_NO_EXCEPTIONS` is not defined in an ATL project, the function throws a [CAtlException](#).

If `_ATL_NO_EXCEPTIONS` is defined, the function causes an assertion failure instead of throwing an exception.

Requirements

Header: `atldef.h`

See also

[Functions](#)

[Debugging and Error Reporting Macros](#)

Device Context Global Functions

3/4/2019 • 2 minutes to read • [Edit Online](#)

This function creates a device context for a given device.

AtlCreateTargetDC	Creates a device context.

AtlCreateTargetDC

Creates a device context for the device specified in the [DVTARGETDEVICE](#) structure.

```
HDC AtlCreateTargetDC(HDC hdc, DVTARGETDEVICE* ptd);
```

Parameters

hdc

[in] The existing handle of a device context, or NULL.

ptd

[in] A pointer to the [DVTARGETDEVICE](#) structure that contains information about the target device.

Return Value

Returns the handle to a device context for the device specified in the [DVTARGETDEVICE](#). If no device is specified, returns the handle to the default display device.

Remarks

If the structure is NULL and *hdc* is NULL, creates a device context for the default display device.

If *hdc* is not NULL and *ptd* is NULL, the function returns the existing *hdc*.

Requirements

Header: atlwin.h

See also

[Functions](#)

Event Handling Global Functions

3/4/2019 • 2 minutes to read • [Edit Online](#)

This function provides an event handler.

IMPORTANT

The function listed in the following table cannot be used in applications that execute in the Windows Runtime.

AtlWaitWithMessageLoop	Waits for an object to be signaled, meanwhile dispatching window messages as needed.

Requirements

Header: atlbase.h

AtlWaitWithMessageLoop

Waits for the object to be signaled, meanwhile dispatching window messages as needed.

IMPORTANT

This function cannot be used in applications that execute in the Windows Runtime.

```
BOOL AtlWaitWithMessageLoop(HANDLE hEvent);
```

Parameters

hEvent

[in] The handle of the object to wait for.

Return Value

Returns TRUE if the object has been signaled.

Remarks

This is useful if you want to wait for an object's event to happen and be notified of it happening, but allow window messages to be dispatched while waiting.

See also

[Functions](#)

Marshaling Global Functions

3/4/2019 • 2 minutes to read • [Edit Online](#)

These functions provide support for marshaling and converting marshaling data into interface pointers.

IMPORTANT

The functions listed in the following table cannot be used in applications that execute in the Windows Runtime.

AtlFreeMarshalStream	Releases the marshal data and the <code>IStream</code> pointer.
AtlMarshalPtrInProc	Creates a new stream object and marshals the specified interface pointer.
AtlUnmarshalPtr	Converts a stream's marshaling data into an interface pointer.

Requirements:

Header: `atlbase.h`

AtlFreeMarshalStream

Releases the marshal data in the stream, then releases the stream pointer.

```
HRESULT AtlFreeMarshalStream(IStream* pStream);
```

Parameters

pStream

[in] A pointer to the `IStream` interface on the stream used for marshaling.

Example

See the example for [AtlMarshalPtrInProc](#).

AtlMarshalPtrInProc

Creates a new stream object, writes the CLSID of the proxy to the stream, and marshals the specified interface pointer by writing the data needed to initialize the proxy into the stream.

```
HRESULT AtlMarshalPtrInProc(  
    IUnknown* pUnk,  
    const IID& iid,  
    IStream** ppStream);
```

Parameters

pUnk

[in] A pointer to the interface to be marshaled.

iid

[in] The GUID of the interface being marshaled.

ppStream

[out] A pointer to the `IStream` interface on the new stream object used for marshaling.

Return Value

A standard HRESULT value.

Remarks

The MSHLFLAGS_TABLESTRONG flag is set so the pointer can be marshaled to multiple streams. The pointer can also be unmarshaled multiple times.

If marshaling fails, the stream pointer is released.

`AtlMarshalPtrInProc` can only be used on a pointer to an in-process object.

Example

```
//marshaling interface from one thread to another

//IStream ptr to hold serialized presentation of interface ptr
IStream* g_pStm;

//forward declaration
DWORD WINAPI ThreadProc(LPVOID lpParameter);

HRESULT WriteInterfacePtrToStream(IMyCircle *pCirc)
{
    //marshal the interface ptr to another thread
    //pCirc has to be pointer to actual object & not a proxy
    HRESULT hr = AtlMarshalPtrInProc(pCirc, IID_IMyCircle, &g_pStm);

    //m_dwThreadID is a DWORD holding thread ID of thread being created.
    CreateThread(NULL, 0, ThreadProc, NULL, 0, &m_dwThreadID);
    return hr;
}

DWORD WINAPI ThreadProc(LPVOID /*lpParameter*/)
{
    // CoInitializeEx is per thread, so initialize COM on this thread
    // (required by AtlUnmarshalPtr)
    HRESULT hr = CoInitializeEx(NULL, COINIT_APARTMENTTHREADED);
    if (SUCCEEDED(hr))
    {
        IMyCircle* pCirc;

        //unmarshal IMyCircle ptr from the stream
        hr = AtlUnmarshalPtr(g_pStm, IID_IMyCircle, (IUnknown**)&pCirc);

        // use IMyCircle ptr to call its methods
        double center;
        pCirc->get_XCenter(&center);

        //release the stream if no other thread requires it
        //to unmarshal the IMyCircle interface pointer
        hr = AtlFreeMarshalStream(g_pStm);

        CoUninitialize();
    }

    return hr;
}
```

AtlUnmarshalPtr

Converts the stream's marshaling data into an interface pointer that can be used by the client.

```
HRESULT AtlUnmarshalPtr(  
    IStream* pStream,  
    const IID& iid,  
    IUnknown** ppUnk);
```

Parameters

pStream

[in] A pointer to the stream being unmarshaled.

iid

[in] The GUID of the interface being unmarshaled.

ppUnk

[out] A pointer to the unmarshaled interface.

Return Value

A standard HRESULT value.

Example

See the example for [AtlMarshalPtrInProc](#).

See also

[Functions](#)

Pixel/HIMETRIC Conversion Global Functions

3/4/2019 • 2 minutes to read • [Edit Online](#)

These functions provide support for converting to and from pixel and HIMETRIC units.

IMPORTANT

The functions listed in the following table cannot be used in applications that execute in the Windows Runtime.

AtlHiMetricToPixel	Converts HIMETRIC units (each unit is 0.01 millimeter) to pixels.
AtlPixelToHiMetric	Converts pixels to HIMETRIC units (each unit is 0.01 millimeter).

AtlHiMetricToPixel

Converts an object's size in HIMETRIC units (each unit is 0.01 millimeter) to a size in pixels on the screen device.

```
extern void AtlHiMetricToPixel(  
    const SIZEL* lpSizeInHiMetric,  
    LPSIZEL lpSizeInPix);
```

Parameters

lpSizeInHiMetric

[in] Pointer to the size of the object in HIMETRIC units.

lpSizeInPix

[out] Pointer to where the object's size in pixels is to be returned.

Example

```
// m_sizeExtent is a member of CComControlBase that holds the  
// control's extents in HIMETRIC units.  
// Use AtlHiMetricToPixel to find the extent of the control in pixels.  
AtlHiMetricToPixel(&m_sizeExtent, &sz);  
ATLTRACE("Width = %d, Height = %d\n", sz.cx, sz.cy);
```

Requirements

Header: atlwin.h

AtlPixelToHiMetric

Converts an object's size in pixels on the screen device to a size in HIMETRIC units (each unit is 0.01 millimeter).

```
extern void AtlPixelToHiMetric(  
    const SIZEL* lpSizeInPix,  
    LPSIZEL lpSizeInHiMetric);
```

Parameters

lpSizeInPix

[in] Pointer to the object's size in pixels.

lpSizeInHiMetric

[out] Pointer to where the object's size in HIMETRIC units is to be returned.

Example

```
// Initialize our control's default size to 100 by 25 pixels
CMyControl::CMyControl()
{
    // width = 100 pixels, height = 25 pixels
    SIZE sz = { 100, 25 };
    // convert pixels to himetric
    AtlPixelToHiMetric(&sz, &m_sizeExtent);
    // store natural extent
    m_sizeNatural = m_sizeExtent;
}
```

Requirements

Header: atlwin.h

See also

[Functions](#)

Registry and TypeLib Global Functions

3/28/2019 • 6 minutes to read • [Edit Online](#)

These functions provide support for loading and registering a type library.

IMPORTANT

The functions listed in the following tables cannot be used in applications that execute in the Windows Runtime.

AfxRegCreateKey	Creates the specified registry key.
AfxRegDeleteKey	Deletes the specified registry key.
AfxRegisterPreviewHandler	A helper to register a preview handler.
AfxUnregisterPreviewHandler	A helper to unregister a preview handler.
AtlRegisterTypeLib	This function is called to register a type library.
AtlUnRegisterTypeLib	This function is called to unregister a type library
AfxRegOpenKey	Opens the specified registry key.
AfxRegOpenKeyEx	Opens the specified registry key.
AtlLoadTypeLib	This function is called to load a type library.
AtlUpdateRegistryFromResourceD	This function is called to update the registry from the supplied resource.
RegistryDataExchange	This function is called to read from, or write to, the system registry. Called by the Registry Data Exchange Macros .

These functions control which node in the registry the program uses to store information.

AtlGetPerUserRegistration	Retrieves whether the application redirects registry access to the HKEY_CURRENT_USER (HKCU) node.
AtlSetPerUserRegistration	Sets whether the application redirects registry access to the HKEY_CURRENT_USER (HKCU) node.

Requirements

Header: atlbase.h

AtlGetPerUserRegistration

Use this function to determine whether the application redirects registry access to the **HKEY_CURRENT_USER**

(**HKCU**) node.

Syntax

```
ATL__inline__ ATLAPI AtlGetPerUserRegistration(bool* pEnabled);
```

Parameters

pEnabled

[out] TRUE indicates that the registry information is directed to the **HKCU** node; FALSE indicates that the application writes registry information to the default node. The default node is **HKEY_CLASSES_ROOT (HKCR)**.

Return Value

S_OK if the method is successful, otherwise the HRESULT error code if an error occurs.

Remarks

Registry redirection is not enabled by default. If you enable this option, registry access is redirected to **HKEY_CURRENT_USER\Software\Classes**.

The redirection is not global. Only the MFC and ATL frameworks are affected by this registry redirection.

Requirements

Header: atlbase.h

AfxRegCreateKey

Creates the specified registry key.

Syntax

```
LONG AFXAPI AfxRegCreateKey(HKEY hKey, LPCTSTR lpSubKey, PHKEY phkResult, CAtlTransactionManager* pTM = NULL);
```

Parameters

hKey

A handle to an open registry key.

lpSubKey

The name of a key that this function opens or creates.

phkResult

A pointer to a variable that receives a handle to the opened or created key.

pTM

Pointer to a `CAtlTransactionManager` object.

Return Value

If the function succeeds, the return value is ERROR_SUCCESS. If the function fails, the return value is a nonzero error code defined in Winerror.h.

Requirements

Header: afxpriv.h

AfxRegDeleteKey

Deletes the specified registry key.

Syntax

```
LONG AFXAPI AfxRegDeleteKey(HKEY hKey, LPCTSTR lpSubKey, CAtlTransactionManager* pTM = NULL);
```

Parameters

hKey

A handle to an open registry key.

lpSubKey

The name of the key to be deleted.

pTM

Pointer to a `CAtlTransactionManager` object.

Return Value

If the function succeeds, the return value is `ERROR_SUCCESS`. If the function fails, the return value is a nonzero error code defined in `Winerror.h`.

Requirements

Header: `afxpriv.h`

A helper to register a preview handler.

Syntax

```
BOOL AFXAPI AfxRegisterPreviewHandler(LPCTSTR lpszCLSID, LPCTSTR lpszShortTypeName, LPCTSTR lpszFilterExt);
```

Parameters

lpszCLSID

Specifies the CLSID of handler.

lpszShortTypeName

Specifies the ProgID of handler.

lpszFilterExt

Specifies the file extension registered with this handler.

Requirements

Header: `afxdisp.h`

AtlRegisterTypeLib

This function is called to register a type library.

```
ATLAPI AtlRegisterTypeLib(HINSTANCE hInstTypeLib, LPCOLESTR lpszIndex);
```

Parameters

hInstTypeLib

The handle to the module instance.

lpszIndex

String in the format `"\\N"`, where N is the integer index of the type library resource. Can be `NULL` if no index is required.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This helper function is utilized by [AtlComModuleUnregisterServer](#) and [CAtlComModule::RegisterTypeLib](#).

Requirements

Header: atlbase.h

AfxRegOpenKey

Opens the specified registry key.

Syntax

```
LONG AFXAPI AfxRegOpenKey(HKEY hKey, LPCTSTR lpSubKey, PHKEY phkResult, CAtlTransactionManager* pTM = NULL);
```

Parameters

hKey

A handle to an open registry key.

lpSubKey

The name of a key that this function opens or creates.

phkResult

A pointer to a variable that receives a handle to the created key.

pTM

Pointer to a `CAtlTransactionManager` object.

Return Value

If the function succeeds, the return value is ERROR_SUCCESS. If the function fails, the return value is a nonzero error code defined in Winerror.h.

Requirements

Header: afxpriv.h

AfxRegOpenKeyEx

Opens the specified registry key.

Syntax

```
LONG AFXAPI AfxRegOpenKeyEx(HKEY hKey, LPCTSTR lpSubKey, DWORD ulOptions, REGSAM samDesired, PHKEY phkResult, CAtlTransactionManager* pTM = NULL);
```

Parameters

hKey

A handle to an open registry key.

lpSubKey

The name of a key that this function opens or creates.

ulOptions

This parameter is reserved and must be zero.

samDesired

A mask that specifies the desired access rights to the key.

phkResult

A pointer to a variable that receives a handle to the opened key.

pTM

Pointer to a `CAtlTransactionManager` object.

Return Value

If the function succeeds, the return value is `ERROR_SUCCESS`. If the function fails, the return value is a nonzero error code defined in `Winerror.h`.

Requirements

Header: `afxpriv.h`

AfxUnregisterPreviewHandler

A helper to unregister a preview handler.

Syntax

```
BOOL AFXAPI AfxUnRegisterPreviewHandler(LPCTSTR lpszCLSID);
```

Parameters

lpszCLSID

Specifies the CLSID of the handler to be unregistered.

Requirements

Header: `afxdisp.h`

AtlSetPerUserRegistration

Sets whether the application redirects registry access to the **HKEY_CURRENT_USER (HKCU)** node.

Syntax

```
ATLINLINE ATLAPI AtlSetPerUserRegistration(bool bEnable);
```

Parameters

bEnable

[in] `TRUE` indicates that the registry information is directed to the **HKCU** node; `FALSE` indicates that the application writes registry information to the default node. The default node is **HKEY_CLASSES_ROOT (HKCR)**.

Return Value

`S_OK` if the method is successful, otherwise the `HRESULT` error code if an error occurs.

Remarks

Registry redirection is not enabled by default. If you enable this option, registry access is redirected to **HKEY_CURRENT_USER\Software\Classes**.

The redirection is not global. Only the MFC and ATL frameworks are affected by this registry redirection.

Requirements

Header: `atlbase.h`

AtlUnRegisterTypeLib

This function is called to unregister a type library.

Syntax

```
ATLAPI AtlUnRegisterTypeLib(  
    HINSTANCE hInstTypeLib,  
    LPCOLESTR lpszIndex);
```

Parameters

hInstTypeLib

The handle to the module instance.

lpszIndex

String in the format "\\N", where N is the integer index of the type library resource. Can be NULL if no index is required.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This helper function is utilized by [CAtlComModule::UnRegisterTypeLib](#) and [AtlComModuleUnregisterServer](#).

Requirements

Header: atlbase.h

AtlLoadTypeLib

This function is called to load a type library.

Syntax

```
ATLINLINE ATLAPI AtlLoadTypeLib(  
    HINSTANCE hInstTypeLib,  
    LPCOLESTR lpszIndex,  
    BSTR* pbstrPath,  
    ITypeLib** ppTypeLib);
```

Parameters

hInstTypeLib

Handle to the module associated with the type library.

lpszIndex

String in the format "\\N", where N is the integer index of the type library resource. Can be NULL if no index is required.

pbstrPath

On successful return, contains the full path of the module associated with the type library.

ppTypeLib

On successful return, contains a pointer to a pointer to the loaded type library.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This helper function is utilized by [AtlRegisterTypeLib](#) and [AtlUnRegisterTypeLib](#).

AtlUpdateRegistryFromResourceD

This function was deprecated in Visual Studio 2013 and is removed in Visual Studio 2015.

<removed>

RegistryDataExchange

This function is called to read from, or write to, the system registry.

Syntax

```
HRESULT RegistryDataExchange(  
    T* pT,  
    enum RDXOperations rdxOp,  
    void* pItem = NULL);
```

Parameters

pT

A pointer to the current object.

rdxOp

An enum value that indicates which operation the function should perform. See the table in the Remarks section for the allowed values.

pItem

Pointer to the data that is to be read from, or written to, the registry. The data can also represent a key to be deleted from the registry. The default value is NULL.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The macros [BEGIN_RDX_MAP](#) and [END_RDX_MAP](#) expand to a function that calls `RegistryDataExchange`.

The possible enum values that indicate the operation the function should perform are shown in the following table:

ENUM VALUE	OPERATION
eReadFromReg	Read data from the registry.
eWriteToReg	Write data to the registry.
eDeleteFromReg	Delete the key from the registry.

Requirements

Header: atlbase.h

See also

[Functions](#)

[Registry Data Exchange Macros](#)

Security Global Functions

3/4/2019 • 6 minutes to read • [Edit Online](#)

These functions provide support for modifying SID and ACL objects.

IMPORTANT

The functions listed in the following table cannot be used in applications that execute in the Windows Runtime.

AtlGetDacl	Call this function to retrieve the discretionary access-control list (DACL) information of a specified object.
AtlSetDacl	Call this function to set the discretionary access-control list (DACL) information of a specified object.
AtlGetGroupSid	Call this function to retrieve the group security identifier (SID) of an object.
AtlSetGroupSid	Call this function to set the group security identifier (SID) of an object.
AtlGetOwnerSid	Call this function to retrieve the owner security identifier (SID) of an object.
AtlSetOwnerSid	Call this function to set the owner security identifier (SID) of an object.
AtlGetSacl	Call this function to retrieve the system access-control list (SACL) information of a specified object.
AtlSetSacl	Call this function to set the system access-control list (SACL) information of a specified object.
AtlGetSecurityDescriptor	Call this function to retrieve the security descriptor of a given object.

Requirements

Header: atlsecurity.h

AtlGetDacl

Call this function to retrieve the discretionary access-control list (DACL) information of a specified object.

IMPORTANT

This function cannot be used in applications that execute in the Windows Runtime.

```
inline bool AtlGetDacl(  
    HANDLE hObject,  
    SE_OBJECT_TYPE ObjectType,  
    CDacl* pDacl) throw();
```

Parameters

hObject

Handle to the object for which to retrieve the security information.

ObjectType

Specifies a value from the [SE_OBJECT_TYPE](#) enumeration that indicates the type of object identified by the *hObject* parameter.

pDacl

Pointer to a DACL object which will contain the retrieved security information.

Return Value

Returns true on success, false on failure.

Remarks

In debug builds, an assertion error will occur if either *hObject* or *pDacl* is invalid.

AtlSetDacl

Call this function to set the discretionary access-control list (DACL) information of a specified object.

IMPORTANT

This function cannot be used in applications that execute in the Windows Runtime.

```
inline bool AtlSetDacl(  
    HANDLE hObject,  
    SE_OBJECT_TYPE ObjectType,  
    const CDacl& rDacl,  
    DWORD dwInheritanceFlowControl = 0) throw(...);
```

Parameters

hObject

Handle to the object for which to set security information.

ObjectType

Specifies a value from the [SE_OBJECT_TYPE](#) enumeration that indicates the type of object identified by the *hObject* parameter.

rDacl

The DACL containing the new security information.

dwInheritanceFlowControl

The inheritance flow control. This value can be 0 (the default), [PROTECTED_DACL_SECURITY_INFORMATION](#) or [UNPROTECTED_DACL_SECURITY_INFORMATION](#).

Return Value

Returns true on success, false on failure.

Remarks

In debug builds, an assertion error will occur if *hObject* is invalid, or if *dwInheritanceFlowControl* is not one of the three permitted values.

Requirements

Header: atlsecurity.h

AtlGetGroupSid

Call this function to retrieve the group security identifier (SID) of an object.

IMPORTANT

This function cannot be used in applications that execute in the Windows Runtime.

```
inline bool AtlGetGroupSid(  
    HANDLE hObject,  
    SE_OBJECT_TYPE ObjectType,  
    CSid* pSid) throw(...);
```

Parameters

hObject

Handle to the object from which to retrieve security information.

ObjectType

Specifies a value from the [SE_OBJECT_TYPE](#) enumeration that indicates the type of object identified by the *hObject* parameter.

pSid

Pointer to a `CSid` object which will contain the new security information.

Return Value

Returns true on success, false on failure.

Requirements

Header: atlsecurity.h

AtlSetGroupSid

Call this function to set the group security identifier (SID) of an object.

IMPORTANT

This function cannot be used in applications that execute in the Windows Runtime.

```
inline bool AtlSetGroupSid(  
    HANDLE hObject,  
    SE_OBJECT_TYPE ObjectType,  
    const CSid& rSid) throw(...);
```

Parameters

hObject

Handle to the object for which to set security information.

ObjectType

Specifies a value from the [SE_OBJECT_TYPE](#) enumeration that indicates the type of object identified by the *hObject* parameter.

rSid

The `CSid` object containing the new security information.

Return Value

Returns true on success, false on failure.

Requirements

Header: atlsecurity.h

AtlGetOwnerSid

Call this function to retrieve the owner security identifier (SID) of an object.

IMPORTANT

This function cannot be used in applications that execute in the Windows Runtime.

```
inline bool AtlGetOwnerSid(  
    HANDLE hObject,  
    SE_OBJECT_TYPE ObjectType,  
    CSid* pSid) throw(...);
```

Parameters

hObject

Handle to the object from which to retrieve security information.

ObjectType

Specifies a value from the [SE_OBJECT_TYPE](#) enumeration that indicates the type of object identified by the *hObject* parameter.

pSid

Pointer to a `CSid` object which will contain the new security information.

Return Value

Returns true on success, false on failure.

Requirements

Header: atlsecurity.h

AtlSetOwnerSid

Call this function to set the owner security identifier (SID) of an object.

IMPORTANT

This function cannot be used in applications that execute in the Windows Runtime.

```
inline bool AtlSetOwnerSid(  
    HANDLE hObject,  
    SE_OBJECT_TYPE ObjectType,  
    const CSid& rSid) throw(...);
```

Parameters

hObject

Handle to the object for which to set security information.

ObjectType

Specifies a value from the [SE_OBJECT_TYPE](#) enumeration that indicates the type of object identified by the *hObject* parameter.

rSid

The `CSid` object containing the new security information.

Return Value

Returns true on success, false on failure.

Requirements

Header: atlsecurity.h

AtlGetSacl

Call this function to retrieve the system access-control list (SACL) information of a specified object.

IMPORTANT

This function cannot be used in applications that execute in the Windows Runtime.

```
inline bool AtlGetSacl(  
    HANDLE hObject,  
    SE_OBJECT_TYPE ObjectType,  
    CSacl* pSacl,  
    bool bRequestNeededPrivileges = true) throw(...);
```

Parameters

hObject

Handle to the object from which to retrieve the security information.

ObjectType

Specifies a value from the [SE_OBJECT_TYPE](#) enumeration that indicates the type of object identified by the *hObject* parameter.

pSacl

Pointer to a SACL object which will contain the retrieved security information.

bRequestNeededPrivileges

If true, the function will attempt to enable the SE_SECURITY_NAME privilege, and restore it on completion.

Return Value

Returns true on success, false on failure.

Remarks

If `AtlGetSacl` is to be called many times on many different objects, it will be more efficient to enable the `SE_SECURITY_NAME` privilege once before calling the function, with *bRequestNeededPrivileges* set to false.

Requirements

Header: atlsecurity.h

AtlSetSacl

Call this function to set the system access-control list (SACL) information of a specified object.

IMPORTANT

This function cannot be used in applications that execute in the Windows Runtime.

```
inline bool AtlSetSacl(  
    HANDLE hObject,  
    SE_OBJECT_TYPE ObjectType,  
    const CSacl& rSacl,  
    DWORD dwInheritanceFlowControl = 0,  
    bool bRequestNeededPrivileges = true) throw(...);
```

Parameters

hObject

Handle to the object for which to set security information.

ObjectType

Specifies a value from the `SE_OBJECT_TYPE` enumeration that indicates the type of object identified by the *hObject* parameter.

rSacl

The SACL containing the new security information.

dwInheritanceFlowControl

The inheritance flow control. This value can be 0 (the default), `PROTECTED_SACL_SECURITY_INFORMATION` or `UNPROTECTED_SACL_SECURITY_INFORMATION`.

bRequestNeededPrivileges

If true, the function will attempt to enable the `SE_SECURITY_NAME` privilege, and restore it on completion.

Return Value

Returns true on success, false on failure.

Remarks

In debug builds, an assertion error will occur if *hObject* is invalid, or if *dwInheritanceFlowControl* is not one of the three permitted values.

If `AtlSetSacl` is to be called many times on many different objects, it will be more efficient to enable the `SE_SECURITY_NAME` privilege once before calling the function, with *bRequestNeededPrivileges* set to false.

Requirements

Header: atlsecurity.h

AtlGetSecurityDescriptor

Call this function to retrieve the security descriptor of a given object.

IMPORTANT

This function cannot be used in applications that execute in the Windows Runtime.

```
inline bool AtlGetSecurityDescriptor(  
    LPCTSTR pszObjectName,  
    SE_OBJECT_TYPE ObjectType,  
    CSecurityDesc* pSecurityDescriptor,  
    SECURITY_INFORMATION requestedInfo = OWNER_SECURITY_INFORMATION |  
    GROUP_SECURITY_INFORMATION | DACL_SECURITY_INFORMATION |  
    SACL_SECURITY_INFORMATION,  
    bool bRequestNeededPrivileges = true) throw(...);
```

Parameters

pszObjectName

Pointer to a null-terminated string that specifies the name of the object from which to retrieve security information.

ObjectType

Specifies a value from the [SE_OBJECT_TYPE](#) enumeration that indicates the type of object identified by the *pszObjectName* parameter.

pSecurityDescriptor

The object which receives the requested security descriptor.

requestedInfo

A set of [SECURITY_INFORMATION](#) bit flags that indicate the type of security information to retrieve. This parameter can be a combination of the following values.

bRequestNeededPrivileges

If true, the function will attempt to enable the SE_SECURITY_NAME privilege, and restore it on completion.

Return Value

Returns true on success, false on failure.

Remarks

If `AtlGetSecurityDescriptor` is to be called many times on many different objects, it will be more efficient to enable the SE_SECURITY_NAME privilege once before calling the function, with *bRequestNeededPrivileges* set to false.

Requirements

Header: atlsecurity.h

See also

[Functions](#)

Security Identifier Global Functions

3/4/2019 • 2 minutes to read • [Edit Online](#)

These functions return common well-known SID objects.

IMPORTANT

The functions listed in the following table cannot be used in applications that execute in the Windows Runtime.

Sids::AccountOps	Returns the DOMAIN_ALIAS_RID_ACCOUNT_OPS SID.
Sids::Admins	Returns the DOMAIN_ALIAS_RID_ADMINS SID.
Sids::AnonymousLogon	Returns the SECURITY_ANONYMOUS_LOGON_RID SID.
Sids::AuthenticatedUser	Returns the SECURITY_AUTHENTICATED_USER_RID SID.
Sids::BackupOps	Returns the DOMAIN_ALIAS_RID_BACKUP_OPS SID.
Sids::Batch	Returns the SECURITY_BATCH_RID SID.
Sids::CreatorGroup	Returns the SECURITY_CREATOR_GROUP_RID SID.
Sids::CreatorGroupServer	Returns the SECURITY_CREATOR_GROUP_SERVER_RID SID.
Sids::CreatorOwner	Returns the SECURITY_CREATOR_OWNER_RID SID.
Sids::CreatorOwnerServer	Returns the SECURITY_CREATOR_OWNER_SERVER_RID SID.
Sids::Dialup	Returns the SECURITY_DIALUP_RID SID.
Sids::Guests	Returns the DOMAIN_ALIAS_RID_GUESTS SID.
Sids::Interactive	Returns the SECURITY_INTERACTIVE_RID SID.
Sids::Local	Returns the SECURITY_LOCAL_RID SID.
Sids::Network	Returns the SECURITY_NETWORK_RID SID.
Sids::NetworkService	Returns the SECURITY_NETWORK_SERVICE_RID SID.
Sids::Null	Returns the SECURITY_NULL_RID SID.
Sids::PreW2KAccess	Returns the DOMAIN_ALIAS_RID_PREW2KCOMPACCESS SID.
Sids::PowerUsers	Returns the DOMAIN_ALIAS_RID_POWER_USERS SID.

Sids::PrintOps	Returns the DOMAIN_ALIAS_RID_PRINT_OPS SID.
Sids::Proxy	Returns the SECURITY_PROXY_RID SID.
Sids::RasServers	Returns the DOMAIN_ALIAS_RID_RAS_SERVERS SID.
Sids::Replicator	Returns the DOMAIN_ALIAS_RID_REPLICATOR SID.
Sids::RestrictedCode	Returns the SECURITY_RESTRICTED_CODE_RID SID.
Sids::Self	Returns the SECURITY_PRINCIPAL_SELF_RID SID.
Sids::ServerLogon	Returns the SECURITY_SERVER_LOGON_RID SID.
Sids::Service	Returns the SECURITY_SERVICE_RID SID.
Sids::System	Returns the SECURITY_LOCAL_SYSTEM_RID SID.
Sids::SystemOps	Returns the DOMAIN_ALIAS_RID_SYSTEM_OPS SID.
Sids::TerminalServer	Returns the SECURITY_TERMINAL_SERVER_RID SID.
Sids::Users	Returns the DOMAIN_ALIAS_RID_USERS SID.
Sids::World	Returns the SECURITY_WORLD_RID SID.

Requirements

Header: atlsecurity.h

Sids::AccountOps

Returns the DOMAIN_ALIAS_RID_ACCOUNT_OPS SID.

```
CSid AccountOps() throw(...);
```

Sids::Admins

Returns the DOMAIN_ALIAS_RID_ADMINS SID.

```
CSid Admins() throw(...);
```

Sids::AnonymousLogon

Returns the SECURITY_ANONYMOUS_LOGON_RID SID.

```
CSid AnonymousLogon() throw(...);
```

Sids::AuthenticatedUser

Returns the SECURITY_AUTHENTICATED_USER_RID SID.

```
CSid AuthenticatedUser() throw(...);
```

Sids::BackupOps

Returns the DOMAIN_ALIAS_RID_BACKUP_OPS SID.

```
CSid BackupOps() throw(...);
```

Sids::Batch

Returns the SECURITY_BATCH_RID SID.

```
CSid Batch() throw(...);
```

Sids::CreatorGroup

Returns the SECURITY_CREATOR_GROUP_RID SID.

```
CSid CreatorGroup() throw(...);
```

Sids::CreatorGroupServer

Returns the SECURITY_CREATOR_GROUP_SERVER_RID SID.

```
CSid CreatorGroupServer() throw(...);
```

Sids::CreatorOwner

Returns the SECURITY_CREATOR_OWNER_RID SID.

```
CSid CreatorOwner() throw(...);
```

Sids::CreatorOwnerServer

Returns the SECURITY_CREATOR_OWNER_SERVER_RID SID.

```
CSid CreatorOwnerServer() throw(...);
```

Sids::Dialup

Returns the SECURITY_DIALUP_RID SID.

```
CSid Dialup() throw(...);
```

Sids::Guests

Returns the DOMAIN_ALIAS_RID_GUESTS SID.

```
CSid Guests() throw(...);
```

Sids::Interactive

Returns the SECURITY_INTERACTIVE_RID SID.

```
CSid Interactive() throw(...);
```

Sids::Local

Returns the SECURITY_LOCAL_RID SID.

```
CSid Local() throw(...);
```

Sids::Network

Returns the SECURITY_NETWORK_RID SID.

```
CSid Network() throw(...);
```

Sids::NetworkService

Returns the SECURITY_NETWORK_SERVICE_RID SID.

```
CSid NetworkService() throw(...);
```

Remarks

Use NetworkService to enable the NT AUTHORITY\NetworkService user to read a CPerfMon security object. NetworkService adds a SecurityAttribute to the ATLServer code which will allow the DLL to login under the NetworkService account on Windows XP Home Edition, Windows XP Professional, Windows Server 2003 and greater operating system.

When custom log counters are created with ATLServer CPerfMon class in the Perfmon MMC, the counters may not appear when viewing the log file although they will appear correctly in the realtime view. CPerfMon custom performance counters don't have the necessary permissions to run under the "Performance Logs and Alerts" service (smlogsvc.exe) on Windows XP Home Edition, Windows XP Professional, Windows Server 2003 (or greater) operating systems. This service runs under the "NT AUTHORITY\NetworkService" account.

