Contents

C/C++ Preprocessor Reference

5/7/2019 • 2 minutes to read • Edit Online

The *C/C++ Preprocessor Reference* explains the preprocessor as it is implemented in Microsoft C/C++. The preprocessor performs preliminary operations on C and C++ files before they are passed to the compiler. You can use the preprocessor to conditionally compile code, insert files, specify compile-time error messages, and apply machine-specific rules to sections of code.

In This Section

Preprocessor Directives
Describes directives, typically used to make source programs easy to change and easy to compile in different execution environments.

Preprocessor Operators
Discusses the four preprocessor-specific operators used in the context of the `#define` directive.

Predefined Macros
Discusses predefined macros as specified by ANSI and Microsoft C++.

Pragmas
Discusses pragmas, which offer a way for each compiler to offer machine- and operating system-specific features while retaining overall compatibility with the C and C++ languages.

Related Sections

C++ Language Reference
Provides reference material for the Microsoft implementation of the C++ language.

C Language Reference
Provides reference material for the Microsoft implementation of the C language.

Building a C/C++ Program
Provides links to topics discussing compiler and linker options.

Visual Studio Projects - C++
Describes the user interface in Visual Studio that enables you to specify the directories that the project system will search to locate files for your C++ project.

Preprocessor

4/4/2019 • 2 minutes to read • Edit Online

The preprocessor is a text processor that manipulates the text of a source file as part of the first phase of translation. The preprocessor does not parse the source text, but it does break it up into tokens for the purpose of locating macro calls. Although the compiler ordinarily invokes the preprocessor in its first pass, the preprocessor can also be invoked separately to process text without compiling.

The reference material on the preprocessor includes the following sections:

- Preprocessor directives
- Preprocessor operators
- Predefined macros
- Pragmas

Microsoft Specific

You can obtain a listing of your source code after preprocessing by using the /E or /EP compiler option. Both options invoke the preprocessor and output the resulting text to the standard output device, which, in most cases, is the console. The difference between the two options is that /E includes `#line` directives and /EP strips these directives out.

END Microsoft Specific

Special Terminology

In the preprocessor documentation, the term "argument" refers to the entity that is passed to a function. In some cases, it is modified by "actual" or "formal," which describes the argument expression specified in the function call and the argument declaration specified in the function definition, respectively.

The term "variable" refers to a simple C-type data object. The term "object" refers to both C++ objects and variables; it is an inclusive term.

See also

C/C++ Preprocessor Reference
Phases of Translation

Phases of Translation

4/4/2019 • 2 minutes to read • Edit Online

C and C++ programs consist of one or more source files, each of which contains some of the text of the program. A source file, together with its include files (files that are included using the `#include` preprocessor directive) but not including sections of code removed by conditional-compilation directives such as `#if`, is called a "translation unit."

Source files can be translated at different times — in fact, it is common to translate only out-of-date files. The translated translation units can be processed into separate object files or object-code libraries. These separate, translated translation units are then linked to form an executable program or a dynamic-link library (DLL). For more information about files that can be used as input to the linker, see LINK Input Files.

Translation units can communicate using:

- Calls to functions that have external linkage.
- Calls to class member functions that have external linkage.
- Direct modification of objects that have external linkage.
- Direct modification of files.
- Interprocess communication (for Microsoft Windows-based applications only).

The following list describes the phases in which the compiler translates files:

Character mapping
Characters in the source file are mapped to the internal source representation. Trigraph sequences are converted to single-character internal representation in this phase.

Line splicing
All lines ending in a backslash (\) and immediately followed by a newline character are joined with the next line in the source file forming logical lines from the physical lines. Unless it is empty, a source file must end in a newline character that is not preceded by a backslash.

Tokenization
The source file is broken into preprocessing tokens and white-space characters. Comments in the source file are replaced with one space character each. Newline characters are retained.

Preprocessing
Preprocessing directives are executed and macros are expanded into the source file. The `#include` statement invokes translation starting with the preceding three translation steps on any included text.

Character-set mapping
All source character set members and escape sequences are converted to their equivalents in the execution character set. For Microsoft C and C++, both the source and the execution character sets are ASCII.

String concatenation
All adjacent string and wide-string literals are concatenated. For example, `"String " "concatenation"` becomes `"String concatenation"`.

Translation
All tokens are analyzed syntactically and semantically; these tokens are converted into object code.

Linkage
All external references are resolved to create an executable program or a dynamic-link library.

The compiler issues warnings or errors during phases of translation in which it encounters syntax errors.

The linker resolves all external references and creates an executable program or DLL by combining one or more separately processed translation units along with standard libraries.

See also

Preprocessor

Preprocessor Directives

10/31/2018 • 2 minutes to read • Edit Online

Preprocessor directives, such as `#define` and `#ifdef`, are typically used to make source programs easy to change and easy to compile in different execution environments. Directives in the source file tell the preprocessor to perform specific actions. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, or suppress compilation of part of the file by removing sections of text. Preprocessor lines are recognized and carried out before macro expansion. Therefore, if a macro expands into something that looks like a preprocessor command, that command is not recognized by the preprocessor.

Preprocessor statements use the same character set as source file statements, with the exception that escape sequences are not supported. The character set used in preprocessor statements is the same as the execution character set. The preprocessor also recognizes negative character values.

The preprocessor recognizes the following directives:

#define	#error	#import	#undef
#elif	#if	#include	#using
#else	#ifdef	#line	#endif
#ifndef	#pragma		

The number sign (**#**) must be the first nonwhite-space character on the line containing the directive; white-space characters can appear between the number sign and the first letter of the directive. Some directives include arguments or values. Any text that follows a directive (except an argument or value that is part of the directive) must be preceded by the single-line comment delimiter (**//**) or enclosed in comment delimiters (**/* */**). Lines containing preprocessor directives can be continued by immediately preceding the end-of-line marker with a backslash (\).

Preprocessor directives can appear anywhere in a source file, but they apply only to the remainder of the source file.

See also

Preprocessor Operators
Predefined Macros
C/C++ Preprocessor Reference

#define Directive (C/C++)

4/4/2019 • 4 minutes to read • Edit Online

The **#define** creates a *macro*, which is the association of an identifier or parameterized identifier with a token string. After the macro is defined, the compiler can substitute the token string for each occurrence of the identifier in the source file.

Syntax

`#define` *identifier token-string*$_{opt}$

`#define` *identifier* `(` *identifier*$_{opt}$ `,` ... `,` *identifier*$_{opt}$ `)` *token-string*$_{opt}$

Remarks

The **#define** directive causes the compiler to substitute *token-string* for each occurrence of *identifier* in the source file. The *identifier* is replaced only when it forms a token. That is, *identifier* is not replaced if it appears in a comment, in a string, or as part of a longer identifier. For more information, see Tokens.

The *token-string* argument consists of a series of tokens, such as keywords, constants, or complete statements. One or more white-space characters must separate *token-string* from *identifier*. This white space is not considered part of the substituted text, nor is any white space that follows the last token of the text.

A `#define` without a *token-string* removes occurrences of *identifier* from the source file. The *identifier* remains defined and can be tested by using the `#if defined` and `#ifdef` directives.

The second syntax form defines a function-like macro with parameters. This form accepts an optional list of parameters that must appear in parentheses. After the macro is defined, each subsequent occurrence of *identifier*(*identifier*$_{opt}$, ..., *identifier*$_{opt}$) is replaced with a version of the *token-string* argument that has actual arguments substituted for formal parameters.

Formal parameter names appear in *token-string* to mark the locations where actual values are substituted. Each parameter name can appear multiple times in *token-string*, and the names can appear in any order. The number of arguments in the call must match the number of parameters in the macro definition. Liberal use of parentheses guarantees that complex actual arguments are interpreted correctly.

The formal parameters in the list are separated by commas. Each name in the list must be unique, and the list must be enclosed in parentheses. No spaces can separate *identifier* and the opening parenthesis. Use line concatenation — place a backslash (`\`) immediately before the newline character — for long directives on multiple source lines. The scope of a formal parameter name extends to the new line that ends *token-string*.

When a macro has been defined in the second syntax form, subsequent textual instances followed by an argument list indicate a macro call. The actual arguments that follows an instance of *identifier* in the source file are matched to the corresponding formal parameters in the macro definition. Each formal parameter in *token-string* that is not preceded by a stringizing (`#`), charizing (`#@`), or token-pasting (`##`) operator, or not followed by a `##` operator, is replaced by the corresponding actual argument. Any macros in the actual argument are expanded before the directive replaces the formal parameter. (The operators are described in Preprocessor Operators.)

The following examples of macros with arguments illustrate the second form of the **#define** syntax:

```
// Macro to define cursor lines
#define CURSOR(top, bottom) (((top) << 8) | (bottom))

// Macro to get a random integer with a specified range
#define getrandom(min, max) \
    ((rand()%(int)(((max) + 1)-(min)))+ (min))
```

Arguments with side effects sometimes cause macros to produce unexpected results. A given formal parameter may appear more than one time in *token-string*. If that formal parameter is replaced by an expression with side effects, the expression, with its side effects, may be evaluated more than one time. (See the examples under Token-Pasting Operator (##).)

The `#undef` directive causes an identifier's preprocessor definition to be forgotten. See The #undef Directive for more information.

If the name of the macro being defined occurs in *token-string* (even as a result of another macro expansion), it is not expanded.

A second **#define** for a macro with the same name generates a warning unless the second token sequence is identical to the first.

Microsoft Specific

Microsoft C/C++ lets you redefine a macro if the new definition is syntactically identical to the original definition. In other words, the two definitions can have different parameter names. This behavior differs from ANSI C, which requires that the two definitions be lexically identical.

For example, the following two macros are identical except for the parameter names. ANSI C does not allow such a redefinition, but Microsoft C/C++ compiles it without error.

```
#define multiply( f1, f2 ) ( f1 * f2 )
#define multiply( a1, a2 ) ( a1 * a2 )
```

On the other hand, the following two macros are not identical and will generate a warning in Microsoft C/C++.

```
#define multiply( f1, f2 ) ( f1 * f2 )
#define multiply( a1, a2 ) ( b1 * b2 )
```

END Microsoft Specific

This example illustrates the **#define** directive:

```
#define WIDTH       80
#define LENGTH      ( WIDTH + 10 )
```

The first statement defines the identifier `WIDTH` as the integer constant 80 and defines `LENGTH` in terms of `WIDTH` and the integer constant 10. Each occurrence of `LENGTH` is replaced by (`WIDTH + 10`). In turn, each occurrence of `WIDTH + 10` is replaced by the expression (`80 + 10`). The parentheses around `WIDTH + 10` are important because they control the interpretation in statements such as the following:

```
var = LENGTH * 20;
```

After the preprocessing stage the statement becomes:

```
var = ( 80 + 10 ) * 20;
```

which evaluates to 1800. Without parentheses, the result is:

```
var = 80 + 10 * 20;
```

which evaluates to 280.

Microsoft Specific

Defining macros and constants with the /D compiler option has the same effect as using a **#define** preprocessing directive at the start of your file. Up to 30 macros can be defined by using the /D option.

END Microsoft Specific

See also

Preprocessor Directives

#error Directive (C/C++)

4/4/2019 • 2 minutes to read • Edit Online

The **#error** directive emits a user-specified error message at compile time and then terminates the compilation.

Syntax

```
#errortoken-string
```

Remarks

The error message that this directive emits includes the *token-string* parameter. The *token-string* parameter is not subject to macro expansion. This directive is most useful during preprocessing for notifying the developer of a program inconsistency or the violation of a constraint. The following example demonstrates error processing during preprocessing:

```
#if !defined(__cplusplus)
#error C++ compiler required.
#endif
```

See also

Preprocessor Directives

#if, #elif, #else, and #endif Directives (C/C++)

4/4/2019 • 5 minutes to read • Edit Online

The **#if** directive, with the **#elif**, **#else**, and **#endif** directives, controls compilation of portions of a source file. If the expression you write (after the **#if**) has a nonzero value, the line group immediately following the **#if** directive is retained in the translation unit.

Grammar

conditional :
  if-part elif-parts$_{opt}$ *else-part*$_{opt}$ *endif-line*

if-part :
  if-line text

if-line :
  #if *constant-expression*
  #ifdef *identifier*
  #ifndef *identifier*

elif-parts :
  elif-line text
  elif-parts elif-line text

elif-line :
  #elif *constant-expression*

else-part :
  else-line text

else-line :
  #else

endif-line :
  #endif

Each **#if** directive in a source file must be matched by a closing **#endif** directive. Any number of **#elif** directives can appear between the **#if** and **#endif** directives, but at most one **#else** directive is allowed. The **#else** directive, if present, must be the last directive before **#endif**.

The **#if**, **#elif**, **#else**, and **#endif** directives can nest in the text portions of other **#if** directives. Each nested **#else**, **#elif**, or **#endif** directive belongs to the closest preceding **#if** directive.

All conditional-compilation directives, such as **#if** and **#ifdef**, must be matched with closing **#endif** directives prior to the end of file; otherwise, an error message is generated. When conditional-compilation directives are contained in include files, they must satisfy the same conditions: There must be no unmatched conditional-compilation directives at the end of the include file.

Macro replacement is performed within the part of the command line that follows an **#elif** command, so a macro call can be used in the *constant-expression*.

The preprocessor selects one of the given occurrences of *text* for further processing. A block specified in *text* can be any sequence of text. It can occupy more than one line. Usually *text* is program text that has meaning to the compiler or the preprocessor.

The preprocessor processes the selected *text* and passes it to the compiler. If *text* contains preprocessor directives, the preprocessor carries out those directives. Only text blocks selected by the preprocessor are compiled.

The preprocessor selects a single *text* item by evaluating the constant expression following each **#if** or **#elif** directive until it finds a true (nonzero) constant expression. It selects all text (including other preprocessor directives beginning with **#**) up to its associated **#elif**, **#else**, or **#endif**.

If all occurrences of *constant-expression* are false, or if no **#elif** directives appear, the preprocessor selects the text block after the **#else** clause. If the **#else** clause is omitted and all instances of *constant-expression* in the **#if** block are false, no text block is selected.

The *constant-expression* is an integer constant expression with these additional restrictions:

- Expressions must have integral type and can include only integer constants, character constants, and the **defined** operator.
- The expression cannot use `sizeof` or a type-cast operator.
- The target environment may not be able to represent all ranges of integers.
- The translation represents type **int** the same as type **long**, and **unsigned int** the same as **unsigned long**.
- The translator can translate character constants to a set of code values different from the set for the target environment. To determine the properties of the target environment, check values of macros from LIMITS.H in an application built for the target environment.
- The expression must not perform any environmental inquiries and must remain insulated from implementation details on the target computer.

defined

The preprocessor operator **defined** can be used in special constant expressions, as shown by the following syntax:

defined(`identifier`)

defined `identifier`

This constant expression is considered true (nonzero) if the *identifier* is currently defined; otherwise, the condition is false (0). An identifier defined as empty text is considered defined. The **defined** directive can be used in an **#if** and an **#elif** directive, but nowhere else.

In the following example, the **#if** and **#endif** directives control compilation of one of three function calls:

```
#if defined(CREDIT)
    credit();
#elif defined(DEBIT)
    debit();
#else
    printerror();
#endif
```

The function call to `credit` is compiled if the identifier `CREDIT` is defined. If the identifier `DEBIT` is defined, the function call to `debit` is compiled. If neither identifier is defined, the call to `printerror` is compiled. Note that `CREDIT` and `credit` are distinct identifiers in C and C++ because their cases are different.

The conditional compilation statements in the following example assume a previously defined symbolic constant named `DLEVEL`.

```
#if DLEVEL > 5
    #define SIGNAL  1
    #if STACKUSE == 1
        #define STACK   200
    #else
        #define STACK   100
    #endif
#else
    #define SIGNAL  0
    #if STACKUSE == 1
        #define STACK   100
    #else
        #define STACK   50
    #endif
#endif
#if DLEVEL == 0
    #define STACK 0
#elif DLEVEL == 1
    #define STACK 100
#elif DLEVEL > 5
    display( debugptr );
#else
    #define STACK 200
#endif
```

The first **#if** block shows two sets of nested **#if**, **#else**, and **#endif** directives. The first set of directives is processed only if `DLEVEL > 5` is true. Otherwise, the statements after **#else** are processed.

The **#elif** and **#else** directives in the second example are used to make one of four choices, based on the value of `DLEVEL`. The constant `STACK` is set to 0, 100, or 200, depending on the definition of `DLEVEL`. If `DLEVEL` is greater than 5, then the statement

```
#elif DLEVEL > 5
display(debugptr);
```

is compiled and `STACK` is not defined.

A common use for conditional compilation is to prevent multiple inclusions of the same header file. In C++, where classes are often defined in header files, constructs like the following can be used to prevent multiple definitions:

```
/*  EXAMPLE.H - Example header file  */
#if !defined( EXAMPLE_H )
#define EXAMPLE_H

class Example
{
...
};

#endif // !defined( EXAMPLE_H )
```

The preceding code checks to see if the symbolic constant `EXAMPLE_H` is defined. If so, the file has already been included and need not be reprocessed. If not, the constant `EXAMPLE_H` is defined to mark EXAMPLE.H as already processed.

__has_include

Visual Studio 2017 version 15.3 and later: Determines whether a library header is available for inclusion:

```
#ifdef __has_include
#  if __has_include(<filesystem>)
#    include <filesystem>
#    define have_filesystem 1
#  elif __has_include(<experimental/filesystem>)
#    include <experimental/filesystem>
#    define have_filesystem 1
#    define experimental_filesystem
#  else
#    define have_filesystem 0
#  endif
#endif
```

See also

Preprocessor Directives

#ifdef and #ifndef Directives (C/C++)

4/4/2019 • 2 minutes to read • Edit Online

The **#ifdef** and **#ifndef** directives perform the same task as the `#if` directive when it is used with **defined**(*identifier*).