Sids::Null

Returns the SECURITY_NULL_RID SID.

```
CSid Null() throw(...);
```

Sids::PreW2KAccess

Returns the DOMAIN_ALIAS_RID_PREW2KCOMPACCESS SID.

```
CSid PreW2KAccess() throw(...);
```

Sids::PowerUsers

Returns the DOMAIN_ALIAS_RID_POWER_USERS SID.

```
CSid PowerUsers() throw(...);
```

Sids::PrintOps

Returns the DOMAIN_ALIAS_RID_PRINT_OPS SID.

```
CSid PrintOps() throw(...);
```

Sids::Proxy

Returns the SECURITY_PROXY_RID SID.

```
CSid Proxy() throw(...);
```

Sids::RasServers

Returns the DOMAIN_ALIAS_RID_RAS_SERVERS SID.

```
CSid RasServers() throw(...);
```

Sids::Replicator

Returns the DOMAIN_ALIAS_RID_REPLICATOR SID.

```
CSid Replicator() throw(...);
```

Sids::RestrictedCode

Returns the SECURITY_RESTRICTED_CODE_RID SID.

```
CSid RestrictedCode() throw(...);
```

Sids::Self

Returns the SECURITY_PRINCIPAL_SELF_RID SID.

```
CSid Self() throw(...);
```

Sids::ServerLogon

Returns the SECURITY_SERVER_LOGON_RID SID.

```
CSid ServerLogon() throw(...);
```

Sids::Service

Returns the SECURITY_SERVICE_RID SID.

```
CSid Service() throw(...);
```

Sids::System

Returns the SECURITY_LOCAL_SYSTEM_RID SID.

```
CSid System() throw(...);
```

Sids::SystemOps

Returns the DOMAIN_ALIAS_RID_SYSTEM_OPS SID.

```
CSid SystemOps() throw(...);
```

Sids::TerminalServer

Returns the SECURITY_TERMINAL_SERVER_RID SID.

```
CSid TerminalServer() throw(...);
```

Sids::Users

Returns the DOMAIN_ALIAS_RID_USERS SID.

```
CSid Users() throw(...);
```

Sids::World

Returns the SECURITY_WORLD_RID SID.

```
CSid World() throw(...);
```

See also

Server Registration Global Functions

3/4/2019 • 2 minutes to read • [Edit Online](#)

These functions provide support for registering and unregistering server objects in the object map.

IMPORTANT

The functions listed in the following table cannot be used in applications that execute in the Windows Runtime.

AtlComModuleRegisterServer	This function is called to register every object in the object map.
AtlComModuleUnregisterServer	This function is called to unregister every object in the object map.
AtlComModuleRegisterClassObjects	This function is called to register class objects.
AtlComModuleRevokeClassObjects	This function is called to revoke class objects from a COM module.
AtlComModuleGetClassObject	This function is called to get the class object.

Requirements

Header: atlbase.h

AtlComModuleRegisterServer

This function is called to register every object in the object map.

```
ATLINLINE ATLAPI AtlComModuleRegisterServer(  
    _ATL_COM_MODULE* pComModule,  
    BOOL bRegTypeLib,  
    const CLSID* pCLSID);
```

Parameters

pComModule

Pointer to the COM module.

bRegTypeLib

TRUE if the type library is to be registered.

pCLSID

Points to the CLSID of the object to be registered. If NULL, all objects in the object map will be registered.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

`AtlComModuleRegisterServer` walks the ATL autogenerated object map and registers each object in the map. If *pCLSID* is not NULL, then only the object referred to by *pCLSID* is registered; otherwise all of the objects are registered.

This function is called by [CAtlComModule::RegisterServer](#).

AtlComModuleUnregisterServer

This function is called to unregister every object in the object map.

```
ATLINLINE ATLAPI AtlComModuleUnregisterServer(  
    _ATL_COM_MODULE* pComModule,  
    BOOL bUnRegTypeLib,  
    const CLSID* pCLSID);
```

Parameters

pComModule

Pointer to the COM module.

bUnRegTypeLib

TRUE if the type library is to be registered.

pCLSID

Points to the CLSID of the object to be unregistered. If NULL all objects in the object map will be unregistered.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

`AtlComModuleUnregisterServer` walks the ATL object map and unregisters each object in the map. If *pCLSID* is not NULL, then only the object referred to by *pCLSID* is unregistered; otherwise all of the objects are unregistered.

This function is called by [CAtlComModule::UnregisterServer](#).

AtlComModuleRegisterClassObjects

This function is called to register class objects.

```
ATLINLINE ATLAPI AtlComModuleRegisterClassObjects(  
    _ATL_COM_MODULE* pComModule,  
    DWORD dwClsContext,  
    DWORD dwFlags);
```

Parameters

pComModule

Pointer to the COM module.

dwClsContext

Specifies the context in which the class object is to be run. Possible values are CLSCTX_INPROC_SERVER, CLSCTX_INPROC_HANDLER, or CLSCTX_LOCAL_SERVER. See [CLSCTX](#) for more details.

dwFlags

Determines the connection types to the class object. Possible values are REGCLS_SINGLEUSE, REGCLS_MULTIPLEUSE, or REGCLS_MULTI_SEPARATE. See [REGCLS](#) for more details.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This helper function is utilized by [CComModule::RegisterClassObjects](#) (obsolete in ATL 7.0) and [CAtlExeModuleT::RegisterClassObjects](#).

AtlComModuleRevokeClassObjects

This function is called to remove the class factory/factories from the Running Object Table.

```
ATLINLINE ATLAPI AtlComModuleRevokeClassObjects(_ATL_COM_MODULE* pComModule);
```

Parameters

pComModule

Pointer to the COM module.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This helper function is utilized by [CComModule::RevokeClassObjects](#) (obsolete in ATL 7.0) and [CAtlExeModuleT::RevokeClassObjects](#).

AtlComModuleGetClassObject

This function is called to return the class factory.

```
ATLINLINE ATLAPI AtlComModuleGetClassObject(  
    _ATL_COM_MODULE* pComModule,  
    REFCLSID rclsid,  
    REFIID riid,  
    LPVOID* ppv);
```

Parameters

pComModule

Pointer to the COM module.

rclsid

The CLSID of the object to be created.

riid

The IID of the requested interface.

ppv

A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppv* is set to NULL.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This helper function is utilized by [CComModule::GetClassObject](#) (obsolete in ATL 7.0) and [CAtlDllModuleT::GetClassObject](#).

See also

WinModule Global Functions

3/4/2019 • 2 minutes to read • [Edit Online](#)

These functions provide support for `_AtlCreateWndData` structure operations.

IMPORTANT

The functions listed in the following table cannot be used in applications that execute in the Windows Runtime.

AtlWinModuleAddCreateWndData	This function is used to initialize and add an <code>_AtlCreateWndData</code> structure.
AtlWinModuleExtractCreateWndData	Call this function to extract an existing <code>_AtlCreateWndData</code> structure.

Requirements

Header: `atlbase.h`

AtlWinModuleAddCreateWndData

This function is used to initialize and add an `_AtlCreateWndData` structure.

```
ATLINLINE ATLAPI_(void) AtlWinModuleAddCreateWndData(  
    _ATL_WIN_MODULE* pWinModule,  
    _AtlCreateWndData* pData,  
    void* pObject);
```

Parameters

pWinModule

Pointer to a module's `_ATL_WIN_MODULE70` structure.

pData

Pointer to the `_AtlCreateWndData` structure to be initialized and added to the current module.

pObject

Pointer to an object's **this** pointer.

Remarks

Initializes an `_AtlCreateWndData` structure, which is used to store the **this** pointer used to refer to class instances, and adds it to the list referenced by a module's `_ATL_WIN_MODULE70` structure. Called by [CAtlWinModule::AddCreateWndData](#).

AtlWinModuleExtractCreateWndData

Call this function to extract an existing `_AtlCreateWndData` structure.

```
ATL_INLINE ATLAPI_(void*) AtlWinModuleExtractCreateWndData(_ATL_WIN_MODULE* pWinModule);
```

Parameters

pWinModule

Pointer to a module's [_ATL_WIN_MODULE70](#) structure.

Return Value

Returns a pointer to the [_AtlCreateWndData](#) structure.

Remarks

This function will extract an existing [_AtlCreateWndData](#) structure from the list referenced by a module's [_ATL_WIN_MODULE70](#) structure.

See also

[Functions](#)

ATL Macros

3/4/2019 • 2 minutes to read • [Edit Online](#)

To find an ATL macro by category, see the following topics.

[Aggregation and Class Factory Macros](#)

Provide ways of controlling aggregation and of declaring class factories.

[Category Macros](#)

Define category maps.

[COM Map Macros](#)

Define COM interface maps.

[Compiler Options Macros](#)

Control specific compiler features.

[Composite Control Macros](#)

Define event sink maps and entries.

[Connection Point Macros](#)

Define connection point maps and entries.

[Debugging and Error Reporting Macros](#)

Provide useful debugging and trace facilities.

[Exception Handling Macros](#)

Provide support for exception handling.

[Message Map Macros](#)

Define message maps and entries.

[Object Map Macros](#)

Define object maps and entries.

[Object Status Macros](#)

Sets flags belonging to ActiveX controls.

[Property Map Macros](#)

Define property maps and entries.

[Registry Data Exchange Macros](#)

Perform Registry Data Exchange operations.

[Registry Macros](#)

Define useful type library and registry facilities.

[Service Map Macros](#)

Define service maps and entries.

[Snap-In Object Macros](#)

Provide support for snap-in extensions.

[String Conversion Macros](#)

Provide string conversion features.

[Window Class Macros](#)

Define window class utilities.

[Windows Messages Macros](#)

Forward window messages.

See also

[ATL COM Desktop Components](#)

[Functions](#)

[Global Variables](#)

[Classes and structs](#)

[Typedefs](#)

Aggregation and Class Factory Macros

3/4/2019 • 8 minutes to read • [Edit Online](#)

These macros provide ways of controlling aggregation and of declaring class factories.

DECLARE_AGGREGATABLE	Declares that your object can be aggregated (the default).
DECLARE_CLASSFACTORY	Declares the class factory to be CComClassFactory , the ATL default class factory.
DECLARE_CLASSFACTORY_EX	Declares your class factory object to be the class factory.
DECLARE_CLASSFACTORY2	Declares CComClassFactory2 to be the class factory.
DECLARE_CLASSFACTORY_AUTO_THREAD	Declares CComClassFactoryAutoThread to be the class factory.
DECLARE_CLASSFACTORY_SINGLETON	Declares CComClassFactorySingleton to be the class factory.
DECLARE_GET_CONTROLLING_UNKNOWN	Declares a virtual <code>GetControllingUnknown</code> function.
DECLARE_NOT_AGGREGATABLE	Declares that your object cannot be aggregated.
DECLARE_ONLY_AGGREGATABLE	Declares that your object must be aggregated.
DECLARE_POLY_AGGREGATABLE	Checks the value of the outer unknown and declares your object aggregatable or not aggregatable, as appropriate.
DECLARE_PROTECT_FINAL_CONSTRUCT	Protects the outer object from deletion during construction of an inner object.
DECLARE_VIEW_STATUS	Specifies the VIEWSTATUS flags to the container.

Requirements

Header: atlcom.h

DECLARE_AGGREGATABLE

Specifies that your object can be aggregated.

```
DECLARE_AGGREGATABLE( x )
```

Parameters

x

[in] The name of the class you are defining as aggregatable.

Remarks

[CComCoClass](#) contains this macro to specify the default aggregation model. To override this default, specify either the [DECLARE_NOT_AGGREGATABLE](#) or [DECLARE_ONLY_AGGREGATABLE](#) macro in your class definition.

Example

```
class ATL_NO_VTABLE CNoAggClass :
    public CComObjectRoot,
    public CComCoClass<CNoAggClass, &CLSID_NoAggClass>
{
public:
    CNoAggClass()
    {
    }

    DECLARE_NOT_AGGREGATABLE(CNoAggClass)
};
```

DECLARE_CLASSFACTORY

Declares [CComClassFactory](#) to be the class factory.

```
DECLARE_CLASSFACTORY()
```

Remarks

[CComCoClass](#) uses this macro to declare the default class factory for your object.

Example

```
class ATL_NO_VTABLE CMyClass :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CMyClass, &CLSID_MyClass>,
    public IDispatchImpl<IMyClass, &IID_IMyClass, &LIBID_NVC_ATL_COMLib, /*wMajor =*/ 1, /*wMinor =*/ 0>,
    public IDispatchImpl<IMyDualInterface, &__uuidof(IMyDualInterface), &LIBID_NVC_ATL_COMLib, /* wMajor = */
1, /* wMinor = */ 0>
{
public:
    DECLARE_CLASSFACTORY()

    // Remainder of class declaration omitted
```

CComClassFactory Class

This class implements the [IClassFactory](#) interface.

```
class CComClassFactory : public IClassFactory,
    public CComObjectRootEx<CComGlobalsThreadModel>
```

Remarks

`CComClassFactory` implements the [IClassFactory](#) interface, which contains methods for creating an object of a particular CLSID, as well as locking the class factory in memory to allow new objects to be created more quickly.

`IClassFactory` must be implemented for every class that you register in the system registry and to which you assign a CLSID.

ATL objects normally acquire a class factory by deriving from [CComCoClass](#). This class includes the macro [DECLARE_CLASSFACTORY](#), which declares `CComClassFactory` as the default class factory. To override this

default, specify one of the `DECLARE_CLASSFACTORYXXX` macros in your class definition. For example, the `DECLARE_CLASSFACTORY_EX` macro uses the specified class for the class factory:

```
class ATL_NO_VTABLE CMyCustomClass :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CMyCustomClass, &CLSID_MyCustomClass>,
    public IDispatchImpl<IMyCustomClass, &IID_IMyCustomClass, &LIBID_NVC_ATL_COMLib, /*wMajor=*/ 1, /*wMinor=*/ 0>
{
public:
    DECLARE_CLASSFACTORY_EX(CMyClassFactory)

    // Remainder of class declaration omitted.
```

The above class definition specifies that `CMyClassFactory` will be used as the object's default class factory.

`CMyClassFactory` must derive from `CComClassFactory` and override `CreateInstance`.

ATL provides three other macros that declare a class factory:

- `DECLARE_CLASSFACTORY2` Uses `CComClassFactory2`, which controls creation through a license.
- `DECLARE_CLASSFACTORY_AUTO_THREAD` Uses `CComClassFactoryAutoThread`, which creates objects in multiple apartments.
- `DECLARE_CLASSFACTORY_SINGLETON` Uses `CComClassFactorySingleton`, which constructs a single `CComObjectGlobal` object.

DECLARE_CLASSFACTORY_EX

Declares `cf` to be the class factory.

```
DECLARE_CLASSFACTORY_EX( cf )
```

Parameters

cf

[in] The name of the class that implements your class factory object.

Remarks

The *cf* parameter must derive from `CComClassFactory` and override the `CreateInstance` method.

`CComCoClass` includes the `DECLARE_CLASSFACTORY` macro, which specifies `CComClassFactory` as the default class factory. However, by including the `DECLARE_CLASSFACTORY_EX` macro in your object's class definition, you override this default.

Example

```
class ATL_NO_VTABLE CMyCustomClass :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CMyCustomClass, &CLSID_MyCustomClass>,
    public IDispatchImpl<IMyCustomClass, &IID_IMyCustomClass, &LIBID_NVC_ATL_COMLib, /*wMajor=*/ 1, /*wMinor=*/ 0>
{
public:
    DECLARE_CLASSFACTORY_EX(CMyClassFactory)

    // Remainder of class declaration omitted.
```

DECLARE_CLASSFACTORY2

Declares [CComClassFactory2](#) to be the class factory.

```
DECLARE_CLASSFACTORY2( lic )
```

Parameters

lic

[in] A class that implements `VerifyLicenseKey`, `GetLicenseKey`, and `IsLicenseValid`.

Remarks

[CComCoClass](#) includes the [DECLARE_CLASSFACTORY](#) macro, which specifies [CComClassFactory](#) as the default class factory. However, by including the [DECLARE_CLASSFACTORY2](#) macro in your object's class definition, you override this default.

Example

```
class ATL_NO_VTABLE CMyClass2 :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CMyClass2, &CLSID_MyClass>,
    public IDispatchImpl<IMyClass, &IID_IMyClass, &LIBID_NVC_ATL_COMLib, /*wMajor =*/ 1, /*wMinor =*/ 0>,
    public IDispatchImpl<IMyDualInterface, &__uuidof(IMyDualInterface), &LIBID_NVC_ATL_COMLib, /* wMajor = */
1, /* wMinor = */ 0>
{
public:
    DECLARE_CLASSFACTORY2(CMyLicense)

    // Remainder of class declaration omitted
```

CComClassFactory2 Class

This class implements the [IClassFactory2](#) interface.

```
template <class license>
class CComClassFactory2 : public IClassFactory2,
    public CComObjectRootEx<CComGlobalsThreadModel>,
    public license
```

Parameters

license

A class that implements the following static functions:

- `static BOOL VerifyLicenseKey( BSTR bstr );`
- `static BOOL GetLicenseKey( DWORD dwReserved, BSTR * pBstr );`
- `static BOOL IsLicenseValid( );`

Remarks

`CComClassFactory2` implements the [IClassFactory2](#) interface, which is an extension of [IClassFactory](#).

`IClassFactory2` controls object creation through a license. A class factory executing on a licensed machine can provide a run-time license key. This license key allows an application to instantiate objects when a full machine license does not exist.

ATL objects normally acquire a class factory by deriving from [CComCoClass](#). This class includes the macro [DECLARE_CLASSFACTORY](#), which declares [CComClassFactory](#) as the default class factory. To use

`CComClassFactory2`, specify the `DECLARE_CLASSFACTORY2` macro in your object's class definition. For example:

```
class ATL_NO_VTABLE CMyClass2 :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CMyClass2, &CLSID_MyClass>,
public IDispatchImpl<IMyClass, &IID_IMyClass, &LIBID_NVC_ATL_COMLib, /*wMajor =*/ 1, /*wMinor =*/ 0>,
public IDispatchImpl<IMyDualInterface, &__uuidof(IMyDualInterface), &LIBID_NVC_ATL_COMLib, /* wMajor = */
1, /* wMinor = */ 0>
{
public:
    DECLARE_CLASSFACTORY2(CMyLicense)

    // Remainder of class declaration omitted
```

`CMyLicense`, the template parameter to `CComClassFactory2`, must implement the static functions `VerifyLicenseKey`, `GetLicenseKey`, and `IsLicenseValid`. The following is an example of a simple license class:

```
class CMyLicense
{
protected:
    static BOOL VerifyLicenseKey(BSTR bstr)
    {
        USES_CONVERSION;
        return !lstrcmp(OLE2T(bstr), _T("My run-time license key"));
    }

    static BOOL GetLicenseKey(DWORD /*dwReserved*/, BSTR* pBstr)
    {
        USES_CONVERSION;
        *pBstr = SysAllocString( T2OLE(_T("My run-time license key")));
        return TRUE;
    }

    static BOOL IsLicenseValid() { return TRUE; }
};
```

`CComClassFactory2` derives from both `CComClassFactory2Base` and `license`. `CComClassFactory2Base`, in turn, derives from `IClassFactory2` and **`CComObjectRootEx< CComGlobalsThreadModel >`**.

DECLARE_CLASSFACTORY_AUTO_THREAD

Declares `CComClassFactoryAutoThread` to be the class factory.

```
DECLARE_CLASSFACTORY_AUTO_THREAD()
```

Remarks

`CComCoClass` includes the `DECLARE_CLASSFACTORY` macro, which specifies `CComClassFactory` as the default class factory. However, by including the `DECLARE_CLASSFACTORY_AUTO_THREAD` macro in your object's class definition, you override this default.

When you create objects in multiple apartments (in an out-of-proc server), add `DECLARE_CLASSFACTORY_AUTO_THREAD` to your class.

Example

```
class ATL_NO_VTABLE CMyAutoClass :
    public CComObjectRootEx<CComMultiThreadModel>,
    public CComCoClass<CMyAutoClass, &CLSID_MyAutoClass>,
    public IMyAutoClass
{
public:
    DECLARE_CLASSFACTORY_AUTO_THREAD()

    // Remainder of class declaration omitted.
```

CComClassFactoryAutoThread Class

This class implements the [IClassFactory](#) interface, and allows objects to be created in multiple apartments.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

```
class CComClassFactoryAutoThread : public IClassFactory,
    public CComObjectRootEx<CComGlobalsThreadModel>
```

Remarks

`CComClassFactoryAutoThread` is similar to [CComClassFactory](#), but allows objects to be created in multiple apartments. To take advantage of this support, derive your EXE module from [CComAutoThreadModule](#).

ATL objects normally acquire a class factory by deriving from [CComCoClass](#). This class includes the macro [DECLARE_CLASSFACTORY](#), which declares [CComClassFactory](#) as the default class factory. To use `CComClassFactoryAutoThread`, specify the [DECLARE_CLASSFACTORY_AUTO_THREAD](#) macro in your object's class definition. For example:

```
class ATL_NO_VTABLE CMyAutoClass :
    public CComObjectRootEx<CComMultiThreadModel>,
    public CComCoClass<CMyAutoClass, &CLSID_MyAutoClass>,
    public IMyAutoClass
{
public:
    DECLARE_CLASSFACTORY_AUTO_THREAD()

    // Remainder of class declaration omitted.
```

DECLARE_CLASSFACTORY_SINGLETON

Declares [CComClassFactorySingleton](#) to be the class factory.

```
DECLARE_CLASSFACTORY_SINGLETON( obj )
```

Parameters

obj

[in] The name of your class object.

Remarks

[CComCoClass](#) includes the [DECLARE_CLASSFACTORY](#) macro, which specifies [CComClassFactory](#) as the default class factory. However, by including the [DECLARE_CLASSFACTORY_SINGLETON](#) macro in your object's class

definition, you override this default.

Example

```
class ATL_NO_VTABLE CMySingletonClass :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CMySingletonClass, &CLSID_MySingletonClass>,
    public IMySingletonClass
{
public:
    DECLARE_CLASSFACTORY_SINGLETON(CMySingletonClass)

    // Remainder of class declaration omitted.
```

CComClassFactorySingleton Class

This class derives from [CComClassFactory](#) and uses [CComObjectGlobal](#) to construct a single object.

IMPORTANT

This class and its members cannot be used in applications that execute in the Windows Runtime.

```
template<class T>
class CComClassFactorySingleton : public CComClassFactory
```

Parameters

T

Your class.

`CComClassFactorySingleton` derives from [CComClassFactory](#) and uses [CComObjectGlobal](#) to construct a single object. Each call to the `CreateInstance` method simply queries this object for an interface pointer.

Remarks

ATL objects normally acquire a class factory by deriving from [CComCoClass](#). This class includes the macro [DECLARE_CLASSFACTORY](#), which declares `CComClassFactory` as the default class factory. To use `CComClassFactorySingleton`, specify the [DECLARE_CLASSFACTORY_SINGLETON](#) macro in your object's class definition. For example:

```
class ATL_NO_VTABLE CMySingletonClass :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CMySingletonClass, &CLSID_MySingletonClass>,
    public IMySingletonClass
{
public:
    DECLARE_CLASSFACTORY_SINGLETON(CMySingletonClass)

    // Remainder of class declaration omitted.
```

DECLARE_GET_CONTROLLING_UNKNOWN

Declares a virtual function `GetControllingUnknown`.

```
DECLARE_GET_CONTROLLING_UNKNOWN()
```

Remarks

Add this macro to your object if you get the compiler error message that `GetControllingUnknown` is undefined (for example, in `CComAggregateCreator`).

DECLARE_NOT_AGGREGATABLE

Specifies that your object cannot be aggregated.

```
DECLARE_NOT_AGGREGATABLE( x )
```

Parameters

x

[in] The name of the class object you are defining as not aggregatable.

Remarks

DECLARE_NOT_AGGREGATABLE causes `CreateInstance` to return an error (CLASS_E_NOAGGREGATION) if an attempt is made to aggregate onto your object.

By default, `CComCoClass` contains the `DECLARE_AGGREGATABLE` macro, which specifies that your object can be aggregated. To override this default behavior, include DECLARE_NOT_AGGREGATABLE in your class definition.

Example

```
class ATL_NO_VTABLE CNoAggClass :
    public CComObjectRoot,
    public CComCoClass<CNoAggClass, &CLSID_NoAggClass>
{
public:
    CNoAggClass()
    {
    }

    DECLARE_NOT_AGGREGATABLE(CNoAggClass)
};
```

DECLARE_ONLY_AGGREGATABLE

Specifies that your object must be aggregated.

```
DECLARE_ONLY_AGGREGATABLE( x )
```

Parameters

x

[in] The name of the class object you are defining as only aggregatable.

Remarks

DECLARE_ONLY_AGGREGATABLE causes an error (E_FAIL) if an attempt is made to `CoCreate` your object as nonaggregated object.

By default, `CComCoClass` contains the `DECLARE_AGGREGATABLE` macro, which specifies that your object can be aggregated. To override this default behavior, include DECLARE_ONLY_AGGREGATABLE in your class definition.

Example

```

class ATL_NO_VTABLE COnlyAggClass :
    public CComObjectRoot,
    public CComCoClass<COnlyAggClass, &CLSID_OnlyAggClass>
{
public:
    COnlyAggClass()
    {
    }

    DECLARE_ONLY_AGGREGATABLE(COnlyAggClass)
};

```

DECLARE_POLY_AGGREGATABLE

Specifies that an instance of **CComPolyObject** < x > is created when your object is created.

```
DECLARE_POLY_AGGREGATABLE( x )
```

Parameters

x

[in] The name of the class object you are defining as aggregatable or not aggregatable.

Remarks

During creation, the value of the outer unknown is checked. If it is NULL, `IUnknown` is implemented for a nonaggregated object. If the outer unknown is not NULL, `IUnknown` is implemented for an aggregated object.

The advantage of using `DECLARE_POLY_AGGREGATABLE` is that you avoid having both `CComAggObject` and `CComObject` in your module to handle the aggregated and nonaggregated cases. A single `CComPolyObject` object handles both cases. This means only one copy of the vtable and one copy of the functions exist in your module. If your vtable is large, this can substantially decrease your module size. However, if your vtable is small, using `CComPolyObject` can result in a slightly larger module size because it is not optimized for an aggregated or nonaggregated object, as are `CComAggObject` and `CComObject`.

The `DECLARE_POLY_AGGREGATABLE` macro is automatically declared in your object if you use the ATL Control Wizard to create a full control.

DECLARE_PROTECT_FINAL_CONSTRUCT

Protects your object from being deleted if (during [FinalConstruct](#)) the internal aggregated object increments the reference count then decrements the count to 0.

```
DECLARE_PROTECT_FINAL_CONSTRUCT()
```

DECLARE_VIEW_STATUS

Place this macro in an ATL ActiveX control's control class to specify the `VIEWSTATUS` flags to the container.

```
DECLARE_VIEW_STATUS( statusFlags )
```

Parameters

statusFlags

[in] The `VIEWSTATUS` flags. See [VIEWSTATUS](#) for a list of flags.

Example

```
DECLARE_VIEW_STATUS(VIEWSTATUS_SOLIDBKGND | VIEWSTATUS_OPAQUE)
```

See also

[Macros](#)

Category Macros

3/12/2019 • 3 minutes to read • [Edit Online](#)

These macros define category maps.

BEGIN_CATEGORY_MAP	Marks the beginning of the category map.
END_CATEGORY_MAP	Marks the end of the category map.
IMPLEMENTED_CATEGORY	Indicates categories that are implemented by the COM object.
REQUIRED_CATEGORY	Indicates categories that are required of the container by the COM object.

Requirements

Header: atlcom.h

BEGIN_CATEGORY_MAP

Marks the beginning of the category map.

```
BEGIN_CATEGORY_MAP(theClass)
```

Parameters

theClass

[in] The name of the class containing the category map.

Remarks

The category map is used to specify which component categories the COM class will implement and which categories it requires from its container.

Add an [IMPLEMENTED_CATEGORY](#) entry to the map for each category implemented by the COM class. Add a [REQUIRED_CATEGORY](#) entry to the map for each category that the class requires its clients to implement. Mark the end of the map with the [END_CATEGORY_MAP](#) macro.

The component categories listed in the map will be registered automatically when the module is registered if the class has an associated [OBJECT_ENTRY_AUTO](#) or [OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO](#).

NOTE

ATL uses the standard component categories manager to register component categories. If the manager is not present on the system when the module is registered, registration succeeds, but the component categories will not be registered for that class.

For more information about component categories, see [What are Component Categories and how do they work](#) in the Windows SDK.

Example

```
BEGIN_CATEGORY_MAP(CMyCtrl)
    IMPLEMENTED_CATEGORY(CATID_Insertable)
END_CATEGORY_MAP()
```

END_CATEGORY_MAP

Marks the end of the category map.

```
END_CATEGORY_MAP()
```

Example

See the example for [BEGIN_CATEGORY_MAP](#).

IMPLEMENTED_CATEGORY

Add an IMPLEMENTED_CATEGORY macro to your component's [category map](#) to specify that it should be registered as implementing the category identified by the *catID* parameter.

```
IMPLEMENTED_CATEGORY(catID)
```

Parameters

catID

[in] A CATID constant or variable holding the globally unique identifier (GUID) for the implemented category. The address of *catID* will be taken and added to the map. See the table below for a selection of stock categories.

Remarks

The component categories listed in the map will be registered automatically when the module is registered if the class has an associated [OBJECT_ENTRY_AUTO](#) or [OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO](#) macro.

Clients can use the category information registered for the class to determine its capabilities and requirements without having to create an instance of it.

For more information about component categories, see [What are Component Categories and how do they work](#) in the Windows SDK.

A Selection of Stock Categories

DESCRIPTION	SYMBOL	REGISTRY GUID
Safe For Scripting	CATID_SafeForScripting	{7DD95801-9882-11CF-9FA9-00AA006C42C4}
Safe For Initialization	CATID_SafeForInitializing	{7DD95802-9882-11CF-9FA9-00AA006C42C4}
Simple Frame Site Containment	CATID_SimpleFrameControl	{157083E0-2368-11cf-87B9-00AA006C8166}
Simple Data Binding	CATID_PropertyNotifyControl	{157083E1-2368-11cf-87B9-00AA006C8166}

DESCRIPTION	SYMBOL	REGISTRY GUID
Advanced Data Binding	CATID_VBDataBound	{157083E2-2368-11cf-87B9-00AA006C8166}
Windowless Controls	CATID_WindowlessObject	{1D06B600-3AE3-11cf-87B9-00AA006C8166}
Internet-Aware Objects	See Internet Aware Objects in the Windows SDK for a sample list.	

Example

```
BEGIN_CATEGORY_MAP(CMyCtrl)
    IMPLEMENTED_CATEGORY(CATID_Insertable)
END_CATEGORY_MAP()
```

REQUIRED_CATEGORY

Add a `REQUIRED_CATEGORY` macro to your component's [category map](#) to specify that it should be registered as requiring the category identified by the *catID* parameter.

```
REQUIRED_CATEGORY( catID )
```

Parameters

catID

[in] A CATID constant or variable holding the globally unique identifier (GUID) for the required category. The address of *catID* will be taken and added to the map. See the table below for a selection of stock categories.

Remarks

The component categories listed in the map will be registered automatically when the module is registered if the class has an associated [OBJECT_ENTRY_AUTO](#) or [OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO](#) macro.

Clients can use the category information registered for the class to determine its capabilities and requirements without having to create an instance of it. For example, a control may require that a container support data binding. The container can find out if it has the capabilities necessary to host the control by querying the category manager for the categories required by that control. If the container does not support a required feature, it can refuse to host the COM object.

For more information about component categories, including a sample list, see [What are Component Categories and how do they work](#) in the Windows SDK.

A Selection of Stock Categories

DESCRIPTION	SYMBOL	REGISTRY GUID
Safe For Scripting	CATID_SafeForScripting	{7DD95801-9882-11CF-9FA9-00AA006C42C4}
Safe For Initialization	CATID_SafeForInitializing	{7DD95802-9882-11CF-9FA9-00AA006C42C4}
Simple Frame Site Containment	CATID_SimpleFrameControl	{157083E0-2368-11cf-87B9-00AA006C8166}

DESCRIPTION	SYMBOL	REGISTRY GUID
Simple Data Binding	CATID_PropertyNotifyControl	{157083E1-2368-11cf-87B9-00AA006C8166}
Advanced Data Binding	CATID_VBDataBound	{157083E2-2368-11cf-87B9-00AA006C8166}
Windowless Controls	CATID_WindowlessObject	{1D06B600-3AE3-11cf-87B9-00AA006C8166}
Internet-Aware Objects	See Internet Aware Objects in the Windows SDK for a sample list.	

Example

```
BEGIN_CATEGORY_MAP(CMyWindow)
    REQUIRED_CATEGORY(CATID_InternetAware)
END_CATEGORY_MAP()
```

See also

[Macros](#)

COM Map Macros

4/1/2019 • 2 minutes to read • [Edit Online](#)

These macros define COM interface maps.

BEGIN_COM_MAP	Marks the beginning of the COM interface map entries.
END_COM_MAP	Marks the end of the COM interface map entries.

Requirements

Header: atlcom.h

BEGIN_COM_MAP

The COM map is the mechanism that exposes interfaces on an object to a client through `QueryInterface`.

```
BEGIN_COM_MAP(x)
```

Parameters

x

[in] The name of the class object you are exposing interfaces on.

Remarks

[CComObjectRootEx::InternalQueryInterface](#) only returns pointers for interfaces in the COM map. Start your interface map with the `BEGIN_COM_MAP` macro, add entries for each of your interfaces with the `COM_INTERFACE_ENTRY` macro or one of its variants, and complete the map with the `END_COM_MAP` macro.

Example

From the ATL [BEEPER](#) sample:

```
BEGIN_COM_MAP(CBeeper)
    COM_INTERFACE_ENTRY(IBeeper)
    COM_INTERFACE_ENTRY(IDispatch)
    COM_INTERFACE_ENTRY_TEAR_OFF(IID_ISupportErrorInfo, CBeeper2)
END_COM_MAP()
```

END_COM_MAP

Ends the definition of your COM interface map.

```
END_COM_MAP()
```

See also

[Macros](#)

COM_INTERFACE_ENTRY Macros

3/4/2019 • 7 minutes to read • [Edit Online](#)

These macros enter an object's interfaces into its COM map so that they can be accessed by `QueryInterface`. The order of entries in the COM map is the order interfaces will be checked for a matching IID during `QueryInterface`.

<code>COM_INTERFACE_ENTRY</code>	Enters interfaces into the COM interface map.
<code>COM_INTERFACE_ENTRY2</code>	Use this macro to disambiguate two branches of inheritance.
<code>COM_INTERFACE_ENTRY_IID</code>	Use this macro to enter the interface into the COM map and specify its IID.
<code>COM_INTERFACE_ENTRY2_IID</code>	Same as <code>COM_INTERFACE_ENTRY2</code> , except you can specify a different IID.
<code>COM_INTERFACE_ENTRY_AGGREGATE</code>	When the interface identified by <i>iid</i> is queried for, <code>COM_INTERFACE_ENTRY_AGGREGATE</code> forwards to <code>punk</code> .
<code>COM_INTERFACE_ENTRY_AGGREGATE_BLIND</code>	Same as <code>COM_INTERFACE_ENTRY_AGGREGATE</code> , except that querying for any IID results in forwarding the query to <i>punk</i> .
<code>COM_INTERFACE_ENTRY_AUTOAGGREGATE</code>	Same as <code>COM_INTERFACE_ENTRY_AGGREGATE</code> , except if <i>punk</i> is NULL, it automatically creates the aggregate described by the <i>clsid</i> .
<code>COM_INTERFACE_ENTRY_AUTOAGGREGATE_BLIND</code>	Same as <code>COM_INTERFACE_ENTRY_AUTOAGGREGATE</code> , except that querying for any IID results in forwarding the query to <i>punk</i> , and if <i>punk</i> is NULL, automatically creating the aggregate described by the <i>clsid</i> .
<code>COM_INTERFACE_ENTRY_BREAK</code>	Causes your program to call <code>DebugBreak</code> when the specified interface is queried for.
<code>COM_INTERFACE_ENTRY_CACHED_TEAR_OFF</code>	Saves the interface-specific data for every instance.
<code>COM_INTERFACE_ENTRY_TEAR_OFF</code>	Exposes your tear-off interfaces.
<code>COM_INTERFACE_ENTRY_CHAIN</code>	Processes the COM map of the base class when the processing reaches this entry in the COM map.
<code>COM_INTERFACE_ENTRY_FUNC</code>	A general mechanism for hooking into ATL's <code>QueryInterface</code> logic.
<code>COM_INTERFACE_ENTRY_FUNC_BLIND</code>	Same as <code>COM_INTERFACE_ENTRY_FUNC</code> , except that querying for any IID results in a call to <i>func</i> .
<code>COM_INTERFACE_ENTRY_NOINTERFACE</code>	Returns <code>E_NOINTERFACE</code> and terminates COM map processing when the specified interface is queried for.

Requirements

Header: atlcom.h

COM_INTERFACE_ENTRY

Enters interfaces into the COM interface map.

Syntax

```
COM_INTERFACE_ENTRY( x )
```

Parameters

x

[in] The name of an interface your class object derives from directly.

Remarks

Typically, this is the entry type you use most often.

Example

```
BEGIN_COM_MAP(CThisExample)
    COM_INTERFACE_ENTRY(IThisExample)
    COM_INTERFACE_ENTRY(IDispatch)
    COM_INTERFACE_ENTRY(ISupportErrorInfo)
END_COM_MAP()
```

Requirements

Header: atlcom.h

COM_INTERFACE_ENTRY2

Use this macro to disambiguate two branches of inheritance.

```
COM_INTERFACE_ENTRY2(x, x2)
```

Parameters

x

[in] The name of an interface you want to expose from your object.

x2

[in] The name of the inheritance branch from which *x* is exposed.

Remarks

For example, if you derive your class object from two dual interfaces, you expose `IDispatch` using `COM_INTERFACE_ENTRY2` since `IDispatch` can be obtained from either one of the interfaces.

Example

```

class ATL_NO_VTABLE CEntry2Example :
    public CEntry2ExampleBase, // CEntry2ExampleBase derives from IDispatch
    public IDispatchImpl<IEntry2Example, &IID_IEntry2Example, &LIBID_NVC_ATL_WindowingLib, /*wMajor =*/ 1,
/*wMinor =*/ 0>,
    public CComCoClass<CEntry2Example, &CLSID_Entry2Example>
{
public:
    CEntry2Example()
    {
    }

    BEGIN_COM_MAP(CEntry2Example)
        COM_INTERFACE_ENTRY(IEntry2Example)
        COM_INTERFACE_ENTRY2(IDispatch, IEntry2Example)
    END_COM_MAP()
};

```

COM_INTERFACE_ENTRY_IID

Use this macro to enter the interface into the COM map and specify its IID.

```
COM_INTERFACE_ENTRY_IID(iid, x)
```

Parameters

iid

[in] The GUID of the interface exposed.

x

[in] The name of the class whose vtable will be exposed as the interface identified by *iid*.

Example

```

BEGIN_COM_MAP(CExample)
    COM_INTERFACE_ENTRY(IExample)
    COM_INTERFACE_ENTRY_IID(IID_IDispatch, CExampleDispatch)
    COM_INTERFACE_ENTRY(IExampleBase)
    COM_INTERFACE_ENTRY(ISupportErrorInfo)
END_COM_MAP()

```

COM_INTERFACE_ENTRY2_IID

Same as [COM_INTERFACE_ENTRY2](#), except you can specify a different IID.

```
COM_INTERFACE_ENTRY2_IID(iid, x, x2)
```

Parameters

iid

[in] The GUID you are specifying for the interface.

x

[in] The name of an interface that your class object derives from directly.

x2

[in] The name of a second interface that your class object derives from directly.

COM_INTERFACE_ENTRY_AGGREGATE

When the interface identified by *iid* is queried for, COM_INTERFACE_ENTRY_AGGREGATE forwards to *punk*.

```
COM_INTERFACE_ENTRY_AGGREGATE(iid, punk)
```

Parameters

iid

[in] The GUID of the interface queried for.

punk

[in] The name of an `IUnknown` pointer.

Remarks

The *punk* parameter is assumed to point to the inner unknown of an aggregate or to NULL, in which case the entry is ignored. Typically, you would `CoCreate` the aggregate in `FinalConstruct`.

Example

```
BEGIN_COM_MAP(COuter1)
    COM_INTERFACE_ENTRY_AGGREGATE(__uuidof(IAgg), m_punkAgg)
END_COM_MAP()
```

COM_INTERFACE_ENTRY_AGGREGATE_BLIND

Same as [COM_INTERFACE_ENTRY_AGGREGATE](#), except that querying for any IID results in forwarding the query to *punk*.

```
COM_INTERFACE_ENTRY_AGGREGATE_BLIND(punk)
```

Parameters

punk

[in] The name of an `IUnknown` pointer.

Remarks

If the interface query fails, processing of the COM map continues.

Example

```
BEGIN_COM_MAP(COuter2)
    COM_INTERFACE_ENTRY_AGGREGATE_BLIND(m_punkAggBlind)
END_COM_MAP()
```

COM_INTERFACE_ENTRY_AUTOAGGREGATE

Same as [COM_INTERFACE_ENTRY_AGGREGATE](#), except if *punk* is NULL, it automatically creates the aggregate described by the *clsid*.

```
COM_INTERFACE_ENTRY_AUTOAGGREGATE(iid, punk, clsid)
```

Parameters

iid

[in] The GUID of the interface queried for.

punk

[in] The name of an `IUnknown` pointer. Must be a member of the class containing the COM map.

clsid

[in] The identifier of the aggregate that will be created if *punk* is NULL.

Remarks

Example

```
BEGIN_COM_MAP(COuter3)
    COM_INTERFACE_ENTRY_AUTOAGGREGATE(__uuidof(IAutoAgg), m_punkAutoAgg, CLSID_CAutoAgg)
END_COM_MAP()
```

COM_INTERFACE_ENTRY_AUTOAGGREGATE_BLIND

Same as [COM_INTERFACE_ENTRY_AUTOAGGREGATE](#), except that querying for any IID results in forwarding the query to *punk*, and if *punk* is NULL, automatically creating the aggregate described by the *clsid*.

```
COM_INTERFACE_ENTRY_AUTOAGGREGATE_BLIND(punk, clsid)
```

Parameters

punk

[in] The name of an `IUnknown` pointer. Must be a member of the class containing the COM map.

clsid

[in] The identifier of the aggregate that will be created if *punk* is NULL.

Remarks

If the interface query fails, processing of the COM map continues.

Example

```
BEGIN_COM_MAP(COuter4)
    COM_INTERFACE_ENTRY_AUTOAGGREGATE_BLIND(m_punkAutoAggB, CLSID_CAutoAggB)
END_COM_MAP()
```

COM_INTERFACE_ENTRY_BREAK

Causes your program to call [DebugBreak](#) when the specified interface is queried for.

```
COM_INTERFACE_ENTRY_BREAK(x)
```

Parameters

x

[in] Text used to construct the interface identifier.

Remarks

The interface IID will be constructed by appending *x* to `IID_`. For example, if *x* is `IPersistStorage`, the IID will be `IID_IPersistStorage`.

COM_INTERFACE_ENTRY_CACHED_TEAR_OFF

Saves the interface-specific data for every instance.

```
COM_INTERFACE_ENTRY_CACHED_TEAR_OFF(iid, x, punk)
```

Parameters

iid

[in] The GUID of the tear-off interface.

x

[in] The name of the class implementing the interface.

punk

[in] The name of an `IUnknown` pointer. Must be a member of the class containing the COM map. Should be initialized to NULL in the class object's constructor.

Remarks

If the interface is not used, this lowers the overall instance size of your object.

Example

```
BEGIN_COM_MAP(COuter)
    COM_INTERFACE_ENTRY(IOuter)
    COM_INTERFACE_ENTRY_CACHED_TEAR_OFF(IID_ITearOff, CTearOff, punkTearOff)
END_COM_MAP()
```

COM_INTERFACE_ENTRY_TEAR_OFF

Exposes your tear-off interfaces.

```
COM_INTERFACE_ENTRY_TEAR_OFF(iid, x)
```

Parameters

iid

[in] The GUID of the tear-off interface.

x

[in] The name of the class implementing the interface.

Remarks

A tear-off interface is implemented as a separate object that is instantiated every time the interface it represents is queried for. Typically, you build your interface as a tear-off if the interface is rarely used, since this saves a vtable pointer in every instance of your main object. The tear-off is deleted when its reference count becomes zero. The class implementing the tear-off should be derived from `CComTearOffObjectBase` and have its own COM map.

Example

```
BEGIN_COM_MAP(CBeeper)
    COM_INTERFACE_ENTRY(IBeeper)
    COM_INTERFACE_ENTRY(IDispatch)
    COM_INTERFACE_ENTRY_TEAR_OFF(IID_ISupportErrorInfo, CBeeper2)
END_COM_MAP()
```

COM_INTERFACE_ENTRY_CHAIN

Processes the COM map of the base class when the processing reaches this entry in the COM map.

```
COM_INTERFACE_ENTRY_CHAIN(classname)
```

Parameters

classname

[in] A base class of the current object.

Remarks

For example, in the following code:

```
BEGIN_COM_MAP(COuterObject)
    COM_INTERFACE_ENTRY2(IDispatch, IOuterObject)
    COM_INTERFACE_ENTRY_CHAIN(CBase)
END_COM_MAP()
```

Note that the first entry in the COM map must be an interface on the object containing the COM map. Thus, you cannot start your COM map entries with `COM_INTERFACE_ENTRY_CHAIN`, which causes the COM map of a different object to be searched at the point where **`COM_INTERFACE_ENTRY_CHAIN(COtherObject)`** appears in your object's COM map. If you want to search the COM map of another object first, add an interface entry for `IUnknown` to your COM map, then chain the other object's COM map. For example:

```
BEGIN_COM_MAP(CThisObject)
    COM_INTERFACE_ENTRY(IUnknown)
    COM_INTERFACE_ENTRY_CHAIN(CBase)
END_COM_MAP()
```

COM_INTERFACE_ENTRY_FUNC

A general mechanism for hooking into ATL's `QueryInterface` logic.

```
COM_INTERFACE_ENTRY_FUNC(iid, dw, func)
```

Parameters

iid

[in] The GUID of the interface exposed.

dw

[in] A parameter passed through to the *func*.

func

[in] The function pointer that will return *iid*.

Remarks

If *iid* matches the IID of the interface queried for, then the function specified by *func* is called. The declaration for the function should be:

```
HRESULT WINAPI func(void* pv, REFIID riid, LPVOID* ppv, DWORD_PTR dw);
```

When your function is called, `pv` points to your class object. The *riid* parameter refers to the interface being queried for, `ppv` is the pointer to the location where the function should store the pointer to the interface, and *dw*

is the parameter you specified in the entry. The function should set *ppv to NULL and return E_NOINTERFACE or S_FALSE if it chooses not to return an interface. With E_NOINTERFACE, COM map processing terminates. With S_FALSE, COM map processing continues, even though no interface pointer was returned. If the function returns an interface pointer, it should return S_OK.

COM_INTERFACE_ENTRY_FUNC_BLIND

Same as [COM_INTERFACE_ENTRY_FUNC](#), except that querying for any IID results in a call to *func*.

```
COM_INTERFACE_ENTRY_FUNC_BLIND(dw, func)
```

Parameters

dw

[in] A parameter passed through to the *func*.

func

[in] The function that gets called when this entry in the COM map is processed.

Remarks

Any failure will cause processing to continue on the COM map. If the function returns an interface pointer, it should return S_OK.

COM_INTERFACE_ENTRY_NOINTERFACE

Returns E_NOINTERFACE and terminates COM map processing when the specified interface is queried for.

```
COM_INTERFACE_ENTRY_NOINTERFACE(x)
```

Parameters

x

[in] Text used to construct the interface identifier.

Remarks

You can use this macro to prevent an interface from being used in a particular case. For example, you can insert this macro into your COM map right before COM_INTERFACE_ENTRY_AGGREGATE_BLIND to prevent a query for the interface from being forwarded to the aggregate's inner unknown.

The interface IID will be constructed by appending *x* to IID_. For example, if *x* is IPersistStorage, the IID will be IID_IPersistStorage.

Compiler Options Macros

5/7/2019 • 5 minutes to read • [Edit Online](#)

These macros control specific compiler features.

_ATL_ALL_WARNINGS	A symbol that enables errors in projects converted from previous versions of ATL.
_ATL_APARTMENT_THREADED	Define if one or more of your objects use apartment threading.
_ATL_CSTRING_EXPLICIT_CONSTRUCTORS	Makes certain <code>CString</code> constructors explicit, preventing any unintentional conversions.
_ATL_ENABLE_PTM_WARNING	Define this macro in order to use C++ standard compliant syntax, which generates the C4867 compiler error when a non-standard syntax is used to initialize a pointer to a member function.
_ATL_FREE_THREADED	Define if one or more of your objects use free or neutral threading.
_ATL_MULTI_THREADED	A symbol that indicates the project will have objects that are marked as Both, Free or Neutral. The macro _ATL_FREE_THREADED should be used instead.
_ATL_NO_AUTOMATIC_NAMESPACE	A symbol that prevents the default use of namespace as ATL.
_ATL_NO_COM_SUPPORT	A symbol that prevents COM-related code from being compiled with your project.
ATL_NO_VTABLE	A symbol that prevents the vtable pointer from being initialized in the class's constructor and destructor.
ATL_NOINLINE	A symbol that indicates a function should not be inlined.
_ATL_SINGLE_THREADED	Define if all of your objects use the single threading model.

[_ATL_ALL_WARNINGS](#)

A symbol that enables errors in projects converted from previous versions of ATL.

```
#define _ATL_ALL_WARNINGS
```

Remarks

Before Visual C++ .NET 2002, ATL disabled a lot of warnings and left them disabled so that they never showed up in user code. Specifically:

- C4127 conditional expression is constant

- C4786 'identifier' : identifier was truncated to 'number' characters in the debug information
- C4201 nonstandard extension used : nameless struct/union
- C4103 'filename' : used #pragma pack to change alignment
- C4291 'declaration' : no matching operator delete found; memory will not be freed if initialization throws an exception
- C4268 'identifier' : 'const' static/global data initialized with compiler-generated default constructor fills the object with zeros
- C4702 unreachable code

In projects converted from previous versions, these warnings are still disabled by the libraries headers.

By adding the following line to the stdafx.h file before including libraries headers, this behavior can be changed.

```
#define _ATL_ALL_WARNINGS
```

If this `#define` is added, the ATL headers are careful to preserve the state of these warnings so that they are not disabled globally (or if the user explicitly disables individual warnings, not to enable them).

New projects have this `#define` set in stdafx.h by default.

`_ATL_APARTMENT_THREADED`

Define if one or more of your objects use apartment threading.

```
_ATL_APARTMENT_THREADED
```

Remarks

Specifies apartment threading. See [Specifying the Project's Threading Model](#) for other threading options, and [Options, ATL Simple Object Wizard](#) for a description of the threading models available for an ATL object.

`_ATL_CSTRING_EXPLICIT_CONSTRUCTORS`

Makes certain `CString` constructors explicit, preventing any unintentional conversions.

```
_ATL_CSTRING_EXPLICIT_CONSTRUCTORS
```

Remarks

When this is defined, all `CString` constructors that take a single parameter are compiled with the `explicit` keyword, which prevents implicit conversions of input arguments. This means for example, that when `_UNICODE` is defined, if you attempt use a `char*` string as a `CString` constructor argument, a compiler error will result. Use this macro in situations where you need to prevent implicit conversions between narrow and wide string types.

By using the `_T` macro on all constructor string arguments, you can define `_ATL_CSTRING_EXPLICIT_CONSTRUCTORS` and avoid compile errors regardless of whether `_UNICODE` is defined.

`_ATL_ENABLE_PTM_WARNING`

Define this macro in order to force the use of ANSI C++ standard-compliant syntax for pointer to member functions. Using this macro will cause the C4867 compiler error to be generated when non-standard syntax is used

to initialize a pointer to a member function.

```
#define _ATL_ENABLE_PTM_WARNING
```

Remarks

The ATL and MFC libraries have been changed to match the Microsoft C++ compiler's improved standard C++ compliance. According to the ANSI C++ standard, the syntax of a pointer to a class member function should be

```
&CMyClass::MyFunc .
```

When [_ATL_ENABLE_PTM_WARNING](#) is not defined (the default case), ATL/MFC disables the C4867 error in macro maps (notably message maps) so that code that was created in earlier versions can continue to build as before. If you define **_ATL_ENABLE_PTM_WARNING**, your code should be C++ standard compliant.

However, the non-standard form has been deprecated, so you need to move existing code to C++ standard compliant syntax. For example, the following:

```
BEGIN_MESSAGE_MAP(CMFCListViewDoc, CDocument)
    ON_COMMAND(ID_MYCOMMAND, OnMycommand)
END_MESSAGE_MAP()
```

Should be changed to:

```
BEGIN_MESSAGE_MAP(CMFCListViewDoc, CDocument)
    ON_COMMAND(ID_MYCOMMAND, &CMFCListViewDoc::OnMycommand)
END_MESSAGE_MAP()
```

Note that for map macros that add the '&' character, you should not add it again in your code.

_ATL_FREE_THREADED

Define if one or more of your objects use free or neutral threading.

```
_ATL_FREE_THREADED
```

Remarks

Specifies free threading. Free threading is equivalent to a multithread apartment model. See [Specifying the Project's Threading Model](#) for other threading options, and [Options, ATL Simple Object Wizard](#) for a description of the threading models available for an ATL object.

_ATL_MULTI_THREADED

A symbol that indicates the project will have objects that are marked as Both, Free or Neutral.

```
_ATL_MULTI_THREADED
```

Remarks

If this symbol is defined, ATL will pull in code that will correctly synchronize access to global data. New code should use the equivalent macro [_ATL_FREE_THREADED](#) instead.

_ATL_NO_AUTOMATIC_NAMESPACE

A symbol that prevents the default use of namespace as ATL.

```
_ATL_NO_AUTOMATIC_NAMESPACE
```

Remarks

If this symbol is not defined, including `atbase.h` will perform **using namespace ATL** by default, which may lead to naming conflicts. To prevent this, define this symbol.

_ATL_NO_COM_SUPPORT

A symbol that prevents COM-related code from being compiled with your project.

```
_ATL_NO_COM_SUPPORT
```

ATL_NO_VTABLE

A symbol that prevents the vtable pointer from being initialized in the class's constructor and destructor.

```
ATL_NO_VTABLE
```

Remarks

If the vtable pointer is prevented from being initialized in the class's constructor and destructor, the linker can eliminate the vtable and all of the functions to which it points. Expands to **__declspec(novtable)**.

Example

```
class ATL_NO_VTABLE CMyClass2 :
```

ATL_NOINLINE

A symbol that indicates a function should not be inlined.

```
ATL_NOINLINE inline
myfunction()
{
    ...
}
```

Parameters

myfunction

The function that should not be inlined.

Remarks

Use this symbol if you want to ensure a function does not get inlined by the compiler, even though it must be declared as inline so that it can be placed in a header file. Expands to **__declspec(noinline)**.

_ATL_SINGLE_THREADED

Define if all of your objects use the single threading model

`_ATL_SINGLE_THREADED`

Remarks

Specifies that the object always runs in the primary COM thread. See [Specifying the Project's Threading Model](#) for other threading options, and [Options, ATL Simple Object Wizard](#) for a description of the threading models available for an ATL object.

See also

[Macros](#)

Composite Control Macros

5/7/2019 • 3 minutes to read • [Edit Online](#)

These macros define event sink maps and entries.

BEGIN_SINK_MAP	Marks the beginning of the event sink map for the composite control.
END_SINK_MAP	Marks the end of the event sink map for the composite control.
SINK_ENTRY	Entry to the event sink map.
SINK_ENTRY_EX	Entry to the event sink map with an additional parameter.
SINK_ENTRY_EX_P	(Visual Studio 2017) Similar to SINK_ENTRY_EX except that it takes a pointer to iid.
SINK_ENTRY_INFO	Entry to the event sink map with manually supplied type information for use with IDispEventSimpleImpl .
SINK_ENTRY_INFO_P	(Visual Studio 2017) Similar to SINK_ENTRY_INFO except that it takes a pointer to iid.

Requirements

Header: atlcom.h

BEGIN_SINK_MAP

Declares the beginning of the event sink map for the composite control.

```
BEGIN_SINK_MAP(_class)
```

Parameters

_class

[in] Specifies the control.

Example

```
BEGIN_SINK_MAP(CMyCompositeCtrl)
//Make sure the Event Handlers have __stdcall calling convention
SINK_ENTRY(IDC_CALENDAR1, DISPID_CLICK, &CMyCompositeCtrl::ClickCalendar1)
SINK_ENTRY(IDC_CALENDAR2, DISPID_CLICK, &CMyCompositeCtrl::ClickCalendar2)
END_SINK_MAP()
```

Remarks

CE ATL implementation of ActiveX event sinks only supports return values of type HRESULT or void from your event handler methods; any other return value is unsupported and its behavior is undefined.

END_SINK_MAP

Declares the end of the event sink map for the composite control.

```
END_SINK_MAP()
```

Example

```
BEGIN_SINK_MAP(CMyCompositeCtrl)
    //Make sure the Event Handlers have __stdcall calling convention
    SINK_ENTRY(IDC_CALENDAR1, DISPID_CLICK, &CMyCompositeCtrl::ClickCalendar1)
    SINK_ENTRY(IDC_CALENDAR2, DISPID_CLICK, &CMyCompositeCtrl::ClickCalendar2)
END_SINK_MAP()
```

Remarks

CE ATL implementation of ActiveX event sinks only supports return values of type HRESULT or void from your event handler methods; any other return value is unsupported and its behavior is undefined.

SINK_ENTRY

Declares the handler function (*fn*) for the specified event (*dispid*), of the control identified by *id*.

```
SINK_ENTRY( id, dispid, fn )
```

Parameters

id

[in] Identifies the control.

dispid

[in] Identifies the specified event.

fn

[in] Name of the event handler function. This function must use the `_stdcall` calling convention and have the appropriate dispinterface-style signature.

Example

```
BEGIN_SINK_MAP(CMyCompositeCtrl)
    //Make sure the Event Handlers have __stdcall calling convention
    SINK_ENTRY(IDC_CALENDAR1, DISPID_CLICK, &CMyCompositeCtrl::ClickCalendar1)
    SINK_ENTRY(IDC_CALENDAR2, DISPID_CLICK, &CMyCompositeCtrl::ClickCalendar2)
END_SINK_MAP()
```

Remarks

CE ATL implementation of ActiveX event sinks only supports return values of type HRESULT or void from your event handler methods; any other return value is unsupported and its behavior is undefined.

SINK_ENTRY_EX and SINK_ENTRY_EX_P

Declares the handler function (*fn*) for the specified event (*dispid*), of the dispatch interface (*iid*), for the control identified by *id*.

```
SINK_ENTRY_EX( id, iid, dispid, fn )
SINK_ENTRY_EX_P( id, piid, dispid, fn ) // (Visual Studio 2017)
```

Parameters

id

[in] Identifies the control.

iid

[in] Identifies the dispatch interface.

piid

[in] Pointer to the dispatch interface.

dispid

[in] Identifies the specified event.

fn

[in] Name of the event handler function. This function must use the `_stdcall` calling convention and have the appropriate dispinterface-style signature.

Example

```
BEGIN_SINK_MAP(CMyCompositCtrl2)
    //Make sure the Event Handlers have __stdcall calling convention
    SINK_ENTRY_EX(IDC_CALENDAR1, __uuidof(DCalendarEvents), DISPID_CLICK,
        &CMyCompositCtrl2::ClickCalendar1)
    SINK_ENTRY_EX(IDC_CALENDAR2, __uuidof(DCalendarEvents), DISPID_CLICK,
        &CMyCompositCtrl2::ClickCalendar2)
END_SINK_MAP()
```

Remarks

CE ATL implementation of ActiveX event sinks only supports return values of type HRESULT or void from your event handler methods; any other return value is unsupported and its behavior is undefined.

SINK_ENTRY_INFO and SINK_ENTRY_INFO_P

Use the SINK_ENTRY_INFO macro within an event sink map to provide the information needed by [IDispEventSimpleImpl](#) to route events to the relevant handler function.

```
SINK_ENTRY_INFO( id, iid, dispid, fn, info )
SINK_ENTRY_INFO_P( id, piid, dispid, fn, info ) // (Visual Studio 2017)
```

Parameters

id

[in] Unsigned integer identifying the event source. This value must match the *nID* template parameter used in the related [IDispEventSimpleImpl](#) base class.

iid

[in] IID that identifies the dispatch interface.

piid

[in] Pointer to IID that identifies the dispatch interface.

dispid

[in] DISPID identifying the specified event.

fn

[in] Name of the event handler function. This function must use the `_stdcall` calling convention and have the appropriate dispinterface-style signature.

info

[in] Type information for the event handler function. This type information is provided in the form of a pointer to an `_ATL_FUNC_INFO` structure. `CC_CDECL` is the only option supported in Windows CE for the `CALLCONV` field of the `_ATL_FUNC_INFO` structure. Any other value is unsupported thus its behavior undefined.

Remarks

The first four macro parameters are the same as those for the [SINK_ENTRY_EX](#) macro. The final parameter provides type information for the event. CE ATL implementation of ActiveX event sinks only supports return values of type `HRESULT` or `void` from your event handler methods; any other return value is unsupported and its behavior is undefined.

See also

[Macros](#)

[Composite Control Global Functions](#)

Connection Point Macros

3/4/2019 • 2 minutes to read • [Edit Online](#)

These macros define connection point maps and entries.

BEGIN_CONNECTION_POINT_MAP	Marks the beginning of the connection point map entries.
CONNECTION_POINT_ENTRY	Enters connection points into the map.
CONNECTION_POINT_ENTRY_P	(Visual Studio 2017) Similar to CONNECTION_POINT_ENTRY but takes a pointer to iid.
END_CONNECTION_POINT_MAP	Marks the end of the connection point map entries.

Requirements

Header: atlcom.h

BEGIN_CONNECTION_POINT_MAP

Marks the beginning of the connection point map entries.

```
BEGIN_CONNECTION_POINT_MAP(x)
```

Parameters

x

[in] The name of the class containing the connection points.

Remarks

Start your connection point map with the [BEGIN_CONNECTION_POINT_MAP](#) macro, add entries for each of your connection points with the [CONNECTION_POINT_ENTRY](#) macro, and complete the map with the [END_CONNECTION_POINT_MAP](#) macro.

For more information about connection points in ATL, see the article [Connection Points](#).

Example

```
BEGIN_CONNECTION_POINT_MAP(CPolyCtl)
    CONNECTION_POINT_ENTRY(__uuidof(_IPolyCtlEvents))
END_CONNECTION_POINT_MAP()
```

CONNECTION_POINT_ENTRY and CONNECTION_POINT_ENTRY_P

Enters a connection point for the specified interface into the connection point map so that it can be accessed.

```
CONNECTION_POINT_ENTRY(iid)
CONNECTION_POINT_ENTRY_P(piid) // (Visual Studio 2017)
```

Parameters

iid

[in] The GUID of the interface being added to the connection point map.

piid

[in] Pointer to the GUID of the interface being added.

Remarks

Connection point entries in the map are used by [IConnectionPointContainerImpl](#). The class containing the connection point map must inherit from `IConnectionPointContainerImpl`.

Start your connection point map with the [BEGIN_CONNECTION_POINT_MAP](#) macro, add entries for each of your connection points with the `CONNECTION_POINT_ENTRY` macro, and complete the map with the [END_CONNECTION_POINT_MAP](#) macro.

For more information about connection points in ATL, see the article [Connection Points](#).

Example

```
class ATL_NO_VTABLE CConnect2 :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CConnect2, &CLSID_Connect2>,
    public IConnectionPointContainerImpl<CConnect2>,
    public IPropertyNotifySinkCP<CConnect2>
{
public:
    BEGIN_CONNECTION_POINT_MAP(CConnect2)
        CONNECTION_POINT_ENTRY(IID_IPropertyNotifySink)
    END_CONNECTION_POINT_MAP()

    // Remainder of class declaration omitted.
```

END_CONNECTION_POINT_MAP

Marks the end of the connection point map entries.

```
END_CONNECTION_POINT_MAP()
```

Remarks

Start your connection point map with the [BEGIN_CONNECTION_POINT_MAP](#) macro, add entries for each of your connection points with the [CONNECTION_POINT_ENTRY](#) macro, and complete the map with the `END_CONNECTION_POINT_MAP` macro.

For more information about connection points in ATL, see the article [Connection Points](#).

Example

```
BEGIN_CONNECTION_POINT_MAP(CMyComponent)
    CONNECTION_POINT_ENTRY(__uuidof(_IMyComponentEvents))
END_CONNECTION_POINT_MAP()
```

See also

[Macros](#)

[Connection Point Global Functions](#)

Debugging and Error Reporting Macros

5/7/2019 • 7 minutes to read • [Edit Online](#)

These macros provide useful debugging and trace facilities.

_ATL_DEBUG_INTERFACES	Writes, to the output window, any interface leaks that are detected when <code>_Module.Term</code> is called.
_ATL_DEBUG_QI	Writes all calls to <code>QueryInterface</code> to the output window.
ATLASSERT	Performs the same functionality as the _ASSERT macro found in the C run-time library.
ATLENSURE	Performs parameters validation. Call <code>AtlThrow</code> if needed
ATLTRACENOTIMPL	Sends a message to the dump device that the specified function is not implemented.
ATLTRACE	Reports warnings to an output device, such as the debugger window, according to the indicated flags and levels. Included for backward compatibility.
ATLTRACE2	Reports warnings to an output device, such as the debugger window, according to the indicated flags and levels.