Syntax

```
#ifdef identifier
#ifndef identifier

// equivalent to
#if defined identifier
#if !defined identifier
```

Remarks

You can use the **#ifdef** and **#ifndef** directives anywhere `#if` can be used. The **#ifdef** *identifier* statement is equivalent to `#if 1` when *identifier* has been defined, and it is equivalent to `#if 0` when *identifier* has not been defined or has been undefined with the `#undef` directive. These directives check only for the presence or absence of identifiers defined with `#define`, not for identifiers declared in the C or C++ source code.

These directives are provided only for compatibility with previous versions of the language. The **defined(** *identifier* **)** constant expression used with the `#if` directive is preferred.

The **#ifndef** directive checks for the opposite of the condition checked by **#ifdef**. If the identifier has not been defined (or its definition has been removed with `#undef`), the condition is true (nonzero). Otherwise, the condition is false (0).

Microsoft Specific

The *identifier* can be passed from the command line using the `/D` option. Up to 30 macros can be specified with `/D`.

This is useful for checking whether a definition exists, because a definition can be passed from the command line. For example:

```
// ifdef_ifndef.CPP
// compile with: /Dtest /c
#ifndef test
#define final
#endif
```

END Microsoft Specific

See also

Preprocessor Directives

#import Directive (C++)

4/4/2019 • 6 minutes to read • Edit Online

C++ Specific

Used to incorporate information from a type library. The content of the type library is converted into C++ classes, mostly describing the COM interfaces.

Syntax

```
#import "filename" [attributes]
#import <filename> [attributes]
```

Parameters

filename
Specifies the type library to import. *filename* can be one of the following:

- The name of a file that contains a type library, such as an .olb, .tlb, or .dll file. The keyword, **file:**, can precede each filename.
- The progid of a control in the type library. The keyword, **progid:**, can precede each progid. For example:

  ```
  #import "progid:my.prog.id.1.5"
  ```

  For more on progids, see Specifying the Localization ID and Version Number.

  Note that when compiling with a cross compiler on a 64-bit operating system, the compiler will be able to read only the 32-bit registry hive. You might want to use the native 64-bit compiler to build and register a 64-bit type library.

- The library ID of the type library. The keyword, **libid:**, can precede each library ID. For example:

  ```
  #import "libid:12341234-1234-1234-1234-123412341234" version("4.0") lcid("9")
  ```

  If you do not specify version or lcid, the rules that are applied to **progid:** are also applied to **libid:**.

- An executable (.exe) file.
- A library (.dll) file containing a type library resource (such as .ocx).
- A compound document holding a type library.
- Any other file format that can be understood by the **LoadTypeLib** API.

attributes
One or more #import attributes. Separate attributes with either a space or comma. For example:

```
#import "..\drawctl\drawctl.tlb" no_namespace, raw_interfaces_only
```

-or-

```
#import "..\drawctl\drawctl.tlb" no_namespace raw_interfaces_only
```

Remarks

Search Order for filename

filename is optionally preceded by a directory specification. The file name must name an existing file. The difference between the two syntax forms is the order in which the preprocessor searches for the type library files when the path is incompletely specified.

SYNTAX FORM	ACTION
Quoted form	Instructs the preprocessor to look for type library files first in the directory of the file that contains the **#import** statement, and then in the directories of whatever files that include (`#include`) that file. The preprocessor then searches along the paths shown below.
Angle-bracket form	Instructs the preprocessor to search for type library files along the following paths: 1. The `PATH` environment variable path list 2. The `LIB` environment variable path list 3. The path specified by the /I (additional include directories) compiler option, except it the compiler is searching for a type library that was referenced from another type library with the no_registry attribute.

Specifying the Localization ID and Version Number

When you specify a progid, you can also specify the localization ID and version number of the progid. For example:

```
#import "progid:my.prog.id" lcid("0") version("4.0)
```

If you do not specify a localization ID, a progid is chosen according to the following rules:

- If there is only one localization ID, that one is used.
- If there is more than one localization ID, the first one with version number 0, 9, or 409 is used.
- If there is more than one localization ID and none of them are 0, 9, or 409, the last one is used.
- If you do not specify a version number, the most recent version is used.

Header Files Created by Import

#import creates two header files that reconstruct the type library contents in C++ source code. The primary header file is similar to that produced by the Microsoft Interface Definition Language (MIDL) compiler, but with additional compiler-generated code and data. The primary header file has the same base name as the type library, plus a .TLH extension. The secondary header file has the same base name as the type library, with a .TLI extension. It contains the implementations for compiler-generated member functions, and is included (`#include`) in the primary header file.

If importing a dispinterface property that uses byref parameters, #import will not generate __declspec(property) statement for the function.

Both header files are placed in the output directory specified by the /Fo (name object file) option. They are then read and compiled by the compiler as if the primary header file was named by a `#include` directive.

The following compiler optimizations come with the **#import** directive:

- The header file, when created, is given the same timestamp as the type library.
- When **#import** is processed, the compiler first checks if the header exists and is up to date. If yes, then it does not need to be re-created.

The **#import** directive also participates in minimal rebuild and can be placed in a precompiled header file. See Creating Precompiled Header Files for more information.

Primary Type Library Header File

The primary type library header file consists of seven sections:

- Heading boilerplate: Consists of comments, `#include` statement for COMDEF.H (which defines some standard macros used in the header), and other miscellaneous setup information.
- Forward references and typedefs: Consists of structure declarations such as `struct IMyInterface` and typedefs.
- Smart pointer declarations: The template class `_com_ptr_t` is a smart-pointer implementation that encapsulates interface pointers and eliminates the need to call `AddRef`, `Release`, `QueryInterface` functions. In addition, it hides the `CoCreateInstance` call in creating a new COM object. This section uses macro statement `_COM_SMARTPTR_TYPEDEF` to establish typedefs of COM interfaces to be template specializations of the _com_ptr_t template class. For example, for interface `IMyInterface`, the .TLH file will contain:

  ```
  _COM_SMARTPTR_TYPEDEF(IMyInterface, __uuidof(IMyInterface));
  ```

  which the compiler will expand to:

  ```
  typedef _com_ptr_t<_com_IIID<IMyInterface, __uuidof(IMyInterface)> > IMyInterfacePtr;
  ```

  Type `IMyInterfacePtr` can then be used in place of the raw interface pointer `IMyInterface*`. Consequently, there is no need to call the various `IUnknown` member functions
- Typeinfo declarations: Primarily consists of class definitions and other items exposing the individual typeinfo items returned by `ITypeLib:GetTypeInfo`. In this section, each typeinfo from the type library is reflected in the header in a form dependent on the `TYPEKIND` information.
- Optional old-style GUID definition: Contains initializations of the named GUID constants. These are names of the form `CLSID_CoClass` and `IID_Interface`, similar to those generated by the MIDL compiler.
- `#include` statement for the secondary type library header.
- Footer boilerplate: Currently includes `#pragma pack(pop)`.

All sections, except the heading boilerplate and footer boilerplate section, are enclosed in a namespace with its name specified by the `library` statement in the original IDL file. You can use the names from the type library header either by an explicit qualification with the namespace name or by including the following statement:

```
using namespace MyLib;
```

immediately after the **#import** statement in the source code.

The namespace can be suppressed by using the no_namespace) attribute of the **#import** directive. However, suppressing the namespace may lead to name collisions. The namespace can also be renamed by the rename_namespace attribute.

The compiler provides the full path to any type library dependency required by the type library it is currently processing. The path is written, in the form of comments, into the type library header (.TLH) that the compiler generates for each processed type library.

If a type library includes references to types defined in other type libraries, then the .TLH file will include comments of the following sort:

```
//
// Cross-referenced type libraries:
//
//  #import "c:\path\typelib0.tlb"
//
```

The actual filename in the **#import** comment is the full path of the cross-referenced type library, as stored in the registry. If you encounter errors that are due to missing type definitions, check the comments at the head of the .TLH to see which dependent type libraries may need to be imported first. Likely errors are syntax errors (for example, C2143, C2146, C2321), C2501 (missing decl-specifiers), or C2433 ('inline' not permitted on data declaration) while compiling the .TLI file.

You must determine which of the dependency comments are not otherwise provided for by system headers and then provide an **#import** directive at some point before the **#import** directive of the dependent type library to resolve the errors.

#import Attributes

#import can optionally include one or more attributes. These attributes tell the compiler to modify the contents of the type-library headers. A backslash (****) symbol can be used to include additional lines in a single **#import** statement. For example:

```
#import "test.lib" no_namespace \
   rename("OldName", "NewName")
```

For more information, see #import Attributes.

END C++ Specific

See also

Preprocessor Directives
Compiler COM Support

#import Attributes (C++)

4/4/2019 • 2 minutes to read • Edit Online

Provides links to attributes used with the `#import` directive.

Microsoft Specific

The following attributes are available to the `#import` directive.

ATTRIBUTE	DESCRIPTION
auto_rename	Renames C++ reserved words by appending two underscores (__) to the variable name to resolve potential name conflicts.
auto_search	Specifies that, when a type library is referenced with #import and itself references another type library, the compiler can do an implicit #import for the other type library.
embedded_idl	Specifies that the type library is written to the .tlh file with the attribute-generated code preserved.
exclude	Excludes items from the type library header files being generated.
high_method_prefix	Specifies a prefix to be used in naming high-level properties and methods.
high_property_prefixes	Specifies alternate prefixes for three property methods.
implementation_only	Suppresses the generation of the .tlh header file (the primary header file).
include()	Disables automatic exclusion.
inject_statement	Inserts its argument as source text into the type-library header.
named_guids	Tells the compiler to define and initialize GUID variables in old style, of the form `LIBID_MyLib`, `CLSID_MyCoClass`, `IID_MyInterface`, and `DIID_MyDispInterface`.
no_auto_exclude	Disables automatic exclusion.
no_dual_interfaces	Changes the way the compiler generates wrapper functions for dual interface methods.
no_implementation	Suppresses the generation of the .tli header, which contains the implementations of the wrapper member functions.
no_namespace	Specifies that the namespace name is not generated by the compiler.

ATTRIBUTE	DESCRIPTION
no_registry	Tells the compiler not to search the registry for type libraries.
no_search_namespace	Has the same functionality as the no_namespace attribute but is used on type libraries that you use the #import directive with the auto_search attribute.
no_smart_pointers	Suppresses the creation of smart pointers for all interfaces in the type library.
raw_dispinterfaces	Tells the compiler to generate low-level wrapper functions for dispinterface methods and properties that call `IDispatch::Invoke` and return the HRESULT error code.
raw_interfaces_only	Suppresses the generation of error-handling wrapper functions and property declarations that use those wrapper functions.
raw_method_prefix	Specifies a different prefix to avoid name collisions.
raw_native_types	Disables the use of COM support classes in the high-level wrapper functions and forces the use of low-level data types instead.
raw_property_prefixes	Specifies alternate prefixes for three property methods.
rename	Works around name collision problems.
rename_namespace	Renames the namespace that contains the contents of the type library.
rename_search_namespace	Has the same functionality as the rename_namespace attribute but is used on type libraries that you use the #import directive with the auto_search attribute.
tlbid	Allows for loading libraries other than the primary type library.

END Microsoft Specific

See also

#import Directive

auto_rename

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Renames C++ reserved words by appending two underscores (__) to the variable name to resolve potential name conflicts.

Syntax

```
auto_rename
```

Remarks

This attribute is used when importing a type library that uses one or more C++ reserved words (keywords or macros) as variable names.

END C++ Specific

See also

#import Attributes
#import Directive

auto_search

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Specifies that, when a type library is referenced with `#import` and itself references another type library, the compiler can do an implicit `#import` for the other type library.

Syntax

```
auto_search
```

Remarks

END C++ Specific

See also

#import Attributes
#import Directive

embedded_idl

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Specifies that the type library is written to the .tlh file with the attribute-generated code preserved.

Syntax

```
embedded_idl[("param")]
```

Parameters

param
Can be one of two values:

- **emitidl**: Type information imported from the typelib will be present in the IDL generated for the attributed project. This is the default and will be in effect if you do not specify a parameter to `embedded_idl`.
- **no_emitidl**: Type information imported from the typelib will not be present in the IDL generated for the attributed project.

Example

```
// import_embedded_idl.cpp
// compile with: /LD
#include <windows.h>
[module(name="MyLib2")];
#import "\school\bin\importlib.tlb" embedded_idl("no_emitidl")
```

Remarks

END C++ Specific

See also

#import Attributes
#import Directive

exclude (#import)

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Excludes items from the type library header files being generated.

Syntax

```
exclude("Name1"[, "Name2",...])
```

Parameters

Name1
First item to be excluded.

Name2
Second item to be excluded (if necessary).

Remarks

Type libraries may include definitions of items defined in system headers or other type libraries. This attribute can take any number of arguments, each being a top-level type library item to be excluded.

END C++ Specific

See also

#import Attributes
#import Directive

high_method_prefix

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Specifies a prefix to be used in naming high-level properties and methods.

Syntax

```
high_method_prefix("Prefix")
```

Parameters

Prefix
Prefix to be used.

Remarks

By default, high-level error-handling properties and methods are exposed by member functions named without a prefix. The names are from the type library.

END C++ Specific

See also

#import Attributes
#import Directive

high_property_prefixes

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Specifies alternate prefixes for three property methods.

Syntax

```
high_property_prefixes("GetPrefix","PutPrefix","PutRefPrefix")
```

Parameters

GetPrefix
Prefix to be used for the `propget` methods.

PutPrefix
Prefix to be used for the `propput` methods.

PutRefPrefix
Prefix to be used for the `propputref` methods.

Remarks

By default, high-level error-handling `propget`, `propput`, and `propputref` methods are exposed by member functions named with prefixes `Get`, `Put`, and `PutRef`, respectively.

END C++ Specific

See also

#import Attributes
#import Directive

implementation_only

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Suppresses the generation of the .tlh header file (the primary header file).

Syntax

```
implementation_only
```

Remarks

This file contains all the declarations used to expose the type-library contents. The .tli header file, with the implementations of the wrapper member functions, will be generated and included in the compilation.

When this attribute is specified, the content of the .tli header is in the same namespace as the one normally used in the .tlh header. In addition, the member functions are not declared as inline.

The **implementation_only** attribute is intended for use in conjunction with the no_implementation attribute as a way of keeping the implementations out of the precompiled header (PCH) file. An `#import` statement with the `no_implementation` attribute is placed in the source region used to create the PCH. The resulting PCH is used by a number of source files. An `#import` statement with the **implementation_only** attribute is then used outside the PCH region. You are required to use this statement only once in one of the source files. This will generate all the required wrapper member functions without additional recompilation for each source file.

> **NOTE**
>
> The **implementation_only** attribute in one `#import` statement must be use in conjunction with another `#import` statement, of the same type library, with the `no_implementation` attribute. Otherwise, compiler errors will be generated. This is because wrapper class definitions generated by the `#import` statement with the `no_implementation` attribute are required to compile the implementations generated by the **implementation_only** attribute.

END C++ Specific

See also

#import Attributes
#import Directive

include()

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Disables automatic exclusion.

Syntax

```
include("Name1"[,"Name2", ...])
```

Parameters

Name1
First item to be forcibly included.

Name2
Second item to be forcibly included (if necessary).

Remarks

Type libraries may include definitions of items defined in system headers or other type libraries. `#import` attempts to avoid multiple definition errors by automatically excluding such items. If items have been excluded, as indicated by Compiler Warning (level 3) C4192, and they should not have been, this attribute can be used to disable the automatic exclusion. This attribute can take any number of arguments, each being the name of the type-library item to be included.

END C++ Specific

See also

#import Attributes
#import Directive

inject_statement

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Inserts its argument as source text into the type-library header.

Syntax

```
inject_statement("source_text")
```

Parameters

source_text
Source text to be inserted into the type library header file.

Remarks

The text is placed at the beginning of the namespace declaration that wraps the type-library contents in the header file.

END C++ Specific

See also

#import Attributes
#import Directive

named_guids

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Tells the compiler to define and initialize GUID variables in old style, of the form `LIBID_MyLib`, `CLSID_MyCoClass`, `IID_MyInterface`, and `DIID_MyDispInterface`.

Syntax

```
named_guids
```

Remarks

END C++ Specific

See also

#import Attributes
#import Directive

no_auto_exclude

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Disables automatic exclusion.

Syntax

```
no_auto_exclude
```

Remarks

Type libraries may include definitions of items defined in system headers or other type libraries. `#import` attempts to avoid multiple definition errors by automatically excluding such items. When this is done, Compiler Warning (level 3) C4192 will be issued for each item to be excluded. You can disable this automatic exclusion by using this attribute.

END C++ Specific

See also

#import Attributes
#import Directive

no_dual_interfaces

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Changes the way the compiler generates wrapper functions for dual interface methods.

Syntax

```
no_dual_interfaces
```

Remarks

Normally, the wrapper will call the method through the virtual function table for the interface. With **no_dual_interfaces**, the wrapper instead calls `IDispatch::Invoke` to invoke the method.

END C++ Specific

See also

#import Attributes
#import Directive

no_implementation

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Suppresses the generation of the .tli header, which contains the implementations of the wrapper member functions.

Syntax

```
no_implementation
```

Remarks

If this attribute is specified, the .tlh header, with the declarations to expose type-library items, will be generated without an `#include` statement to include the .tli header file.

This attribute is used in conjunction with implementation_only.

END C++ Specific

See also

#import Attributes
#import Directive

no_namespace

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Specifies that the namespace name is not generated by the compiler.

Syntax

```
no_namespace
```

Remarks

The type-library contents in the `#import` header file are normally defined in a namespace. The namespace name is specified in the `library` statement of the original IDL file. If the **no_namespace** attribute is specified, then this namespace is not generated by the compiler.

If you want to use a different namespace name, then use the rename_namespace attribute instead.

END C++ Specific

See also

#import Attributes
#import Directive

no_registry

4/4/2019 • 2 minutes to read • Edit Online

no_registry tells the compiler not to search the registry for type libraries imported with `#import`.

Syntax

```
#import filename no_registry
```

Parameters

filename
A type library.

Remarks

If a referenced type library is not found in the include directories, the compilation will fail even if the type library is in the registry. **no_registry** propagates to other type libraries implicitly imported with `auto_search`.

The compiler will never search the registry for type libraries that are specified by file name and passed directly to `#import`.

When `auto_search` is specified, the additional `#import`s will be generated with the **no_registry** setting of the initial `#import` (if the initial `#import` directive was **no_registry**, an `auto_search`-generated `#import` is also **no_registry**.)

no_registry is useful if you want to import cross referenced type libraries without the risk of the compiler finding an older version of the file in the registry. **no_registry** is also useful if the type library is not registered.

See also

#import Attributes
#import Directive

no_search_namespace

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Has the same functionality as the no_namespace attribute but is used on type libraries that you use the `#import` directive with the auto_search attribute.

Syntax

```
no_search_namespace
```

Remarks

END C++ Specific

See also

#import Attributes
#import Directive

no_smart_pointers

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Suppresses the creation of smart pointers for all interfaces in the type library.

Syntax

```
no_smart_pointers
```

Remarks

By default, when you use `#import`, you get a smart pointer declaration for all interfaces in the type library. These smart pointers are of type _com_ptr_t Class.

END C++ Specific

See also

#import Attributes
#import Directive

raw_dispinterfaces

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Tells the compiler to generate low-level wrapper functions for dispinterface methods and properties that call `IDispatch::Invoke` and return the HRESULT error code.

Syntax

```
raw_dispinterfaces
```

Remarks

If this attribute is not specified, only high-level wrappers are generated, which throw C++ exceptions in case of failure.

END C++ Specific

See also

#import Attributes
#import Directive

raw_interfaces_only

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Suppresses the generation of error-handling wrapper functions and property declarations that use those wrapper functions.

Syntax

```
raw_interfaces_only
```

Remarks

The **raw_interfaces_only** attribute also causes the default prefix used in naming the non-property functions to be removed. Normally, the prefix is **raw_**. If this attribute is specified, the function names are directly from the type library.

This attribute allows you to expose only the low-level contents of the type library.

END C++ Specific

See also

#import Attributes
#import Directive

raw_method_prefix

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Specifies a different prefix to avoid name collisions.

Syntax

```
raw_method_prefix("Prefix")
```

Parameters

Prefix
The prefix to be used.

Remarks

Low-level properties and methods are exposed by member functions named with a default prefix of **raw_** to avoid name collisions with the high-level error-handling member functions.

NOTE

The effects of the **raw_method_prefix** attribute will not be changed by the presence of the raw_interfaces_only attribute. The **raw_method_prefix** always takes precedence over `raw_interfaces_only` in specifying a prefix. If both attributes are used in the same `#import` statement, then the prefix specified by the **raw_method_prefix** attribute is used.

END C++ Specific

See also

#import Attributes
#import Directive

raw_native_types

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Disables the use of COM support classes in the high-level wrapper functions and forces the use of low-level data types instead.

Syntax

```
raw_native_types
```

Remarks

By default, the high-level error-handling methods use the COM support classes _bstr_t and _variant_t in place of the `BSTR` and `VARIANT` data types and raw COM interface pointers. These classes encapsulate the details of allocating and deallocating memory storage for these data types, and greatly simplify type casting and conversion operations.

END C++ Specific

See also

#import Attributes
#import Directive

raw_property_prefixes

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Specifies alternate prefixes for three property methods.

Syntax

```
raw_property_prefixes("GetPrefix","PutPrefix","PutRefPrefix")
```

Parameters

GetPrefix
Prefix to be used for the `propget` methods.

PutPrefix
Prefix to be used for the `propput` methods.

PutRefPrefix
Prefix to be used for the `propputref` methods.

Remarks

By default, low-level `propget`, `propput`, and `propputref` methods are exposed by member functions named with prefixes of **get_**, **put_**, and **putref_** respectively. These prefixes are compatible with the names used in the header files generated by MIDL.

END C++ Specific

See also

#import Attributes
#import Directive

rename (#import)

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Works around name collision problems.

Syntax

```
rename("OldName","NewName")
```

Parameters

OldName
Old name in the type library.

NewName
Name to be used instead of the old name.

Remarks

If this attribute is specified, the compiler replaces all occurrences of *OldName* in a type library with the user-supplied *NewName* in the resulting header files.

This attribute can be used when a name in the type library coincides with a macro definition in the system header files. If this situation is not resolved, then various syntax errors will be generated, such as Compiler Error C2059 and Compiler Error C2061.

NOTE
The replacement is for a name used in the type library, not for a name used in the resulting header file.

For example, suppose a property named `MyParent` exists in a type library, and a macro `GetMyParent` is defined in a header file and used before `#import`. Since `GetMyParent` is the default name of a wrapper function for the error-handling `get` property, a name collision will occur. To work around the problem, use the following attribute in the `#import` statement:

```
rename("MyParent","MyParentX")
```

which renames the name `MyParent` in the type library. An attempt to rename the `GetMyParent` wrapper name will fail:

```
rename("GetMyParent","GetMyParentX")
```

This is because the name `GetMyParent` only occurs in the resulting type library header file.

END C++ Specific

See also

#import Attributes
#import Directive

rename_namespace

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Renames the namespace that contains the contents of the type library.

Syntax

```
rename_namespace("NewName")
```

Parameters

NewName
The new name of the namespace.

Remarks

It takes a single argument, *NewName*, which specifies the new name for the namespace.

To remove the namespace, use the no_namespace attribute instead.

END C++ Specific

See also

#import Attributes
#import Directive

rename_search_namespace

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Has the same functionality as the rename_namespace attribute but is used on type libraries that you use the `#import` directive with the auto_search attribute.

Syntax

```
rename_search_namespace("NewName")
```

Parameters

NewName
The new name of the namespace.

Remarks

END C++ Specific

See also

#import Attributes
#import Directive

tlbid

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Allows for loading libraries other than the primary type library.

Syntax

```
tlbid(number)
```

Parameters

number
The number of the type library in `filename`.

Remarks

If multiple type libraries are built into a single DLL, it possible to load libraries other than the primary type library by using **tlbid**.

For example:

```
#import <MyResource.dll> tlbid(2)
```

is equivalent to:

```
LoadTypeLib("MyResource.dll\\2");
```

END C++ Specific

See also

#import Attributes
#import Directive

#include Directive (C/C++)

4/4/2019 • 4 minutes to read • Edit Online

Tells the preprocessor to treat the contents of a specified file as if they appear in the source program at the point where the directive appears.

Syntax

```
#include  "path-spec"
#include  <path-spec>
```

Remarks

You can organize constant and macro definitions into include files and then use **#include** directives to add them to any source file. Include files are also useful for incorporating declarations of external variables and complex data types. The types may be defined and named only once in an include file created for that purpose.

The *path-spec* is a file name that may optionally be preceded by a directory specification. The file name must name an existing file. The syntax of the *path-spec* depends on the operating system on which the program is compiled.

For information about how to reference assemblies in a C++ application that's compiled by using /clr, see #using.

Both syntax forms cause that directive to be replaced by the entire contents of the specified include file. The difference between the two forms is the order in which the preprocessor searches for header files when the path is incompletely specified. The following table shows the difference between the two syntax forms.

SYNTAX FORM	ACTION
Quoted form	The preprocessor searches for include files in this order: 1) In the same directory as the file that contains the **#include** statement. 2) In the directories of the currently opened include files, in the reverse order in which they were opened. The search begins in the directory of the parent include file and continues upward through the directories of any grandparent include files. 3) Along the path that's specified by each **/I** compiler option. 4) Along the paths that are specified by the INCLUDE environment variable.
Angle-bracket form	The preprocessor searches for include files in this order: 1) Along the path that's specified by each **/I** compiler option. 2) When compiling occurs on the command line, along the paths that are specified by the INCLUDE environment variable.

The preprocessor stops searching as soon as it finds a file that has the given name. If you enclose a complete, unambiguous path specification for the include file between double quotation marks (**" "**), the preprocessor searches only that path specification and ignores the standard directories.

If the file name that's enclosed in double quotation marks is an incomplete path specification, the preprocessor first searches the "parent" file's directory. A parent file is the file that contains the **#include** directive. For example, if you include a file named *file2* in a file named *file1*, *file1* is the parent file.

Include files can be "nested"; that is, an **#include** directive can appear in a file that's named by another **#include** directive. For example, *file2* could include *file3*. In this case, *file1* would still be the parent of *file2*, but it would be the "grandparent" of *file3*.

When include files are nested and when compiling occurs on the command line, directory searching begins with the directories of the parent file and then proceeds through the directories of any grandparent files. That is, searching begins relative to the directory that contains the source that's currently being processed. If the file is not found, the search moves to directories that are specified by the /I (Additional include directories) compiler option. Finally, the directories that are specified by the INCLUDE environment variable are searched.

From the Visual Studio development environment, the INCLUDE environment variable is ignored. For information about how to set the directories that are searched for include files—this also applies to the LIB environment variable—see VC++ Directories Property Page.

This example shows file inclusion by using angle brackets:

```
#include <stdio.h>
```

This example adds the contents of the file named STDIO.H to the source program. The angle brackets cause the preprocessor to search the directories that are specified by the INCLUDE environment variable for STDIO.H, after it searches directories that are specified by the **/I** compiler option.

The next example shows file inclusion by using the quoted form:

```
#include "defs.h"
```

This example adds the contents of the file that's specified by DEFS.H to the source program. The quotation marks mean that the preprocessor first searches the directory that contains the parent source file.

Nesting of include files can continue up to 10 levels. When the nested **#include** is processed, the preprocessor continues to insert the enclosing include file into the original source file.

Microsoft Specific

To locate includable source files, the preprocessor first searches the directories that are specified by the **/I** compiler option. If the **/I** option is not present or fails, the preprocessor uses the INCLUDE environment variable to find any include files within angle brackets. The INCLUDE environment variable and **/I** compiler option can contain multiple paths, separated by semicolons (**;**). If more than one directory appears as part of the **/I** option or within the INCLUDE environment variable, the preprocessor searches them in the order in which they appear.

For example, the command

```
CL /ID:\MSVC\INCLUDE MYPROG.C
```

causes the preprocessor to search the directory D:\MSVC\INCLUDE\ for include files such as STDIO.H. The commands

```
SET INCLUDE=D:\MSVC\INCLUDE
CL MYPROG.C
```

have the same effect. If both sets of searches fail, a fatal compiler error is generated.

If the file name is fully specified for an include file that has a path that includes a colon (for example, F:\MSVC\SPECIAL\INCL\TEST.H), the preprocessor follows the path.

For include files that are specified as `#include "path-spec"`, directory searching begins with the directory of the parent file and then proceeds through the directories of any grandparent files. That is, searching begins relative to the directory that contains the source file that contains the **#include** directive that's being processed. If there is no grandparent file and the file has not been found, the search continues as if the file name were enclosed in angle brackets.

END Microsoft Specific

See also

Preprocessor Directives
/I (Additional include directories)

#line Directive (C/C++)

4/4/2019 • 2 minutes to read • Edit Online

The **#line** directive tells the preprocessor to change the compiler's internally stored line number and filename to a given line number and filename.

Syntax

> **#line** *digit-sequence* ["*filename*"]

Remarks

The compiler uses the line number and optional filename to refer to errors that it finds during compilation. The line number usually refers to the current input line, and the filename refers to the current input file. The line number is incremented after each line is processed.

The *digit-sequence* value can be any integer constant. Macro replacement can be performed on the preprocessing tokens, but the result must evaluate to the correct syntax. The *filename* can be any combination of characters and must be enclosed in double quotation marks (**" "**). If *filename* is omitted, the previous filename remains unchanged.

You can alter the source line number and filename by writing a **#line** directive. The translator uses the line number and filename to determine the values of the predefined macros `__FILE__` and `__LINE__`. You can use these macros to insert self-descriptive error messages into the program text. For more information on these predefined macros, see Predefined Macros.

The `__FILE__` macro expands to a string whose contents are the filename, surrounded by double quotation marks (**" "**).

If you change the line number and filename, the compiler ignores the previous values and continues processing with the new values. The **#line** directive is typically used by program generators to cause error messages to refer to the original source file instead of to the generated program.

The following examples illustrate **#line** and the `__LINE__` and `__FILE__` macros.

In this statement, the internally stored line number is set to 151 and the filename is changed to copy.c.

```
#line 151 "copy.c"
```

In this example, the macro `ASSERT` uses the predefined macros `__LINE__` and `__FILE__` to print an error message about the source file if a given assertion is not true.

```
#define ASSERT(cond) if( !(cond) )\
{printf( "assertion error line %d, file(%s)\n", \
__LINE__, __FILE__ );}
```

See also

Preprocessor Directives

Null Directive

4/4/2019 • 2 minutes to read • Edit Online

The null preprocessor directive is a single number sign (**#**) alone on a line. It has no effect.

Syntax

```
#
```

See also

Preprocessor Directives

#undef Directive (C/C++)

4/4/2019 • 2 minutes to read • Edit Online

Removes (undefines) a name previously created with `#define`.

Syntax

```
#undef
identifier
```

Remarks

The **#undef** directive removes the current definition of *identifier*. Consequently, subsequent occurrences of *identifier* are ignored by the preprocessor. To remove a macro definition using **#undef**, give only the macro *identifier* ; do not give a parameter list.

You can also apply the **#undef** directive to an identifier that has no previous definition. This ensures that the identifier is undefined. Macro replacement is not performed within **#undef** statements.

The **#undef** directive is typically paired with a `#define` directive to create a region in a source program in which an identifier has a special meaning. For example, a specific function of the source program can use manifest constants to define environment-specific values that do not affect the rest of the program. The **#undef** directive also works with the `#if` directive to control conditional compilation of the source program. See The #if, #elif, #else, and #endif Directives for more information.

In the following example, the **#undef** directive removes definitions of a symbolic constant and a macro. Note that only the identifier of the macro is given.

```
#define WIDTH 80
#define ADD( X, Y ) ((X) + (Y))
.
.
.
#undef WIDTH
#undef ADD
```

Microsoft Specific

Macros can be undefined from the command line using the `/U` option, followed by the macro names to be undefined. The effect of issuing this command is equivalent to a sequence of `#undef` *macro-name* statements at the beginning of the file.

END Microsoft Specific

See also

Preprocessor Directives

#using Directive (C++/CLI)

4/4/2019 • 2 minutes to read • Edit Online

Imports metadata into a program compiled with /clr.

Syntax

```
#using file [as_friend]
```

Parameters

file
An MSIL .dll, .exe, .netmodule, or .obj. For example,

```
#using <MyComponent.dll>
```

as_friend
Specifies that all types in *file* are accessible. For more information, see Friend Assemblies (C++).

Remarks

file can be a Microsoft intermediate language (MSIL) file that you import for its managed data and managed constructs. If a .dll file contains an assembly manifest, then all the .dlls referenced in the manifest are imported and the assembly you are building will list *file* in the metadata as an assembly reference.

If *file* does not contain an assembly (if *file* is a module) and if you do not intend to use type information from the module in the current (assembly) application, you have the option of just indicating that the module is part the assembly; use /ASSEMBLYMODULE. The types in the module would then be available to any application that referenced the assembly.

An alternative to use **#using** is the /FU compiler option.

.exe assemblies passed to **#using** should be compiled by using one of the .NET Visual Studio compilers (Visual Basic or Visual C#, for example). Attempting to import metadata from an .exe assembly compiled with `/clr` will result in a file load exception.

NOTE
A component that is referenced with **#using** can be run with a different version of the file imported at compile time, causing a client application to give unexpected results.

In order for the compiler to recognize a type in an assembly (not a module), it needs to be forced to resolve the type, which you can do, for example, by defining an instance of the type. There are other ways to resolve type names in an assembly for the compiler, for example, if you inherit from a type in an assembly, the type name will then become known to the compiler.

When importing metadata built from source code that used __declspec(thread), the thread semantics are not persisted in metadata. For example, a variable declared with **__declspec(thread)**, compiled in a program that is build for the .NET Framework common language runtime, and then imported via **#using**, will no longer have **__declspec(thread)** semantics on the variable.

All imported types (both managed and native) in a file referenced by **#using** are available, but the compiler treats native types as declarations not definitions.

mscorlib.dll is automatically referenced when compiling with `/clr`.

The LIBPATH environment variable specifies the directories that will be searched when the compiler tries to resolve file names passed to **#using**.

The compiler will search for references along the following path:

- A path specified in the **#using** statement.
- The current directory.
- The .NET Framework system directory.
- Directories added with the /AI compiler option.
- Directories on LIBPATH environment variable.

Example

If you build an assembly (C) and reference an assembly (B) that itself references another assembly (A), you will not have to explicitly reference assembly A unless you explicitly use one of A's types in C.

```
// using_assembly_A.cpp
// compile with: /clr /LD
public ref class A {};
```

Example

```
// using_assembly_B.cpp
// compile with: /clr /LD
#using "using_assembly_A.dll"
public ref class B {
public:
   void Test(A a) {}
   void Test() {}
};
```

Example

In the following sample, there is no compiler error for not referencing using_assembly_A.dll because the program does not use any of the types defined in using_assembly_A.cpp.

```
// using_assembly_C.cpp
// compile with: /clr
#using "using_assembly_B.dll"
int main() {
   B b;
   b.Test();
}
```

See also

Preprocessor Directives

Preprocessor Operators

4/4/2019 • 2 minutes to read • Edit Online

Four preprocessor-specific operators are used in the context of the `#define` directive (see the following list for a summary of each). The stringizing, charizing, and token-pasting operators are discussed in the next three sections. For information on the `defined` operator, see The #if, #elif, #else, and #endif Directives.