[_ATL_DEBUG_INTERFACES](#)

Define this macro before including any ATL header files to trace all `AddRef` and `Release` calls on your components' interfaces to the output window.

```
#define _ATL_DEBUG_INTERFACES
```

Remarks

The trace output will appear as shown below:

```
ATL: QIThunk - 2008 AddRef : Object = 0x00d81ba0 Refcount = 1 CBug - IBUG
```

The first part of each trace will always be `ATL: QIThunk`. Next is a value identifying the particular *interface thunk* being used. An interface thunk is an object used to maintain a reference count and provide the tracing capability used here. A new interface thunk is created on every call to `QueryInterface` except for requests for the `IUnknown` interface (in this case, the same thunk is returned every time to comply with COM's identity rules).

Next you'll see `AddRef` or `Release` indicating which method was called. Following that, you'll see a value identifying the object whose interface reference count was changed. The value traced is the **this** pointer of the object.

The reference count that is traced is the reference count on that thunk after `AddRef` or `Release` was called. Note that this reference count may not match the reference count for the object. Each thunk maintains its own reference count to help you fully comply with COM's reference-counting rules.

The final piece of information traced is the name of the object and the interface being affected by the `AddRef` or `Release` call.

Any interface leaks that are detected when the server shuts down and `_Module.Term` is called will be logged like this:

```
ATL: QIThunk - 2005 LEAK : Object = 0x00d81ca0 Refcount = 1 MaxRefCount = 1 CBug - IBUG
```

The information provided here maps directly to the information provided in the previous trace statements, so you can examine the reference counts throughout the whole lifetime of an interface thunk. In addition, you get an indication of the maximum reference count on that interface thunk.

NOTE

`_ATL_DEBUG_INTERFACES` can be used in retail builds.

_ATL_DEBUG_QI

Writes all calls to `QueryInterface` to the output window.

```
#define _ATL_DEBUG_QI
```

Remarks

If a call to `QueryInterface` failed, the output window will display:

interface name - `failed`

ATLASSERT

The ATLASSERT macro performs the same functionality as the `_ASSERTTE` macro found in the C run-time library.

```
ATLASSERT(booleanExpression);
```

Parameters

booleanExpression

Expression (including pointers) that evaluates to nonzero or 0.

Remarks

In debug builds, ATLASSERT evaluates *booleanExpression* and generates a debug report when the result is false.

Requirements

Header: `atldef.h`

ATLENSURE

This macro is used to validate parameters passed to a function.

```
ATLENSURE(booleanExpression);  
ATLENSURE_THROW(booleanExpression, hr);
```

Parameters

booleanExpression

Specifies a boolean expression to be tested.

hr

Specifies an error code to return.

Remarks

These macros provide a mechanism to detect and notify the user of incorrect parameter usage.

The macro calls ATLASSERT and if the condition fails calls `AtlThrow`.

In the ATLENSURE case, `AtlThrow` is called with E_FAIL.

In the ATLENSURE_THROW case, `AtlThrow` is called with the specified HRESULT.

The difference between ATLENSURE and ATLASSERT is that ATLENSURE throws an exception in Release builds as well as in Debug builds.

Example

```
void MyImportantFunction(char* psz)
{
    ATLENSURE(NULL != psz);

    char mysz[64];
    strcpy_s(mysz, sizeof(mysz), psz);
}
```

Requirements

Header: afx.h

ATLTRACENOTIMPL

In debug builds of ATL, sends the string "*funcname* is not implemented" to the dump device and returns E_NOTIMPL.

```
ATLTRACENOTIMPL(funcname);
```

Parameters

funcname

[in] A string containing the name of the function that is not implemented.

Remarks

In release builds, simply returns E_NOTIMPL.

Example

```
ATLTRACENOTIMPL(_T("IOleControl::GetControlInfo"));
```

Requirements

Header: atltrace.h

ATLTRACE

Reports warnings to an output device, such as the debugger window, according to the indicated flags and levels. Included for backward compatibility.

```
ATLTRACE(exp);

ATLTRACE(
    DWORD category,
    UINT level,
    LPCSTR lpszFormat, ...);
```

Parameters

exp

[in] The string and variables to send to the output window or any application that traps these messages.

category

[in] Type of event or method on which to report. See the Remarks for a list of categories.

level

[in] The level of tracing to report. See the Remarks for details.

lpszFormat

[in] The formatted string to send to the dump device.

Remarks

See [ATLTRACE2](#) for a description of ATLTRACE. ATLTRACE and ATLTRACE2 have the same behavior, ATLTRACE is included for backward compatibility.

ATLTRACE2

Reports warnings to an output device, such as the debugger window, according to the indicated flags and levels.

```
ATLTRACE2(exp);

ATLTRACE2(
    DWORD category,
    UINT level,
    LPCSTR lpszFormat, ...);
```

Parameters

exp

[in] The string to send to the output window or any application that traps these messages.

category

[in] Type of event or method on which to report. See the Remarks for a list of categories.

level

[in] The level of tracing to report. See the Remarks for details.

lpszFormat

[in] The `printf`-style format string to use to create a string to send to the dump device.

Remarks

The short form of ATLTRACE2 writes a string to the debugger's output window. The second form of ATLTRACE2 also writes output to the debugger's output window, but is subject to the settings of the ATL/MFC Trace Tool (see [ATLTraceTool Sample](#)). For example, if you set *level* to 4 and the ATL/MFC Trace Tool to level 0, you will not see the message. *level* can be 0, 1, 2, 3, or 4. The default, 0, reports only the most serious problems.

The *category* parameter lists the trace flags to set. These flags correspond to the types of methods for which you want to report. The tables below list the valid trace flags you can use for the *category* parameter.

ATL Trace Flags

ATL CATEGORY	DESCRIPTION
<code>atlTraceGeneral</code>	Reports on all ATL applications. The default.
<code>atlTraceCOM</code>	Reports on COM methods.
<code>atlTraceQI</code>	Reports on QueryInterface calls.
<code>atlTraceRegistrar</code>	Reports on the registration of objects.
<code>atlTraceRefCount</code>	Reports on changing reference count.
<code>atlTraceWindowing</code>	Reports on windows methods; for example, reports an invalid message map ID.
<code>atlTraceControls</code>	Reports on controls; for example, reports when a control or its window is destroyed.
<code>atlTraceHosting</code>	Reports hosting messages; for example, reports when a client in a container is activated.
<code>atlTraceDBClient</code>	Reports on OLE DB Consumer Template; for example, when a call to GetData fails, the output can contain the HRESULT.
<code>atlTraceDBProvider</code>	Reports on OLE DB Provider Template; for example, reports if the creation of a column failed.
<code>atlTraceSnapin</code>	Reports for MMC SnapIn application.
<code>atlTraceNotImpl</code>	Reports that the indicated function is not implemented.
<code>atlTraceAllocation</code>	Reports messages printed by the memory debugging tools in atldbgmem.h.

MFC Trace Flags

MFC CATEGORY	DESCRIPTION
<code>traceAppMsg</code>	General purpose, MFC messages. Always recommended.
<code>traceDumpContext</code>	Messages from CDumpContext .
<code>traceWinMsg</code>	Messages from MFC's message handling code.
<code>traceMemory</code>	Messages from MFC's memory management code.
<code>traceCmdRouting</code>	Messages from MFC's Windows command routing code.

MFC CATEGORY	DESCRIPTION
<code>traceHtml</code>	Messages from MFC's DHTML dialog support.
<code>traceSocket</code>	Messages from MFC's socket support.
<code>traceOle</code>	Messages from MFC's OLE support.
<code>traceDatabase</code>	Messages from MFC's database support.
<code>traceInternet</code>	Messages from MFC's Internet support.

To declare a custom trace category, declare a global instance of the `CTraceCategory` class as follows:

```
CTraceCategory MY_CATEGORY(_T("MyCategoryName"), 1);
```

The category name, MY_CATEGORY in this example, is the name you specify to the *category* parameter. The first parameter is the category name that will appear in the ATL/MFC Trace Tool. The second parameter is the default trace level. This parameter is optional, and the default trace level is 0.

To use a user-defined category:

```
ATLTRACE2(MY_CATEGORY, 2, _T("a message in a custom category"));
```

To specify that you want to filter the trace messages, insert definitions for these macros into Stdafx.h before the `#include <atlbase.h>` statement.

Alternatively, you can set the filter in the preprocessor directives in the **Property Pages** dialog box. Click the **Preprocessor** tab and then insert the global into the **Preprocessor Definitions** edit box.

Atlbase.h contains default definitions of the ATLTRACE2 macros and these definitions will be used if you don't define these symbols before atlbase.h is processed.

In release builds, ATLTRACE2 compiles to `(void) 0`.

ATLTRACE2 limits the contents of the string to be sent to the dump device to no more than 1023 characters, after formatting.

ATLTRACE and ATLTRACE2 have the same behavior, ATLTRACE is included for backward compatibility.

Example

```
int i = 1;
ATLTRACE2(atlTraceGeneral, 4, "Integer = %d\n", i);
// Output: 'Integer = 1'
```

See also

[Macros](#)

[Debugging and Error Reporting Global Functions](#)

Exception Handling Macros

3/4/2019 • 2 minutes to read • [Edit Online](#)

These macros provide support for exception handling.

_ATLCATCH	Statement(s) to handle errors occurring in the associated <code>_ATLTRY</code> .
_ATLCATCHALL	Statement(s) to handle errors occurring in the associated <code>_ATLTRY</code> .
_ATLTRY	Marks a guarded code section where an error could possibly occur.

Requirements:

Header: `atldef.h`

`_ATLCATCH`

Statement(s) to handle errors occurring in the associated `_ATLTRY`.

```
_ATLCATCH(e)
```

Parameters

e

The exception to catch.

Remarks

Used in conjunction with `_ATLTRY`. Resolves to C++ `catch(CAtlException e)` for handling a given type of C++ exceptions.

`_ATLCATCHALL`

Statement(s) to handle errors occurring in the associated `_ATLTRY`.

```
_ATLCATCHALL
```

Remarks

Used in conjunction with `_ATLTRY`. Resolves to C++ `catch(...)` for handling all types of C++ exceptions.

`_ATLTRY`

Marks a guarded code section where an error could possibly occur.

```
_ATLTRY
```

Remarks

Used in conjunction with [_ATLCATCH](#) or [_ATLCATCHALL](#). Resolves to the C++ symbol [try](#).

See also

[Macros](#)

Message Map Macros (ATL)

3/4/2019 • 21 minutes to read • [Edit Online](#)

These macros define message maps and entries.

ALT_MSG_MAP	Marks the beginning of an alternate message map.
BEGIN_MSG_MAP	Marks the beginning of the default message map.
CHAIN_MSG_MAP_ALT	Chains to an alternate message map in the base class.
CHAIN_MSG_MAP_ALT_MEMBER	Chains to an alternate message map in a data member of the class.
CHAIN_MSG_MAP	Chains to the default message map in the base class.
CHAIN_MSG_MAP_DYNAMIC	Chains to the message map in another class at run time.
CHAIN_MSG_MAP_MEMBER	Chains to the default message map in a data member of the class.
COMMAND_CODE_HANDLER	Maps a WM_COMMAND message to a handler function, based on the notification code.
COMMAND_HANDLER	Maps a WM_COMMAND message to a handler function, based on the notification code and the identifier of the menu item, control, or accelerator.
COMMAND_ID_HANDLER	Maps a WM_COMMAND message to a handler function, based on the identifier of the menu item, control, or accelerator.
COMMAND_RANGE_CODE_HANDLER	Maps a WM_COMMAND message to a handler function, based on the notification code and a contiguous range of control identifiers.
COMMAND_RANGE_HANDLER	Maps a WM_COMMAND message to a handler function, based on a contiguous range of control identifiers.
DECLARE_EMPTY_MSG_MAP	Implements an empty message map.
DEFAULT_REFLECTION_HANDLER	Provides a default handler for reflected messages that are not handled otherwise.
END_MSG_MAP	Marks the end of a message map.
FORWARD_NOTIFICATIONS	Forwards notification messages to the parent window.
MESSAGE_HANDLER	Maps a Windows message to a handler function.

MESSAGE_RANGE_HANDLER	Maps a contiguous range of Windows messages to a handler function.
NOTIFY_CODE_HANDLER	Maps a WM_NOTIFY message to a handler function, based on the notification code.
NOTIFY_HANDLER	Maps a WM_NOTIFY message to a handler function, based on the notification code and the control identifier.
NOTIFY_ID_HANDLER	Maps a WM_NOTIFY message to a handler function, based on the control identifier.
NOTIFY_RANGE_CODE_HANDLER	Maps a WM_NOTIFY message to a handler function, based on the notification code and a contiguous range of control identifiers.
NOTIFY_RANGE_HANDLER	Maps a WM_NOTIFY message to a handler function, based on a contiguous range of control identifiers.
REFLECT_NOTIFICATIONS	Reflects notification messages back to the window that sent them.
REFLECTED_COMMAND_CODE_HANDLER	Maps a reflected WM_COMMAND message to a handler function, based on the notification code.
REFLECTED_COMMAND_HANDLER	Maps a reflected WM_COMMAND message to a handler function, based on the notification code and the identifier of the menu item, control, or accelerator.
REFLECTED_COMMAND_ID_HANDLER	Maps a reflected WM_COMMAND message to a handler function, based on the identifier of the menu item, control, or accelerator.
REFLECTED_COMMAND_RANGE_CODE_HANDLER	Maps a reflected WM_COMMAND message to a handler function, based on the notification code and a contiguous range of control identifiers.
REFLECTED_COMMAND_RANGE_HANDLER	Maps a reflected WM_COMMAND message to a handler function, based on a contiguous range of control identifiers.
REFLECTED_NOTIFY_CODE_HANDLER	Maps a reflected WM_NOTIFY message to a handler function, based on the notification code.
REFLECTED_NOTIFY_HANDLER	Maps a reflected WM_NOTIFY message to a handler function, based on the notification code and the control identifier.
REFLECTED_NOTIFY_ID_HANDLER	Maps a reflected WM_NOTIFY message to a handler function, based on the control identifier.
REFLECTED_NOTIFY_RANGE_CODE_HANDLER	Maps a reflected WM_NOTIFY message to a handler function, based on the notification code and a contiguous range of control identifiers.
REFLECTED_NOTIFY_RANGE_HANDLER	Maps a reflected WM_NOTIFY message to a handler function, based on a contiguous range of control identifiers.

Requirements

Header: atlwin.h

ALT_MSG_MAP

Marks the beginning of an alternate message map.

```
ALT_MSG_MAP(msgMapID)
```

Parameters

msgMapID

[in] The message map identifier.

Remarks

ATL identifies each message map by a number. The default message map (declared with the `BEGIN_MSG_MAP` macro) is identified by 0. An alternate message map is identified by *msgMapID*.

Message maps are used to process messages sent to a window. For example, [CContainedWindow](#) allows you to specify the identifier of a message map in the containing object. [CContainedWindow::WindowProc](#) then uses this message map to direct the contained window's messages either to the appropriate handler function or to another message map. For a list of macros that declare handler functions, see [BEGIN_MSG_MAP](#).

Always begin a message map with `BEGIN_MSG_MAP`. You can then declare subsequent alternate message maps.

The [END_MSG_MAP](#) macro marks the end of the message map. Note that there is always exactly one instance of `BEGIN_MSG_MAP` and `END_MSG_MAP`.

For more information about using message maps in ATL, see [Message Maps](#).

Example

The following example shows the default message map and one alternate message map, each containing one handler function:

```
BEGIN_MSG_MAP(CMyOneAltClass)
    MESSAGE_HANDLER(WM_PAINT, OnPaint)
ALT_MSG_MAP(1)
    MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
END_MSG_MAP()
```

The next example shows two alternate message maps. The default message map is empty.

```
BEGIN_MSG_MAP(CMyClass)
ALT_MSG_MAP(1)
    MESSAGE_HANDLER(WM_PAINT, OnPaint)
    MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
ALT_MSG_MAP(2)
    MESSAGE_HANDLER(WM_CREATE, OnCreate)
END_MSG_MAP()
```

Requirements

Header: atlwin.h

BEGIN_MSG_MAP

Marks the beginning of the default message map.

```
BEGIN_MSG_MAP(theClass)
```

Parameters

theClass

[in] The name of the class containing the message map.

Remarks

[CWindowImpl::WindowProc](#) uses the default message map to process messages sent to the window. The message map directs messages either to the appropriate handler function or to another message map.

The following macros map a message to a handler function. This function must be defined in *theClass*.

MACRO	DESCRIPTION
MESSAGE_HANDLER	Maps a Windows message to a handler function.
MESSAGE_RANGE_HANDLER	Maps a contiguous range of Windows messages to a handler function.
COMMAND_HANDLER	Maps a WM_COMMAND message to a handler function, based on the notification code and the identifier of the menu item, control, or accelerator.
COMMAND_ID_HANDLER	Maps a WM_COMMAND message to a handler function, based on the identifier of the menu item, control, or accelerator.
COMMAND_CODE_HANDLER	Maps a WM_COMMAND message to a handler function, based on the notification code.
COMMAND_RANGE_HANDLER	Maps a contiguous range of WM_COMMAND messages to a handler function, based on the identifier of the menu item, control, or accelerator.
NOTIFY_HANDLER	Maps a WM_NOTIFY message to a handler function, based on the notification code and the control identifier.
NOTIFY_ID_HANDLER	Maps a WM_NOTIFY message to a handler function, based on the control identifier.
NOTIFY_CODE_HANDLER	Maps a WM_NOTIFY message to a handler function, based on the notification code.
NOTIFY_RANGE_HANDLER	Maps a contiguous range of WM_NOTIFY messages to a handler function, based on the control identifier.

The following macros direct messages to another message map. This process is called "chaining."

MACRO	DESCRIPTION
CHAIN_MSG_MAP	Chains to the default message map in the base class.

MACRO	DESCRIPTION
CHAIN_MSG_MAP_MEMBER	Chains to the default message map in a data member of the class.
CHAIN_MSG_MAP_ALT	Chains to an alternate message map in the base class.
CHAIN_MSG_MAP_ALT_MEMBER	Chains to an alternate message map in a data member of the class.
CHAIN_MSG_MAP_DYNAMIC	Chains to the default message map in another class at run time.

The following macros direct "reflected" messages from the parent window. For example, a control normally sends notification messages to its parent window for processing, but the parent window can reflect the message back to the control.

MACRO	DESCRIPTION
REFLECTED_COMMAND_HANDLER	Maps a reflected WM_COMMAND message to a handler function, based on the notification code and the identifier of the menu item, control, or accelerator.
REFLECTED_COMMAND_ID_HANDLER	Maps a reflected WM_COMMAND message to a handler function, based on the identifier of the menu item, control, or accelerator.
REFLECTED_COMMAND_CODE_HANDLER	Maps a reflected WM_COMMAND message to a handler function, based on the notification code.
REFLECTED_COMMAND_RANGE_HANDLER	Maps a reflected WM_COMMAND message to a handler function, based on a contiguous range of control identifiers.
REFLECTED_COMMAND_RANGE_CODE_HANDLER	Maps a reflected WM_COMMAND message to a handler function, based on the notification code and a contiguous range of control identifiers.
REFLECTED_NOTIFY_HANDLER	Maps a reflected WM_NOTIFY message to a handler function, based on the notification code and the control identifier.
REFLECTED_NOTIFY_ID_HANDLER	Maps a reflected WM_NOTIFY message to a handler function, based on the control identifier.
REFLECTED_NOTIFY_CODE_HANDLER	Maps a reflected WM_NOTIFY message to a handler function, based on the notification code.
REFLECTED_NOTIFY_RANGE_HANDLER	Maps a reflected WM_NOTIFY message to a handler function, based on a contiguous range of control identifiers.
REFLECTED_NOTIFY_RANGE_CODE_HANDLER	Maps a reflected WM_NOTIFY message to a handler function, based on the notification code and a contiguous range of control identifiers.

Example

```

class CMyExtWindow : public CMyBaseWindow
{
public:
    BEGIN_MSG_MAP(CMyExtWindow)
        MESSAGE_HANDLER(WM_PAINT, OnPaint)
        MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
        CHAIN_MSG_MAP(CMyBaseWindow)
    END_MSG_MAP()

    LRESULT OnPaint(UINT /*nMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
        BOOL& /*bHandled*/)
    {
        return 0;
    }

    LRESULT OnSetFocus(UINT /*nMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
        BOOL& /*bHandled*/)
    {
        return 0;
    }
};

```

When a `CMyExtWindow` object receives a `WM_PAINT` message, the message is directed to `CMyExtWindow::OnPaint` for the actual processing. If `OnPaint` indicates the message requires further processing, the message will then be directed to the default message map in `CMyBaseWindow`.

In addition to the default message map, you can define an alternate message map with [ALT_MSG_MAP](#). Always begin a message map with `BEGIN_MSG_MAP`. You can then declare subsequent alternate message maps. The following example shows the default message map and one alternate message map, each containing one handler function:

```

BEGIN_MSG_MAP(CMyOneAltClass)
    MESSAGE_HANDLER(WM_PAINT, OnPaint)
ALT_MSG_MAP(1)
    MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
END_MSG_MAP()

```

The next example shows two alternate message maps. The default message map is empty.

```

BEGIN_MSG_MAP(CMyClass)
ALT_MSG_MAP(1)
    MESSAGE_HANDLER(WM_PAINT, OnPaint)
    MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
ALT_MSG_MAP(2)
    MESSAGE_HANDLER(WM_CREATE, OnCreate)
END_MSG_MAP()

```

The [END_MSG_MAP](#) macro marks the end of the message map. Note that there is always exactly one instance of `BEGIN_MSG_MAP` and `END_MSG_MAP`.

For more information about using message maps in ATL, see [Message Maps](#).

Requirements

Header: `atlwin.h`

CHAIN_MSG_MAP_ALT

Defines an entry in a message map.


```
CHAIN_MSG_MAP_ALT(theChainClass, msgMapID)
```

Parameters

theChainClass

[in] The name of the base class containing the message map.

msgMapID

[in] The message map identifier.

Remarks

CHAIN_MSG_MAP_ALT directs messages to an alternate message map in a base class. You must have declared this alternate message map with [ALT_MSG_MAP\(msgMapID\)](#). To direct messages to a base class's default message map (declared with [BEGIN_MSG_MAP](#)), use CHAIN_MSG_MAP. For an example, see [CHAIN_MSG_MAP](#).

NOTE

Always begin a message map with BEGIN_MSG_MAP. You can then declare subsequent alternate message maps with ALT_MSG_MAP. The [END_MSG_MAP](#) macro marks the end of the message map. Every message map must have exactly one instance of BEGIN_MSG_MAP and END_MSG_MAP.

For more information about using message maps in ATL, see [Message Maps](#).

Requirements

Header: atlwin.h

CHAIN_MSG_MAP_ALT_MEMBER

Defines an entry in a message map.

```
CHAIN_MSG_MAP_ALT_MEMBER(theChainMember, msgMapID)
```

Parameters

theChainMember

[in] The name of the data member containing the message map.

msgMapID

[in] The message map identifier.

Remarks

CHAIN_MSG_MAP_ALT_MEMBER directs messages to an alternate message map in a data member. You must have declared this alternate message map with [ALT_MSG_MAP\(msgMapID\)](#). To direct messages to a data member's default message map (declared with [BEGIN_MSG_MAP](#)), use CHAIN_MSG_MAP_MEMBER. For an example, see [CHAIN_MSG_MAP_MEMBER](#).

NOTE

Always begin a message map with BEGIN_MSG_MAP. You can then declare subsequent alternate message maps with ALT_MSG_MAP. The [END_MSG_MAP](#) macro marks the end of the message map. Every message map must have exactly one instance of BEGIN_MSG_MAP and END_MSG_MAP.

For more information about using message maps in ATL, see [Message Maps](#).

Requirements

Header: atlwin.h

CHAIN_MSG_MAP

Defines an entry in a message map.

```
CHAIN_MSG_MAP(theChainClass)
```

Parameters

theChainClass

[in] The name of the base class containing the message map.

Remarks

CHAIN_MSG_MAP directs messages to a base class's default message map (declared with [BEGIN_MSG_MAP](#)). To direct messages to a base class's alternate message map (declared with [ALT_MSG_MAP](#)), use [CHAIN_MSG_MAP_ALT](#).

NOTE

Always begin a message map with BEGIN_MSG_MAP. You can then declare subsequent alternate message maps with ALT_MSG_MAP. The [END_MSG_MAP](#) macro marks the end of the message map. Every message map must have exactly one instance of BEGIN_MSG_MAP and END_MSG_MAP.

For more information about using message maps in ATL, see [Message Maps](#).

Example

```
class CMyExtClass : public CMyBaseClass
{
public:
    BEGIN_MSG_MAP(CMyExtClass)
        MESSAGE_HANDLER(WM_PAINT, OnPaint)
        // chain to default message map in CMyBaseClass
        CHAIN_MSG_MAP(CMyBaseClass)
    ALT_MSG_MAP(1)
        // chain to first alternative message map in CMyBaseClass
        CHAIN_MSG_MAP(CMyBaseClass)
    ALT_MSG_MAP(2)
        MESSAGE_HANDLER(WM_CHAR, OnChar)
        // chain to alternate message map in CMyBaseClass
        CHAIN_MSG_MAP_ALT(CMyBaseClass, 1)
    END_MSG_MAP()

    LRESULT OnPaint(UINT /*nMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
        BOOL& /*bHandled*/)
    {
        return 0;
    }

    LRESULT OnChar(UINT /*nMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
        BOOL& /*bHandled*/)
    {
        return 0;
    }
};
```

This example illustrates the following:

- If a window procedure is using `CMyClass`'s default message map and `OnPaint` does not handle a message, the message is directed to `CMyBaseClass`'s default message map for processing.
- If a window procedure is using the first alternate message map in `CMyClass`, all messages are directed to `CMyBaseClass`'s default message map.
- If a window procedure is using `CMyClass`'s second alternate message map and `OnChar` does not handle a message, the message is directed to the specified alternate message map in `CMyBaseClass`. `CMyBaseClass` must have declared this message map with `ALT_MSG_MAP(1)`.

Requirements

Header: `atlwin.h`

CHAIN_MSG_MAP_DYNAMIC

Defines an entry in a message map.

```
CHAIN_MSG_MAP_DYNAMIC(dynaChainID)
```

Parameters

dynaChainID

[in] The unique identifier for an object's message map.

Remarks

`CHAIN_MSG_MAP_DYNAMIC` directs messages, at run time, to the default message map in another object. The object and its message map are associated with *dynaChainID*, which you define through [CDynamicChain::SetChainEntry](#). You must derive your class from `CDynamicChain` in order to use `CHAIN_MSG_MAP_DYNAMIC`. For an example, see the [CDynamicChain](#) overview.

NOTE

Always begin a message map with [BEGIN_MSG_MAP](#). You can then declare subsequent alternate message maps with `ALT_MSG_MAP`. The [END_MSG_MAP](#) macro marks the end of the message map. Every message map must have exactly one instance of `BEGIN_MSG_MAP` and `END_MSG_MAP`.

For more information about using message maps in ATL, see [Message Maps](#).

Requirements

Header: `atlwin.h`

CHAIN_MSG_MAP_MEMBER

Defines an entry in a message map.

```
CHAIN_MSG_MAP_MEMBER(theChainMember)
```

Parameters

theChainMember

[in] The name of the data member containing the message map.

Remarks

`CHAIN_MSG_MAP_MEMBER` directs messages to a data member's default message map (declared with

[BEGIN_MSG_MAP](#)). To direct messages to a data member's alternate message map (declared with [ALT_MSG_MAP](#)), use [CHAIN_MSG_MAP_ALT_MEMBER](#).

NOTE

Always begin a message map with [BEGIN_MSG_MAP](#). You can then declare subsequent alternate message maps with [ALT_MSG_MAP](#). The [END_MSG_MAP](#) macro marks the end of the message map. Every message map must have exactly one instance of [BEGIN_MSG_MAP](#) and [END_MSG_MAP](#).

For more information about using message maps in ATL, see [Message Maps](#).

Example

```
class CMyContainerClass : public CWindowImpl<CMyContainerClass>
{
public:
    CMyContainedClass m_obj;

    BEGIN_MSG_MAP(CMyContainerClass)
        MESSAGE_HANDLER(WM_PAINT, OnPaint)
        // chain to default message map of m_obj
        CHAIN_MSG_MAP_MEMBER(m_obj)
    ALT_MSG_MAP(1)
        // chain to default message map of m_obj
        CHAIN_MSG_MAP_MEMBER(m_obj)
    ALT_MSG_MAP(2)
        MESSAGE_HANDLER(WM_CHAR, OnChar)
        // chain to alternate message map of m_obj
        CHAIN_MSG_MAP_ALT_MEMBER(m_obj, 1)
    END_MSG_MAP()

    LRESULT OnPaint(UINT /*nMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
        BOOL& /*bHandled*/)
    {
        return 0;
    }
    LRESULT OnChar(UINT /*nMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
        BOOL& /*bHandled*/)
    {
        return 0;
    }
};
```

This example illustrates the following:

- If a window procedure is using `CMyClass`'s default message map and `OnPaint` does not handle a message, the message is directed to `m_obj`'s default message map for processing.
- If a window procedure is using the first alternate message map in `CMyClass`, all messages are directed to `m_obj`'s default message map.
- If a window procedure is using `CMyClass`'s second alternate message map and `OnChar` does not handle a message, the message is directed to the specified alternate message map of `m_obj`. Class `CMyContainedClass` must have declared this message map with `ALT_MSG_MAP(1)`.

Requirements

Header: atlwin.h

COMMAND_CODE_HANDLER

Similar to [COMMAND_HANDLER](#), but maps a [WM_COMMAND](#) message based only on the notification code.

```
COMMAND_CODE_HANDLER(code, func)
```

Parameters

code

[in] The notification code.

func

[in] The name of the message-handler function.

Requirements

Header: atlwin.h

COMMAND_HANDLER

Defines an entry in a message map.

```
COMMAND_HANDLER(id, code, func)
```

Parameters

id

[in] The identifier of the menu item, control, or accelerator.

code

[in] The notification code.

func

[in] The name of the message-handler function.

Remarks

COMMAND_HANDLER maps a [WM_COMMAND](#) message to the specified handler function, based on the notification code and the control identifier. For example:

```
class ATL_NO_VTABLE CPolyProp :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CPolyProp, &CLSID_PolyProp>,
    public IPropertyPageImpl<CPolyProp>,
    public CDialogImpl<CPolyProp>
{
public:
    BEGIN_COM_MAP(CPolyProp)
        COM_INTERFACE_ENTRY(IPropertyPage)
    END_COM_MAP()

    BEGIN_MSG_MAP(CPolyProp)
        COMMAND_HANDLER(IDC_SIDES, EN_CHANGE, OnEnChangeSides)
        CHAIN_MSG_MAP(IPropertyPageImpl<CPolyProp>)
    END_MSG_MAP()

    // When a CPolyProp object receives a WM_COMMAND message identified
    // by IDC_SIDES and EN_CHANGE, the message is directed to
    // CPolyProp::OnEnChangeSides for the actual processing.
    LRESULT OnEnChangeSides(WORD /*wNotifyCode*/, WORD /*wID*/, HWND /*hWndCtl*/,
        BOOL& /*bHandled*/);
```

Any function specified in a COMMAND_HANDLER macro must be defined as follows:

```
LRESULT CommandHandler(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);
```

The message map sets `bHandled` to TRUE before `CommandHandler` is called. If `CommandHandler` does not fully handle the message, it should set `bHandled` to FALSE to indicate the message needs further processing.

NOTE

Always begin a message map with `BEGIN_MSG_MAP`. You can then declare subsequent alternate message maps with `ALT_MSG_MAP`. The `END_MSG_MAP` macro marks the end of the message map. Every message map must have exactly one instance of `BEGIN_MSG_MAP` and `END_MSG_MAP`.

In addition to `COMMAND_HANDLER`, you can use `MESSAGE_HANDLER` to map a `WM_COMMAND` message without regard to an identifier or code. In this case, `MESSAGE_HANDLER(WM_COMMAND, OnHandlerFunction)` will direct all `WM_COMMAND` messages to `OnHandlerFunction`.

For more information about using message maps in ATL, see [Message Maps](#).

Requirements

Header: `atlwin.h`

COMMAND_ID_HANDLER

Similar to `COMMAND_HANDLER`, but maps a `WM_COMMAND` message based only on the identifier of the menu item, control, or accelerator.

```
COMMAND_ID_HANDLER(id, func)
```

Parameters

id

[in] The identifier of the menu item, control, or accelerator sending the message.

func

[in] The name of the message-handler function.

Requirements

Header: `atlwin.h`

COMMAND_RANGE_CODE_HANDLER

Similar to `COMMAND_RANGE_HANDLER`, but maps `WM_COMMAND` messages with a specific notification code from a range of controls to a single handler function.

```
COMMAND_RANGE_CODE_HANDLER(idFirst, idLast, code, func)
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

code

[in] The notification code.

func

[in] The name of the message-handler function.

Remarks

This range is based on the identifier of the menu item, control, or accelerator sending the message.

Requirements

Header: atlwin.h

COMMAND_RANGE_HANDLER

Similar to [COMMAND_HANDLER](#), but maps [WM_COMMAND](#) messages from a range of controls to a single handler function.