OPERATOR	ACTION
Stringizing operator (#)	Causes the corresponding actual argument to be enclosed in double quotation marks
Charizing operator (#@)	Causes the corresponding argument to be enclosed in single quotation marks and to be treated as a character (Microsoft Specific)
Token-pasting operator (##)	Allows tokens used as actual arguments to be concatenated to form other tokens
defined operator	Simplifies the writing of compound expressions in certain macro directives

See also

Preprocessor Directives
Predefined Macros
C/C++ Preprocessor Reference

Stringizing Operator (#)

4/4/2019 • 2 minutes to read • Edit Online

The number-sign or "stringizing" operator (**#**) converts macro parameters to string literals without expanding the parameter definition. It is used only with macros that take arguments. If it precedes a formal parameter in the macro definition, the actual argument passed by the macro invocation is enclosed in quotation marks and treated as a string literal. The string literal then replaces each occurrence of a combination of the stringizing operator and formal parameter within the macro definition.

NOTE

The Microsoft C (versions 6.0 and earlier) extension to the ANSI C standard that previously expanded macro formal arguments appearing inside string literals and character constants is no longer supported. Code that relied on this extension should be rewritten using the stringizing (**#**) operator.

White space preceding the first token of the actual argument and following the last token of the actual argument is ignored. Any white space between the tokens in the actual argument is reduced to a single white space in the resulting string literal. Thus, if a comment occurs between two tokens in the actual argument, it is reduced to a single white space. The resulting string literal is automatically concatenated with any adjacent string literals from which it is separated only by white space.

Further, if a character contained in the argument usually requires an escape sequence when used in a string literal (for example, the quotation mark (**"**) or backslash (\) character), the necessary escape backslash is automatically inserted before the character.

The Visual C++ stringizing operator does not behave correctly when it is used with strings that include escape sequences. In this situation, the compiler generates Compiler Error C2017.

Examples

The following example shows a macro definition that includes the stringizing operator and a main function that invokes the macro:

Such invocations would be expanded during preprocessing, producing the following code:

```
int main() {
   printf_s( "In quotes in the printf function call\n" "\n" );
   printf_s( "\"In quotes when printed to the screen\"\n" "\n" );
   printf_s( "\"This: \\\" prints an escaped double quote\"" "\n" );
}
```

```
// stringizer.cpp
#include <stdio.h>
#define stringer( x ) printf_s( #x "\n" )
int main() {
   stringer( In quotes in the printf function call );
   stringer( "In quotes when printed to the screen" );
   stringer( "This: \"  prints an escaped double quote" );
}
```

```
In quotes in the printf function call
"In quotes when printed to the screen"
"This: \"  prints an escaped double quote"
```

The following sample shows how you can expand a macro parameter:

```
// stringizer_2.cpp
// compile with: /E
#define F abc
#define B def
#define FB(arg) #arg
#define FB1(arg) FB(arg)
FB(F B)
FB1(F B)
```

See also

Preprocessor Operators

Charizing Operator (#@)

4/4/2019 • 2 minutes to read • Edit Online

Microsoft Specific

The charizing operator can be used only with arguments of macros. If `#@` precedes a formal parameter in the definition of the macro, the actual argument is enclosed in single quotation marks and treated as a character when the macro is expanded. For example:

```
#define makechar(x)  #@x
```

causes the statement

```
a = makechar(b);
```

to be expanded to

```
a = 'b';
```

The single-quotation character cannot be used with the charizing operator.

END Microsoft Specific

See also

Preprocessor Operators

Token-Pasting Operator (##)

4/4/2019 • 2 minutes to read • Edit Online

The double-number-sign or "token-pasting" operator (**##**), which is sometimes called the "merging" operator, is used in both object-like and function-like macros. It permits separate tokens to be joined into a single token and therefore cannot be the first or last token in the macro definition.

If a formal parameter in a macro definition is preceded or followed by the token-pasting operator, the formal parameter is immediately replaced by the unexpanded actual argument. Macro expansion is not performed on the argument prior to replacement.

Then, each occurrence of the token-pasting operator in *token-string* is removed, and the tokens preceding and following it are concatenated. The resulting token must be a valid token. If it is, the token is scanned for possible replacement if it represents a macro name. The identifier represents the name by which the concatenated tokens will be known in the program before replacement. Each token represents a token defined elsewhere, either within the program or on the compiler command line. White space preceding or following the operator is optional.

This example illustrates use of both the stringizing and token-pasting operators in specifying program output:

```
#define paster( n ) printf_s( "token" #n " = %d", token##n )
int token9 = 9;
```

If a macro is called with a numeric argument like

```
paster( 9 );
```

the macro yields

```
printf_s( "token" "9" " = %d", token9 );
```

which becomes

```
printf_s( "token9 = %d", token9 );
```

Example

```
// preprocessor_token_pasting.cpp
#include <stdio.h>
#define paster( n ) printf_s( "token" #n " = %d", token##n )
int token9 = 9;

int main()
{
   paster(9);
}
```

```
token9 = 9
```

See also

Preprocessor Operators

Macros (C/C++)

4/4/2019 • 2 minutes to read • Edit Online

Preprocessing expands macros in all lines that are not preprocessor directives (lines that do not have a **#** as the first non-white-space character) and in parts of some directives that are not skipped as part of a conditional compilation. "Conditional compilation" directives allow you to suppress compilation of parts of a source file by testing a constant expression or identifier to determine which text blocks are passed on to the compiler and which text blocks are removed from the source file during preprocessing.

The `#define` directive is typically used to associate meaningful identifiers with constants, keywords, and commonly used statements or expressions. Identifiers that represent constants are sometimes called "symbolic constants" or "manifest constants." Identifiers that represent statements or expressions are called "macros." In this preprocessor documentation, only the term "macro" is used.

When the name of the macro is recognized in the program source text or in the arguments of certain other preprocessor commands, it is treated as a call to that macro. The macro name is replaced by a copy of the macro body. If the macro accepts arguments, the actual arguments following the macro name are substituted for formal parameters in the macro body. The process of replacing a macro call with the processed copy of the body is called "expansion" of the macro call.

In practical terms, there are two types of macros. "Object-like" macros take no arguments, whereas "function-like" macros can be defined to accept arguments so that they look and act like function calls. Because macros do not generate actual function calls, you can sometimes make programs run faster by replacing function calls with macros. (In C++, inline functions are often a preferred method.) However, macros can create problems if you do not define and use them with care. You may have to use parentheses in macro definitions with arguments to preserve the proper precedence in an expression. Also, macros may not correctly handle expressions with side effects. See the `getrandom` example in The #define Directive for more information.

Once you have defined a macro, you cannot redefine it to a different value without first removing the original definition. However, you can redefine the macro with exactly the same definition. Thus, the same definition can appear more than once in a program.

The `#undef` directive removes the definition of a macro. Once you have removed the definition, you can redefine the macro to a different value. The #define Directive and The #undef Directive discuss the `#define` and `#undef` directives, respectively.

For more information, see,

- Macros and C++
- Variadic Macros
- Predefined Macros

See also

C/C++ Preprocessor Reference

Macros and C++

4/4/2019 • 2 minutes to read • Edit Online

C++ offers new capabilities, some of which supplant those offered by the ANSI C preprocessor. These new capabilities enhance the type safety and predictability of the language:

- In C++, objects declared as **const** can be used in constant expressions. This allows programs to declare constants that have type and value information, and enumerations that can be viewed symbolically with the debugger. Using the preprocessor `#define` directive to define constants is not as precise. No storage is allocated for a **const** object unless an expression that takes its address is found in the program.
- The C++ inline function capability supplants function-type macros. The advantages of using inline functions over macros are:
  - Type safety. Inline functions are subject to the same type checking as normal functions. Macros are not type safe.
  - Correct handling of arguments that have side effects. Inline functions evaluate the expressions supplied as arguments prior to entering the function body. Therefore, there is no chance that an expression with side effects will be unsafe.

For more information on inline functions, see inline, __inline, __forceinline.

For backward compatibility, all preprocessor facilities that existed in ANSI C and in earlier C++ specifications are preserved for Microsoft C++.

See also

Predefined Macros
Macros (C/C++)

Variadic Macros

4/4/2019 • 2 minutes to read • Edit Online

Variadic macros are function-like macros that contain a variable number of arguments.

Remarks

To use variadic macros, the ellipsis may be specified as the final formal argument in a macro definition, and the replacement identifier `__VA_ARGS__` may be used in the definition to insert the extra arguments. `__VA_ARGS__` is replaced by all of the arguments that match the ellipsis, including commas between them.

The C Standard specifies that at least one argument must be passed to the ellipsis, to ensure that the macro does not resolve to an expression with a trailing comma. The Visual C++ implementation will suppress a trailing comma if no arguments are passed to the ellipsis.

Example

```
// variadic_macros.cpp
#include <stdio.h>
#define EMPTY

#define CHECK1(x, ...) if (!(x)) { printf(__VA_ARGS__); }
#define CHECK2(x, ...) if ((x)) { printf(__VA_ARGS__); }
#define CHECK3(...) { printf(__VA_ARGS__); }
#define MACRO(s, ...) printf(s, __VA_ARGS__)

int main() {
    CHECK1(0, "here %s %s %s", "are", "some", "varargs1(1)\n");
    CHECK1(1, "here %s %s %s", "are", "some", "varargs1(2)\n");   // won't print

    CHECK2(0, "here %s %s %s", "are", "some", "varargs2(3)\n");   // won't print
    CHECK2(1, "here %s %s %s", "are", "some", "varargs2(4)\n");

    // always invokes printf in the macro
    CHECK3("here %s %s %s", "are", "some", "varargs3(5)\n");

    MACRO("hello, world\n");

    MACRO("error\n", EMPTY); // would cause error C2059, except VC++
                             // suppresses the trailing comma
}
```

```
here are some varargs1(1)
here are some varargs2(4)
here are some varargs3(5)
hello, world
error
```

See also

Macros (C/C++)

Predefined Macros

4/8/2019 • 12 minutes to read • Edit Online

The Microsoft C/C++ compiler (MSVC) predefines certain preprocessor macros, depending on the language (C or C++), the compilation target, and the chosen compiler options.

MSVC supports the predefined preprocessor macros required by the ANSI/ISO C99 standard and the ISO C++14 and C++17 standards. The implementation also supports several more Microsoft-specific preprocessor macros. Some macros are defined only for specific build environments or compiler options. Except where noted, the macros are defined throughout a translation unit as if they were specified as **/D** compiler option arguments. When defined, the macros are expanded to the specified values by the preprocessor before compilation. The predefined macros take no arguments and can't be redefined.

Standard predefined identifier

The compiler supports this predefined identifier specified by ISO C99 and ISO C++11.

- **__func__** The unqualified and unadorned name of the enclosing function as a function-local **static const** array of **char**.

  ```
  void example(){
      printf("%s\n", __func__);
  } // prints "example"
  ```

Standard predefined macros

The compiler supports these predefined macros specified by the ISO C99 and ISO C++17 standards.

- **__cplusplus** Defined as an integer literal value when the translation unit is compiled as C++. Otherwise, undefined.
- **__DATE__** The compilation date of the current source file. The date is a constant length string literal of the form *Mmm dd yyyy*. The month name *Mmm* is the same as the abbreviated month name generated by the C Runtime Library (CRT) asctime function. The first character of date *dd* is a space if the value is less than 10. This macro is always defined.
- **__FILE__** The name of the current source file. **__FILE__** expands to a character string literal. To ensure that the full path to the file is displayed, use /FC (Full Path of Source Code File in Diagnostics). This macro is always defined.
- **__LINE__** Defined as the integer line number in the current source file. The value of the **__LINE__** macro can be changed by using a `#line` directive. This macro is always defined.
- **__STDC__** Defined as 1 only when compiled as C and if the /Za compiler option is specified. Otherwise, undefined.
- **__STDC_HOSTED__** Defined as 1 if the implementation is a *hosted implementation*, one that supports the entire required standard library. Otherwise, defined as 0.
- **__STDCPP_THREADS__** Defined as 1 if and only if a program can have more than one thread of execution, and compiled as C++. Otherwise, undefined.
- **__TIME__** The time of translation of the preprocessed translation unit. The time is a character string literal

of the form *hh:mm:ss*, the same as the time returned by the CRT asctime function. This macro is always defined.

Microsoft-specific predefined macros

MSVC supports these additional predefined macros.

- __**ATOM**__ Defined as 1 when the /favor:ATOM compiler option is set and the compiler target is x86 or x64. Otherwise, undefined.

- __**AVX**__ Defined as 1 when the /arch:AVX or /arch:AVX2 compiler options are set and the compiler target is x86 or x64. Otherwise, undefined.

- __**AVX2**__ Defined as 1 when the /arch:AVX2 compiler option is set and the compiler target is x86 or x64. Otherwise, undefined.

- _**CHAR_UNSIGNED** Defined as 1 if the default **char** type is unsigned. This value is defined when the /J (Default char Type Is unsigned) compiler option is set. Otherwise, undefined.

- __**CLR_VER** Defined as an integer literal that represents the version of the Common Language Runtime (CLR) used to compile the app. The value is encoded in the form `Mmmbbbbb`, where `M` is the major version of the runtime, `mm` is the minor version of the runtime, and `bbbbb` is the build number. __**CLR_VER** is defined if the /clr compiler option is set. Otherwise, undefined.

  ```
  // clr_ver.cpp
  // compile with: /clr
  using namespace System;
  int main() {
     Console::WriteLine(__CLR_VER);
  }
  ```

- _**CONTROL_FLOW_GUARD** Defined as 1 when the /guard:cf (Enable Control Flow Guard) compiler option is set. Otherwise, undefined.

- __**COUNTER**__ Expands to an integer literal that starts at 0. The value is incremented by 1 every time it's used in a source file, or in included headers of the source file. __**COUNTER**__ remembers its state when you use precompiled headers. This macro is always defined.

  This example uses `__COUNTER__` to assign unique identifiers to three different objects of the same type. The `exampleClass` constructor takes an integer as a parameter. In `main`, the application declares three objects of type `exampleClass`, using `__COUNTER__` as the unique identifier parameter:

```
// macro__COUNTER__.cpp
// Demonstration of __COUNTER__, assigns unique identifiers to
// different objects of the same type.
// Compile by using: cl /EHsc /W4 macro__COUNTER__.cpp
#include <stdio.h>

class exampleClass {
    int m_nID;
public:
    // initialize object with a read-only unique ID
    exampleClass(int nID) : m_nID(nID) {}
    int GetID(void) { return m_nID; }
};

int main()
{
    // __COUNTER__ is initially defined as 0
    exampleClass e1(__COUNTER__);

    // On the second reference, __COUNTER__ is now defined as 1
    exampleClass e2(__COUNTER__);

    // __COUNTER__ is now defined as 2
    exampleClass e3(__COUNTER__);

    printf("e1 ID: %i\n", e1.GetID());
    printf("e2 ID: %i\n", e2.GetID());
    printf("e3 ID: %i\n", e3.GetID());

    // Output
    // ------------------------------
    // e1 ID: 0
    // e2 ID: 1
    // e3 ID: 2

    return 0;
}
```

- **__cplusplus_cli** Defined as the integer literal value 200406 when compiled as C++ and a /clr compiler option is set. Otherwise, undefined. When defined, **__cplusplus_cli** is in effect throughout the translation unit.

```
// cplusplus_cli.cpp
// compile by using /clr
#include "stdio.h"
int main() {
   #ifdef __cplusplus_cli
      printf("%d\n", __cplusplus_cli);
   #else
      printf("not defined\n");
   #endif
}
```

- **__cplusplus_winrt** Defined as the integer literal value 201009 when compiled as C++ and the /ZW (Windows Runtime Compilation) compiler option is set. Otherwise, undefined.

- **_CPPRTTI** Defined as 1 if the /GR (Enable Run-Time Type Information) compiler option is set. Otherwise, undefined.

- **_CPPUNWIND** Defined as 1 if one or more of the /GX (Enable Exception Handling), /clr (Common Language Runtime Compilation), or /EH (Exception Handling Model) compiler options are set. Otherwise, undefined.

- **_DEBUG** Defined as 1 when the /LDd, /MDd, or /MTd compiler option is set. Otherwise, undefined.

- **_DLL** Defined as 1 when the /MD or /MDd (Multithreaded DLL) compiler option is set. Otherwise, undefined.

- **__FUNCDNAME__** Defined as a string literal that contains the decorated name of the enclosing function. The macro is defined only within a function. The **__FUNCDNAME__** macro isn't expanded if you use the /EP or /P compiler option.

  This example uses the `__FUNCDNAME__`, `__FUNCSIG__`, and `__FUNCTION__` macros to display function information.

```
// Demonstrates functionality of __FUNCTION__, __FUNCDNAME__, and __FUNCSIG__ macros
void exampleFunction()
{
    printf("Function name: %s\n", __FUNCTION__);
    printf("Decorated function name: %s\n", __FUNCDNAME__);
    printf("Function signature: %s\n", __FUNCSIG__);

    // Sample Output
    // -------------------------------------------------
    // Function name: exampleFunction
    // Decorated function name: ?exampleFunction@@YAXXZ
    // Function signature: void __cdecl exampleFunction(void)
}
```

- **__FUNCSIG__** Defined as a string literal that contains the signature of the enclosing function. The macro is defined only within a function. The **__FUNCSIG__** macro isn't expanded if you use the /EP or /P compiler option. When compiled for a 64-bit target, the calling convention is `__cdecl` by default. For an example of usage, see the `__FUNCDNAME__` macro.

- **__FUNCTION__** Defined as a string literal that contains the undecorated name of the enclosing function. The macro is defined only within a function. The **__FUNCTION__** macro isn't expanded if you use the /EP or /P compiler option. For an example of usage, see the `__FUNCDNAME__` macro.