```
COMMAND_RANGE_HANDLER( idFirst, idLast, func)
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

func

[in] The name of the message-handler function.

Remarks

This range is based on the identifier of the menu item, control, or accelerator sending the message.

Requirements

Header: atlwin.h

DECLARE_EMPTY_MSG_MAP

Declares an empty message map.

```
DECLARE_EMPTY_MSG_MAP()
```

Remarks

DECLARE_EMPTY_MSG_MAP is a convenience macro that calls the macros [BEGIN_MSG_MAP](#) and [END_MSG_MAP](#) to create an empty message map:

```
BEGIN_MSG_MAP(CExample)
END_MSG_MAP()
```

DEFAULT_REFLECTION_HANDLER

Provides a default handler for the child window (control) that will receive reflected messages; the handler will properly pass unhandled messages to `DefWindowProc`.

```
DEFAULT_REFLECTION_HANDLER()
```

Requirements

Header: atlwin.h

END_MSG_MAP

Marks the end of a message map.

```
END_MSG_MAP()
```

Remarks

Always use the [BEGIN_MSG_MAP](#) macro to mark the beginning of a message map. Use [ALT_MSG_MAP](#) to declare subsequent alternate message maps.

Note that there is always exactly one instance of [BEGIN_MSG_MAP](#) and [END_MSG_MAP](#).

For more information about using message maps in ATL, see [Message Maps](#).

Example

The following example shows the default message map and one alternate message map, each containing one handler function:

```
BEGIN_MSG_MAP(CMyOneAltClass)
    MESSAGE_HANDLER(WM_PAINT, OnPaint)
ALT_MSG_MAP(1)
    MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
END_MSG_MAP()
```

The next example shows two alternate message maps. The default message map is empty.

```
BEGIN_MSG_MAP(CMyClass)
ALT_MSG_MAP(1)
    MESSAGE_HANDLER(WM_PAINT, OnPaint)
    MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
ALT_MSG_MAP(2)
    MESSAGE_HANDLER(WM_CREATE, OnCreate)
END_MSG_MAP()
```

Requirements

Header: atlwin.h

FORWARD_NOTIFICATIONS

Forwards notification messages to the parent window.

```
FORWARD_NOTIFICATIONS()
```

Remarks

Specify this macro as part of your message map.

Requirements

Header: atlwin.h

MESSAGE_HANDLER

Defines an entry in a message map.

```
MESSAGE_HANDLER( msg, func )
```

Parameters

msg

[in] The Windows message.

func

[in] The name of the message-handler function.

Remarks

MESSAGE_HANDLER maps a Windows message to the specified handler function.

Any function specified in a MESSAGE_HANDLER macro must be defined as follows:

```
LRESULT MessageHandler(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);
```

The message map sets `bHandled` to TRUE before `MessageHandler` is called. If `MessageHandler` does not fully handle the message, it should set `bHandled` to FALSE to indicate the message needs further processing.

NOTE

Always begin a message map with [BEGIN_MSG_MAP](#). You can then declare subsequent alternate message maps with [ALT_MSG_MAP](#). The [END_MSG_MAP](#) macro marks the end of the message map. Every message map must have exactly one instance of BEGIN_MSG_MAP and END_MSG_MAP.

In addition to MESSAGE_HANDLER, you can use [COMMAND_HANDLER](#) and [NOTIFY_HANDLER](#) to map [WM_COMMAND](#) and [WM_NOTIFY](#) messages, respectively.

For more information about using message maps in ATL, see [Message Maps](#).

Example

```
class CMyBaseWindow : public CWindowImpl<CMyBaseWindow>
{
public:
    BEGIN_MSG_MAP(CMyBaseWindow)
        MESSAGE_HANDLER(WM_CREATE, OnCreate)
    END_MSG_MAP()

    // When a CMyBaseWindow object receives a WM_CREATE message, the message
    // is directed to CMyBaseWindow::OnCreate for the actual processing.
    LRESULT OnCreate(UINT /*nMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
        BOOL& /*bHandled*/)
    {
        return 0;
    }
};
```

Requirements

Header: atlwin.h

MESSAGE_RANGE_HANDLER

Similar to [MESSAGE_HANDLER](#), but maps a range of Windows messages to a single handler function.

```
MESSAGE_RANGE_HANDLER( msgFirst, msgLast, func )
```

Parameters

msgFirst

[in] Marks the beginning of a contiguous range of messages.

msgLast

[in] Marks the end of a contiguous range of messages.

func

[in] The name of the message-handler function.

Requirements

Header: atlwin.h

NOTIFY_CODE_HANDLER

Similar to [NOTIFY_HANDLER](#), but maps a [WM_NOTIFY](#) message based only on the notification code.

```
NOTIFY_CODE_HANDLER(cd, func)
```

Parameters

cd

[in] The notification code.

func

[in] The name of the message-handler function.

Requirements

Header: atlwin.h

NOTIFY_HANDLER

Defines an entry in a message map.

```
NOTIFY_HANDLER( id, cd, func )
```

Parameters

id

[in] The identifier of the control sending the message.

cd

[in] The notification code.

func

[in] The name of the message-handler function.

Remarks

NOTIFY_HANDLER maps a [WM_NOTIFY](#) message to the specified handler function, based on the notification code and the control identifier.

Any function specified in a NOTIFY_HANDLER macro must be defined as follows:

```
LRESULT NotifyHandler(int idCtrl, LPNMHDR pnmh, BOOL& bHandled);
```

The message map sets `bHandled` to TRUE before `NotifyHandler` is called. If `NotifyHandler` does not fully handle the message, it should set `bHandled` to FALSE to indicate the message needs further processing.

NOTE

Always begin a message map with `BEGIN_MSG_MAP`. You can then declare subsequent alternate message maps with `ALT_MSG_MAP`. The `END_MSG_MAP` macro marks the end of the message map. Every message map must have exactly one instance of `BEGIN_MSG_MAP` and `END_MSG_MAP`.

In addition to `NOTIFY_HANDLER`, you can use `MESSAGE_HANDLER` to map a `WM_NOTIFY` message without regard to an identifier or code. In this case, `MESSAGE_HANDLER(WM_NOTIFY, OnHandlerFunction)` will direct all `WM_NOTIFY` messages to `OnHandlerFunction`.

For more information about using message maps in ATL, see [Message Maps](#).

Example

```
class CMyDialog2 : public CDialogImpl<CMyDialog2>
{
public:
    enum { IDD = IDD_MYDLG };

    BEGIN_MSG_MAP(CMyDialog2)
        NOTIFY_HANDLER(IDC_TREE1, NM_CLICK, OnNMClickTree1)
    END_MSG_MAP()

public:
    // When a CMyDialog2 object receives a WM_NOTIFY message
    // identified by IDC_TREE1 and NM_CLICK, the message is
    // directed to CMyDialog2::OnNMClickTree1 for the actual
    // processing.
    LRESULT OnNMClickTree1(int /*idCtrl*/, LPNMHDR pNMHDR, BOOL& /*bHandled*/);
};
```

Requirements

Header: `atlwin.h`

NOTIFY_ID_HANDLER

Similar to [NOTIFY_HANDLER](#), but maps a `WM_NOTIFY` message based only on the control identifier.

```
NOTIFY_ID_HANDLER( id, func )
```

Parameters

id

[in] The identifier of the control sending the message.

func

[in] The name of the message-handler function.

Requirements

Header: `atlwin.h`

NOTIFY_RANGE_CODE_HANDLER

Similar to [NOTIFY_RANGE_HANDLER](#), but maps `WM_NOTIFY` messages with a specific notification code from a

range of controls to a single handler function.

```
NOTIFY_RANGE_CODE_HANDLER( idFirst, idLast, cd, func )
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

cd

[in] The notification code.

func

[in] The name of the message-handler function.

Remarks

This range is based on the identifier of the control sending the message.

Requirements

Header: atlwin.h

NOTIFY_RANGE_HANDLER

Similar to [NOTIFY_HANDLER](#), but maps [WM_NOTIFY](#) messages from a range of controls to a single handler function.

```
NOTIFY_RANGE_HANDLER( idFirst, idLast, func )
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

func

[in] The name of the message-handler function.

Remarks

This range is based on the identifier of the control sending the message.

Requirements

Header: atlwin.h

REFLECT_NOTIFICATIONS

Reflects notification messages back to the child window (control) that sent them.

```
REFLECT_NOTIFICATIONS()
```

Remarks

Specify this macro as part of the parent window's message map.

Requirements

Header: atlwin.h

REFLECTED_COMMAND_CODE_HANDLER

Similar to [COMMAND_CODE_HANDLER](#), but maps commands reflected from the parent window.

```
REFLECTED_COMMAND_CODE_HANDLER( code, func )
```

Parameters

code

[in] The notification code.

func

[in] The name of the message-handler function.

Requirements

Header: atlwin.h

REFLECTED_COMMAND_HANDLER

Similar to [COMMAND_HANDLER](#), but maps commands reflected from the parent window.

```
REFLECTED_COMMAND_HANDLER( id, code, func )
```

Parameters

id

[in] The identifier of the menu item, control, or accelerator.

code

[in] The notification code.

func

[in] The name of the message-handler function.

Requirements

Header: atlwin.h

REFLECTED_COMMAND_ID_HANDLER

Similar to [COMMAND_ID_HANDLER](#), but maps commands reflected from the parent window.

```
REFLECTED_COMMAND_ID_HANDLER( id, func )
```

Parameters

id

[in] The identifier of the menu item, control, or accelerator.

func

[in] The name of the message-handler function.

Requirements

Header: atlwin.h

REFLECTED_COMMAND_RANGE_CODE_HANDLER

Similar to [COMMAND_RANGE_CODE_HANDLER](#), but maps commands reflected from the parent window.

```
REFLECTED_COMMAND_RANGE_CODE_HANDLER( idFirst, idLast, code, func )
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

code

[in] The notification code.

func

[in] The name of the message-handler function.

Requirements

Header: atlwin.h

REFLECTED_COMMAND_RANGE_HANDLER

Similar to [COMMAND_RANGE_HANDLER](#), but maps commands reflected from the parent window.

```
REFLECTED_COMMAND_RANGE_HANDLER( idFirst, idLast, func )
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

func

[in] The name of the message-handler function.

Requirements

Header: atlwin.h

REFLECTED_NOTIFY_CODE_HANDLER

Similar to [NOTIFY_CODE_HANDLER](#), but maps notifications reflected from the parent window.

```
REFLECTED_NOTIFY_CODE_HANDLER_EX( cd, func )
```

Parameters

cd

[in] The notification code.

func

[in] The name of the message-handler function.

Requirements

Header: atlwin.h

REFLECTED_NOTIFY_HANDLER

Similar to [NOTIFY_HANDLER](#), but maps notifications reflected from the parent window.

```
REFLECTED_NOTIFY_HANDLER( id, cd, func )
```

Parameters

id

[in] The identifier of the menu item, control, or accelerator.

cd

[in] The notification code.

func

[in] The name of the message-handler function.

Requirements

Header: atlwin.h

REFLECTED_NOTIFY_ID_HANDLER

Similar to [NOTIFY_ID_HANDLER](#), but maps notifications reflected from the parent window.

```
REFLECTED_NOTIFY_ID_HANDLER( id, func )
```

Parameters

id

[in] The identifier of the menu item, control, or accelerator.

func

[in] The name of the message-handler function.

Requirements

Header: atlwin.h

REFLECTED_NOTIFY_RANGE_CODE_HANDLER

Similar to [NOTIFY_RANGE_CODE_HANDLER](#), but maps notifications reflected from the parent window.

```
REFLECTED_NOTIFY_RANGE_CODE_HANDLER( idFirst, idLast, cd, func )
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

cd

[in] The notification code.

func

[in] The name of the message-handler function.

Requirements

Header: atlwin.h

REFLECTED_NOTIFY_RANGE_HANDLER

Similar to [NOTIFY_RANGE_HANDLER](#), but maps notifications reflected from the parent window.

```
REFLECTED_NOTIFY_RANGE_HANDLER( idFirst, idLast, func )
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

func

[in] The name of the message-handler function.

See also

[Macros](#)

Object Map Macros

3/4/2019 • 2 minutes to read • [Edit Online](#)

These macros define object maps and entries.

<code>DECLARE_OBJECT_DESCRIPTION</code>	Allows you to specify a class object's text description, which will be entered into the object map.
<code>OBJECT_ENTRY_AUTO</code>	Enters an ATL object into the object map, updates the registry, and creates an instance of the object.
<code>OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO</code>	Allows you to specify that the object should be registered and initialized, but it should not be externally creatable via <code>CoCreateInstance</code> .

Requirements

Header: `atlcom.h`

DECLARE_OBJECT_DESCRIPTION

Allows you to specify a text description for your class object.

```
DECLARE_OBJECT_DESCRIPTION( x )
```

Parameters

`x`

[in] The class object's description.

Remarks

ATL enters this description into the object map through the `OBJECT_ENTRY_AUTO` macro.

`DECLARE_OBJECT_DESCRIPTION` implements a `GetObjectDescription` function, which you can use to override the `CCoClass::GetObjectDescription` method.

The `GetObjectDescription` function is called by `IComponentRegistrar::GetComponents`. `IComponentRegistrar` is an Automation interface that allows you to register and unregister individual components in a DLL. When you create a Component Registrar object with the ATL Project Wizard, the wizard will automatically implement the `IComponentRegistrar` interface. `IComponentRegistrar` is typically used by Microsoft Transaction Server.

For more information about the ATL Project Wizard, see the article [Creating an ATL Project](#).

Example

```

class ATL_NO_VTABLE CMyDescribedClass :
    public CComObjectRoot,
    public CComCoClass<CMyDescribedClass, &CLSID_MyDescribedClass>
{
public:
    CMyDescribedClass()
    {
    }

    // Override CComCoClass::GetObjectDescription
    DECLARE_OBJECT_DESCRIPTION("My Described Object 1.0")
};

```

OBJECT_ENTRY_AUTO

Enters an ATL object into the object map, updates the registry, and creates an instance of the object.

```
OBJECT_ENTRY_AUTO( clsid, class )
```

Parameters

clsid

[in] The CLSID of a COM class implemented by the C++ class named *class*.

class

[in] The name of the C++ class implementing the COM class represented by *clsid*.

Remarks

Object entry macros are placed at global scope in the project to provide support for the registration, initialization, and creation of a class.

OBJECT_ENTRY_AUTO enters the function pointers of the creator class and class-factory creator class

`CreateInstance` functions for this object into the auto-generated ATL object map. When

[CAtlComModule::RegisterServer](#) is called, it updates the system registry for each object in the object map.

The table below describes how the information added to the object map is obtained from the class given as the second parameter to this macro.

INFORMATION FOR	OBTAINED FROM
COM registration	Registry Macros
Class factory creation	Class Factory Macros
Instance creation	Aggregation Macros
Component category registration	Category Macros
Class-level initialization and cleanup	ObjectMain

OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO

Allows you to specify that the object should be registered and initialized, but it should not be externally creatable via `CoCreateInstance`.

```
OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO( clsid, class )
```

Parameters

clsid

[in] The CLSID of a COM class implemented by the C++ class named *class*.

class

[in] The name of the C++ class implementing the COM class represented by *clsid*.

Remarks

Object entry macros are placed at global scope in the project to provide support for the registration, initialization, and creation of a class.

OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO allows you to specify that an object should be registered and initialized (see [OBJECT_ENTRY_AUTO](#) for more information), but it should not be creatable via `CoCreateInstance`.

See also

[Macros](#)

Object Status Macros

3/4/2019 • 2 minutes to read • [Edit Online](#)

This macro sets flags belonging to ActiveX controls.

DECLARE_OLEMISC_STATUS	Used in ATL ActiveX controls to set the OLEMISC flags.

Requirements

Header: atlcom.h

DECLARE_OLEMISC_STATUS

Used in ATL ActiveX controls to set the OLEMISC flags.

```
DECLARE_OLEMISC_STATUS( miscstatus )
```

Parameters

miscstatus

All applicable OLEMISC flags.

Remarks

This macro is used to set the OLEMISC flags for an ActiveX control. Refer to [IOleObject::GetMiscStatus](#) for more details.

Example

```
DECLARE_OLEMISC_STATUS(OLEMISC_RECOMPOSEONRESIZE |  
    OLEMISC_CANTLINKINSIDE |  
    OLEMISC_INSIDEOUT |  
    OLEMISC_ACTIVATEWHENVISIBLE |  
    OLEMISC_SETCLIENTSITEFIRST  
)
```

See also

[Macros](#)

Property Map Macros

3/4/2019 • 3 minutes to read • [Edit Online](#)

These macros define property maps and entries.

BEGIN_PROP_MAP	Marks the beginning of the ATL property map.
PROP_DATA_ENTRY	Indicates the extent, or dimensions, of an ActiveX control.
PROP_ENTRY_TYPE	Enters a property description, property DISPID, and property page CLSID into the property map.
PROP_ENTRY_TYPE_EX	Enters a property description, property DISPID, property page CLSID, and <code>IDispatch</code> IID into the property map.
PROP_PAGE	Enters a property page CLSID into the property map.
END_PROP_MAP	Marks the end of the ATL property map.

Requirements

Header: `atlcom.h`

BEGIN_PROP_MAP

Marks the beginning of the object's property map.

```
BEGIN_PROP_MAP(theClass)
```

Parameters

theClass

[in] Specifies the class containing the property map.

Remarks

The property map stores property descriptions, property DISPIDs, property page CLSIDs, and `IDispatch` IIDs. Classes [IPerPropertyBrowsingImpl](#), [IPersistPropertyBagImpl](#), [IPersistStreamInitImpl](#), and [ISpecifyPropertyPagesImpl](#) use the property map to retrieve and set this information.

When you create an object with the ATL Project Wizard, the wizard will create an empty property map by specifying `BEGIN_PROP_MAP` followed by [END_PROP_MAP](#).

`BEGIN_PROP_MAP` does not save out the extent (that is, the dimensions) of a property map, because an object using a property map may not have a user interface, so it would have no extent. If the object is an ActiveX control with a user interface, it has an extent. In this case, you must specify [PROP_DATA_ENTRY](#) in your property map to supply the extent.

Example

```

BEGIN_PROP_MAP(CMyPropCtrl)
    PROP_DATA_ENTRY("_cx", m_sizeExtent.cx, VT_UI4)
    PROP_DATA_ENTRY("_cy", m_sizeExtent.cy, VT_UI4)
    PROP_ENTRY_TYPE("Property1", 1, CLSID_MyPropPage1, VT_BSTR)
    PROP_ENTRY_TYPE_EX("Caption", DISPID_CAPTION, CLSID_MyPropPage2, IID_IMyDual1, VT_BSTR)
    PROP_ENTRY_INTERFACE_CALLBACK("CorrectParamCallback", 0, CLSID_MyPropPage1, AllowedCLSID, VT_DISPATCH)
    PROP_ENTRY_INTERFACE_CALLBACK_EX("CorrectParamCallbackEx", 1, IID_IMyDual1, CLSID_MyPropPage2,
AllowedCLSID, VT_UNKNOWN)
    PROP_PAGE(CLSID_MyPropPage3)
END_PROP_MAP()

```

PROP_DATA_ENTRY

Indicates the extent, or dimensions, of an ActiveX control.

```
PROP_DATA_ENTRY( szDesc, member, vt)
```

Parameters

szDesc

[in] The property description.

member

[in] The data member containing the extent; for example, `m_sizeExtent`.

vt

[in] Specifies the VARIANT type of the property.

Remarks

This macro causes the specified data member to be persisted.

When you create an ActiveX control, the wizard inserts this macro after the property map macro

[BEGIN_PROP_MAP](#) and before the property map macro [END_PROP_MAP](#).

Example

In the following example, the extent of the object (`m_sizeExtent`) is being persisted.

```

BEGIN_PROP_MAP(CMyWindow)
    PROP_DATA_ENTRY("_cx", m_sizeExtent.cx, VT_UI4)
    PROP_DATA_ENTRY("_cy", m_sizeExtent.cy, VT_UI4)
END_PROP_MAP()

```

```

BEGIN_PROP_MAP(CMyCompositeCtrl)
    PROP_DATA_ENTRY("Width", m_nWidth, VT_UI4)
    PROP_DATA_ENTRY("Height", m_nHeight, VT_UI4)
END_PROP_MAP()

```

PROP_ENTRY_TYPE

Use this macro to enter a property description, property DISPID, and property page CLSID into the object's property map.

```
PROP_ENTRY_TYPE( szDesc, dispid, clsid, vt)
```

Parameters

szDesc

[in] The property description.

dispid

[in] The property's DISPID.

clsid

[in] The CLSID of the associated property page. Use the special value CLSID_NULL for a property that does not have an associated property page.

vt

[in] The property's type.

Remarks

The PROP_ENTRY macro was insecure and deprecated. It has been replaced with PROP_ENTRY_TYPE.

The [BEGIN_PROP_MAP](#) macro marks the beginning of the property map; the [END_PROP_MAP](#) macro marks the end.

Example

See the example for [BEGIN_PROP_MAP](#).

PROP_ENTRY_TYPE_EX

Similar to [PROP_ENTRY_TYPE](#), but allows you specify a particular IID if your object supports multiple dual interfaces.

```
PROP_ENTRY_TYPE_EX( szDesc, dispid, clsid, iidDispatch, vt)
```

Parameters

szDesc

[in] The property description.

dispid

[in] The property's DISPID.

clsid

[in] The CLSID of the associated property page. Use the special value CLSID_NULL for a property that does not have an associated property page.

iidDispatch

[in] The IID of the dual interface defining the property.

vt

[in] The property's type.

Remarks

The PROP_ENTRY_EX macro was insecure and deprecated. It has been replaced with PROP_ENTRY_TYPE_EX.

The [BEGIN_PROP_MAP](#) macro marks the beginning of the property map; the [END_PROP_MAP](#) macro marks the end.

Example

The following example groups entries for `IMyDua11` followed by an entry for `IMyDua12`. Grouping by dual interface will improve performance.

```
BEGIN_PROP_MAP(Cat1Edit)
    PROP_ENTRY_TYPE_EX("Caption", DISPID_CAPTION, CLSID_MyPropPage2, IID_IMyDual1, VT_BSTR)
    PROP_ENTRY_TYPE_EX("Enabled", DISPID_ENABLED, CLSID_MyPropPage2, IID_IMyDual1, VT_BOOL)
    PROP_ENTRY_TYPE_EX("Width", DISPID_DRAWWIDTH, CLSID_MyPropPage2, IID_IMyDual2, VT_UINT)
END_PROP_MAP()
```

PROP_PAGE

Use this macro to enter a property page CLSID into the object's property map.

```
PROP_PAGE(clsid)
```

Parameters

clsid

[in] The CLSID of a property page.

Remarks

PROP_PAGE is similar to [PROP_ENTRY_TYPE](#), but does not require a property description or DISPID.

NOTE

If you have already entered a CLSID with PROP_ENTRY_TYPE or [PROP_ENTRY_TYPE_EX](#), you do not need to make an additional entry with PROP_PAGE.

The [BEGIN_PROP_MAP](#) macro marks the beginning of the property map; the [END_PROP_MAP](#) macro marks the end.

Example

```
BEGIN_PROP_MAP(CMyCtrl)
    OtherPropMapEntries
    PROP_PAGE(CLSID_DatePage)
    PROP_PAGE(CLSID_StockColorPage)
END_PROP_MAP()
```

END_PROP_MAP

Marks the end of the object's property map.

```
END_PROP_MAP()
```

Remarks

When you create an object with the ATL Project Wizard, the wizard will create an empty property map by specifying [BEGIN_PROP_MAP](#) followed by END_PROP_MAP.

Example

See the example for [BEGIN_PROP_MAP](#).

See also

[Macros](#)

Registry Data Exchange Macros

3/4/2019 • 3 minutes to read • [Edit Online](#)

These macros perform Registry Data Exchange operations.

<code>BEGIN_RDX_MAP</code>	Marks the beginning of the Registry Data Exchange map.
<code>END_RDX_MAP</code>	Marks the end of the Registry Data Exchange map.
<code>RDX_BINARY</code>	Associates the specified registry entry with a specified member variable of type BYTE.
<code>RDX_CSTRING_TEXT</code>	Associates the specified registry entry with a specified member variable of type CString.
<code>RDX_DWORD</code>	Associates the specified registry entry with a specified member variable of type DWORD.
<code>RDX_TEXT</code>	Associates the specified registry entry with a specified member variable of type TCHAR.

Requirements

Header: atlplus.h

BEGIN_RDX_MAP

Marks the beginning of the Registry Data Exchange map.

```
BEGIN_RDX_MAP
```

Remarks

The following macros are used within the Registry Data Exchange map to read and write entries in the system registry:

MACRO	DESCRIPTION
<code>RDX_BINARY</code>	Associates the specified registry entry with a specified member variable of type BYTE.
<code>RDX_DWORD</code>	Associates the specified registry entry with a specified member variable of type DWORD.
<code>RDX_CSTRING_TEXT</code>	Associates the specified registry entry with a specified member variable of type CString.
<code>RDX_TEXT</code>	Associates the specified registry entry with a specified member variable of type TCHAR.

The global function [RegistryDataExchange](#), or the member function of the same name created by the BEGIN_RDX_MAP and END_RDX_MAP macros, should be used whenever your code needs to exchange data between the system registry and the variables specified in the RDX map.

END_RDX_MAP

Marks the end of the Registry Data Exchange map.

```
END_RDX_MAP
```

RDX_BINARY

Associates the specified registry entry with a specified member variable of type BYTE.

```
RDX_BINARY(  
    rootkey,  
    subkey,  
    valuenam,  
    member,  
    member_size )
```

Parameters

rootkey

The registry key root.

subkey

The registry subkey.

valuenam

The registry key.

member

The member variable to associate with the specified registry entry.

member_size

The size, in bytes, of the member variable.

Remarks

This macro is used in conjunction with the BEGIN_RDX_MAP and END_RDX_MAP macros to associate a member variable with a given registry entry. The global function [RegistryDataExchange](#), or the member function of the same name created by the BEGIN_RDX_MAP and END_RDX_MAP macros, should be used to perform exchange of data between the system registry and the member variables in the RDX map.

RDX_CSTRING_TEXT

Associates the specified registry entry with a specified member variable of type CString.

```
RDX_CSTRING_TEXT(  
    rootkey,  
    subkey,  
    valuenam,  
    member,  
    member_size )
```

Parameters

rootkey

The registry key root.

subkey

The registry subkey.

valuename

The registry key.

member

The member variable to associate with the specified registry entry.

member_size

The size, in bytes, of the member variable.

Remarks

This macro is used in conjunction with the `BEGIN_RDX_MAP` and `END_RDX_MAP` macros to associate a member variable with a given registry entry. The global function [RegistryDataExchange](#), or the member function of the same name created by the `BEGIN_RDX_MAP` and `END_RDX_MAP` macros, should be used to perform exchange of data between the system registry and the member variables in the RDX map.

RDX_DWORD

Associates the specified registry entry with a specified member variable of type `DWORD`.

```
RDX_DWORD(  
    rootkey,  
    subkey,  
    valuename,  
    member,  
    member_size )
```

Parameters

rootkey

The registry key root.

subkey

The registry subkey.

valuename

The registry key.

member

The member variable to associate with the specified registry entry.

member_size

The size, in bytes, of the member variable.

Remarks

This macro is used in conjunction with the `BEGIN_RDX_MAP` and `END_RDX_MAP` macros to associate a member variable with a given registry entry. The global function [RegistryDataExchange](#), or the member function of the same name created by the `BEGIN_RDX_MAP` and `END_RDX_MAP` macros, should be used to perform exchange of data between the system registry and the member variables in the RDX map.

RDX_TEXT

Associates the specified registry entry with a specified member variable of type `TCHAR`.

```
RDX_TEXT(  
    rootkey,  
    subkey,  
    valuenam,  
    member,  
    member_size )
```

Parameters

rootkey

The registry key root.

subkey

The registry subkey.

valuenam

The registry key.

member

The member variable to associate with the specified registry entry.

member_size

The size, in bytes, of the member variable.

Remarks

This macro is used in conjunction with the BEGIN_RDX_MAP and END_RDX_MAP macros to associate a member variable with a given registry entry. The global function [RegistryDataExchange](#), or the member function of the same name created by the BEGIN_RDX_MAP and END_RDX_MAP macros, should be used to perform exchange of data between the system registry and the member variables in the RDX map.

See also

[Macros](#)

[RegistryDataExchange](#)

Registry Macros

3/4/2019 • 4 minutes to read • [Edit Online](#)

These macros define useful type library and registry facilities.

_ATL_STATIC_REGISTRY	Indicates that you want the registration code for your object to be in the object to avoid a dependency on ATL.DLL.
DECLARE_LIBID	Provides a way for ATL to obtain the <i>libid</i> of the type library.
DECLARE_NO_REGISTRY	Avoids default ATL registration.
DECLARE_REGISTRY	Enters or removes the main object's entry in the system registry.
DECLARE_REGISTRY_APPID_RESOURCEID	Specifies the information required to automatically register the <i>appid</i> .
DECLARE_REGISTRY_RESOURCE	Finds the named resource and runs the registry script within it.
DECLARE_REGISTRY_RESOURCEID	Finds the resource identified by an ID number and runs the registry script within it.

Requirements

Header: atlcom.h

[_ATL_STATIC_REGISTRY](#)

A symbol that indicates you want the registration code for your object to be in the object to avoid a dependency on ATL.DLL.

```
#define _ATL_STATIC_REGISTRY
```

Remarks

When you define ATL_STATIC_REGISTRY, you should use the following code:

```
#ifdef _ATL_STATIC_REGISTRY
#include <statreg.h>
#endif
```

[DECLARE_LIBID](#)

Provides a way for ATL to obtain the *libid* of the type library.

```
DECLARE_LIBID( libid )
```

Parameters

libid

The GUID of the type library.

Remarks

Use DECLARE_LIBID in a `CAAtlModuleT`-derived class.

Example

Non-attributed wizard-generated ATL projects will have a sample of using this macro.

DECLARE_NO_REGISTRY

Use DECLARE_NO_REGISTRY if you want to avoid any default ATL registration for the class in which this macro appears.

```
DECLARE_NO_REGISTRY()
```

DECLARE_REGISTRY

Enters the standard class registration into the system registry or removes it from the system registry.

```
DECLARE_REGISTRY(  
    class,  
    pid,  
    vpid,  
    nid,  
    flags )
```

Parameters

class

[in] Included for backward compatibility.

pid

[in] An LPCTSTR that is a version-specific program identifier.

vpid

[in] An LPCTSTR that is a version-independent program identifier.

nid

[in] A UINT that is an index of the resource string in the registry to use as the description of the program.

flags

[in] A DWORD containing the program's threading model in the registry. Must be one of the following values: `THREADFLAGS_APARTMENT`, `THREADFLAGS_BOTH`, or `AUTPRXFLAG`.

Remarks

The standard registration consists of the CLSID, program ID, version-independent program ID, description string, and thread model.

When you create an object or control using the ATL Add Class Wizard, the wizard automatically implements script-based registry support and adds the [DECLARE_REGISTRY_RESOURCEID](#) macro to your files. If you do not want script-based registry support, you need to replace this macro with `DECLARE_REGISTRY`.

DECLARE_REGISTRY only inserts the five basic keys described above into the registry. You must manually write code to insert other keys into the registry.

DECLARE_REGISTRY_APPID_RESOURCEID

Specifies the information required to automatically register the *appid*.

```
DECLARE_REGISTRY_APPID_RESOURCEID(  
    resid,  
    appid )
```

Parameters

resid

The resource id of the .rgs file that contains information about the *appid*.

appid

A GUID.

Remarks

Use DECLARE_REGISTRY_APPID_RESOURCEID in a `CAtlModuleT`-derived class.

Example

Classes added to ATL projects with the Add Class code wizard will have a sample of using this macro.

DECLARE_REGISTRY_RESOURCE

Gets the named resource containing the registry file and runs the script to either enter objects into the system registry or remove them from the system registry.

```
DECLARE_REGISTRY_RESOURCE( x )
```

Parameters

x

[in] String identifier of your resource.

Remarks

When you create an object or control using the ATL Project Wizard, the wizard will automatically implement script-based registry support and add the [DECLARE_REGISTRY_RESOURCEID](#) macro, which is similar to DECLARE_REGISTRY_RESOURCE, to your files.

You can statically link with the ATL Registry Component (Registrar) for optimized registry access. To statically link to the Registrar code, add the following line to your stdafx.h file:

```
#define _ATL_STATIC_REGISTRY
```

If you want ATL to substitute replacement values at run time, do not specify the DECLARE_REGISTRY_RESOURCE or DECLARE_REGISTRY_RESOURCEID macro. Instead, create an array of `_ATL_REGMAP_ENTRIES` structures, where each entry contains a variable placeholder paired with a value to replace the placeholder at run time. Then call [CAtlModule::UpdateRegistryFromResourceD](#) or [CAtlModule::UpdateRegistryFromResourceS](#), passing the array. This adds all of the replacement values in the `_ATL_REGMAP_ENTRIES` structures to the Registrar's replacement map.

For more information about replaceable parameters and scripting, see the article [The ATL Registry Component](#)

(Registrar).