- **_INTEGRAL_MAX_BITS** Defined as the integer literal value 64, the maximum size (in bits) for a non-vector integral type. This macro is always defined.

```
// integral_max_bits.cpp
#include <stdio.h>
int main() {
    printf("%d\n", _INTEGRAL_MAX_BITS);
}
```

- **__INTELLISENSE__** Defined as 1 during an IntelliSense compiler pass in the Visual Studio IDE. Otherwise, undefined. You can use this macro to guard code the IntelliSense compiler doesn't understand, or use it to toggle between the build and IntelliSense compiler. For more information, see Troubleshooting Tips for IntelliSense Slowness.

- **_ISO_VOLATILE** Defined as 1 if the /volatile:iso compiler option is set. Otherwise, undefined.

- **_KERNEL_MODE** Defined as 1 if the /kernel (Create Kernel Mode Binary) compiler option is set. Otherwise, undefined.

- **_M_AMD64** Defined as the integer literal value 100 for compilations that target x64 processors. Otherwise, undefined.

- **_M_ARM** Defined as the integer literal value 7 for compilations that target ARM processors. Otherwise, undefined.

- **_M_ARM_ARMV7VE** Defined as 1 when the /arch:ARMv7VE compiler option is set for compilations that target ARM processors. Otherwise, undefined.
- **_M_ARM_FP** Defined as an integer literal value that indicates which /arch compiler option was set for ARM processor targets. Otherwise, undefined.
  - A value in the range 30-39 if no `/arch` ARM option was specified, indicating the default architecture for ARM was set (`VFPv3`).
  - A value in the range 40-49 if `/arch:VFPv4` was set.
  - For more information, see /arch (ARM).
- **_M_ARM64** Defined as 1 for compilations that target 64-bit ARM processors. Otherwise, undefined.
- **_M_CEE** Defined as 001 if any /clr (Common Language Runtime Compilation) compiler option is set. Otherwise, undefined.
- **_M_CEE_PURE** Deprecated beginning in Visual Studio 2015. Defined as 001 if the /clr:pure compiler option is set. Otherwise, undefined.
- **_M_CEE_SAFE** Deprecated beginning in Visual Studio 2015. Defined as 001 if the /clr:safe compiler option is set. Otherwise, undefined.
- **_M_FP_EXCEPT** Defined as 1 if the /fp:except or /fp:strict compiler option is set. Otherwise, undefined.
- **_M_FP_FAST** Defined as 1 if the /fp:fast compiler option is set. Otherwise, undefined.
- **_M_FP_PRECISE** Defined as 1 if the /fp:precise compiler option is set. Otherwise, undefined.
- **_M_FP_STRICT** Defined as 1 if the /fp:strict compiler option is set. Otherwise, undefined.
- **_M_IX86** Defined as the integer literal value 600 for compilations that target x86 processors. This macro isn't defined for x64 or ARM compilation targets.
- **_M_IX86_FP** Defined as an integer literal value that indicates the /arch compiler option that was set, or the default. This macro is always defined when the compilation target is an x86 processor. Otherwise, undefined. When defined, the value is:
  - 0 if the `/arch:IA32` compiler option was set.
  - 1 if the `/arch:SSE` compiler option was set.
  - 2 if the `/arch:SSE2`, `/arch:AVX`, or `/arch:AVX2` compiler option was set. This value is the default if an `/arch` compiler option wasn't specified. When `/arch:AVX` is specified, the macro **__AVX__** is also defined. When `/arch:AVX2` is specified, both **__AVX__** and **__AVX2__** are also defined.
  - For more information, see /arch (x86).
- **_M_X64** Defined as the integer literal value 100 for compilations that target x64 processors. Otherwise, undefined.
- **_MANAGED** Defined as 1 when the /clr compiler option is set. Otherwise, undefined.
- **_MSC_BUILD** Defined as an integer literal that contains the revision number element of the compiler's version number. The revision number is the fourth element of the period-delimited version number. For example, if the version number of the Microsoft C/C++ compiler is 15.00.20706.01, the **_MSC_BUILD** macro evaluates to 1. This macro is always defined.
- **_MSC_EXTENSIONS** Defined as 1 if the on-by-default /Ze (Enable Language Extensions) compiler option is set. Otherwise, undefined.

- **_MSC_FULL_VER** Defined as an integer literal that encodes the major, minor, and build number elements of the compiler's version number. The major number is the first element of the period-delimited version number, the minor number is the second element, and the build number is the third element. For example, if the version number of the Microsoft C/C++ compiler is 15.00.20706.01, the **_MSC_FULL_VER** macro evaluates to 150020706. Enter `cl /?` at the command line to view the compiler's version number. This macro is always defined.
- **_MSC_VER** Defined as an integer literal that encodes the major and minor number elements of the compiler's version number. The major number is the first element of the period-delimited version number and the minor number is the second element. For example, if the version number of the Microsoft C/C++ compiler is 17.00.51106.1, the **_MSC_VER** macro evaluates to 1700. Enter `cl /?` at the command line to view the compiler's version number. This macro is always defined.

VISUAL STUDIO VERSION	_MSC_VER
Visual Studio 6.0	1200
Visual Studio .NET 2002 (7.0)	1300
Visual Studio .NET 2003 (7.1)	1310
Visual Studio 2005 (8.0)	1400
Visual Studio 2008 (9.0)	1500
Visual Studio 2010 (10.0)	1600
Visual Studio 2012 (11.0)	1700
Visual Studio 2013 (12.0)	1800
Visual Studio 2015 (14.0)	1900
Visual Studio 2017 RTW (15.0)	1910
Visual Studio 2017 version 15.3	1911
Visual Studio 2017 version 15.5	1912
Visual Studio 2017 version 15.6	1913
Visual Studio 2017 version 15.7	1914
Visual Studio 2017 version 15.8	1915
Visual Studio 2017 version 15.9	1916
Visual Studio 2019 RTW (16.0)	1920

To test for compiler releases or updates in a given version of Visual Studio or after, use the **>=** operator. You can use it in a conditional directive to compare **_MSC_VER** against that known version. If you have several mutually exclusive versions to compare, order your comparisons in descending order of version number. For example, this code checks for compilers released in Visual Studio 2017 and later. Next, it checks for compilers released in or after Visual Studio 2015. Then it checks for all compilers released before Visual Studio 2015:

```
#if _MSC_VER >= 1910
// . . .
#elif _MSC_VER >= 1900
// . . .
#else
// . . .
#endif
```

For more information, see Visual C++ Compiler Version in the Microsoft C++ Team Blog.

- **_MSVC_LANG** Defined as an integer literal that specifies the C++ language standard targeted by the compiler. It's set only in code compiled as C++. The macro is the integer literal value 201402L by default, or when the /std:c++14 compiler option is specified. The macro is set to 201703L if the /std:c++17 compiler option is specified. It's set to a higher, unspecified value when the /std:c++latest option is specified. Otherwise, the macro is undefined. The **_MSVC_LANG** macro and /std (Specify Language Standard Version) compiler options are available beginning in Visual Studio 2015 Update 3.

- **__MSVC_RUNTIME_CHECKS** Defined as 1 when one of the /RTC compiler options is set. Otherwise, undefined.

- **_MT** Defined as 1 when /MD or /MDd (Multithreaded DLL) or /MT or /MTd (Multithreaded) is specified. Otherwise, undefined.

- **_NATIVE_WCHAR_T_DEFINED** Defined as 1 when the /Zc:wchar_t compiler option is set. Otherwise, undefined.

- **_OPENMP** Defined as integer literal 200203, if the /openmp (Enable OpenMP 2.0 Support) compiler option is set. This value represents the date of the OpenMP specification implemented by MSVC. Otherwise, undefined.

```
// _OPENMP_dir.cpp
// compile with: /openmp
#include <stdio.h>
int main() {
    printf("%d\n", _OPENMP);
}
```

- **_PREFAST_** Defined as 1 when the /analyze compiler option is set. Otherwise, undefined.

- **__TIMESTAMP__** Defined as a string literal that contains the date and time of the last modification of the current source file, in the abbreviated, constant length form returned by the CRT asctime function, for example, `Fri 19 Aug 13:32:58 2016`. This macro is always defined.

- **_VC_NODEFAULTLIB** Defined as 1 when the /Zl (Omit Default Library Name) compiler option is set. Otherwise, undefined.

- **_WCHAR_T_DEFINED** Defined as 1 when the default /Zc:wchar_t compiler option is set. The **_WCHAR_T_DEFINED** macro is defined but has no value if the `/Zc:wchar_t-` compiler option is set, and **wchar_t** is defined in a system header file included in your project. Otherwise, undefined.

- **_WIN32** Defined as 1 when the compilation target is 32-bit ARM, 64-bit ARM, x86, or x64. Otherwise, undefined.

- **_WIN64** Defined as 1 when the compilation target is 64-bit ARM or x64. Otherwise, undefined.

- **_WINRT_DLL** Defined as 1 when compiled as C++ and both /ZW (Windows Runtime Compilation) and

/LD or /LDd compiler options are set. Otherwise, undefined.

No preprocessor macros that identify the ATL or MFC library version are predefined by the compiler. ATL and MFC library headers define these version macros internally. They're undefined in preprocessor directives made before the required header is included.

- **_ATL_VER** Defined in <atldef.h> as an integer literal that encodes the ATL version number.
- **_MFC_VER** Defined in <afxver_.h> as an integer literal that encodes the MFC version number.

See also

Macros (C/C++)
Preprocessor Operators
Preprocessor Directives

Grammar Summary (C/C++)

4/4/2019 • 2 minutes to read • Edit Online

This section describes the formal grammar of the preprocessor. It covers the syntax of preprocessing directives and operators discussed in The Preprocessor and in Pragma Directives.

The following topics are included:

- Definitions
- Conventions
- Preprocessor Grammar

See also

C/C++ Preprocessor Reference

Definitions for the Grammar Summary

4/4/2019 • 2 minutes to read • Edit Online

Terminals are endpoints in a syntax definition. No other resolution is possible. Terminals include the set of reserved words and user-defined identifiers.

Nonterminals are placeholders in the syntax. Most are defined elsewhere in this syntax summary. Definitions can be recursive. The following nonterminals are defined in the Lexical Conventions section of the *C++ Language Reference*:

`constant`, *constant-expression*, *identifier*, *keyword*, `operator`, `punctuator`

An optional component is indicated by the subscripted $_{\text{opt}}$. For example, the following indicates an optional expression enclosed in curly braces:

{ *expression*$_{\text{opt}}$ **}**

See also

Grammar Summary (C/C++)

Conventions

4/4/2019 • 2 minutes to read • Edit Online

The conventions use different font attributes for different components of the syntax. The symbols and fonts are as follows:

ATTRIBUTE	DESCRIPTION
nonterminal	Italic type indicates nonterminals.
#include	Terminals in bold type are literal reserved words and symbols that must be entered as shown. Characters in this context are always case sensitive.
opt	Nonterminals followed by $_{opt}$ are always optional.
default typeface	Characters in the set described or listed in this typeface can be used as terminals in statements.

A colon (**:**) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines.

See also

Grammar Summary (C/C++)

Preprocessor Grammar

4/4/2019 • 2 minutes to read • Edit Online

control-line:
  #define *identifier token-string*$_{opt}$
  #define *identifier*(*identifier*$_{opt}$ **,** ... **,** *identifier*$_{opt}$ **)** *token-string*$_{opt}$
  #include " ** *path-spec* **"
  #include < *path-spec* **>**
  #line *digit-sequence* **"** *filename* **"**$_{opt}$
  #undef *identifier*
  #error *token-string*
  #pragma *token-string*

constant-expression:
  defined(*identifier* **)**
  defined *identifier*
  any other constant expression

conditional :
  if-part elif-parts$_{opt}$ *else-part*$_{opt}$ *endif-line*

if-part :
  if-line text

if-line :
  #if *constant-expression*
  #ifdef *identifier*
  #ifndef *identifier*

elif-parts :
  elif-line text
  elif-parts elif-line text

elif-line :
  #elif *constant-expression*

else-part :
  else-line text

else-line :
  #else

endif-line :
  #endif

digit-sequence :
  digit
  digit-sequence digit

digit : one of
  0 1 2 3 4 5 6 7 8 9

token-string :
  String of tokens

token :
  keyword
  identifier
  constant
  operator
  punctuator

filename :
  Legal operating system filename

path-spec :
  Legal file path

text :
  Any sequence of text

> **NOTE**
>
> The following nonterminals are expanded in the Lexical Conventions section of the *C++ Language Reference*: *constant*, *constant-expression*, *identifier*, *keyword*, *operator*, and *punctuator*.

See also

Grammar Summary (C/C++)

Pragma Directives and the __Pragma Keyword

4/4/2019 • 2 minutes to read • Edit Online

Pragma directives specify machine- or operating-specific compiler features. The **__pragma** keyword, which is specific to the Microsoft compiler, enables you to code pragma directives within macro definitions.

Syntax

```
#pragma token-string
__pragma(token-string)
```

Remarks

Each implementation of C and C++ supports some features unique to its host machine or operating system. Some programs, for example, must exercise precise control over the memory areas where data is put or to control the way certain functions receive parameters. The **#pragma** directives offer a way for each compiler to offer machine- and operating system-specific features while retaining overall compatibility with the C and C++ languages.

Pragmas are machine- or operating system-specific by definition, and are usually different for every compiler. Pragmas can be used in conditional statements, to provide new preprocessor functionality, or to provide implementation-defined information to the compiler.

The `token-string` is a series of characters that gives a specific compiler instruction and arguments, if any. The number sign (**#**) must be the first non-white-space character on the line that contains the pragma; white-space characters can separate the number sign and the word "pragma". Following **#pragma**, write any text that the translator can parse as preprocessing tokens. The argument to **#pragma** is subject to macro expansion.

If the compiler finds a pragma that it does not recognize, it issues a warning and continues compilation.

The Microsoft C and C++ compilers recognize the following pragmas:

alloc_text	auto_inline	bss_seg
check_stack	code_seg	comment
component	conform [1]	const_seg
data_seg	deprecated	detect_mismatch
fenv_access	float_control	fp_contract
function	hdrstop	include_alias
init_seg [1]	inline_depth	inline_recursion
intrinsic	loop [1]	make_public

managed	message	
omp	once	
optimize	pack	pointers_to_members [1]
pop_macro	push_macro	region, endregion
runtime_checks	section	setlocale
strict_gs_check	unmanaged	vtordisp [1]
warning		

[1] Supported only by the C++ compiler.

Pragmas and Compiler Options

Some pragmas provide the same functionality as compiler options. When a pragma is encountered in source code, it overrides the behavior specified by the compiler option. For example, if you specified /Zp8, you can override this compiler setting for specific sections of the code with pack:

```
cl /Zp8 ...

<file> - packing is 8
// ...
#pragma pack(push, 1) - packing is now 1
// ...
#pragma pack(pop) - packing is 8
</file>
```

The __pragma() Keyword

Microsoft specific

The compiler also supports the **__pragma** keyword, which has the same functionality as the **#pragma** directive, but can be used inline in a macro definition. The **#pragma** directive cannot be used in a macro definition because the compiler interprets the number sign character ('#') in the directive to be the stringizing operator (#).

The following code example demonstrates how the **__pragma** keyword can be used in a macro. This code is excerpted from the mfcdual.h header in the ACDUAL sample in "Compiler COM Support Samples":

```
#define CATCH_ALL_DUAL \
CATCH(COleException, e) \
{ \
_hr = e->m_sc; \
} \
AND_CATCH_ALL(e) \
{ \
__pragma(warning(push)) \
__pragma(warning(disable:6246)) /*disable _ctlState prefast warning*/ \
AFX_MANAGE_STATE(pThis->m_pModuleState); \
__pragma(warning(pop)) \
_hr = DualHandleException(_riidSource, e); \
} \
END_CATCH_ALL \
return _hr; \
```

End Microsoft specific

See also

C/C++ Preprocessor Reference
C Pragmas
Keywords

alloc_text

4/4/2019 • 2 minutes to read • Edit Online

Names the code section where the specified function definitions are to reside. The pragma must occur between a function declarator and the function definition for the named functions.

Syntax

```
#pragma alloc_text( "
textsection
", function1, ... )
```

Remarks

The **alloc_text** pragma does not handle C++ member functions or overloaded functions. It is applicable only to functions declared with C linkage — that is, functions declared with the **extern "C"** linkage specification. If you attempt to use this pragma on a function with C++ linkage, a compiler error is generated.

Since function addressing using `__based` is not supported, specifying section locations requires the use of the **alloc_text** pragma. The name specified by *textsection* should be enclosed in double quotation marks.

The **alloc_text** pragma must appear after the declarations of any of the specified functions and before the definitions of these functions.

Functions referenced in an **alloc_text** pragma should be defined in the same module as the pragma. If this is not done and an undefined function is later compiled into a different text section, the error may or may not be caught. Although the program will usually run correctly, the function will not be allocated in the intended sections.

Other limitations on **alloc_text** are as follows:

- It cannot be used inside a function.
- It must be used after the function has been declared, but before the function has been defined.

See also

Pragma Directives and the __Pragma Keyword

auto_inline

4/4/2019 • 2 minutes to read • Edit Online

Excludes any functions defined within the range where **off** is specified from being considered as candidates for automatic inline expansion.

Syntax

```
#pragma auto_inline( [{on | off}] )
```

Remarks

To use the **auto_inline** pragma, place it before and immediately after (not in) a function definition. The pragma takes effect at the first function definition after the pragma is seen.

See also

Pragma Directives and the __Pragma Keyword

bss_seg

10/31/2018 • 2 minutes to read • Edit Online

Specifies the segment where uninitialized variables are stored in the .obj file.

Syntax

```
#pragma bss_seg( [ [ { push | pop }, ] [ identifier, ] ] [ "segment-name" [, "segment-class" ] )
```

Parameters

push
(Optional) Puts a record on the internal compiler stack. A *push** can have an *identifier* and *segment-name*.

pop
(Optional) Removes a record from the top of the internal compiler stack.

identifier
(Optional) When used with **push**, assigns a name to the record on the internal compiler stack. *identifier* enables multiple records to be popped with a single **pop** command. When used with **pop**, the directive pops records off the internal stack until *identifier* is removed; if *identifier* is not found on the internal stack, nothing is popped.

"*segment-name*"
(Optional) The name of a segment. When used with **pop**, the stack is popped and *segment-name* becomes the active segment name.

"*segment-class*"
(Optional) Included for compatibility with C++ prior to version 2.0. It is ignored.

Remarks

.Obj files can be viewed with the dumpbin application. The default segment in the .obj file for uninitialized data is .bss. In some cases use of **bss_seg** can speed load times by grouping uninitialized data into one section.

bss_seg with no parameters resets the segment to .bss.

Data allocated using the **bss_seg** pragma does not retain any information about its location.

You can also specify sections for initialized data (data_seg), functions (code_seg), and const variables (const_seg).

See /SECTION for a list of names you should not use when creating a section.

Example

```
// pragma_directive_bss_seg.cpp
int i;                     // stored in .bss
#pragma bss_seg(".my_data1")
int j;                     // stored in .my_data1

#pragma bss_seg(push, stack1, ".my_data2")
int l;                     // stored in .my_data2

#pragma bss_seg(pop, stack1)   // pop stack1 from stack
int m;                     // stored in .my_data1

int main() {
}
```

See also

Pragma Directives and the __Pragma Keyword

check_stack

4/4/2019 • 2 minutes to read • Edit Online

Instructs the compiler to turn off stack probes if `off` (or `-`) is specified, or to turn on stack probes if `on` (or `+`) is specified.

Syntax

```
#pragma check_stack([ {on | off}] )
#pragma check_stack{+ | -}
```

Remarks

If no argument is given, stack probes are treated according to the default. This pragma takes effect at the first function defined after the pragma is seen. Stack probes are neither a part of macros nor of functions that are generated inline.

If you don't give an argument for the **check_stack** pragma, stack checking reverts to the behavior specified on the command line. For more information, see Compiler Reference. The interaction of the `#pragma check_stack` and the /Gs option is summarized in the following table.

Using the check_stack Pragma

SYNTAX	COMPILED WITH /GS OPTION?	ACTION
`#pragma check_stack( )` or `#pragma check_stack`	Yes	Turns off stack checking for functions that follow
`#pragma check_stack( )` or `#pragma check_stack`	No	Turns on stack checking for functions that follow
`#pragma check_stack(on)` or `#pragma check_stack +`	Yes or No	Turns on stack checking for functions that follow
`#pragma check_stack(off)` or `#pragma check_stack -`	Yes or No	Turns off stack checking for functions that follow

See also

Pragma Directives and the __Pragma Keyword

code_seg

4/4/2019 • 2 minutes to read • Edit Online

Specifies the text segment where functions are stored in the .obj file.

Syntax

```
#pragma code_seg( [ [ { push | pop }, ] [ identifier, ] ] [ "segment-name" [, "segment-class" ] )
```

Parameters

push
(Optional) Puts a record on the internal compiler stack. A **push** can have an *identifier* and *segment-name*.

pop
(Optional) Removes a record from the top of the internal compiler stack.

identifier
(Optional) When used with **push**, assigns a name to the record on the internal compiler stack. When used with **pop**, pops records off the internal stack until *identifier* is removed; if *identifier* is not found on the internal stack, nothing is popped.

identifier enables multiple records to be popped with just one **pop** command.

"*segment-name*"
(Optional) The name of a segment. When used with **pop**, the stack is popped and *segment-name* becomes the active text segment name.

"*segment-class*"
(Optional) Ignored, but included for compatibility with versions of C++ earlier than version 2.0.

Remarks

The **code_seg** pragma directive does not control placement of object code generated for instantiated templates, nor code generated implicitly by the compiler—for example, special member functions. We recommend that you use the __declspec(code_seg(...)) attribute instead because it gives you control over placement of all object code. This includes compiler-generated code.

A *segment* in an .obj file is a named block of data that's loaded into memory as a unit. A *text segment* is a segment that contains executable code. In this article, the terms *segment* and *section* are used interchangeably.

The **code_seg** pragma directive tells the compiler to put all subsequent object code from the translation unit into a text segment named *segment-name*. By default, the text segment used for functions in an .obj file is named .text.

A **code_seg** pragma directive without parameters resets the text segment name for the subsequent object code to .text.

You can use the DUMPBIN.EXE application to view .obj files. Versions of DUMPBIN for each supported target architecture are included with Visual Studio.

Example

This example shows how to use the **code_seg** pragma directive to control where object code is put:

```
// pragma_directive_code_seg.cpp
void func1() {                  // stored in .text
}

#pragma code_seg(".my_data1")
void func2() {                  // stored in my_data1
}

#pragma code_seg(push, r1, ".my_data2")
void func3() {                  // stored in my_data2
}

#pragma code_seg(pop, r1)      // stored in my_data1
void func4() {
}

int main() {
}
```

For a list of names that should not be used to create a section, see /SECTION.

You can also specify sections for initialized data (data_seg), uninitialized data (bss_seg), and const variables (const_seg).

See also

code_seg (__declspec)
Pragma Directives and the __Pragma Keyword

comment (C/C++)

4/4/2019 • 3 minutes to read • Edit Online

Places a comment record into an object file or executable file.

Syntax

```
#pragma comment( comment-type [,"commentstring"] )
```

Remarks

The *comment-type* is one of the predefined identifiers, described below, that specifies the type of comment record. The optional *commentstring* is a string literal that provides additional information for some comment types. Because *commentstring* is a string literal, it obeys all the rules for string literals with respect to escape characters, embedded quotation marks (`"`), and concatenation.

compiler

Places the name and version number of the compiler in the object file. This comment record is ignored by the linker. If you supply a *commentstring* parameter for this record type, the compiler generates a warning.

exestr

Places *commentstring* in the object file. At link time this string is placed in the executable file. The string is not loaded into memory when the executable file is loaded; however, it can be found with a program that finds printable strings in files. One use for this comment-record type is to embed a version number or similar information in an executable file.

`exestr` is deprecated and will be removed in a future release; the linker does not process the comment record.

lib

Places a library-search record in the object file. This comment type must be accompanied by a *commentstring* parameter containing the name (and possibly the path) of the library that you want the linker to search. The library name follows the default library-search records in the object file; the linker searches for this library just as if you had named it on the command line provided that the library is not specified with /nodefaultlib. You can place multiple library-search records in the same source file; each record appears in the object file in the same order in which it is encountered in the source file.

If the order of the default library and an added library is important, compiling with the /Zl switch will prevent the default library name from being placed in the object module. A second comment pragma then can be used to insert the name of the default library after the added library. The libraries listed with these pragmas will appear in the object module in the same order they are found in the source code.

linker

Places a linker option in the object file. You can use this comment-type to specify a linker option instead of passing it to the command line or specifying it in the development environment. For example, you can specify the /include option to force the inclusion of a symbol:

```
#pragma comment(linker, "/include:__mySymbol")
```

Only the following (*comment-type*) linker options are available to be passed to the linker identifier:

- /DEFAULTLIB
- /EXPORT
- /INCLUDE
- /MANIFESTDEPENDENCY
- /MERGE
- /SECTION

user

Places a general comment in the object file. The *commentstring* parameter contains the text of the comment. This comment record is ignored by the linker.

The following pragma causes the linker to search for the EMAPI.LIB library while linking. The linker searches first in the current working directory and then in the path specified in the LIB environment variable.

```
#pragma comment( lib, "emapi" )
```

The following pragma causes the compiler to place the name and version number of the compiler in the object file:

```
#pragma comment( compiler )
```

NOTE

For comments that take a *commentstring* parameter, you can use a macro in any place where you would use a string literal, provided that the macro expands to a string literal. You can also concatenate any combination of string literals and macros that expand to string literals. For example, the following statement is acceptable:

```
#pragma comment( user, "Compiled on " __DATE__ " at " __TIME__ )
```

See also

Pragma Directives and the __Pragma Keyword

component

4/9/2019 • 2 minutes to read • Edit Online

Controls the collection of browse information or dependency information from within source files.

Syntax

> **#pragma component(browser,** { **on** | **off** }[**, references** [**,** *name*]] **)**
> **#pragma component(minrebuild, on** | **off)**
> **#pragma component(mintypeinfo, on** | **off)**

Remarks

Browser

You can turn collecting on or off, and you can specify particular names to be ignored as information is collected.

Using on or off controls the collection of browse information from the pragma forward. For example:

```
#pragma component(browser, off)
```

stops the compiler from collecting browse information.

> **NOTE**
> To turn on the collecting of browse information with this pragma, browse information must first be enabled.

The `references` option can be used with or without the *name* argument. Using `references` without *name* turns on or off the collecting of references (other browse information continues to be collected, however). For example:

```
#pragma component(browser, off, references)
```

stops the compiler from collecting reference information.

Using `references` with *name* and `off` prevents references to *name* from appearing in the browse information window. Use this syntax to ignore names and types you are not interested in and to reduce the size of browse information files. For example:

```
#pragma component(browser, off, references, DWORD)
```

ignores references to DWORD from that point forward. You can turn collecting of references to DWORD back on by using `on`:

```
#pragma component(browser, on, references, DWORD)
```

This is the only way to resume collecting references to *name*; you must explicitly turn on any *name* that you have turned off.

To prevent the preprocessor from expanding *name* (such as expanding NULL to 0), put quotes around it:

```
#pragma component(browser, off, references, "NULL")
```

Minimal Rebuild

The deprecated /Gm (Enable Minimal Rebuild) feature requires the compiler to create and store C++ class dependency information, which takes disk space. To save disk space, you can use `#pragma component( minrebuild, off )` whenever you don't need to collect dependency information, for instance, in unchanging header files. Insert `#pragma component(minrebuild, on)` after unchanging classes to turn dependency collection back on.

Reduce Type Information

The `mintypeinfo` option reduces the debugging information for the region specified. The volume of this information is considerable, impacting .pdb and .obj files. You cannot debug classes and structures in the mintypeinfo region. Use of the mintypeinfo option can be helpful to avoid the following warning:

```
LINK : warning LNK4018: too many type indexes in PDB "filename", discarding subsequent type information
```

For more information, see the /Gm (Enable Minimal Rebuild) compiler option.

See also

Pragma Directives and the __Pragma Keyword

conform

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Specifies the run-time behavior of the /Zc:forScope compiler option.

Syntax

> **#pragma conform(** *name* [**, show**] [**,** { **on** | **off** }] [[**,** { **push** | **pop** }] [**,** *identifier*]] **)**

Parameters

name
Specifies the name of the compiler option to be modified. The only valid *name* is `forScope`.

show
(Optional) Causes the current setting of *name* (true or false) to be displayed by means of a warning message during compilation. For example, `#pragma conform(forScope, show)`.

on, **off**
(Optional) Setting *name* to **on** enables the /Zc:forScope compiler option. The default is **off**.

push
(Optional) Pushes the current value of *name* onto the internal compiler stack. If you specify *identifier*, you can specify the **on** or **off** value for *name* to be pushed onto the stack. For example, `#pragma conform(forScope, push, myname, on)`.

pop
(Optional) Sets the value of *name* to the value at the top of the internal compiler stack and then pops the stack. If identifier is specified with **pop**, the stack will be popped back until it finds the record with *identifier*, which will also be popped; the current value for *name* in the next record on the stack becomes the new value for *name*. If you specify **pop** with an *identifier* that is not in a record on the stack, the **pop** is ignored.

identifier
(Optional) Can be included with a **push** or **pop** command. If *identifier* is used, then an **on** or **off** specifier can also be used.

Example

```
// pragma_directive_conform.cpp
// compile with: /W1
// C4811 expected
#pragma conform(forScope, show)
#pragma conform(forScope, push, x, on)
#pragma conform(forScope, push, x1, off)
#pragma conform(forScope, push, x2, off)
#pragma conform(forScope, push, x3, off)
#pragma conform(forScope, show)
#pragma conform(forScope, pop, x1)
#pragma conform(forScope, show)

int main() {}
```

See also

Pragma Directives and the __Pragma Keyword

const_seg

4/4/2019 • 2 minutes to read • Edit Online

Specifies the segment where const variables are stored in the .obj file.

Syntax

```
#pragma const_seg ( [ [ { push | pop}, ] [ identifier, ] ] [ "segment-name" [, "segment-class" ] )
```

Parameters

push
(Optional) Puts a record on the internal compiler stack. A **push** can have an *identifier* and *segment-name*.

pop
(Optional) Removes a record from the top of the internal compiler stack.

identifier
(Optional) When used with **push**, assigns a name to the record on the internal compiler stack. When used with **pop**, pops records off the internal stack until *identifier* is removed; if *identifier* is not found on the internal stack, nothing is popped.

Using *identifier* enables multiple records to be popped with a single **pop** command.

"*segment-name*"
(Optional) The name of a segment. When used with **pop**, the stack is popped and *segment-name* becomes the active segment name.

"*segment-class*"
(Optional) Included for compatibility with C++ prior to version 2.0. It is ignored.

Remarks

The meaning of the terms *segment* and *section* are interchangeable in this topic.

OBJ files can be viewed with the dumpbin application. The default segment in the .obj file for `const` variables is .rdata. Some `const` variables, such as scalars, are automatically inlined into the code stream. Inlined code will not appear in .rdata.

Defining an object requiring dynamic initialization in a `const_seg` results in undefined behavior.

`#pragma const_seg` with no parameters resets the segment to .rdata.

Example

```
// pragma_directive_const_seg.cpp
// compile with: /EHsc
#include <iostream>

const int i = 7;               // inlined, not stored in .rdata
const char sz1[]= "test1";     // stored in .rdata

#pragma const_seg(".my_data1")
const char sz2[]= "test2";     // stored in .my_data1

#pragma const_seg(push, stack1, ".my_data2")
const char sz3[]= "test3";     // stored in .my_data2

#pragma const_seg(pop, stack1) // pop stack1 from stack
const char sz4[]= "test4";     // stored in .my_data1

int main() {
    using namespace std;
   // const data must be referenced to be put in .obj
   cout << sz1 << endl;
   cout << sz2 << endl;
   cout << sz3 << endl;
   cout << sz4 << endl;
}
```

```
test1
test2
test3
test4
```

Comments

See /SECTION for a list of names you should not use when creating a section.

You can also specify sections for initialized data (data_seg), uninitialized data (bss_seg), and functions (code_seg).

See also

Pragma Directives and the __Pragma Keyword

data_seg

10/31/2018 • 2 minutes to read • Edit Online

Specifies the data segment where initialized variables are stored in the .obj file.

Syntax

```
#pragma data_seg( [ [ { push | pop }, ] [ identifier, ] ] [ "segment-name" [, "segment-class" ] )
```

Parameters

push
(Optional) Puts a record on the internal compiler stack. A **push** can have an *identifier* and *segment-name*.

pop
(Optional) Removes a record from the top of the internal compiler stack.

identifier
(Optional) When used with **push**, assigns a name to the record on the internal compiler stack. When used with **pop**, pops records off the internal stack until *identifier* is removed; if *identifier* is not found on the internal stack, nothing is popped.

identifier enables multiple records to be popped with a single **pop** command.

"*segment-name*"
(Optional) The name of a segment. When used with **pop**, the stack is popped and *segment-name* becomes the active segment name.

"*segment-class*"
(Optional) Included for compatibility with C++ prior to version 2.0. It is ignored.

Remarks

The meaning of the terms *segment* and *section* are interchangeable in this topic.

OBJ files can be viewed with the dumpbin application. The default segment in the .obj file for initialized variables is .data. Variables that are uninitialized are considered to be initialized to zero and are stored in .bss.

data_seg with no parameters resets the segment to .data.

Example

```
// pragma_directive_data_seg.cpp
int h = 1;                     // stored in .data
int i = 0;                     // stored in .bss
#pragma data_seg(".my_data1")
int j = 1;                     // stored in .my_data1

#pragma data_seg(push, stack1, ".my_data2")
int l = 2;                     // stored in .my_data2

#pragma data_seg(pop, stack1)   // pop stack1 off the stack
int m = 3;                     // stored in .my_data1

int main() {
}
```

Data allocated using **data_seg** does not retain any information about its location.

See /SECTION for a list of names you should not use when creating a section.

You can also specify sections for const variables (const_seg), uninitialized data (bss_seg), and functions (code_seg).

See also

Pragma Directives and the __Pragma Keyword

deprecated (C/C++)

4/4/2019 • 2 minutes to read • Edit Online

The **deprecated** pragma lets you indicate that a function, type, or any other identifier may no longer be supported in a future release or should no longer be used.

NOTE

For information about the C++14 `[[deprecated]]` attribute, and guidance on when to use that attribute vs the Microsoft declspec or pragma, see C++ Standard Attributes attribute.

Syntax

```
#pragma deprecated( identifier1 [,identifier2, ...] )
```

Remarks

When the compiler encounters an identifier specified by a **deprecated** pragma, it issues compiler warning C4995.

You can deprecate macro names. Place the macro name in quotes or else macro expansion will occur.

Because the **deprecated** pragma works on all matching identifiers, and does not take signatures into account, it is not the best option for deprecating specific versions of overloaded functions. Any matching function name that is brought into scope triggers the warning.

We recommend you use the C++14 `[[deprecated]]` attribute, when possible, instead of the **deprecated** pragma. The Microsoft-specific __declspec(deprecated) declaration modifier is also a better choice in many cases than the **deprecated** pragma. The `[[deprecated]]` attribute and `__declspec(deprecated)` modifier allow you to specify deprecated status for particular forms of overloaded functions. The diagnostic warning only appears on references to the specific overloaded function the attribute or modifier applies to.