DECLARE_REGISTRY_RESOURCEID

Same as [DECLARE_REGISTRY_RESOURCE](#) except that it uses a wizard-generated UINT to identify the resource, rather than a string name.

```
DECLARE_REGISTRY_RESOURCEID( x )
```

Parameters

x

[in] Wizard-generated identifier of your resource.

Remarks

When you create an object or control using the ATL Project Wizard, the wizard will automatically implement script-based registry support and add the DECLARE_REGISTRY_RESOURCEID macro to your files.

You can statically link with the ATL Registry Component (Registrar) for optimized registry access. To statically link to the Registrar code, add the following line to your stdafx.h file:

```
#define _ATL_STATIC_REGISTRY
```

If you want ATL to substitute replacement values at run time, do not specify the DECLARE_REGISTRY_RESOURCE or DECLARE_REGISTRY_RESOURCEID macro. Instead, create an array of `_ATL_REGMAP_ENTRIES` structures, where each entry contains a variable placeholder paired with a value to replace the placeholder at run time. Then call [CAtlModule::UpdateRegistryFromResourceD](#) or [CAtlModule::UpdateRegistryFromResourceS](#), passing the array. This adds all of the replacement values in the `_ATL_REGMAP_ENTRIES` structures to the Registrar's replacement map.

For more information about replaceable parameters and scripting, see the article [The ATL Registry Component \(Registrar\)](#).

See also

[Macros](#)

Service Map Macros

3/4/2019 • 2 minutes to read • [Edit Online](#)

These macros define service maps and entries.

<code>BEGIN_SERVICE_MAP</code>	Marks the beginning of an ATL service map.
<code>END_SERVICE_MAP</code>	Marks the end of an ATL service map.
<code>SERVICE_ENTRY</code>	Indicates that the object supports a specific service ID.
<code>SERVICE_ENTRY_CHAIN</code>	Instructs <code>IServiceProviderImpl::QueryService</code> to chain to the specified object.

Requirements

Header: `atlcom.h`

BEGIN_SERVICE_MAP

Marks the beginning of the service map.

```
BEGIN_SERVICE_MAP(theClass)
```

Parameters

theClass

[in] Specifies the class containing the service map.

Remarks

Use the service map to implement service provider functionality on your COM object. First, you must derive your class from `IServiceProviderImpl`. There are two types of entries:

- `SERVICE_ENTRY` Indicates support for the specified service ID (SID).
- `SERVICE_ENTRY_CHAIN` Instructs `IServiceProviderImpl::QueryService` to chain to another, specified object.

Example

```
BEGIN_SERVICE_MAP(CMyService)
    SERVICE_ENTRY(SID_SBindHost) // This object supports the SBindHost service
    SERVICE_ENTRY_CHAIN(m_spClientSite) // Everything else, just ask the container
END_SERVICE_MAP()
```

END_SERVICE_MAP

Marks the end of the service map.

```
END_SERVICE_MAP()
```

Example

See the example for [BEGIN_SERVICE_MAP](#).

SERVICE_ENTRY

Indicates that the object supports the service id specified by *SID*.

```
SERVICE_ENTRY( SID )
```

Parameters

SID

The service ID.

Example

See the example for [BEGIN_SERVICE_MAP](#).

SERVICE_ENTRY_CHAIN

Instructs [IServiceProviderImpl::QueryService](#) to chain to the object specified by *punk*.

```
SERVICE_ENTRY_CHAIN( punk )
```

Parameters

punk

A pointer to the **IUnknown** interface to which to chain.

Example

See the example for [BEGIN_SERVICE_MAP](#).

IServiceProviderImpl::QueryService

Creates or accesses the specified service and returns an interface pointer to the specified interface for the service.

```
STDMETHOD(QueryService)(  
    REFGUID guidService,  
    REFIID riid,  
    void** ppvObject);
```

Parameters

guidService

[in] Pointer to a service identifier (SID).

riid

[in] Identifier of the interface to which the caller is to gain access.

ppvObj

[out] Indirect pointer to the requested interface.

Return Value

The returned HRESULT value is one of the following:

RETURN VALUE	MEANING
S_OK	The service was successfully created or retrieved.
E_INVALIDARG	One or more of the arguments is invalid.
E_OUTOFMEMORY	Memory is insufficient to create the service.
E_UNEXPECTED	An unknown error occurred.
E_NOINTERFACE	The requested interface is not part of this service, or the service is unknown.

Remarks

`QueryService` returns an indirect pointer to the requested interface in the specified service. The caller is responsible for releasing this pointer when it is no longer required.

When you call `QueryService`, you pass both a service identifier (*guidService*) and an interface identifier (*riid*). The *guidService* specifies the service to which you want access, and the *riid* identifies an interface that is part of the service. In return, you receive an indirect pointer to the interface.

The object that implements the interface might also implement interfaces that are part of other services. Consider the following:

- Some of these interfaces might be optional. Not all interfaces defined in the service description are necessarily present on every implementation of the service or on every returned object.
- Unlike calls to `QueryInterface`, passing a different service identifier does not necessarily mean that a different Component Object Model (COM) object is returned.
- The returned object might have additional interfaces that are not part of the definition of the service.

Two different services, such as `SID_SMyService` and `SID_SYourService`, can both specify the use of the same interface, even though the implementation of the interface might have nothing in common between the two services. This works, because a call to `QueryService` (`SID_SMyService`, `IID_IDispatch`) can return a different object than `QueryService` (`SID_SYourService`, `IID_IDispatch`). Object identity is not assumed when you specify a different service identifier.

See also

[Macros](#)

Snap-In Object Macros

3/4/2019 • 2 minutes to read • [Edit Online](#)

These macros provide support for snap-in extensions.

BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP	Marks the beginning of the snap-in extension data class map for a Snap-In object.
BEGIN_SNAPINTOOLBARID_MAP	Marks the beginning of the toolbar map for a Snap-In object.
END_EXTENSION_SNAPIN_NODEINFO_MAP	Marks the end of the snap-in extension data class map for a Snap-In object.
END_SNAPINTOOLBARID_MAP	Marks the end of the toolbar map for a Snap-In object.
EXTENSION_SNAPIN_DATACLASS	Creates a data member for the data class of the snap-in extension.
EXTENSION_SNAPIN_NODEINFO_ENTRY	Enters a snap-in extension data class into the snap-in extension data class map of the Snap-In object.
SNAPINMENUID	Declares the ID of the context menu used by the Snap-In object.
SNAPINTOOLBARID_ENTRY	Enters a toolbar into the toolbar map of the Snap-In object.

Requirements

Header: atlsnap.h

BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP

Marks the beginning of the snap-in extension data class map.

```
BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP(classname)
```

Parameters

classname

[in] The name of the snap-in extension data class.

Remarks

Start your snap-in extension map with the [BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP](#) macro, add entries for each of your snap-in extension data types with the [EXTENSION_SNAPIN_NODEINFO_ENTRY](#) macro, and complete the map with the [END_EXTENSION_SNAPIN_NODEINFO_MAP](#) macro.

Example

```

class CMyExtSnapinExtData :
    public CSnapInItemImpl<CMyExtSnapinExtData>,
    public CMySnapinBase
{
public:
    CMyExtSnapinExtData() {}
};

class CMyExtSnapin :
    public CComObjectRoot,
    public CSnapInObjectRoot<1, CMyExtSnapin>,
    public IComponentDataImpl<CMyExtSnapin, CMyExtSnapin>
{
public:
    CMyExtSnapin() {}

    BEGIN_COM_MAP(CMyExtSnapin)
    END_COM_MAP()

    EXTENSION_SNAPIN_DATACLASS(CMyExtSnapinExtData)

    BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP(CMyExtSnapin)
        EXTENSION_SNAPIN_NODEINFO_ENTRY(CMyExtSnapinExtData)
    END_EXTENSION_SNAPIN_NODEINFO_MAP()
};

```

BEGIN_SNAPINTOOLBARID_MAP

Declares the beginning of the toolbar ID map for the Snap-In object.

```
BEGIN_SNAPINTOOLBARID_MAP(_class)
```

Parameters

_class

[in] Specifies the Snap-In object class.

Example

```

class CMySnapinData :
    public CSnapInItemImpl<CMySnapinData>,
    public CMySnapinBase
{
public:
    CMySnapinData() {}

    BEGIN_SNAPINTOOLBARID_MAP(CMySnapinData)
        // IDR_MYSNAPINTOOLBAR is the resource ID of a toolbar resource.
        SNAPINTOOLBARID_ENTRY(IDR_MYSNAPINTOOLBAR)
    END_SNAPINTOOLBARID_MAP()
};

```

END_EXTENSION_SNAPIN_NODEINFO_MAP

Marks the end of the snap-in extension data class map.

```
END_EXTENSION_SNAPIN_NODEINFO_MAP()
```

Remarks

Start your snap-in extension map with the [BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP](#) macro, add entries for each of your extension snap-in data types with the [EXTENSION_SNAPIN_NODEINFO_ENTRY](#) macro, and complete the map with the [END_EXTENSION_SNAPIN_NODEINFO_MAP](#) macro.

Example

See the example for [BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP](#).

END_SNAPINTOOLBARID_MAP

Declares the end of the toolbar ID map for the Snap-In object.

```
END_SNAPINTOOLBARID_MAP( _class )
```

Parameters

_class

[in] Specifies the Snap-In object class.

Example

See the example for [BEGIN_SNAPINTOOLBARID_MAP](#).

EXTENSION_SNAPIN_DATACLASS

Adds a data member to the snap-in extension data class for an **ISnapInItemImpl**-derived class.

```
EXTENSION_SNAPIN_DATACLASS(dataClass )
```

Parameters

dataClass

[in] The data class of the snap-in extension.

Remarks

This class should also be entered into a snap-in extension data class map. Start your snap-in extension data class map with the [BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP](#) macro, add entries for each of your snap-in extension data types with the [EXTENSION_SNAPIN_NODEINFO_ENTRY](#) macro, and complete the map with the [END_EXTENSION_SNAPIN_NODEINFO_MAP](#) macro.

Example

```

class CMyExtSnapinExtData :
    public CSnapInItemImpl<CMyExtSnapinExtData>,
    public CMySnapinBase
{
public:
    CMyExtSnapinExtData() {}
};

class CMyExtSnapin :
    public CComObjectRoot,
    public CSnapInObjectRoot<1, CMyExtSnapin>,
    public IComponentDataImpl<CMyExtSnapin, CMyExtSnapin>
{
public:
    CMyExtSnapin() {}

    BEGIN_COM_MAP(CMyExtSnapin)
    END_COM_MAP()

    EXTENSION_SNAPIN_DATACLASS(CMyExtSnapinExtData)

    BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP(CMyExtSnapin)
        EXTENSION_SNAPIN_NODEINFO_ENTRY(CMyExtSnapinExtData)
    END_EXTENSION_SNAPIN_NODEINFO_MAP()
};

```

EXTENSION_SNAPIN_NODEINFO_ENTRY

Adds a snap-in extension data class to the snap-in extension data class map.

```
EXTENSION_SNAPIN_NODEINFO_ENTRY( dataClass )
```

Parameters

dataClass

[in] The data class of the snap-in extension.

Remarks

Start your snap-in extension data class map with the [BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP](#) macro, add entries for each of your snap-in extension data types with the `EXTENSION_SNAPIN_NODEINFO_ENTRY` macro, and complete the map with the [END_EXTENSION_SNAPIN_NODEINFO_MAP](#) macro.

Example

See the example for [BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP](#).

SNAPINMENUID

Use this macro to declare the context menu resource of the Snap-In object.

```
SNAPINMENUID( id )
```

Parameters

id

[in] Identifies the context menu of the Snap-In object.

SNAPINTOOLBARID_ENTRY

Use this macro to enter a toolbar ID into the Snap-In object's toolbar ID map.

```
SNAPINTOOLBARID_ENTRY( id )
```

Parameters

id

[in] Identifies the toolbar control.

Remarks

The [BEGIN_SNAPINTOOLBARID_MAP](#) macro marks the beginning of the toolbar ID map; the [END_SNAPINTOOLBARID_MAP](#) macro marks the end.

Example

See the example for [BEGIN_SNAPINTOOLBARID_MAP](#).

See also

[Macros](#)

String Conversion Macros

3/4/2019 • 2 minutes to read • [Edit Online](#)

These macros provide string conversion features.

ATL and MFC String Conversion Macros

The string conversion macros discussed here are valid for both ATL and MFC. For more information on MFC string conversion, see [TN059: Using MFC MBCS/Unicode Conversion Macros](#) and [MFC Macros and Globals](#).

DEVMODE and TEXTMETRIC String Conversion Macros

These macros create a copy of a [DEVMODE](#) or [TEXTMETRIC](#) structure and convert the strings within the new structure to a new string type. The macros allocate memory on the stack for the new structure and return a pointer to the new structure.

```
MACRONAME( address_of_structure )
```

Remarks

For example:

```
DEVMODEW* lpw = DEVMODEA2W(lpa);
```

and:

```
TEXTMETRICW* lptmw = TEXTMETRICA2W(lptma);
```

In the macro names, the string type in the source structure is on the left (for example, **A**) and the string type in the destination structure is on the right (for example, **W**). **A** stands for LPSTR, **OLE** stands for LPOLESTR, **T** stands for LPTSTR, and **W** stands for LPWSTR.

Thus, DEVMODEA2W copies a [DEVMODE](#) structure with LPSTR strings into a [DEVMODE](#) structure with LPWSTR strings, TEXTMETRICOLE2T copies a [TEXTMETRIC](#) structure with LPOLESTR strings into a [TEXTMETRIC](#) structure with LPTSTR strings, and so on.

The two strings converted in the [DEVMODE](#) structure are the device name ([dmDeviceName](#)) and the form name ([dmFormName](#)). The [DEVMODE](#) string conversion macros also update the structure size ([dmSize](#)).

The four strings converted in the [TEXTMETRIC](#) structure are the first character ([tmFirstChar](#)), the last character ([tmLastChar](#)), the default character ([tmDefaultChar](#)), and the break character ([tmBreakChar](#)).

The behavior of the [DEVMODE](#) and [TEXTMETRIC](#) string conversion macros depends on the compiler directive in effect, if any. If the source and destination types are the same, no conversion takes place. Compiler directives change **T** and **OLE** as follows:

COMPILER DIRECTIVE IN EFFECT	T BECOMES	OLE BECOMES
none	A	W

COMPILER DIRECTIVE IN EFFECT	T BECOMES	OLE BECOMES
_UNICODE	W	W
OLE2ANSI	A	A
_UNICODE and OLE2ANSI	W	A

The following table lists the `DEVMODE` and `TEXTMETRIC` string conversion macros.

DEVMODEA2W	TEXTMETRICA2W
DEVMODEOLE2T	TEXTMETRICOLE2T
DEVMODET2OLE	TEXTMETRICT2OLE
DEVMODEW2A	TEXTMETRICW2A

See also

[Macros](#)

Window Class Macros

3/4/2019 • 3 minutes to read • [Edit Online](#)

These macros define window class utilities.

DECLARE_WND_CLASS	Allows you to specify the name of a new window class.
DECLARE_WND_CLASS2	(Visual Studio 2017) Allows you to specify the name of a new window class and the enclosing class whose window procedure the new class will use.
DECLARE_WND_SUPERCLASS	Allows you to specify the name of an existing window class on which a new window class will be based.
DECLARE_WND_CLASS_EX	Allows you to specify the parameters of a class.

Requirements

Header: atlwin.h

DECLARE_WND_CLASS

Allows you to specify the name of a new window class. Place this macro in an ATL ActiveX control's control class.

```
DECLARE_WND_CLASS( WndClassName )
```

Parameters

WndClassName

[in] The name of the new window class. If NULL, ATL will generate a window class name.

Remarks

If you are using the /permissive- compiler option, then DECLARE_WND_CLASS will cause a compiler error; use DECLARE_WND_CLASS2 instead.

DECLARE_WND_CLASS allows you to specify the name of a new window class whose information will be managed by [CWndClassInfo](#). DECLARE_WND_CLASS defines the new window class by implementing the following static function:

```
static CWndClassInfo& GetWndClassInfo();
```

DECLARE_WND_CLASS specifies the following styles for the new window:

- CS_HREDRAW
- CS_VREDRAW
- CS_DBLCLKS

DECLARE_WND_CLASS also specifies the default window's background color. Use the

[DECLARE_WND_CLASS_EX](#) macro to provide your own styles and background color.

[CWindowImpl](#) uses the [DECLARE_WND_CLASS](#) macro to create a window based on a new window class. To override this behavior, use the [DECLARE_WND_SUPERCLASS](#) macro, or provide your own implementation of the [GetWndClassInfo](#) function.

For more information about using windows in ATL, see the article [ATL Window Classes](#).

DECLARE_WND_CLASS2

(Visual Studio 2017) Similar to [DECLARE_WND_CLASS](#), but with an extra parameter that avoids a dependent name error when compiling with the `/permissive-` option.

```
DECLARE_WND_CLASS2( WndClassName, EnclosingClass )
```

Parameters

WndClassName

[in] The name of the new window class. If NULL, ATL will generate a window class name.

EnclosingClass

[in] The name of the window class that encloses the new window class. Cannot be NULL.

Remarks

If you are using the `/permissive-` option, then [DECLARE_WND_CLASS](#) will cause a compilation error because it contains a dependent name. [DECLARE_WND_CLASS2](#) requires you to explicitly name the class that this macro is used in and does not cause the error under the `/permissive-` flag. Otherwise this macro is identical to [DECLARE_WND_CLASS](#).

DECLARE_WND_SUPERCLASS

Allows you to specify the parameters of a class. Place this macro in an ATL ActiveX control's control class.

```
DECLARE_WND_SUPERCLASS( WndClassName, OrigWndClassName )
```

Parameters

WndClassName

[in] The name of the window class that will superclass *OrigWndClassName*. If NULL, ATL will generate a window class name.

OrigWndClassName

[in] The name of an existing window class.

Remarks

This macro allows you to specify the name of a window class that will superclass an existing window class. [CWndClassInfo](#) manages the information of the superclass.

[DECLARE_WND_SUPERCLASS](#) implements the following static function:

```
static CWndClassInfo& GetWndClassInfo();
```

By default, [CWindowImpl](#) uses the [DECLARE_WND_CLASS](#) macro to create a window based on a new window class. By specifying the [DECLARE_WND_SUPERCLASS](#) macro in a `CWindowImpl`-derived class, the window class will be based on an existing class but will use your window procedure. This technique is called superclassing.

Besides using the `DECLARE_WND_CLASS` and `DECLARE_WND_SUPERCLASS` macros, you can override the [GetWndClassInfo](#) function with your own implementation.

For more information about using windows in ATL, see the article [ATL Window Classes](#).

DECLARE_WND_CLASS_EX

Allows you to specify the name of an existing window class on which a new window class will be based. Place this macro in an ATL ActiveX control's control class.

```
DECLARE_WND_CLASS_EX( WndClassName, style, bkgnd )
```

Parameters

WndClassName

[in] The name of the new window class. If NULL, ATL will generate a window class name.

style

[in] The style of the window.

bkgnd

[in] The background color of the window.

Remarks

This macro allows you to specify the class parameters of a new window class, whose information will be managed by [CWndClassInfo](#). `DECLARE_WND_CLASS_EX` defines the new window class by implementing the following static function:

```
static CWndClassInfo& GetWndClassInfo();
```

If you want to use the default styles and background color, use the `DECLARE_WND_CLASS` macro. For more information about using windows in ATL, see the article [ATL Window Classes](#).

See also

[Macros](#)

Windows Messages Macros

3/4/2019 • 2 minutes to read • [Edit Online](#)

This macro forwards window messages.

<code>WM_FORWARDMSG</code>	Use to forward a message received by a window to another window for processing.

Requirements

Header: `atlbase.h`

WM_FORWARDMSG

This macro forwards a message received by a window to another window for processing.

`WM_FORWARDMSG`

Return Value

Nonzero if the message was processed, zero if not.

Remarks

Use `WM_FORWARDMSG` to forward a message received by a window to another window for processing. The `LPARAM` and `WPARAM` parameters are used as follows:

PARAMETER	USAGE
<code>WPARAM</code>	Data defined by user
<code>LPARAM</code>	A pointer to a <code>MSG</code> structure that contains information about a message

Example

In the following example, `m_hWndOther` represents the other window that receives this message.

```
LRESULT CMyWindow::OnMsg(UINT nMsg, WPARAM wParam, LPARAM lParam,
    BOOL& bHandled)
{
    MSG msg = { m_hWnd, nMsg, wParam, lParam, 0, { 0, 0 } };
    LRESULT lRet = SendMessage(m_hWndOther, WM_FORWARDMSG, 0, (LPARAM)&msg);
    if(lRet == 0) // not handled
        bHandled = FALSE;
    return 0;
}
```

See also

[Macros](#)

ATL Operators

3/4/2019 • 3 minutes to read • [Edit Online](#)

This section contains the reference topics for the ATL global operators.

OPERATOR	DESCRIPTION
<code>operator ==</code>	Compares two <code>CSid</code> objects or <code>SID</code> structures for equality.
<code>operator !=</code>	Compares two <code>CSid</code> objects or <code>SID</code> structures for inequality.
<code>operator <</code>	Tests if the <code>CSid</code> object or <code>SID</code> structure on the left side of the operator is less than the <code>CSid</code> object or <code>SID</code> structure on the right side (for C++ Standard Library compatibility).
<code>operator ></code>	Tests if the <code>CSid</code> object or <code>SID</code> structure on the left side of the operator is greater than the <code>CSid</code> object or <code>SID</code> structure on the right side (for C++ Standard Library compatibility).
<code>operator <=</code>	Tests if the <code>CSid</code> object or <code>SID</code> structure on the left side of the operator is less than or equal to the <code>CSid</code> object or <code>SID</code> structure on the right side (for C++ Standard Library compatibility).
<code>operator >=</code>	Tests if the <code>CSid</code> object or <code>SID</code> structure on the left side of the operator is greater than or equal to the <code>CSid</code> object or <code>SID</code> structure on the right side (for C++ Standard Library compatibility).

Requirements

Header: `atlsecurity.h`.

`operator ==`

Compares `CSid` objects or `SID` (security identifier) structures for equality.

```
bool operator==(const CSid& lhs, const CSid& rhs) throw();
```

Parameters

lhs

The first `CSid` object or `SID` structure to compare.

rhs

The second `CSid` object or `SID` structure to compare.

Return Value

Returns TRUE if the objects are equal, FALSE if they are not equal.

operator !=

Compares `CSid` objects or `SID` (security identifier) structures for inequality.

```
bool operator==(const CSid& lhs, const CSid& rhs) throw();
```

Parameters

lhs

The first `CSid` object or `SID` structure to compare.

rhs

The second `CSid` object or `SID` structure to compare.

Return Value

Returns TRUE if the objects are not equal, FALSE if they are equal.

operator <

Tests if the `CSid` object or `SID` structure on the left side of the operator is less than the `CSid` object or `SID` structure on the right side (for C++ Standard Library compatibility).

```
bool operator<(const CSid& lhs, const CSid& rhs) throw();
```

Parameters

lhs

The first `CSid` object or `SID` structure to compare.

rhs

The second `CSid` object or `SID` structure to compare.

Return Value

Returns TRUE if the address of the *lhs* object is less than the address of the *rhs* object, FALSE otherwise.

Remarks

This operator acts on the address of the `CSid` object or `SID` structure, and is implemented to provide compatibility with C++ Standard Library collection classes.

operator >

Tests if the `CSid` object or `SID` structure on the left side of the operator is greater than the `CSid` object or `SID` structure on the right side (for C++ Standard Library compatibility).

```
bool operator<(const CSid& lhs, const CSid& rhs) throw();
```

Parameters

lhs

The first `CSid` object or `SID` structure to compare.

rhs

The second `CSid` object or `SID` structure to compare.

Return Value

Returns TRUE if the address of the *lhs* is greater than the address of the *rhs*, FALSE otherwise.

Remarks

This operator acts on the address of the `CSid` object or `SID` structure, and is implemented to provide compatibility with C++ Standard Library collection classes.

operator <=

Tests if the `CSid` object or `SID` structure on the left side of the operator is less than or equal to the `CSid` object or `SID` structure on the right side (for C++ Standard Library compatibility).

```
bool operator<(const CSid& lhs, const CSid& rhs) throw();
```

Parameters

lhs

The first `CSid` object or `SID` structure to compare.

rhs

The second `CSid` object or `SID` structure to compare.

Return Value

Returns TRUE if the address of the *lhs* is less than or equal to the address of the *rhs*, FALSE otherwise.

Remarks

This operator acts on the address of the `CSid` object or `SID` structure, and is implemented to provide compatibility with C++ Standard Library collection classes.

operator >=

Tests if the `CSid` object or `SID` structure on the left side of the operator is greater than or equal to the `CSid` object or `SID` structure on the right side (for C++ Standard Library compatibility).

```
bool operator<(const CSid& lhs, const CSid& rhs) throw();
```

Parameters

lhs

The first `CSid` object or `SID` structure to compare.

rhs

The second `CSid` object or `SID` structure to compare.

Return Value

Returns TRUE if the address of the *lhs* is greater than or equal to the address of the *rhs*, FALSE otherwise.

Remarks

This operator acts on the address of the `CSid` object or `SID` structure, and is implemented to provide compatibility with C++ Standard Library collection classes.

ATL Global Variables

10/31/2018 • 2 minutes to read • [Edit Online](#)

_pAtlModule

A global variable storing a pointer to the current module.

```
__declspec(selectany) CAtlModule * _pAtlModule
```

Remarks

Methods on this global variable can be used to provide the functionality that the (now obsolete) class CComModule provided in Visual C++ 6.0.

Example

```
LONG lLocks = _pAtlModule->GetLockCount();
```

Requirements

Header: atlbase.h

ATL Typedefs

3/4/2019 • 4 minutes to read • [Edit Online](#)

The Active Template Library includes the following typedefs.

_ATL_BASE_MODULE	Defined as a typedef based on _ATL_BASE_MODULE70 .
_ATL_COM_MODULE	Defined as a typedef based on _ATL_COM_MODULE70 .
_ATL_MODULE	Defined as a typedef based on _ATL_MODULE70 .
_ATL_WIN_MODULE	Defined as a typedef based on _ATL_WIN_MODULE70 .
ATL_URL_PORT	The type used by CUrl for specifying a port number.
CComDispatchDriver	This class manages COM interface pointers.
CComGlobalsThreadModel	Calls the appropriate thread model methods, regardless of the threading model being used.
CComObjectThreadModel	Calls the appropriate thread model methods, regardless of the threading model being used.
CContainedWindow	This class is a specialization of <code>CContainedWindowT</code> .
CPath	A specialization of CPathT using <code>CString</code> .
CPathA	A specialization of CPathT using <code>CStringA</code> .
CPathW	A specialization of CPathT using <code>CStringW</code> .
CSimpleValArray	Represents an array for storing simple types.
DefaultThreadTraits	The default thread traits class.
LPCURL	A pointer to a constant CUrl object.
LPURL	A pointer to a CUrl object.

[_ATL_BASE_MODULE](#)

Defined as a typedef based on [_ATL_BASE_MODULE70](#).

```
typedef ATL::_ATL_BASE_MODULE70 _ATL_BASE_MODULE;
```

Remarks

Used in every ATL project. Based on [_ATL_BASE_MODULE70](#).

Classes that are part of the ATL 7.0 Module Classes derive from the `_ATL_BASE_MODULE` structure. For more information on ATL Module Classes, refer to [COM Modules Classes](#).

Requirements

Header: `atlcore.h`

`_ATL_COM_MODULE`

Defined as a typedef based on `_ATL_COM_MODULE70`.

```
typedef ATL::_ATL_COM_MODULE70 _ATL_COM_MODULE;
```

Remarks

Used by ATL projects which use COM features. Based on [_ATL_COM_MODULE70](#).

Requirements

Header: `atlbase.h`

`_ATL_MODULE`

Defined as a typedef based on `_ATL_MODULE70`.

```
typedef ATL::_ATL_MODULE70 _ATL_MODULE;
```

Requirements

Header:

Remarks

Based on [_ATL_MODULE70](#).

`_ATL_WIN_MODULE`

Defined as a typedef based on `_ATL_WIN_MODULE70`.

```
typedef ATL::_ATL_WIN_MODULE70 _ATL_WIN_MODULE;
```

Remarks

Used by any ATL projects which use windowing features. Based on [_ATL_WIN_MODULE70](#).

Requirements

Header: `atlbase.h`

`ATL_URL_PORT`

The type used by [CUrl](#) for specifying a port number.

```
typedef WORD ATL_URL_PORT;
```

Requirements

Header: atlutil.h

CComDispatchDriver

This class manages COM interface pointers.

```
typedef CComQIPtr<IDispatch, &__uuidof(IDispatch)> CComDispatchDriver;
```

Requirements

Header: atlbase.h

CComGlobalsThreadModel

Calls the appropriate thread model methods, regardless of the threading model being used.

```
#if defined(_ATL_SINGLE_THREADED)
typedef CComSingleThreadModel CComGlobalsThreadModel;
#elif defined(_ATL_APARTMENT_THREADED)
typedef CComMultiThreadModel CComGlobalsThreadModel;
#elif defined(_ATL_FREE_THREADED)
typedef CComMultiThreadModel CComGlobalsThreadModel;
#else
#pragma message ("No global threading model defined")
#endif
```

Remarks

Depending on the threading model used by your application, the **typedef** name `CComGlobalsThreadModel` references either `CComSingleThreadModel` or `CComMultiThreadModel`. These classes provide additional `typedef` names to reference a critical section class.

NOTE

`CComGlobalsThreadModel` does not reference class `CComMultiThreadModelNoCS`.

Using `CComGlobalsThreadModel` frees you from specifying a particular threading model class. Regardless of the threading model being used, the appropriate methods will be called.

In addition to `CComGlobalsThreadModel`, ATL provides the **typedef** name `CComObjectThreadModel`. The class referenced by each `typedef` depends on the threading model used, as shown in the following table:

TYPEDF	SINGLE THREADING	APARTMENT THREADING	FREE THREADING
<code>CComObjectThreadModel</code>	S	S	M
<code>CComGlobalsThreadModel</code>	S	M	M

S = `CComSingleThreadModel`; M = `CComMultiThreadModel`

Use `CComObjectThreadModel` within a single object class. Use `CComGlobalsThreadModel` in an object that is globally available to your program, or when you want to protect module resources across multiple threads.

Requirements

Header: `atlbase.h`

CComObjectThreadModel

Calls the appropriate thread model methods, regardless of the threading model being used.

```
#if defined(_ATL_SINGLE_THREADED)
typedef CComSingleThreadModel CComObjectThreadModel;
#elif defined(_ATL_APARTMENT_THREADED)
typedef CComSingleThreadModel CComObjectThreadModel;
#elif defined(_ATL_FREE_THREADED)
typedef CComMultiThreadModel CComObjectThreadModel;
#else
#pragma message ("No global threading model defined")
#endif
```

Remarks

Depending on the threading model used by your application, the `typedef` name `CComObjectThreadModel` references either `CComSingleThreadModel` or `CComMultiThreadModel`. These classes provide additional `typedef` names to reference a critical section class.

NOTE

`CComObjectThreadModel` does not reference class `CComMultiThreadModelNoCS`.

Using `CComObjectThreadModel` frees you from specifying a particular threading model class. Regardless of the threading model being used, the appropriate methods will be called.

In addition to `CComObjectThreadModel`, ATL provides the **typedef** name `CComGlobalsThreadModel`. The class referenced by each **typedef** depends on the threading model used, as shown in the following table:

TYPEDEF	SINGLE THREADING	APARTMENT THREADING	FREE THREADING
<code>CComObjectThreadModel</code>	S	S	M
<code>CComGlobalsThreadModel</code>	S	M	M

S= `CComSingleThreadModel`; M= `CComMultiThreadModel`

Use `CComObjectThreadModel` within a single object class. Use `CComGlobalsThreadModel` in an object that is either globally available to your program, or when you want to protect module resources across multiple threads.

Requirements

Header: `atlbase.h`

CContainedWindow

This class is a specialization of `CContainedWindowT`.

```
typedef CContainedWindowT<CWindow> CContainedWindow;
```

Requirements

Header: atlwin.h

Remarks

`CContainedWindow` is a specialization of [CContainedWindowT](#). If you want to change the base class or traits, use `CContainedWindowT` directly.

CPath

A specialization of [CPathT](#) using `CString`.

```
typedef CPathT<CString> CPath;
```

Requirements

Header: atlpsh.h

CPathA

A specialization of [CPathT](#) using `CStringA`.

```
typedef CPathT<CStringA> CPathA;
```

Requirements

Header: atlpsh.h

CPathW

A specialization of [CPathT](#) using `CStringW`.

```
typedef ATL::CPathT<CStringW> CPathW;
```

Requirements

Header: atlpsh.h

CSimpleValArray

Represents an array for storing simple types.

```
#define CSimpleValArray CSimpleArray
```

Remarks

`CSimpleValArray` is provided for creating and managing arrays containing simple data types. It is a simple #define of [CSimpleArray](#).

Requirements

Header: atlsimpcoll.h

LPCURL

A pointer to a constant [CUrl](#) object.

```
typedef const CUrl* LPCURL;
```

Requirements

Header: atlutil.h

DefaultThreadTraits

The default thread traits class.

Syntax

```
#if defined(_MT)
    typedef CRTThreadTraits DefaultThreadTraits;
#else
    typedef Win32ThreadTraits DefaultThreadTraits;
#endif
```

Remarks

If the current project uses the multithreaded CRT, DefaultThreadTraits is defined as CRTThreadTraits. Otherwise, Win32ThreadTraits is used.

Requirements

Header: atlbase.h

LPURL

A pointer to a [CUrl](#) object.

```
typedef CUrl* LPURL;
```

Requirements

Header: atlutil.h

See also

[ATL COM Desktop Components](#)

[Functions](#)

[Global Variables](#)

[Classes and structs](#)

[Macros](#)

ATL Wizards and Dialog Boxes

3/4/2019 • 2 minutes to read • [Edit Online](#)

The Active Template Library (ATL) wizards generate boilerplate code for various kinds of COM objects. You can run the wizards by opening the shortcut menu for a project in **Solution Explorer** and choosing **Add, Class**.

Related Articles

TITLE	DESCRIPTION
Creating an ATL Project	Describes the ATL project wizard and its settings.
ATL Simple Object	Creates a basic object.
ATL Property Page	Creates a basic property page.
ATL OLE DB Provider	Creates a basic OLE DB provider.
ATL OLE DB Consumer	Creates a basic OLE DB consumer.
ATL Dialog Box	Creates a basic dialog box.
ATL Control	Creates a basic ActiveX control.
ATL COM+ 1.0 Component	Creates a basic ATL COM+ 1.0 component
ATL Active Server Page Component	Creates a basic ATL Active Server Page component.
ATL COM Desktop Components	Links to the ATL documentation.

ATL Active Server Page Component Wizard

5/15/2019 • 3 minutes to read • [Edit Online](#)

This wizard is not available in Visual Studio 2019 and later.

This wizard inserts into the project an Active Server Pages (ASP) component. The Microsoft Internet Information Services (IIS) uses ASP components as part of its enhanced Web page development architecture.

By using this wizard, you can specify the component's threading model and its aggregation support. You can also indicate support for the error information interface, connection points, and free-threaded marshaling.

Remarks

Beginning with Visual Studio 2008, the registration script produced by this wizard registers its COM components under **HKEY_CURRENT_USER** instead of **HKEY_LOCAL_MACHINE**. To modify this behavior, set the **Register component for all users** option of the ATL Wizard.

Names

Specify the names for the object, interface, and classes to be added to your project. Except for **Short name**, all other boxes can be edited independently of the others. If you change the text for **Short name**, the change is reflected in the names of all other boxes in this page.

If you change the **Coclass** name in the COM section, the change is reflected in the **Type** and **ProgID** boxes, but the **Interface** name does not change. This naming behavior is designed to make all the names easily identifiable for you as you develop your control.

C++

Provides information for the C++ class created for the object.

- **Short name**

Sets the root name for the object. The name that you provide determines the **Class** and **Coclass** names, the **.cpp file** and **.h file** names, the **Interface** name, the **Type** names, and the **ProgID**, unless you change those fields individually.

- **.h file**

Sets the name of the header file for the new object's class. By default, this name is based on the name that you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you select an existing file, the wizard will not save it to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

- **Class**

Sets the name of the class to be created. This name is based on the name that you provide in **Short name**, preceded by 'C', the typical prefix for a class name.

- **.cpp file**

Sets the name of the implementation file for the new object's class. By default, this name is based on the name that you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice. The file is not saved to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class implementation should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

- **Attributed**

Indicates whether the object uses attributes. If you are adding an object to an attributed ATL project, this option is selected and not available to change. That is, you can add only attributed objects to a project created with attribute support.

If you select this option for an ATL project that does not have attribute support, the wizard prompts you to specify whether you want to add attribute support to the project.

By default for nonattributed projects, any objects you add after you set this option are designated as attributed (the check box is selected). You can clear this box to add an object that does not use attributes.

See [Application Settings, ATL Project Wizard](#) and [Basic Mechanics of Attributes](#) for more information.

COM

Provides information about the COM functionality for the object.

- **Coclass**

Sets the name of the component class that contains a list of interfaces supported by the object. If your project or this object uses attributes, you cannot change this option because ATL does not include the **coclass** attribute.

- **Type**

Sets the object description that will appear in the registry for the coclass.

- **Interface**

Sets the interface you create for your object. This interface contains your custom methods.

- **ProgID**

Sets the name that containers can use instead of the CLSID of the object.

See also

[ATL Active Server Page Component](#)

ASP, ATL Active Server Page Component Wizard

3/4/2019 • 2 minutes to read • [Edit Online](#)

Use this page of the ATL Active Server Page Component Wizard to specify optional settings for handling information and state related to your ASP component.

- **Optional methods**

Adds the optional ASP methods, **OnStartPage** and **OnEndPage**, to your object. This option must be selected to set any Active Server Pages intrinsic objects. By default, it is selected.

- **OnStartPage/OnEndPage**

[OnStartPage](#) is called the first time the script tries to access the object. **OnEndPage** is called when the object is finished processing the script.

- **Intrinsic object**

You must select the **OnStartPage/OnEndPage** option to set any ASP intrinsic objects.

OPTION	DESCRIPTION
Request	Provides access to the Active Server Pages intrinsic Request object. The Request object is used to pass an HTTP request.
Response	Provides access to the Active Server Pages intrinsic Response object. In response to a request, the Response object sends information to the browser to display to the user.
Session	Provides access to the Active Server Pages intrinsic Session object. The Session object maintains information about the current user session, including storing and retrieving state information.
Application	Provides access to the Active Server Pages intrinsic Application object. The Application object manages state that is shared across multiple ASP objects.
Server	Provides access to the Active Server Pages intrinsic Server object. The Server object allows you to create other ASP objects.

See also

[ATL Active Server Page Component Wizard](#)

[ATL Active Server Page Component](#)

Options, ATL Active Server Page Component Wizard

3/4/2019 • 2 minutes to read • [Edit Online](#)

Use this page of the ATL Active Server Page Component Wizard to design for increased efficiency and error support for the object.

For more information on ATL projects and ATL COM classes, see [ATL COM Desktop Components](#).

- **Threading model**

Indicates the method for managing threads. By default, the project uses **Apartment** threading.

See [Specifying the Project's Threading Model](#) for more information.

OPTION	DESCRIPTION
Single	Specifies that the object uses the single threading model. In the single threading model, an object always runs in the primary COM thread. See Single-Threaded Apartments and InprocServer32 for more information.
Apartment	Specifies that the object uses apartment threading. Equivalent to single thread apartment. Each object of an apartment-threaded component is assigned an apartment for its thread, for the life of the object; however, multiple threads can be used for multiple objects. Each apartment is tied to a specific thread and has a Windows message pump (default). See Single-Threaded Apartments for more information.
Both	Specifies that the object can use either apartment or free threading, depending from which kind of a thread it is created.
Free	Specifies that the object uses free threading. Free threading is equivalent to a multithread apartment model. See Multithreaded Apartments for more information.
Neutral	Specifies that the object follows the guidelines for multithreaded apartments, but it can execute on any kind of thread.

- **Aggregation**

Indicates whether the object uses [aggregation](#). The aggregate object chooses which interfaces to expose to clients, and the interfaces are exposed as if the aggregate object implemented them. Clients of the aggregate object communicate only with the aggregate object.

OPTION	DESCRIPTION
Yes	Specifies that the object can be aggregated. The default.
No	Specifies that the object is not aggregated.

OPTION	DESCRIPTION
Only	Specifies that the object must be aggregated.

- **Support**

Additional support options:

OPTION	DESCRIPTION
ISupportErrorInfo	Creates support for the ISupportErrorInfo interface so the object can return error information to the client.
Connection points	Enables connection points for your object by making your object's class derive from IConnectionPointContainerImpl .
Free-threaded marshaler	Creates a free-threaded marshaler object to marshal interface pointers efficiently between threads in the same process. Available to object specifying either Both or Free as the threading model.

See also

[ATL Active Server Page Component Wizard](#)

[ATL Active Server Page Component](#)

ATL COM+ 1.0 Component Wizard

5/15/2019 • 3 minutes to read • [Edit Online](#)

This wizard is not available in Visual Studio 2019 and later.

Use this wizard to add an object to your project that supports COM+ 1.0 services, including transactions.

You can specify whether the object supports dual interfaces and Automation. You can also indicate support for the error information interface, enhanced object control, transactions, and asynchronous message queuing.

Remarks

Beginning with Visual Studio 2008, the registration script produced by this wizard will register its COM components under **HKEY_CURRENT_USER** instead of **HKEY_LOCAL_MACHINE**. To modify this behavior, set the **Register component for all users** option of the ATL Wizard.

Names

Specify the names for the object, interface, and classes to be added to your project. With the exception of **Short name**, all other boxes can be edited independently of the others. If you change the text for **Short name**, the change is reflected in the names of all other boxes in this page. If you change the **Coclass** name in the COM section, the change is reflected in the **Type** and **ProgID** boxes, but the **Interface** name does not change. This naming behavior is designed to make all the names easily identifiable for you as you develop your control.

- **Short name**

Sets the abbreviated name for the object. The name that you provide determines the **Class** and **Coclass** names, the **.cpp file** and **.h file** names, the **Interface** name, the **Type** names, and the **ProgID**, unless you change those fields individually.

- **.h file**

Sets the name of the header file for the new object's class. By default, this name is based on the name that you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you choose an existing file, the wizard will not save it to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

- **Class**

Sets the name of the class to be created. This name is based on the name you provide in **Short name**, preceded by 'C', the typical prefix for a class name.

- **.cpp file**

Sets the name of the implementation file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice. The file is not saved to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class implementation should be appended to the contents of

the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

- **Attributed**

Indicates whether the object uses attributes. If you are adding an object to an attributed ATL project, this option is selected and not available to change. That is, you can add only attributed objects to a project created with attribute support.

If you select this option for an ATL project that does not have attribute support, the wizard prompts you to specify whether you want to add attribute support to the project.

Any objects you add following setting this option are designated as attributed by default (the check box is selected). You can clear this box to add an object that does not use attributes.

See [Application Settings, ATL Project Wizard](#) and [Basic Mechanics of Attributes](#) for more information.

COM

Provides information about the COM functionality for the object.

- **Coclass**

Sets the name of the component class that contains a list of interfaces supported by the object.

NOTE

If you create your project using attributes, or if you indicate on this wizard page that the COM+ 1.0 component uses attributes, you cannot change this option because ATL does not include the `coclass` attribute.

- **Type**

Sets the object description that will appear in the registry

- **Interface**

Sets the interface you create for your object. This interface contains your custom methods.

- **ProgID**

Sets the name that containers can use instead of the CLSID of the object.

See also

[ATL COM+ 1.0 Component](#)

COM+ 1.0, ATL COM+ 1.0 Component Wizard

5/15/2019 • 2 minutes to read • [Edit Online](#)

This wizard is not available in Visual Studio 2019 and later.

Use this page of the ATL COM+ 1.0 Component Wizard to specify interface type and additional interfaces to be supported.

For more information on ATL projects and ATL COM classes, see [ATL COM Desktop Components](#).

- **Interface**

Indicates the type of interface the object supports. By default, the object supports a dual interface.

OPTION	DESCRIPTION
Dual	Specifies that the object supports a dual interface (its vtable has custom interface functions and late-binding <code>IDispatch</code> methods). Allows both COM clients and Automation controllers to access the object.
Custom	Specifies that the object supports a custom interface (its vtable has custom interface functions). A custom interface can be faster than a dual interface, especially across process boundaries. - Automation compatible Adds automation support to the custom interface. For attributed projects, sets the oleautomation attribute in the coclass.

- **Queueable**

Indicates that clients can call this component asynchronously using message queues. Adds the component attributed macro `custom(TLBATTR_QUEUEABLE, 0)` to the .h file (attributed projects) or to the .idl file (nonattributed projects).

- **Support**

Indicates additional support for error handling and object control.

OPTION	DESCRIPTION
ISupportErrorInfo	Creates support for the ISupportErrorInfo interface so the object can return error information to the client.
IObjectControl	Provides your object access to the three IObjectControl methods: Activate , CanBePooled , and Deactivate .
IObjectConstruct	Creates support for the IObjectConstruct interface to manage passing in parameters from other methods or objects.

- **Transaction**

Indicates that the object supports transactions. Includes the file `mtxattr.h` in the .idl file (nonattributed

projects).

OPTION	DESCRIPTION
Supported	Specifies that the object is never the root of a transaction stream by adding the component attribute macro custom(TLBATTR_TRANS_SUPPORTED,0) to the .h file (attributed projects) or to the .idl file (nonattributed projects).
Required	Specifies that the object may or may not be the root of a transaction stream by adding the component attribute macro custom(TLBATTR_TRANS_REQUIRED,0) to the .h file (attributed projects) or to the .idl file (nonattributed projects).
Not supported	Specifies that the object excludes transactions. Adds the component attribute macro custom(TLBATTR_TRANS_NOTSUPP,0) to the .h file (attributed projects) or to the .idl file (nonattributed projects).
Requires new	Specifies that the object is always the root of a transaction stream by adding the component attribute macro custom(TLBATTR_TRANS_REQNEW,0) to the .h file (attributed projects) or to the .idl file (nonattributed projects).

See also

[ATL COM+ 1.0 Component Wizard](#)

[ATL COM+ 1.0 Component](#)

ATL Control Wizard

3/4/2019 • 4 minutes to read • [Edit Online](#)

Inserts into an ATL project (or an MFC project with ATL support) an ATL control. You can use this wizard to insert one of three kinds of controls:

- Standard control
- Composite control
- DHTML control

Additionally, you can specify a minimal control, removing the interfaces from the [Interfaces](#) list, which are provided as defaults for controls to open in most containers. You can set the interfaces you want supported for the control in the **Interfaces** page of the wizard.

Remarks

The registration script produced by this wizard will register its COM components under HKEY_CURRENT_USER instead of HKEY_LOCAL_MACHINE. To modify this behavior, set the **Register component for all users** option of the ATL Wizard.

Names

Specify the names for the object, interface, and classes to be added to your project. Except for **Short name**, all other boxes can be changed independently. If you change the text for **Short name**, the change is reflected in the names of all other boxes in this page. If you change the **Coclass** name in the COM section, the change is reflected in the **Type** box, but the **Interface** name and **ProgID** do not change. This naming behavior is designed to make all the names easily identifiable for you as you develop your control.

NOTE

Coclass is editable on only nonattributed controls. If your project attributed, you cannot edit **Coclass**.

C++

Provides information for the C++ class created to implement the object.

- **Short name**

Sets the abbreviated name for the object. The name that you provide determines the class and **Coclass** names, the file (.CPP and .H) names, the interface name, and the **Type** names, unless you change those fields individually.

- **Class**

Sets the name of the class that implements the object. This name is based on the name that you provide in **Short name**, preceded by 'C', the typical prefix for a class name.

- **.h file**

Sets the name of the header file for the new object's class. By default, this name is based on the name that you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you select an existing file, the wizard will not save it

to the selected location until you click **Finish**.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

- **.cpp file**

Sets the name of the implementation file for the new object's class. By default, this name is based on the name that you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice. The file is not saved to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class implementation should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

- **Attributed**

Indicates whether the object uses attributes. If you are adding an object to an attributed ATL project, this option is selected and not available to change. That is, you can add only attributed objects to a project created with attribute support.

You can add an attributed object only to an ATL project that uses attributes. If you select this option for an ATL project that does not have attribute support, the wizard prompts you to specify whether you want to add attribute support to the project.

By default, any objects you add after you set this option are designated as attributed (the check box is selected). You can clear this box to add an object that does not use attributes.

See [Application Settings, ATL Project Wizard](#) and [Basic Mechanics of Attributes](#) for more information.

COM

Provides information about the COM functionality for the object.

- **Coclass**

Sets the name of the component class that contains a list of interfaces supported by the object.

NOTE

If you create your project using attributes, or if you indicate on this wizard page that the control uses attributes, you cannot change this option because ATL does not include the **coclass** attribute.

- **Interface**

Sets the name of the interface for the object. By default an interface name is prepended with "I".

- **Type**

Sets the object description that will appear in the registry

- **ProgID**

Sets the name that containers can use instead of the CLSID of the object. This field is not automatically populated. If you do not manually populate this field, the control may not be available to other tools. For example, ActiveX controls that are generated without a `ProgID` are not available in the **Insert ActiveX Control** dialog box. For more information about the dialog box, see [Insert ActiveX Control Dialog Box](#).

See also

ATL Control

Adding Functionality to the Composite Control

Fundamentals of ATL COM Objects

Appearance, ATL Control Wizard

3/4/2019 • 2 minutes to read • [Edit Online](#)

Use this page of the wizard to identify additional user element options for the control. This page is available for controls identified as **Standard controls** under **Control type** on the [Options, ATL Control Wizard](#) page.

UIElement List

- **View status**

Sets the appearance of the control within the container.

- **Opaque:** Sets the VIEWSTATUS_OPAQUE bit in the [VIEWSTATUS](#) enumeration and draws the entire control rectangle passed to the [CComControlBase::OnDraw](#) method. The control appears completely opaque, and none of the container shows behind the control boundaries.

This setting helps the container draw the control more quickly. If this option is not selected, the control can contain transparent parts.

Only an opaque control can have a solid background.

- **Solid Background:** Sets the VIEWSTATUS_SOLIDBKGND bit in the VIEWSTATUS enumeration. The control's background appears as a solid color with no pattern.

This option is available only if the **Opaque** option is also selected.

- **Add control based on**

Sets the control to be based on a Windows control type by adding a [CContainedWindow](#) data member to the class implementing the control. It also adds a message map and message handler functions to handle Windows messages for the control. Choose from the list the type of Windows control you want to create, if any.

- **Button**
- **ListBox**
- **SysAnimate32**
- **SysListView32**
- **ComboBox**
- **RichEdit**
- **SysDateTimePick32**
- **SysMonthCal32**
- **ComboBoxEx32**
- **ScrollBar**
- **SysHeader32**
- **SysTabControl32**
- **Edit**

- **Static**
- **SysIPAddress32**
- **SysTreeView32**

- **Misc status**

Sets additional appearance and behavior options for the control.

- **Invisible at run-time:** Sets the control to be invisible at run time. You can use invisible controls to perform operations in the background, such as firing events at timed intervals.
- **Acts like button:** Sets the `OLEMISC_ACTSLIKEBUTTON` bit in the `OLEMISC` enumeration to enable a control to act like a button. If the container has marked the control's client site as a default button, selecting this option enables your button control to display itself as a default button by drawing itself with a thicker frame. See [CComControlBase::GetAmbientDisplayAsDefault](#) for more information.
- **Acts like label:** Sets the `OLEMISC_ACTSLIKELABEL` bit in the `OLEMISC` enumeration to enable a control to replace the container's native label. The container determines what to do with this flag, if anything.

- **Other**

Sets additional behavior options for the control.

- **Normalized DC:** Sets the control to create a normalized device context when it is called to draw itself. This action standardizes the control's appearance, but it makes drawing less efficient.
- **Window only:** Specifies that your control cannot be windowless. If you do not select this option, your control is automatically windowless in containers that support windowless objects, and it is automatically windowed in containers that do not support windowless objects. Selecting this option forces your control to be windowed, even in containers that support windowless objects.
- **Insertable:** Select this option to have your control appear in the **Insert Object** dialog box of applications such as Word and Excel. Your control can then be inserted by any application that supports embedded objects through this dialog box.

See also

[ATL Control Wizard](#)

[SUBEDIT Sample: Superclasses a Standard Windows Control](#)

Interfaces, ATL Control Wizard

3/4/2019 • 2 minutes to read • [Edit Online](#)

This page of the wizard identifies the interfaces that the control supports. By default, the supported interfaces are those typically used by most containers.

NOTE

If you selected **Minimal control** on the [Options](#) tab, no interfaces appear by default in the **Supported** list box.

- **Not supported**

Indicates the available interfaces that are not currently supported for the control.

- **Supported**

Indicates the interfaces that are currently supported for the control.

TRANSFER BUTTON	DESCRIPTION
>	Adds to the Supported list the interface name currently selected in the Not Supported list.
>>	Adds to the Supported list all interface names available in the Not Supported list.
<	Removes the interface name currently selected in the Supported list.
<<	Removes all interface names currently listed in the Supported list.

See also

[ATL Control Wizard](#)

Options, ATL Control Wizard

3/4/2019 • 2 minutes to read • [Edit Online](#)

Use this page of the wizard to define the type of control you are creating and the level of interface support it contains.

UIElement List

Control type

The kind of control you want to create.

- **Standard control:** An ActiveX control.
- **Composite control:** An ActiveX control that can contain (similar to a dialog box) other ActiveX controls or Windows controls. A composite control includes the following:
 - A template for the dialog box that implements the composite control.
 - A custom resource, REGISTRY, which automatically registers the composite control when invoked.
 - A C++ class that implements the composite control.
 - A COM interface, exposed by the composite control.
 - An HTML test page containing the composite control.

By default, this control sets `CComControlBase::m_bWindowOnly` to true, to indicate that this is a windowed control. It implements a sink map. For more information, see [Support for DHTML Control](#).

- **DHTML control:** An ATL DHTML control specifies the user interface, using HTML. The DHTML UI class contains a COM map. By default, this control sets `CComControlBase::m_bWindowOnly` to true, to indicate that this is a windowed control.

For more information, see [Identifying the Elements of the DHTML Control Project](#).

Minimal control

Supports only the interfaces that are absolutely needed by most containers. You can set **Minimal control** for any of the control types: you can create a minimal standard control, a minimal composite control, or a minimal DHTML control.

Aggregation

Adds aggregation support for the control you are creating. For more information, see [Aggregation](#).

- **Yes:** Create a control that can be aggregated.
- **No:** Create a control that cannot be aggregated.
- **Only:** Create a control that can only be instantiated through aggregation.

Threading model

Specifies that the threading model used by the control.

- **Single:** The control will run only in the primary COM thread.
- **Apartment:** The control can be created in any single thread apartment. The default.

Interface

The type of interface this control exposes to the container.

- **Dual:** Creates an interface that exposes properties and methods through `IDispatch` and directly through the VTBL.
- **Custom:** Creates an interface that exposes methods directly through a VTBL.

If you select **Custom**, then you can specify that the control is **Automation compatible**. If you select **Automation compatible**, then the wizard adds the `oleautomation` attribute to the interface in the IDL, and the interface can be marshaled by the universal marshaler in `oleaut32.dll`. See [Marshaling Details](#) in the Windows SDK for more information.

Additionally, if you select **Automation compatible**, then all parameters for all methods in the control must be VARIANT compatible.

Support

Sets additional miscellaneous support for the control.

- **Connection points:** Enables connection points for your object by making your object's class derive from `IConnectionPointContainerImpl` and allowing it to expose a source interface.
- **Licensed:** Adds support to the control for [licensing](#). Licensed controls can only be hosted if the client machine has the correct license.

See also

[ATL Control Wizard](#)

Stock Properties, ATL Control Wizard

3/4/2019 • 2 minutes to read • [Edit Online](#)

This page of the wizard identifies the stock properties supported for the control. By default, no properties are identified.

- **Not supported**

Indicates the available properties that are not currently supported for the control.

- **Supported**

Indicates the properties that are currently supported for the control.

TRANSFER BUTTON	DESCRIPTION
>	Adds to the Supported list the property name currently selected in the Not Supported list.
>>	Adds to the Supported list all property names available in the Not Supported list.
<	Removes the property name currently selected in the Supported list.
<<	Removes all property names currently listed in the Supported list.

See also

[ATL Control Wizard](#)

ATL Dialog Wizard

3/4/2019 • 2 minutes to read • [Edit Online](#)

This wizard inserts into the project an ATL dialog box object, derived from [CAxDialogImpl](#). A dialog box derived from `CAxDialogImpl` can host ActiveX controls.

The wizard creates a dialog resource with default **OK** and **Cancel** buttons. You can edit the dialog resource and add ActiveX controls using the [Dialog Editor](#) in Resource View.

The wizard inserts into the header file a [message map](#) and declarations for handling the default click events. See [Implementing a Dialog Box](#) for more information about ATL dialog boxes.

- **Short name**

Sets the abbreviated name for the ATL dialog object. The name you provide determines the class name and the file (.cpp and .h) names, unless you change those fields individually.

- **Class**

Sets the name of the class to be created. This name is based on the name you provide in **Short name**, preceded by 'C', the typical prefix for a class name.

- **.h file**

Sets the name of the header file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you choose an existing file, the wizard will not save it to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

- **.cpp file**

Sets the name of the implementation file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice. The file is not saved to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class implementation should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

See also

[ATL Dialog Box](#)

ATL OLE DB Consumer Wizard

5/15/2019 • 6 minutes to read • [Edit Online](#)

This wizard is not available in Visual Studio 2019 and later.

This wizard sets up an OLE DB consumer class with the data bindings necessary to access the specified data source through the specified OLE DB provider.

NOTE

This wizard requires you to click the **Data Source** button to select a data source before entering names in the `Class` and `.h file` fields.

UIElement List

- **Data Source**

The **Data Source** button lets you set up the specified data source using the specified OLE DB provider. When you click this button, the **Data Link Properties** dialog box appears. For more information on building connection strings and the **Data Link Properties** dialog box, see [Data Link API Overview](#) in the Windows SDK documentation.

The following additional information describes the tabs in the **Data Link Properties** dialog box.

- **Provider** tab

Select an appropriate provider to manage the connection to the data source. The type of provider is typically determined by the type of database to which you are connecting. Click the **Next** button or click the **Connection** tab.

- **Connection** tab

The contents of this tab depend on the provider you selected. Although there are many types of providers, this section covers connections for the two most common: SQL and ODBC data. The others are similar variations on the fields described here.

For SQL data:

1. **Select or enter a server name:** Click the drop-down list menu to display all registered data servers on the network, and select one.
2. **Enter information to log on to the server:** Enter a user name and password to log on to the data server.

NOTE

There is a security problem with the "Allow saving of password" feature of the Data Link Properties dialog box. In "Enter information to log on to the server," there are two radio buttons:

- **Use Windows NT integrated security**
- **Use a specific user name and password**

If you select **Use a specific user name and password**, you have the option of saving the password (using the check box for "Allow saving password"); however, this option is not secure. It is recommended that you select **Use Windows NT integrated security**; this option is secure because it encrypts the password. There might be situations in which you want to select "Allow saving password." For example, if you are releasing a library with a private database solution, you should not access the database directly but instead use a middle-tier application to verify the user (through whatever authentication scheme you choose) and then limit the sort of data available to the user.

3. **Select the database on the server:** Click the drop-down list menu to display all registered databases on the data server, and select one.

- or -

Attach a database file as a database name: Specify a file to be used as the database; enter the explicit pathname.

For ODBC data:

1. **Specify the source of data:** You can use a data source name or a connection string.

Use data source name: This drop-down list displays data sources registered in your machine. You can set up data sources ahead of time using the ODBC Data Source Administrator

- or -

Use connection string: Either enter a connection string you have already obtained, or click the **Build** button; the **Select Data Source** dialog box appears. Select a file or machine data source and click **OK**.

NOTE

You can obtain a connection string by viewing the properties of an existing connection in **Server Explorer**, or you can create a connection by double-clicking **Add Connection** in **Server Explorer**.

2. **Enter information to log on to the server:** Enter a user name and password to log on to the data server.
3. Enter the initial catalog to use.
4. Click **Test Connection**; if the test succeeds, click **OK**. If not, check your logon information, try another database, or try another data server.

○ **Advanced** tab

Network settings: Specify the **Impersonation level** (the level of impersonation that the server is allowed to use when impersonating the client; corresponds directly to RPC impersonation levels) and **Protection level** (the level of protection of data sent between client and server; corresponds directly to RPC protection levels).

Other: In **Connect timeout**, specify the number of seconds of idle time allowed before a timeout occurs. In **Access permissions**, specify the access permissions on the data connection.

For more information about advanced initialization properties, refer to the documentation provided with each specific OLE DB provider.

- **All** tab

This tab displays a summary of the initialization properties for the data source and connection you have specified. You can edit these values.

Click **OK** to finish. The **Select Database Object** dialog box appears. From this dialog box, select the table, view, or stored procedure that the consumer will use.

- **Class**

After you select a data source, this box is populated with a default class name based on the table or stored procedure that you selected (see **Select a data source** below). You can edit the class name.

- **.h file**

After you select a data source, this box is populated with a default header class name based on the table or stored procedure that you selected (see **Select a data source** below). You can edit the header file's name or select an existing header file.

- **Attributed**

This option specifies whether the wizard will create consumer classes using attributes or template declarations. When you select this option, the wizard uses attributes instead of template declarations (this is the default option). When you deselect this option, the wizard uses template declarations instead of attributes.

- If you select a consumer **Type of Table**, the wizard uses the `db_source` and `db_table` attributes to create the table and table accessor class declarations, and uses `db_column` to create the column map. For example, it creates this map:

```
```cpp
// Inject table class and table accessor class declarations
[db_source("<initialization_string>"), db_table("dbo.Orders")]
...
// Column map
[db_column(1, status=m_dwOrderIDStatus, length=m_dwOrderIDLength)] LONG m_OrderID;
[db_column(2, status=m_dwCustomerIDStatus, length=m_dwCustomerIDLength)] TCHAR m_CustomerID[6];
...
```
```

instead of using the `CTable` template class to declare the table and table accessor class, and the `BEGIN_COLUMN_MAP` and `END_COLUMN_MAP` macros to create the column map, as in this example:

```

` ``cpp
// Table accessor class
class COrdersAccessor; // Table class
class COrders : public CTable<CAccessor<COrdersAccessor>>;
// ...
// Column map
BEGIN_COLUMN_MAP(COrderDetailsAccessor)
    COLUMN_ENTRY_LENGTH_STATUS(1, m_OrderID, m_dwOrderIDLength, m_dwOrderIDStatus)
    COLUMN_ENTRY_LENGTH_STATUS(2, m_CustomerID, m_dwCustomerIDLength, m_dwCustomerIDStatus)
    // ...
END_COLUMN_MAP()
` ``

```

- If you select a consumer **Type** of **Command**, the wizard uses the `db_source` and `db_command` attributes, and uses `db_column` to create the column map. For example, it creates this map:

```

` ``cpp
[db_source("<initialization_string>"), db_command("SQL_command")]
...
// Column map using db_column is the same as for consumer type of 'table'
` ``

```

instead of using the command and command accessor class declarations in the command class' .h file, for example:

```

` ``cpp
// Command accessor class:
class CListOrdersAccessor;
// Command class:
class CListOrders : public CCommand<CAccessor<CListOrdersAccessor>>;
// ...
// Column map using BEGIN_COLUMN_MAP ... END_COLUMN_MAP is the same as
// for consumer type of 'table'
` ``

```

See [Basic Mechanics of Attributes](#) for more information.

• Type

Select one of these radio buttons to specify whether the consumer class will be derived from `CTable` or `CCommand` (default).

○ Table

Select this option if you want to use `CTable` or `db_table` to create the table and table accessor class declarations.

○ Command

Select this option if you want to use `CCommand` or `db_command` to create the command and command accessor class declarations. This is the default selection.

• Support

Select the check boxes to specify the kinds of updates to be supported in the consumer (the default is none). Each of the following will set `DBPROP_IRowsetChange` and the appropriate entries for `DBPROP_UPDATABILITY` in the property set map.

○ Change

Specifies that the consumer support updates of row data in the rowset.

- **Insert**

Specifies that the consumer support insertion of rows into the rowset.

- **Delete**

Specifies that the consumer support deletion of rows from the rowset.

See also

[ATL OLE DB Consumer](#)

[Adding Functionality with Code Wizards](#)

[Connection Strings and Data Links \(OLE DB\)](#)

ATL OLE DB Provider Wizard

5/15/2019 • 2 minutes to read • [Edit Online](#)

This wizard is not available in Visual Studio 2019 and later.

Remarks

Beginning with Visual Studio 2008, the registration script produced by this wizard will register its COM components under **HKEY_CURRENT_USER** instead of **HKEY_LOCAL_MACHINE**. To modify this behavior, set the **Register component for all users** option of the ATL Wizard.

The following table describes the options for the ATL OLE DB Provider Wizard:

- **Short name**

Type the short name of the provider to be created. The other edit boxes in the wizard will automatically populate based on what you type here. You can edit the other name boxes if you want.

- **Coclass**

The name of the coclass. The ProgID name will change to match this name.

- **Attributed**

This option specifies whether the wizard will create provider classes using attributes or template declarations. When you select this option, the wizard uses attributes instead of template declarations (this is the default option if you created an attributed project). When you clear this option, the wizard uses template declarations instead of attributes (this is the default option if you created a non-attributed project).

If you select this option when you created a non-attributed project, the wizard warns you that the project will be converted to an attributed project and asks you whether to continue or not.

- **ProgID**

The ProgID, or programmatic identifier, is a text string that your application can use instead of a GUID. The ProgID name has the form *Projectname.Coclassname*.

- **Version**

The version number of your provider. The default is 1.