Example

```
// pragma_directive_deprecated.cpp
// compile with: /W3
#include <stdio.h>
void func1(void) {
}

void func2(void) {
}

int main() {
   func1();
   func2();
   #pragma deprecated(func1, func2)
   func1();   // C4995
   func2();   // C4995
}
```

The following sample shows how to deprecate a class:

```
// pragma_directive_deprecated2.cpp
// compile with: /W3
#pragma deprecated(X)
class X {  // C4995
public:
   void f(){}
};

int main() {
   X x;   // C4995
}
```

See also

Pragma Directives and the __Pragma Keyword

detect_mismatch

4/4/2019 • 2 minutes to read • Edit Online

Places a record in an object. The linker checks these records for potential mismatches.

Syntax

```
#pragma detect_mismatch("name", "value")
```

Remarks

When you link the project, the linker throws a `LNK2038` error if the project contains two objects that have the same `name` but each has a different `value`. Use this pragma to prevent inconsistent object files from linking.

Both name and value are string literals and obey the rules for string literals with respect to escape characters and concatenation. They are case-sensitive and cannot contain a comma, equal sign, quotation marks, or the **null** character.

Example

This example creates two files that have different version numbers for the same version label.

```
// pragma_directive_detect_mismatch_a.cpp
#pragma detect_mismatch("myLib_version", "9")
int main ()
{
   return 0;
}

// pragma_directive_detect_mismatch_b.cpp
#pragma detect_mismatch("myLib_version", "1")
```

If you compile both of these files by using the command line `cl pragma_directive_detect_mismatch_a.cpp pragma_directive_detect_mismatch_b.cpp`, you will receive the error `LNK2038`.

See also

Pragma Directives and the __Pragma Keyword

execution_character_set

4/4/2019 • 2 minutes to read • Edit Online

Specifies the execution character set used for string and character literals. This directive is not needed for literals marked with the u8 prefix.

Syntax

```
#pragma execution_character_set("target")
```

Parameters

target
Specifies the target execution character set. Currently the only target execution set supported is "utf-8".

Remarks

This compiler directive is obsolete starting in Visual Studio 2015 Update 2. We recommend that you use the `/execution-charset:utf-8` or `/utf-8` compiler options together with using the `u8` prefix on narrow character and string literals that contain extended characters. For more information about the `u8` prefix, see String and Character Literals. For more information about the compiler options, see /execution-charset (Set Execution Character Set) and /utf-8 (Set Source and Executable character sets to UTF-8).

The `#pragma execution_character_set("utf-8")` directive tells the compiler to encode narrow character and narrow string literals in your source code as UTF-8 in the executable. This output encoding is independent of the source file encoding used.

By default, the compiler encodes narrow characters and narrow strings by using the current code page as the execution character set. This means that Unicode or DBCS characters in a literal that are outside the range of the current code page are converted to the default replacement character in the output. Unicode and DBCS characters are truncated to their low-order byte. This is almost certainly not what you intend. You can specify UTF-8 encoding for literals in the source file by using a `u8` prefix. The compiler passes these UTF-8 encoded strings to the output unchanged. Narrow character literals prefixed by using u8 must fit in one byte, or they are truncated on output.

By default, Visual Studio uses the current code page as the source character set used to interpret your source code for output. When a file is read in, Visual Studio interprets it according to the current code page unless the file code page was set, or unless a byte-order mark (BOM) or UTF-16 characters are detected at the beginning of the file. Because UTF-8 can't be set as the current code page, when the automatic detection encounters source files encoded as UTF-8 without a BOM, Visual Studio assumes that they are encoded by using the current code page. Characters in the source file that are outside the range of the specified or automatically detected code page can cause compiler warnings and errors.

See also

Pragma Directives and the __Pragma Keyword
/execution-charset (Set Execution Character Set)
/utf-8 (Set Source and Executable character sets to UTF-8)

fenv_access

10/31/2018 • 2 minutes to read • Edit Online

Disables (**on**) or enables (**off**) optimizations that could change floating-point environment flag tests and mode changes.

Syntax

> **#pragma fenv_access (** { **on** | **off** } **)**

Remarks

By default, **fenv_access** is **off**. If the compiler can assume that your code does not access or manipulate the floating-point environment, then it can perform many floating-point code optimizations. Set **fenv_access** to **on** to inform the compiler that your code accesses the floating-point environment to test status flags, exceptions, or to set control mode flags. The compiler disables these optimizations so that your code can access the floating-point environment consistently.

For more information on floating-point behavior, see /fp (Specify Floating-Point Behavior).

The kinds of optimizations that are subject to **fenv_access** are:

- Global common subexpression elimination
- Code motion
- Constant folding

Other floating-point pragmas include:

- float_control
- fp_contract

Examples

This example sets **fenv_access** to **on** to set the floating-point control register for 24-bit precision:

```
// pragma_directive_fenv_access_x86.cpp
// compile with: /O2
// processor: x86
#include <stdio.h>
#include <float.h>
#include <errno.h>
#pragma fenv_access (on)

int main() {
   double z, b = 0.1, t = 0.1;
   unsigned int currentControl;
   errno_t err;

   err = _controlfp_s(&currentControl, _PC_24, _MCW_PC);
   if (err != 0) {
      printf_s("The function _controlfp_s failed!\n");
      return -1;
   }
   z = b * t;
   printf_s ("out=%.15e\n",z);
}
```

```
out=9.999999776482582e-003
```

If you comment out `#pragma fenv_access (on)` from the previous sample, note that the output is different because the compiler does compile-time evaluation, which does not use the control mode.

```
// pragma_directive_fenv_access_2.cpp
// compile with: /O2
#include <stdio.h>
#include <float.h>

int main() {
   double z, b = 0.1, t = 0.1;
   unsigned int currentControl;
   errno_t err;

   err = _controlfp_s(&currentControl, _PC_24, _MCW_PC);
   if (err != 0) {
      printf_s("The function _controlfp_s failed!\n");
      return -1;
   }
   z = b * t;
   printf_s ("out=%.15e\n",z);
}
```

```
out=1.000000000000000e-002
```

See also

Pragma Directives and the __Pragma Keyword

float_control

4/4/2019 • 2 minutes to read • Edit Online

Specifies floating-point behavior for a function.

Syntax

> **#pragma float_control** [**(** [*value* **,** *setting* [**, push**]] | [**push** | **pop**] **)**]

Options

value, *setting* [, **push**]
Specifies floating-point behavior. *value* can be **precise**, **strict**, or **except**. For more information, see /fp (Specify Floating-Point Behavior). The *setting* can either be **on** or **off**.

If *value* is **strict**, the settings for both **strict** and **except** are specified by *setting*. **except** can only be set to **on** when **precise** or **strict** is also set to **on**.

If the optional **push** token is added, the current setting for *value* is pushed on to the internal compiler stack.

push
Push the current **float_control** setting on to the internal compiler stack

pop
Removes the **float_control** setting from the top of the internal compiler stack and makes that the new **float_control** setting.

Remarks

You cannot use **float_control** to turn **precise** off when **except** is on. Similarly, **precise** cannot be turned off when fenv_access is on. To go from strict model to a fast model by using the **float_control** pragma, use the following code:

```
#pragma float_control(except, off)
#pragma fenv_access(off)
#pragma float_control(precise, off)
```

To go from fast model to a strict model with the **float_control** pragma, use the following code:

```
#pragma float_control(precise, on)
#pragma fenv_access(on)
#pragma float_control(except, on)
```

If no options are specified, **float_control** has no effect.

Other floating-point pragmas include:

- fenv_access
- fp_contract

Example

The following sample shows how to catch an overflow floating-point exception by using pragma **float_control**.

```
// pragma_directive_float_control.cpp
// compile with: /EHa
#include <stdio.h>
#include <float.h>

double func( ) {
   return 1.1e75;
}

#pragma float_control (except, on)

int main( ) {
   float u[1];
   unsigned int currentControl;
   errno_t err;

   err = _controlfp_s(&currentControl, ~_EM_OVERFLOW, _MCW_EM);
   if (err != 0)
      printf_s("_controlfp_s failed!\n");

   try  {
      u[0] = func();
      printf_s ("Fail");
      return(1);
   }

   catch (...)  {
      printf_s ("Pass");
      return(0);
   }
}
```

```
Pass
```

See also

Pragma Directives and the __Pragma Keyword

fp_contract

10/31/2018 • 2 minutes to read • Edit Online

Determines whether floating-point contraction takes place. A floating-point contraction is an instruction such as FMA (Fused-Multiply-Add) that combines two separate floating point operations into a single instruction. Use of these instructions can affect floating-point precision, because instead of rounding after each operation, the processor may round only once after both operations.

Syntax

> **#pragma fp_contract (** { **on** | **off** } **)**

Remarks

By default, **fp_contract** is **on**. This tells the compiler to use floating-point contraction instructions where possible. Set **fp_contract** to **off** to preserve individual floating-point instructions.

For more information on floating-point behavior, see /fp (Specify Floating-Point Behavior).

Other floating-point pragmas include:

- fenv_access
- float_control

Example

The code generated from this sample does not use a fused-multiply-add instruction even when it is available on the target processor. If you comment out `#pragma fp_contract (off)`, the generated code may use a fused-multiply-add instruction if it is available.

```
// pragma_directive_fp_contract.cpp
// on x86 and x64 compile with: /O2 /fp:fast /arch:AVX2
// other platforms compile with: /O2

#include <stdio.h>

// remove the following line to enable FP contractions
#pragma fp_contract (off)

int main() {
   double z, b, t;

   for (int i = 0; i < 10; i++) {
      b = i * 5.5;
      t = i * 56.025;

      z = t * i + b;
      printf("out = %.15e\n", z);
   }
}
```

```
out = 0.000000000000000e+00
out = 6.152500000000000e+01
out = 2.351000000000000e+02
out = 5.207249999999999e+02
out = 9.184000000000000e+02
out = 1.428125000000000e+03
out = 2.049900000000000e+03
out = 2.783725000000000e+03
out = 3.629600000000000e+03
out = 4.587525000000000e+03
```

See also

Pragma Directives and the __Pragma Keyword

function (C/C++)

4/4/2019 • 2 minutes to read • Edit Online

Specifies that calls to functions specified in the pragma's argument list be generated.

Syntax

```
#pragma function( function1 [, function2, ...] )
```

Remarks

If you use the `intrinsic` pragma (or /Oi) to tell the compiler to generate intrinsic functions (intrinsic functions are generated as inline code, not as function calls), you can use the **function** pragma to explicitly force a function call. Once a function pragma is seen, it takes effect at the first function definition containing a specified intrinsic function. The effect continues to the end of the source file or to the appearance of an `intrinsic` pragma specifying the same intrinsic function. The **function** pragma can be used only outside of a function — at the global level.

For lists of the functions that have intrinsic forms, see #pragma intrinsic.

Example

```
// pragma_directive_function.cpp
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// use intrinsic forms of memset and strlen
#pragma intrinsic(memset, strlen)

// Find first word break in string, and set remaining
// chars in string to specified char value.
char *set_str_after_word(char *string, char ch) {
   int i;
   int len = strlen(string);  /* NOTE: uses intrinsic for strlen */

   for(i = 0; i < len; i++) {
      if (isspace(*(string + i)))
         break;
   }

   for(; i < len; i++)
      *(string + i) = ch;

   return string;
}

// do not use strlen intrinsic
#pragma function(strlen)

// Set all chars in string to specified char value.
char *set_str(char *string, char ch) {
   // Uses intrinsic for memset, but calls strlen library function
   return (char *) memset(string, ch, strlen(string));
}

int main() {
   char *str = (char *) malloc(20 * sizeof(char));

   strcpy_s(str, sizeof("Now is the time"), "Now is the time");
   printf("str is '%s'\n", set_str_after_word(str, '*'));
   printf("str is '%s'\n", set_str(str, '!'));
}
```

```
str is 'Now************'
str is '!!!!!!!!!!!!!!!'
```

See also

Pragma Directives and the __Pragma Keyword

hdrstop

4/4/2019 • 2 minutes to read • Edit Online

Gives you additional control over precompilation file names and over the location at which the compilation state is saved.

Syntax

```
#pragma hdrstop [( "filename" )]
```

Remarks

The *filename* is the name of the precompiled header file to use or create (depending on whether /Yu or /Yc is specified). If *filename* does not contain a path specification, the precompiled header file is assumed to be in the same directory as the source file.

If a C or C++ file contains a **hdrstop** pragma when compiled with `/Yc`, the compiler saves the state of the compilation up to the location of the pragma. The compiled state of any code that follows the pragma is not saved.

Use *filename* to name the precompiled header file in which the compiled state is saved. A space between **hdrstop** and *filename* is optional. The file name specified in the **hdrstop** pragma is a string and is therefore subject to the constraints of any C or C++ string. In particular, you must enclose it in quotation marks and use the escape character (backslash) to specify directory names. For example:

```
#pragma hdrstop( "c:\\projects\\include\\myinc.pch" )
```

The name of the precompiled header file is determined according to the following rules, in order of precedence:

1. The argument to the `/Fp` compiler option
2. The *filename* argument to `#pragma hdrstop`
3. The base name of the source file with a .PCH extension

For the `/Yc` and `/Yu` options, if neither of the two compilation options nor the **hdrstop** pragma specifies a file name, the base name of the source file is used as the base name of the precompiled header file.

You can also use preprocessing commands to perform macro replacement as follows:

```
#define INCLUDE_PATH "c:\\progra~`1\\devstsu~1\\vc\\include\\"
#define PCH_FNAME "PROG.PCH"
.
.
.
#pragma hdrstop( INCLUDE_PATH PCH_FNAME )
```

The following rules govern where the **hdrstop** pragma can be placed:

- It must appear outside any data or function declaration or definition.
- It must be specified in the source file, not within a header file.

Example

```
#include <windows.h>                 // Include several files
#include "myhdr.h"

__inline Disp( char *szToDisplay )   // Define an inline function
{
    ...                              // Some code to display string
}
#pragma hdrstop
```

In this example, the **hdrstop** pragma appears after two files have been included and an inline function has been defined. This might, at first, seem to be an odd placement for the pragma. Consider, however, that using the manual precompilation options, `/Yc` and `/Yu`, with the **hdrstop** pragma makes it possible for you to precompile entire source files — even inline code. The Microsoft compiler does not limit you to precompiling only data declarations.

See also

Pragma Directives and the __Pragma Keyword

include_alias

4/4/2019 • 2 minutes to read • Edit Online

Specifies that when *alias_filename* is found in a `#include` directive, the compiler substitutes *actual_filename* in its place.

Syntax

> #pragma include_alias("*alias_filename*", "*actual_filename*") #pragma include_alias(<*alias_filename*>, <*actual_filename*>)

Remarks

The **include_alias** pragma directive allows you to substitute files that have different names or paths for the file names included by source files. For example, some file systems allow longer header filenames than the 8.3 FAT file system limit. The compiler cannot simply truncate the longer names to 8.3, because the first eight characters of the longer header filenames may not be unique. Whenever the compiler encounters the *alias_filename* string, it substitutes *actual_filename*, and looks for the header file *actual_filename* instead. This pragma must appear before the corresponding `#include` directives. For example:

```
// First eight characters of these two files not unique.
#pragma include_alias( "AppleSystemHeaderQuickdraw.h", "quickdra.h" )
#pragma include_alias( "AppleSystemHeaderFruit.h", "fruit.h" )

#pragma include_alias( "GraphicsMenu.h", "gramenu.h" )

#include "AppleSystemHeaderQuickdraw.h"
#include "AppleSystemHeaderFruit.h"
#include "GraphicsMenu.h"
```

The alias being searched for must match the specification exactly, in case as well as in spelling and in use of double quotation marks or angle brackets. The **include_alias** pragma performs simple string matching on the filenames; no other filename validation is performed. For example, given the following directives,

```
#pragma include_alias("mymath.h", "math.h")
#include "./mymath.h"
#include "sys/mymath.h"
```

no aliasing (substitution) is performed, since the header file strings do not match exactly. Also, header filenames used as arguments to the `/Yu` and `/Yc` compiler options, or the `hdrstop` pragma, are not substituted. For example, if your source file contains the following directive,

```
#include <AppleSystemHeaderStop.h>
```

the corresponding compiler option should be

> /YcAppleSystemHeaderStop.h

You can use the **include_alias** pragma to map any header filename to another. For example:

```
#pragma include_alias( "api.h", "c:\version1.0\api.h" )
#pragma include_alias( <stdio.h>, <newstdio.h> )
#include "api.h"
#include <stdio.h>
```

Do not mix filenames enclosed in double quotation marks with filenames enclosed in angle brackets. For example, given the above two `#pragma include_alias` directives, the compiler performs no substitution on the following `#include` directives:

```
#include <api.h>
#include "stdio.h"
```

Furthermore, the following directive generates an error:

```
#pragma include_alias(<header.h>, "header.h")  // Error
```

Note that the filename reported in error messages, or as the value of the predefined `__FILE__` macro, is the name of the file after the substitution has been performed. For example, see the output after the following directives:

```
#pragma include_alias( "VERYLONGFILENAME.H", "myfile.h" )
#include "VERYLONGFILENAME.H"
```

An error in VERYLONGFILENAME.H produces the following error message:

```
myfile.h(15) : error C2059 : syntax error
```

Also note that transitivity is not supported. Given the following directives,

```
#pragma include_alias( "one.h", "two.h" )
#pragma include_alias( "two.h", "three.h" )
#include "one.h"
```

the compiler searches for the file two.h rather than three.h.

See also

Pragma Directives and the __Pragma Keyword

init_seg

4/4/2019 • 3 minutes to read • Edit Online

C++ Specific

Specifies a keyword or code section that affects the order in which startup code is executed.

Syntax

```
#pragma init_seg({ compiler | lib | user | "section-name" [, func-name]} )
```

Remarks

The meaning of the terms *segment* and *section* are interchangeable in this topic.