- **DataSource class**

The name of the data source class, of the form *CShortnameSource*.

- **DataSource .h file**

The header file for the data source class. You can edit this file's name or select an existing header file.

- **Session class**

The name of the session class, of the form *CShortnameSession*.

- **Session .h file**

The header file for the session class. You can edit this file's name or select an existing header file.

- **Command class**

The name of the command class, of the form *CShortnameCommand*.

- **Command .h file**

The header file for the command class. This name cannot be edited and depends on the name of the rowset header file.

- **Rowset class**

The name of the rowset class, of the form *CShortnameRowset*.

- **Rowset .h file**

The header file for the rowset class. You can edit this file's name or select an existing header file.

- **Rowset .cpp file**

The provider's implementation file. You can edit this file's name or select an existing implementation file.

See also

[ATL OLE DB Provider](#)

ATL Project Wizard

5/7/2019 • 2 minutes to read • [Edit Online](#)

The Active Template Library (ATL) is a set of template-based C++ classes that simplify writing small and fast COM objects. The ATL Project Wizard creates a project with the structures to contain COM objects.

Overview

This wizard page describes the current [application settings for the ATL project](#) you are creating. By default, the project has the following settings:

- Dynamic-link library Specifies that your server is a DLL and therefore an in-process server.
- Attributed Specifies that your project uses attributes.

To change these defaults, click **Application Settings** in the left column of the wizard and make changes in that page of the ATL Project Wizard.

For information on the default project settings, including the choice of character set, and linking defaults, see [Default ATL Project Configurations](#).

After you create an ATL project, you can add objects or controls to your project using Visual C++ [code wizards](#). You can make the following types of enhancements to a basic ATL project using code wizards:

- [Add object and controls to an ATL project](#)
- [Add a new interface in an ATL project](#)
- [Add a COM+ 1.0 component to an ATL project](#)

Additionally, consider these tasks when you create and enhance an ATL project:

- [Make an ATL object noncreatable](#)
- [Optimize the compiler for an ATL project](#)

You can specify project properties (for example, [whether to link statically to the CRT](#)) in the [Project Properties](#) page, and you can set [build configurations](#) for an ATL project.

See also

[Visual Studio Projects - C++](#)

[C++ project types in Visual Studio](#)

[Fundamentals of ATL COM Objects](#)

[Programming with ATL and C Run-Time Code Tutorial](#)

Application Settings, ATL Project Wizard

3/4/2019 • 2 minutes to read • [Edit Online](#)

Use the **Application Settings** page of the ATL Project Wizard to design and add basic features to a new ATL project.

Server type

Choose from one of three server types:

- **Dynamic-link library (DLL)**

Select to create an in-process server.

- **Executable (EXE)**

Select to create a local out-of-process server. This option does not allow support for MFC or COM+ 1.0. It does not allow for the merging of proxy/stub code.

- **Service (EXE)**

Select to create a Windows application that runs in the background when Windows starts. This option does not allow support for MFC or COM+ 1.0 or does not allow for the merging of proxy/stub code.

Additional options

NOTE

All additional options are available for DLL projects only.

- **Allow merging of proxy/stub code**

Select the **Allow merging of proxy/stub code** check box as a convenience when marshaling interfaces is required. This option puts the MIDL-generated proxy and stub code in the same executable as the server.

- **Support MFC**

Select to specify that your object includes MFC support. This option links your project to the MFC libraries so that you can access any of the classes and functions they contain.

- **Support COM+ 1.0**

Select to modify the project build settings to support COM+ 1.0 components. In addition to the standard list of libraries, the wizard adds the COM+ 1.0 component-specific library comsvcs.lib

In addition, the mtxex.dll is delay loaded on the host system when your application is launched.

- **Support component registrar**

If your ATL project contains support for COM+ 1.0 components, you can set this option. The component registrar allows your COM+ 1.0 object to obtain a list of components, register components, or unregister components (individually or all at once).

See also

ATL Project Wizard

Creating an ATL Project

Default ATL Project Configurations

ATL Property Page Wizard

5/15/2019 • 3 minutes to read • [Edit Online](#)

This wizard is not available in Visual Studio 2019 and later.

This wizard [adds a property page into an ATL project](#) or to an MFC project with ATL support. An ATL property page provides a user interface for setting the properties (or calling the methods) of one or more COM objects.

Remarks

Beginning with Visual Studio 2008, the registration script produced by this wizard will register its COM components under **HKEY_CURRENT_USER** instead of **HKEY_LOCAL_MACHINE**. To modify this behavior, set the **Register component for all users** option of the ATL Wizard.

Names

Specify the names for the object, interface, and classes to be added to your project. Except for **Short name**, all other boxes can be edited independently. If you change the text for **Short name**, the change is reflected in the names of all other boxes in this page. If you change the **Coclass** name in the COM section, the change is reflected in the **Type** and **ProgID** boxes. This naming behavior is designed to make all the names easily identifiable for you as you develop your property page.

NOTE

Coclass is editable on only nonattributed projects. If your project attributed, you cannot edit **Coclass**.

C++

Provides information for the C++ class created to implement the object.

Term	Definition
Short name	Sets the abbreviated name for the object. The name that you provide determines the class and Coclass names, the file (.cpp and .h) names, the Type name, and the ProgID , unless you change those fields individually.
.h file	Sets the name of the header file for the new object's class. By default, this name is based on the name that you provide in Short name. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you select an existing file, the wizard will not save it to the selected location until you click Finish in the wizard. The wizard does not overwrite a file. If you select the name of an existing file, when you click Finish, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click Yes to append the file; click No to return to the wizard and specify another file name.

Class	Sets the name of the class that implements the object. This name is based on the name that you provide in Short name , preceded by 'C', the typical prefix for a class name.
.cpp file	Sets the name of the implementation file for the new object's class. By default, this name is based on the name that you provide in Short name. Click the ellipsis button to save the file name to the location of your choice. The file is not saved to the selected location until you click Finish in the wizard. The wizard does not overwrite a file. If you select the name of an existing file, when you click Finish, the wizard prompts you to indicate whether the class implementation should be appended to the contents of the file. Click Yes to append the file; click No to return to the wizard and specify another file name.
Attributed	Indicates whether the object uses attributes. If you are adding an object to an attributed ATL project, this option is selected and not available to change, that is, you can add only attributed objects to a project created with attribute support. You can add an attributed object only to an ATL project that uses attributes. If you select this option for an ATL project that does not have attribute support, the wizard prompts you to specify whether you want to add attribute support to the project. By default, any objects you add after you set this option are designated as attributed (the check box is selected). You can clear this box to add an object that does not use attributes. See Application Settings, ATL Project Wizard and Basic Mechanics of Attributes for more information.

COM

Provides information about the COM functionality for the object.

- **Coclass**

Sets the name of the component class that contains a list of interfaces supported by the object.

NOTE

If you create your project using attributes, or if you indicate on this wizard page that the property page uses attributes, you cannot change this option because ATL does not include the `coclass` attribute.

- **Type**

Sets the object description that will appear in the registry

- **ProgID**

Sets the name that containers can use instead of the CLSID of the object.

See also

[Options](#), [ATL Property Page Wizard](#)

Strings, ATL Property Page Wizard

Example: Implementing a Property Page

Options, ATL Property Page Wizard

5/15/2019 • 2 minutes to read • [Edit Online](#)

The ATL Property Page wizard is not available in Visual Studio 2019 and later.

Use this page of the wizard to define the threading model and aggregation level of property page you are creating.

- **Threading model**

Specifies the threading model used by the property page.

See [Specifying the Project's Threading Model](#) for more information.

OPTION	DESCRIPTION
Single	The property page runs only in the primary COM thread.
Apartment	The property page can be created in any single thread apartment. The default.

- **Aggregation**

Adds aggregation support for the property page you are creating. See [Aggregation](#) for more information.

OPTION	DESCRIPTION
Yes	Create a property page that can be aggregated.
No	Create a property page that cannot be aggregated.
Only	Create a property page that can only be instantiated through aggregation.

See also

[ATL Property Page Wizard](#)

[Strings, ATL Property Page Wizard](#)

Strings, ATL Property Page Wizard

5/15/2019 • 2 minutes to read • [Edit Online](#)

The ATL Property Page wizard is not available in Visual Studio 2019 and later.

Provides the text associated with the property page.

- **Title**

Sets the text that appears on the tab of the property page.

- **Doc string**

Sets a text string describing the page. This string can be displayed in the property sheet dialog box. The property frame could use the description in a status line or tool tip. The standard property frame currently does not use this string.

- **Help file**

Sets the name of the help file that describes how to use the property page. This name should not include a path. When the user presses **Help**, the frame opens the help file in the directory named in the value of the HelpDir key in the property page registry entries under its CLSID.

See also

[ATL Property Page Wizard](#)

[Options, ATL Property Page Wizard](#)

ATL Simple Object Wizard

3/4/2019 • 4 minutes to read • [Edit Online](#)

This wizard inserts into the project a minimal COM object. Use this page of the wizard to specify the names that identify the C++ class and files for your object and its COM functionality.

Use the [Options](#) page of this wizard to specify the object's threading model, its aggregation support, and whether it supports dual interfaces and Automation. You can also indicate support for the error information interface, connection points, Internet Explorer support, and free-threaded marshaling.

Remarks

Beginning with Visual Studio 2008, the registration script produced by this wizard will register its COM components under **HKEY_CURRENT_USER** instead of **HKEY_LOCAL_MACHINE**. To modify this behavior, set the **Register component for all users** option of the ATL Wizard.

Names

Specify the names for the object, interface, and classes to be added to your project. Except for **Short name**, all other boxes can be edited independently of the others. If you change the text for **Short name**, the change is reflected in the names of all other boxes in this page. If you change the **Coclass** name in the COM section, the change is reflected in the **Type** and **ProgID** boxes, but the **Interface** name does not change. This naming behavior is designed to make all the names easily identifiable for you as you develop your control.

NOTE

Coclass is editable on only nonattributed projects. If your project attributed, you cannot edit **Coclass**.

C++

Provides information for the C++ class created for the object.

- **Short name**

Sets the abbreviated name for the object. The name that you provide determines the `Class` and `Coclass` names, the **.cpp file** and **.h file** names, the **Interface** name, the **Type** names, and the **ProgID**, unless you change those fields individually.

- **.h file**

Sets the name of the header file for the new object's class. By default, this name is based on the name that you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you select an existing file, the wizard will not save it to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

- **Class**

Sets the name of the class to be created. This name is based on the name that you provide in **Short name**,

preceded by 'C', the typical prefix for a class name.

- **.cpp file**

Sets the name of the implementation file for the new object's class. By default, this name is based on the name that you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice. The file is not saved to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class implementation should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

- **Attributed**

Indicates whether the object uses attributes. If you are adding an object to an attributed ATL project, this option is selected and not available to change. That is, you can add only attributed objects to a project created with attribute support.

You can add an attributed object only to an ATL project that uses attributes. If you select this option for an ATL project that does not have attribute support, the wizard prompts you to specify whether you want to add attribute support to the project.

By default, any objects you add after you set this option are designated as attributed (the check box is selected). You can clear this box to add an object that does not use attributes.

See [Application Settings, ATL Project Wizard](#) and [Basic Mechanics of Attributes](#) for more information.

COM

Provides information about the COM functionality for the object.

- **Coclass**

Sets the name of the component class that contains a list of interfaces supported by the object.

NOTE

If you create your project using attributes, or if you indicate on this wizard page that the object uses attributes, you cannot change this option because ATL does not include the `coclass` attribute.

- **Type**

Sets the object description that will appear in the registry

- **Interface**

Sets the interface you create for your object. This interface contains your custom methods.

- **ProgID**

Sets the name that containers can use instead of the CLSID of the object.

See also

[ATL Simple Object](#)

Options, ATL Simple Object Wizard

3/4/2019 • 2 minutes to read • [Edit Online](#)

Use this page of the ATL Simple Object Wizard to design for increased efficiency and error support for the object.

For more information on ATL projects and ATL COM classes, see [ATL COM Desktop Components](#).

- **Threading model**

Indicates the method for managing threads. By default, the project uses **Apartment** threading.

See [Specifying the Project's Threading Model](#) for more information.

OPTION	DESCRIPTION
Single	Specifies that the object always runs in the primary COM thread. See Single-Threaded Apartments and InprocServer32 for more information.
Apartment	Specifies that the object uses apartment threading. Equivalent to single thread apartment. Each object of an apartment-threaded component is assigned an apartment for its thread, for the life of the object; however, multiple threads can be used for multiple objects. Each apartment is tied to a specific thread and has a Windows message pump (default). See Single-Threaded Apartments for more information.
Both	Specifies that the object can use either apartment or free threading, depending from which kind of a thread it is created.
Free	Specifies that the object uses free threading. Free threading is equivalent to a multithread apartment model. See Multithreaded Apartments for more information.
Neutral	Specifies that the object follows the guidelines for multithreaded apartments, but it can execute on any kind of thread.

- **Aggregation**

Indicates whether the object uses [aggregation](#). The aggregate object chooses which interfaces to expose to clients, and the interfaces are exposed as if the aggregate object implemented them. Clients of the aggregate object communicate only with the aggregate object.

OPTION	DESCRIPTION
Yes	Specifies that the object can be aggregated. The default.
No	Specifies that the object is not aggregated.

OPTION	DESCRIPTION
Only	Specifies that the object must be aggregated.

- **Interface**

Indicates the type of interface the object supports. By default, the object supports a dual interface.

OPTION	DESCRIPTION
Dual	Specifies that the object supports a dual interface (its vtable has custom interface functions plus late-binding <code>IDispatch</code> methods). Allows both COM clients and Automation controllers to access the object. The default.
Custom	Specifies that the object supports a custom interface (its vtable has custom interface functions). A custom interface can be faster than a dual interface, especially across process boundaries. - Automation compatible Allows Automation controllers to access an object that has the custom interface support.

- **Support**

Indicates additional support for the object.

OPTION	DESCRIPTION
ISupportErrorInfo	Creates support for the ISupportErrorInfo interface so the object can return error information to the client.
Connection points	Enables connection points for your object by making your object's class derive from IConnectionPointContainerImpl .
Free-threaded marshaler	Creates a free-threaded marshaler object to marshal interface pointers efficiently between threads in the same process. Available to object specifying Both as the threading model.
IObjectWithSite (IE object support)	Implements IObjectWithSiteImpl , which provides a simple way to support communication between an object and its site in a container.

See also

[ATL Simple Object Wizard](#)

[ATL Simple Object](#)

[In-Process Server Threading Issues](#)

Adding a New Interface in an ATL Project

5/7/2019 • 2 minutes to read • [Edit Online](#)

When you add an interface to your object or control, you create stubbed-out functions for each method in that interface. In your object or control, you can add only interfaces currently found in an existing type library. Also, the class in which you add the interface must implement the `BEGIN_COM_MAP` macro or, if the project is attributed, it must have the `coclass` attribute.

You can add a new interface to your control in one of two ways: manually or using code wizards in Class View.

To use code wizards in Class View to add an interface to an existing object or control

1. In [Class View](#), right-click the class name of a control. For example, a full control or composite control, or any other control class that implements a `BEGIN_COM_MAP` macro in its header file.
2. On the shortcut menu, click **Add**, and then click **Implement Interface**.
3. Select the interfaces to implement in the [Implement Interface Wizard](#). If the interface does not exist in any available typelib, then you must add it manually to the .idl file.

To add a new interface manually

1. Add the definition of your new interface to the .idl file.
2. Derive your object or control from the interface.
3. Create a new `COM_INTERFACE_ENTRY` for the interface or, if the project is attributed, add the `coclass` attribute.
4. Implement methods on the interface.

See also

[ATL Project Wizard](#)

[C++ project types in Visual Studio](#)

[Programming with ATL and C Run-Time Code](#)

[Fundamentals of ATL COM Objects](#)

[Default ATL Project Configurations](#)

Adding an ATL Active Server Page Component

5/15/2019 • 2 minutes to read • [Edit Online](#)

The ATL Active Server Pages component wizard is not available in Visual Studio 2019 and later.

To add an Active Template Library (ATL) object to your project, your project must have been created as an ATL COM application or as an MFC application that contains ATL support. You can use the [ATL Project Wizard](#) to create an ATL application, you can select **Add ATL Support to MFC** from the [Add Class Dialog Box](#) dialog box, or you can [add an ATL object to your MFC application](#) to implement ATL support for an MFC application.

Active Server Pages components are part of the Internet Information Services architecture, which provides the following advanced Web development features:

- You can embed ASP components into your HTML pages to create dynamic, browser-independent content.
- You can use ASP pages to provide standards-based database connectivity.
- You can use the ASP error-handling capabilities for your Web-based applications.

To add an ATL Active Server Pages component to your project

1. In **Solution Explorer** right-click the name of the project to which you want to add the ATL Active Server Pages component.
2. From the shortcut menu, click **Add**, and then click **Add Class**.
3. In the [Add Class](#) dialog box, in the **Templates** pane, click **ATL Active Server Page Component**, and then click **Open** to display the [ATL Active Server Page Component Wizard](#).

See also

[Adding a Class](#)

[Adding a New Interface in an ATL Project](#)

[Adding Connection Points to an Object](#)

[Adding a Method](#)

[MFC Class](#)

[Adding a Generic C++ Class](#)

Adding an ATL COM+ 1.0 Component

5/15/2019 • 2 minutes to read • [Edit Online](#)

The ATL COM+ 1.0 Component wizard is not available in Visual Studio 2019 and later.

This wizard adds an object to your project that supports COM+ 1.0 services, including transactions.

To add an ATL COM+ 1.0 component to your project

1. In either **Solution Explorer** or [Class View](#), right-click the name of the project to which you want to add the ATL COM+ 1.0 component.
2. On the shortcut menu, click **Add**, and then click **Add Class**.
3. In the [Add Class](#) dialog box, in the **Templates** pane, click **ATL COM+ 1.0 Component**, and then click **Open** to display the [ATL COM+ 1.0 Component Wizard](#).

See also

[Adding a Class](#)

[Adding a Method](#)

Adding an ATL Control

3/4/2019 • 2 minutes to read • [Edit Online](#)

Use this wizard to add a user interface object to a project that supports interfaces for all potential containers. To support these interfaces, the project must have been created as an ATL application or as an MFC application that contains ATL support. You can use the [ATL Project Wizard](#) to create an ATL application, or [add an ATL object to your MFC application](#) to implement ATL support for an MFC application.

To add an ATL control to your project

1. In either **Solution Explorer** or [Class View](#), right-click the name of the project to which you want to add the ATL simple object.
2. Click **Add** from the shortcut menu, and then click **Add Class**.
3. In the [Add Class](#) dialog box, in the templates pane, click **ATL Control**, and then click **Add** to display the [ATL Control Wizard](#).

Using the **ATL Control Wizard**, you can create one of three types of controls:

- A standard control
- A composite control
- A DHTML control

Additionally, you can reduce the size of the control and remove interfaces that are not used by most containers by selecting **Minimal control** on the **Options** page of the wizard.

See also

[Adding Functionality to the Composite Control](#)

[Fundamentals of ATL COM Objects](#)

Adding an ATL Dialog Box

3/4/2019 • 2 minutes to read • [Edit Online](#)

To add an ATL dialog to your project, your project must be either an ATL project or an MFC project that includes ATL support. You can use the [ATL Project Wizard](#) to create an ATL application, or [add an ATL object to your MFC application](#) to implement ATL support for an MFC application.

By default, the ATL Dialog Wizard implements a dialog box derived from [CAxDialogImpl](#). This class includes support for hosting ActiveX and Windows controls. If you do not want the overhead of ActiveX control support, once the wizard has generated your code, replace all instances of `CAxDialogImpl` with either [CSimpleDialog](#) or [CDialogImpl](#) as your base class.

NOTE

`CSimpleDialog` creates only modal dialog boxes that support only Windows common controls. `CDialogImpl` creates either modal or modeless dialog boxes.

To add an ATL dialog resource to your project

1. Create an ATL project using the [ATL Project Wizard](#).
2. From [Class View](#), right-click the project name and click **Add** from the shortcut menu. Click **Add Class**.
3. In the **Templates** pane of the [Add Class](#) dialog box, click **ATL Dialog**. Click **Open** to display the [ATL Dialog Wizard](#).

For more information, see [Implementing a Dialog Box](#).

See also

[Adding a Class](#)
[Window Classes](#)
[Message Maps](#)

Adding an ATL OLE DB Consumer

5/15/2019 • 2 minutes to read • [Edit Online](#)

The ATL OLE DB Consumer wizard is not available in Visual Studio 2019 and later. You can still add the functionality manually. For more information, see [Creating a Consumer Without Using a Wizard](#).

Use this wizard to add an ATL OLE DB consumer to a project. An ATL OLE DB consumer consists of an OLE DB accessor class and data bindings necessary to access a data source. The project must have been created as an ATL COM application, or as an MFC or Win32 application that contains ATL support (which the ATL OLE DB Consumer Wizard adds automatically).

NOTE

You can add an OLE DB consumer to an MFC project. If you do, the ATL OLE DB Consumer Wizard adds the necessary COM support to your project. This assumes that when you created the MFC project, you selected the **ActiveX controls** check box (in the **Advanced Features** page of the MFC Project Application Wizard), which is checked by default. Selecting this option ensures that the application calls `CoInitialize` and `CoUninitialize`. If you did not select **ActiveX controls** when you created the MFC project, you need to call `CoInitialize` and `CoUninitialize` in your main code.

To add an ATL OLE DB consumer to your project

1. In **Class View**, right-click the project. On the shortcut menu, click **Add** and then click **Add Class**.
2. In the Visual C++ folder, double-click the **ATL OLE DB Consumer** icon or select it and click **Open**.

The ATL OLE DB Consumer Wizard opens.

3. Define settings as described in [ATL OLE DB Consumer Wizard](#).
4. Click **Finish** to close the wizard. The newly created OLE DB consumer code will be inserted in your project.

See also

[Adding Functionality with Code Wizards](#)

Adding an ATL OLE DB Provider

5/15/2019 • 2 minutes to read • [Edit Online](#)

The ATL OLE DB Provider wizard is not available in Visual Studio 2019 and later.

Use this wizard to add an ATL OLE DB provider to a project. An ATL OLE DB provider consists of a data source, session, command, and rowset classes. The project must have been created as an ATL COM application.

To add an ATL OLE DB provider to your project

1. In **Class View**, right-click the project. On the shortcut menu, click **Add** and then click **Add Class**.
2. In the **Visual C++** folder, double-click the **ATL OLE DB Provider** icon or select it and click **Open**.

The ATL OLE DB Provider Wizard opens.

3. Define settings as described in [ATL OLE DB Provider Wizard](#).
4. Click **Finish** to close the wizard, which will insert the newly created OLE DB provider code in your project.

See also

[Adding Functionality with Code Wizards](#)

Adding an ATL Property Page

5/10/2019 • 2 minutes to read • [Edit Online](#)

NOTE

The ATL Property Page wizard is not available in Visual Studio 2019 and later.

To add an Active Template Library (ATL) property page to your project, your project must have been created as an ATL application or as an MFC application that contains ATL support. You can use the [ATL Project Wizard](#) to create an ATL application or [add an ATL object to your MFC application](#) to implement ATL support for an MFC application.

If you are adding a property page for a control, your control must support the [ISpecifyPropertyPagesImpl](#) interface. By default, this interface is in the derivation list of your control class when you [create an ATL control](#) using the [ATL Control Wizard](#).

NOTE

If your control class does not have [ISpecifyPropertyPagesImpl](#) in its derivation list, you must add it manually.

To add an ATL property page to your project

1. In either **Solution Explorer** or [Class View](#), right-click the name of the project to which you want to add the ATL property page.
2. From the shortcut menu, click **Add** and then click **Add Class**.
3. In the [Add Class](#) dialog box, in the **Templates** pane, click **ATL Property Page** and then click **Open** to display the [ATL Property Page Wizard](#).

Once you create a property page for a control, you must provide the [PROP_PAGE](#) entry in the property map for the control.

See also

[Property Pages](#)

[Fundamentals of ATL COM Objects](#)

[Example: Implementing a Property Page](#)

Adding an ATL Simple Object

3/4/2019 • 2 minutes to read • [Edit Online](#)

To add an ATL (Active Template Library) object to your project, your project must have been created as an ATL application or as an MFC application that contains ATL support. You can use the [ATL Project Wizard](#) to create an ATL application, or [add an ATL object to your MFC application](#) to implement ATL support for an MFC application.

You can define COM interfaces for your new ATL object when you first create it, or add them later by using the [Implement Interface](#) command from the **Class View** shortcut menu.

To add an ATL simple object to your ATL COM project

1. In either **Solution Explorer** or [Class View](#), right-click the name of the project to which you want to add the ATL simple object.
2. From the shortcut menu, click **Add**, and then click **Add Class**.
3. In the [Add Class](#) dialog box, in the **Templates** pane, click **ATL Simple Object**, and then click **Open** to display the [ATL Simple Object Wizard](#).
4. Set additional options for your project on the [Options](#) page of the **ATL Simple Object** wizard.
5. Click **Finish** to add the object to your project.

See also

[Adding a Class](#)

[Adding a New Interface in an ATL Project](#)

[Adding Connection Points to an Object](#)

[Adding a Method](#)

[MFC Class](#)

[Adding a Generic C++ Class](#)

Adding Objects and Controls to an ATL Project

5/15/2019 • 2 minutes to read • [Edit Online](#)

NOTE

The ATL COM+ 1.0 Component Wizard, ATL OLE DB Consumer wizard, and ATL Active Server Page Component wizard are not available in Visual Studio 2019 and later.

You can use one of the ATL code wizards to add an object or a control to your ATL- or MFC-based projects. For each COM object or control you add, the wizard generates .cpp and .h files, as well as an .rgs file for script-based registry support. The following ATL code wizards are available in Visual Studio:

ATL Simple Object	ATL Dialog	ATL Control
ATL Property Page	ATL Active Server Page Component	ATL OLE DB Consumer
Add ATL Support to MFC	ATL COM+ 1.0 Component Wizard	ATL OLE DB Provider

NOTE

Before adding an ATL object to your project, you should review the details and requirements for the object in its related Help topics.

To add an object or a control using the ATL Control Wizard

1. In **Solution Explorer**, right-click the project node and click **Add** from the shortcut menu. Click **Add Class**.

The [Add Class](#) dialog box appears.

2. With the **ATL** folder selected in the **Categories** pane, select an object to insert from the **Templates** pane. Click **Open**. The code wizard for the selected object appears.

NOTE

If you want to add an ATL object to an MFC project, you must add ATL support to the existing project. You can do this by following the instructions in [Adding ATL Support to Your MFC Project](#).

Alternately, if you attempt to add an ATL object to your MFC project without previously adding ATL support, Visual Studio prompts you to specify whether you want ATL support added to your project. Click **Yes** to add ATL support to the project and open the selected ATL wizard.

See also

[ATL Project Wizard](#)

[C++ project types in Visual Studio](#)

[Fundamentals of ATL COM Objects](#)

[Programming with ATL and C Run-Time Code](#)

Creating an ATL Project

5/15/2019 • 2 minutes to read • [Edit Online](#)

The easiest way to create an ATL project is to use the ATL Project Wizard, located in the **Win32 Projects** folder of the **New Project** dialog box.

To create an ATL project using the ATL Project Wizard

1. In Visual Studio, choose **File > New > Project** from the main menu.
2. Select the **ATL Project** icon in the **Templates** pane to open the **ATL Project Wizard**.
3. Define your application settings using the [Application Settings](#) page of the **ATL Project Wizard**.

NOTE

Skip this step to keep the wizard default settings.

4. Click **Finish** to close the wizard and open your new project in the development environment.

Once your project is created, you can view the files created in **Solution Explorer**. For more information about the files the wizard creates for your project, see the project-generated file ReadMe.txt. For more information about the file types, see [File Types Created for Visual Studio C++ projects](#). For more information about the configurations for the new ATL project, and how to change them, see [Default ATL Project Configurations](#).

See also

[Adding Functionality with Code Wizards](#)
[Property Pages](#)

COM+ 1.0 Support in ATL Projects

3/4/2019 • 2 minutes to read • [Edit Online](#)

You can use the [ATL Project Wizard](#) to create a project with basic support for COM+ 1.0 components.

COM+ 1.0 is designed for developing distributed component-based applications. It also provides a run-time infrastructure for deploying and managing these applications.

If you select the **Support COM+ 1.0** check box, the wizard modifies the build script in the link step. Specifically, the COM+ 1.0 project links to the following libraries:

- comsvcs.lib
- Mtxguid.lib

If you select the **Support COM+ 1.0** check box, you can also select **Support component registrar**. The component registrar allows your COM+ 1.0 object to get a list of components, register components, or unregister components (individually or all at once).

See also

[Fundamentals of ATL COM Objects](#)

[Programming with ATL and C Run-Time Code](#)

[Default ATL Project Configurations](#)

Default ATL Project Configurations

3/12/2019 • 2 minutes to read • [Edit Online](#)

The ATL Project Wizard creates two project configurations by default:

CONFIGURATION	CHARACTER SET	USE OF ATL
Release	MBCS	DLL
Debug	MBCS	DLL

Character set, **Use of ATL** and can all be changed in the **Project Settings** dialog under the **General** tab. You can also add your own configurations using the Configuration Manager. For details, see [Build Configurations](#).

See also

[Programming with ATL and C Run-Time Code](#)

[Set compiler and build properties](#)

[Configuration Manager Dialog Box](#)

[Compiling and Building](#)

Making an ATL Object Noncreatable

5/7/2019 • 2 minutes to read • [Edit Online](#)

You can change the attributes of an ATL-based COM object so that a client cannot directly create the object. In this case, the object would be returned through a method call on another object rather than created directly.

To make an object noncreatable

1. Remove the [OBJECT_ENTRY_AUTO](#) for the object. If you want the object to be noncreatable but the control to be registered, replace `OBJECT_ENTRY_AUTO` with [OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO](#).
2. Add the [noncreatable](#) attribute to the coclass in the .idl file. For example:

```
[uuid(A1992E3D-3CF0-11D0-826F-00A0C90F2851),  
helpstring("MyObject"),  
noncreatable]  
coclass MyObject  
{  
    [default] interface IMyInterface;  
}
```

See also

[ATL Project Wizard](#)

[C++ project types in Visual Studio](#)

[Programming with ATL and C Run-Time Code](#)

[Fundamentals of ATL COM Objects](#)

[Default ATL Project Configurations](#)

MFC Support in ATL Projects

3/4/2019 • 2 minutes to read • [Edit Online](#)

If you select **Support MFC** in the ATL Project Wizard, your project declares the application as an MFC application object (class). The project initializes the MFC library and instantiates a class (class *ProjName*) that is derived from [CWinApp](#).

This option is available for nonattributed ATL DLL projects only.

```
class CProjNameApp : public CWinApp
{
public:

    // Overrides
    virtual BOOL InitInstance();
    virtual int ExitInstance();
    DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP(CProjNameApp, CWinApp)
END_MESSAGE_MAP()

CProjNameApp theApp;

BOOL CProjNameApp::InitInstance()
{
    return CWinApp::InitInstance();
}

int CProjNameApp::ExitInstance()
{
    return CWinApp::ExitInstance();
}
```

You can view the application object class and its `InitInstance` and `ExitInstance` functions in Class View.

See also

[Adding a Class](#)

[Creating an ATL Project](#)

[Default ATL Project Configurations](#)

Specifying Compiler Optimization for an ATL Project

5/7/2019 • 2 minutes to read • [Edit Online](#)

By default, the [ATL Control Wizard](#) generates new classes with the ATL_NO_VTABLE macro, as follows:

```
class ATL_NO_VTABLE CProjName
{
    ...
};
```

ATL then defines _ATL_NO_VTABLE as follows:

```
#ifndef _ATL_DISABLE_NO_VTABLE
#define ATL_NO_VTABLE
#else
#define ATL_NO_VTABLE __declspec(novtable)
#endif
```

If you do not define _ATL_DISABLE_NO_VTABLE, the ATL_NO_VTABLE macro expands to `declspec(novtable)`. Using `declspec(novtable)` in a class declaration prevents the vtable pointer from being initialized in the class constructor and destructor. When you build your project, the linker eliminates the vtable and all functions to which the vtable points.

You must use ATL_NO_VTABLE, and consequently `declspec(novtable)`, with only base classes that are not directly creatable. You must not use `declspec(novtable)` with the most-derived class in your project, because this class (usually [CCComObject](#), [CCComAggObject](#), or [CCComPolyObject](#)) initializes the vtable pointer for your project.

You must not call virtual functions from the constructor of any object that uses `declspec(novtable)`. You should move those calls to the [FinalConstruct](#) method.

If you are unsure whether you should use the `declspec(novtable)` modifier, you can remove the ATL_NO_VTABLE macro from any class definition, or you can globally disable it by specifying

```
#define _ATL_DISABLE_NO_VTABLE
```

in stdafx.h, before all other ATL header files are included.

See also

[ATL Project Wizard](#)

[C++ project types in Visual Studio](#)

[Programming with ATL and C Run-Time Code](#)

[Fundamentals of ATL COM Objects](#)

[novtable](#)

[Default ATL Project Configurations](#)