Because initialization of global static objects can involve executing code, you must specify a keyword that defines when the objects are to be constructed. It is particularly important to use the **init_seg** pragma in dynamic-link libraries (DLLs) or libraries requiring initialization.

The options to the **init_seg** pragma are:

compiler
Reserved for Microsoft C run-time library initialization. Objects in this group are constructed first.

lib
Available for third-party class-library vendors' initializations. Objects in this group are constructed after those marked as *compiler* but before any others.

user
Available to any user. Objects in this group are constructed last.

section-name Allows explicit specification of the initialization section. Objects in a user-specified *section-name* are not implicitly constructed; however, their addresses are placed in the section named by *section-name*.

The section name you give will contain pointers to helper functions that will construct the global objects declared in that module after the pragma.

For a list of names you should not use when creating a section, see /SECTION.

func-name Specifies a function to be called in place of `atexit` when the program exits. This helper function also calls atexit with a pointer to the destructor for the global object. If you specify a function identifier in the pragma of the form,

```
int __cdecl myexit (void (__cdecl *pf)(void))
```

then your function will be called instead of the C run-time library's `atexit`. This allows you to build a list of the destructors that will need to be called when you are ready to destroy the objects.

If you need to defer initialization (for example, in a DLL) you may choose to specify the section name explicitly. You must then call the constructors for each static object.

There are no quotes around the identifier for the `atexit` replacement.

Your objects will still be placed in the sections defined by the other XXX_seg pragmas.

The objects that are declared in the module will not be automatically initialized by the C run-time. You will need to do that yourself.

By default, `init_seg` sections are read only. If the section name is .CRT, the compiler will silently change the attribute to read only, even if it is marked as read, write.

You cannot specify **init_seg** more than once in a translation unit.

Even if your object does not have a user-defined constructor, a constructor not explicitly defined in code, the compiler may generate one (for example to bind v-table pointers). Therefore, your code will have to call the compiler-generated constructor.

Example

```
// pragma_directive_init_seg.cpp
#include <stdio.h>
#pragma warning(disable : 4075)

typedef void (__cdecl *PF)(void);
int cxpf = 0;   // number of destructors we need to call
PF pfx[200];    // pointers to destructors.

int myexit (PF pf) {
   pfx[cxpf++] = pf;
   return 0;
}

struct A {
   A() { puts("A()"); }
   ~A() { puts("~A()"); }
};

// ctor & dtor called by CRT startup code
// because this is before the pragma init_seg
A aaaa;

// The order here is important.
// Section names must be 8 characters or less.
// The sections with the same name before the $
// are merged into one section. The order that
// they are merged is determined by sorting
// the characters after the $.
// InitSegStart and InitSegEnd are used to set
// boundaries so we can find the real functions
// that we need to call for initialization.

#pragma section(".mine$a", read)
__declspec(allocate(".mine$a")) const PF InitSegStart = (PF)1;

#pragma section(".mine$z",read)
__declspec(allocate(".mine$z")) const PF InitSegEnd = (PF)1;

// The comparison for 0 is important.
// For now, each section is 256 bytes. When they
// are merged, they are padded with zeros. You
// can't depend on the section being 256 bytes, but
// you can depend on it being padded with zeros.

void InitializeObjects () {
   const PF *x = &InitSegStart;
   for (++x ; x < &InitSegEnd ; ++x)
      if (*x) (*x)();
}
```

```
void DestroyObjects () {
   while (cxpf>0) {
      --cxpf;
      (pfx[cxpf])();
   }
}

// by default, goes into a read only section
#pragma init_seg(".mine$m", myexit)

A bbbb;
A cccc;

int main () {
   InitializeObjects();
   DestroyObjects();
}
```

```
A()
A()
A()
~A()
~A()
~A()
```

See also

Pragma Directives and the __Pragma Keyword

inline_depth

4/4/2019 • 2 minutes to read • Edit Online

Specifies the inline heuristic search depth, such that, no function will be inlined if it is at a depth (in the call graph) greater than *n*.

Syntax

```
#pragma inline_depth( [n] )
```

Remarks

This pragma controls the inlining of functions marked inline and __inline or inlined automatically under the `/Ob2` option.

n can be a value between 0 and 255, where 255 means unlimited depth in the call graph, and zero inhibits inline expansion. When *n* is not specified, the default (254) is used.

The **inline_depth** pragma controls the number of times a series of function calls can be expanded. For example, if the inline depth is four, and if A calls B and B then calls C, all three calls will be expanded inline. However, if the closest inline expansion is two, only A and B are expanded, and C remains as a function call.

To use this pragma, you must set the `/Ob` compiler option to 1 or 2. The depth set using this pragma takes effect at the first function call after the pragma.

The inline depth can be decreased during expansion but not increased. If the inline depth is six and during expansion the preprocessor encounters an **inline_depth** pragma with a value of eight, the depth remains six.

The **inline_depth** pragma has no effect on functions marked with `__forceinline`.

NOTE

Recursive functions can be substituted inline to a maximum depth of 16 calls.

See also

Pragma Directives and the __Pragma Keyword
inline_recursion

inline_recursion

4/4/2019 • 2 minutes to read • Edit Online

Controls the inline expansion of direct or mutually recursive function calls.

Syntax

```
#pragma inline_recursion( [{on | off}] )
```

Remarks

Use this pragma to control functions marked as inline and __inline or functions that the compiler automatically expands under the `/Ob2` option. Use of this pragma requires an /Ob compiler option setting of either 1 or 2. The default state for **inline_recursion** is off. This pragma takes effect at the first function call after the pragma is seen and does not affect the definition of the function.

The **inline_recursion** pragma controls how recursive functions are expanded. If **inline_recursion** is off, and if an inline function calls itself (either directly or indirectly), the function is expanded only one time. If **inline_recursion** is on, the function is expanded multiple times until it reaches the value set with the inline_depth pragma, the default value for recursive functions that is defined by the `inline_depth` pragma, or a capacity limit.

See also

Pragma Directives and the __Pragma Keyword
inline_depth
/Ob (Inline Function Expansion)

intrinsic

10/31/2018 • 2 minutes to read • Edit Online

Specifies that calls to functions specified in the pragma's argument list are intrinsic.

Syntax

```
#pragma intrinsic( function1 [, function2, ...] )
```

Remarks

The **intrinsic** pragma tells the compiler that a function has known behavior. The compiler may call the function and not replace the function call with inline instructions, if it will result in better performance.

The library functions with intrinsic forms are listed below. Once an **intrinsic** pragma is seen, it takes effect at the first function definition containing a specified intrinsic function. The effect continues to the end of the source file or to the appearance of a `function` pragma specifying the same intrinsic function. The **intrinsic** pragma can be used only outside of a function definition — at the global level.

The following functions have intrinsic forms and the intrinsic forms are used when you specify /Oi:

_disable	_outp	fabs	strcmp
_enable	_outpw	labs	strcpy
_inp	_rotl	memcmp	strlen
_inpw	_rotr	memcpy	
_lrotl	_strset	memset	
_lrotr	abs	strcat	

Programs that use intrinsic functions are faster because they do not have the overhead of function calls but may be larger due to the additional code generated.

x86 Specific

The `_disable` and `_enable` intrinsics generate kernel-mode instructions to disable/enable interrupts and could be useful in kernel-mode drivers.

Example

Compile the following code from the command line with `cl -c -FAs sample.c` and look at sample.asm to see that they turn into x86 instructions CLI and STI:

```
// pragma_directive_intrinsic.cpp
// processor: x86
#include <dos.h>   // definitions for _disable, _enable
#pragma intrinsic(_disable)
#pragma intrinsic(_enable)
void f1(void) {
   _disable();
   // do some work here that should not be interrupted
   _enable();
}
int main() {
}
```

End x86 Specific

The floating-point functions listed below do not have true intrinsic forms. Instead they have versions that pass arguments directly to the floating-point chip rather than pushing them onto the program stack:

acos	cosh	pow	tanh
asin	fmod	sinh	

The floating-point functions listed below have true intrinsic forms when you specify /Oi, /Og, and /fp:fast (or any option that includes /Og: /Ox, /O1, and /O2):

atan	exp	log10	sqrt
atan2	log	sin	tan
cos			

You can use /fp:strict or /Za to override generation of true intrinsic floating-point options. In this case, the functions are generated as library routines that pass arguments directly to the floating-point chip instead of pushing them onto the program stack.

See #pragma function for information and an example on how to enable/disable intrinsics for a block of source text.

See also

Pragma Directives and the __Pragma Keyword
Compiler Intrinsics

loop

4/4/2019 • 2 minutes to read • Edit Online

Controls how loop code is to be considered by the auto-parallelizer, and/or excludes a loop from consideration by the auto-vectorizer.

Syntax

```
#pragma loop( hint_parallel(n) )
#pragma loop( no_vector )
#pragma loop( ivdep )
```

Parameters

hint_parallel(n)
Hints to the compiler that this loop should be parallelized across *n* threads, where *n* is a positive integer literal or zero. If *n* is zero, the maximum number of threads is used at run time. This is a hint to the compiler, not a command, and there is no guarantee that the loop will be parallelized. If the loop has data dependencies, or structural issues—for example, the loop stores to a scalar that's used beyond the loop body—then the loop will not be parallelized.

The compiler ignores this option unless the /Qpar compiler switch is specified.

no_vector
By default, the auto-vectorizer is on and will attempt to vectorize all loops that it evaluates as benefitting from it. Specify this pragma to disable the auto-vectorizer for the loop that follows it.

ivdep
Hints to the compiler to ignore vector dependencies for this loop. Use this in conjunction with *hint_parallel*.

Remarks

To use the **loop** pragma, place it immediately before—not in—a loop definition. The pragma takes effect for the scope of the loop that follows it. You can apply multiple pragmas to a loop, in any order, but you must state each one in a separate pragma statement.

See also

Auto-Parallelization and Auto-Vectorization
Pragma Directives and the __Pragma Keyword

make_public

4/4/2019 • 2 minutes to read • Edit Online

Indicates that a native type should have public assembly accessibility.

Syntax

```
#pragma make_public(type)
```

Parameters

type is the name of the type you want to have public assembly accessibility.

Remarks

make_public is useful for when the native type you want to reference is from a .h file that you cannot change. If you want to use the native type in the signature of a public function in a type with public assembly visibility, the native type must also have public assembly accessibility or the compiler will issue a warning.

make_public must be specified at global scope and is only in effect from the point at which it is declared through to the end of the source code file.

The native type may be implicitly or explicitly private; see Type Visibility for more information.

Examples

The following sample is the contents of a .h file that contains the definitions for two native structs.

```
// make_public_pragma.h
struct Native_Struct_1 { int i; };
struct Native_Struct_2 { int i; };
```

The following code sample consumes the header file and shows that unless you explicitly mark the native structs as public, using **make_public**, the compiler will generate a warning when you attempt to use the native structs in the signature of public function in a public managed type.

```
// make_public_pragma.cpp
// compile with: /c /clr /W1
#pragma warning (default : 4692)
#include "make_public_pragma.h"
#pragma make_public(Native_Struct_1)

public ref struct A {
   void Test(Native_Struct_1 u) {u.i = 0;}   // OK
   void Test(Native_Struct_2 u) {u.i = 0;}   // C4692
};
```

See also

Pragma Directives and the __Pragma Keyword

managed, unmanaged

4/4/2019 • 2 minutes to read • Edit Online

Enable function-level control for compiling functions as managed or unmanaged.

Syntax

```
#pragma managed
#pragma unmanaged
#pragma managed([push,] on | off)
#pragma managed(pop)
```

Remarks

The /clr compiler option provides module-level control for compiling functions either as managed or unmanaged.

An unmanaged function will be compiled for the native platform, and execution of that portion of the program will be passed to the native platform by the common language runtime.

Functions are compiled as managed by default when `/clr` is used.

When applying these pragmas:

- Add the pragma preceding a function but not within a function body.
- Add the pragma after `#include` statements. Do not use these pragmas before `#include` statements.

The compiler ignores the **managed** and **unmanaged** pragmas if `/clr` is not used in the compilation.

When a template function is instantiated, the pragma state at the time of definition for the template determines if it is managed or unmanaged.

For more information, see Initialization of Mixed Assemblies.

Example

```cpp
// pragma_directives_managed_unmanaged.cpp
// compile with: /clr
#include <stdio.h>

// func1 is managed
void func1() {
   System::Console::WriteLine("In managed function.");
}

// #pragma unmanaged
// push managed state on to stack and set unmanaged state
#pragma managed(push, off)

// func2 is unmanaged
void func2() {
   printf("In unmanaged function.\n");
}

// #pragma managed
#pragma managed(pop)

// main is managed
int main() {
   func1();
   func2();
}
```

```
In managed function.
In unmanaged function.
```

See also

Pragma Directives and the __Pragma Keyword

message

4/4/2019 • 2 minutes to read • Edit Online

Sends a string literal to the standard output without terminating the compilation.

Syntax

```
#pragma message( messagestring )
```

Remarks

A typical use of the **message** pragma is to display informational messages at compile time.

The *messagestring* parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination.

If you use a predefined macro in the **message** pragma, the macro should return a string, else you will have to convert the output of the macro to a string.

The following code fragment uses the **message** pragma to display messages during compilation:

```
// pragma_directives_message1.cpp
// compile with: /LD
#if _M_IX86 >= 500
#pragma message("_M_IX86 >= 500")
#endif

#pragma message("")

#pragma message( "Compiling " __FILE__ )
#pragma message( "Last modified on " __TIMESTAMP__ )

#pragma message("")

// with line number
#define STRING2(x) #x
#define STRING(x) STRING2(x)

#pragma message (__FILE__ "[" STRING(__LINE__) "]: test")

#pragma message("")
```

See also

Pragma Directives and the __Pragma Keyword

omp

4/4/2019 • 2 minutes to read • Edit Online

Takes one or more OpenMP directives, along with any optional directive clauses.

Syntax

```
#pragma omp directive
```

Remarks

See OpenMP Directives for more information.

See also

Pragma Directives and the __Pragma Keyword

once

4/4/2019 • 2 minutes to read • Edit Online

Specifies that the file will be included (opened) only once by the compiler when compiling a source code file.

Syntax

```
#pragma once
```

Remarks

The use of `#pragma once` can reduce build times as the compiler will not open and read the file after the first `#include` of the file in the translation unit. This is referred to as *multiple-include optimization*. It has an effect similar to the `#include guard` idiom, which uses preprocessor macro definitions to prevent multiple inclusion of the contents of the file. This also helps to prevent violations of the *one definition rule*—the requirement that all templates, types, functions, and objects have no more than one definition in your code.

For example:

```
// header.h
#pragma once
// Code placed here is included only once per translation unit
```

We recommend the `#pragma once` directive for new code because it doesn't pollute the global namespace with a preprocessor symbol. It requires less typing, is less distracting, and can't cause symbol collisions—errors caused when different header files use the same preprocessor symbol as the guard value. It is not part of the C++ Standard, but it is implemented portably by several common compilers.

There is no advantage to use of both the #include guard idiom and `#pragma once` in the same file. The compiler recognizes the #include guard idiom and implements the multiple include optimization the same way as the `#pragma once` directive if no non-comment code or preprocessor directive comes before or after the standard form of the idiom:

```
// header.h
// Demonstration of the #include guard idiom.
// Note that the defined symbol can be arbitrary.
#ifndef HEADER_H_     // equivalently, #if !defined HEADER_H_
#define HEADER_H_
// Code placed here is included only once per translation unit
#endif // HEADER_H_
```

We recommend the `#include guard` idiom when code must be portable to compilers that do not implement the `#pragma once` directive, to maintain consistency with existing code, or when the multiple-include optimization is impossible. This can occur in complex projects when file system aliasing or aliased include paths prevent the compiler from identifying identical include files by canonical path.

Be careful not to use `#pragma once` or the `#include guard` idiom in header files that are designed to be included multiple times, using preprocessor symbols to control their effects. For an example of this design, see the <assert.h> header file. Also be careful to manage include paths to avoid creating multiple paths to included files, which can defeat the multiple-include optimization for both `#include guard`s and `#pragma once`.

See also

Pragma Directives and the __Pragma Keyword

optimize

4/4/2019 • 2 minutes to read • Edit Online

Specifies optimizations to be performed on a function-by-function basis.

Syntax

```
#pragma optimize( "[optimization-list]", {on | off} )
```

Remarks

The **optimize** pragma must appear outside a function and takes effect at the first function defined after the pragma is seen. The *on* and *off* arguments turn options specified in the *optimization-list* on or off.

The *optimization-list* can be zero or more of the parameters shown in the following table.

Parameters of the optimize Pragma

PARAMETER(S)	TYPE OF OPTIMIZATION
g	Enable global optimizations.
s or *t*	Specify short or fast sequences of machine code.
y	Generate frame pointers on the program stack.

These are the same letters used with the /O compiler options. For example, the following pragma is equivalent to the `/Os` compiler option:

```
#pragma optimize( "ts", on )
```

Using the **optimize** pragma with the empty string (**""**) is a special form of the directive:

When you use the *off* parameter, it turns all the optimizations, *g*, *s*, *t*, and *y*, off.

When you use the *on* parameter, it resets the optimizations to those that you specified with the /O compiler option.

```
#pragma optimize( "", off )
.
.
.
#pragma optimize( "", on )
```

See also

Pragma Directives and the __Pragma Keyword

pack

4/4/2019 • 3 minutes to read • Edit Online

Specifies packing alignment for structure, union, and class members.

Syntax

```
#pragma pack( [ show ] | [ push | pop ] [, identifier ] , n  )
```

Parameters

show
(Optional) Displays the current byte value for packing alignment. The value is displayed by a warning message.

push
(Optional) Pushes the current packing alignment value on the internal compiler stack, and sets the current packing alignment value to *n*. If *n* is not specified, the current packing alignment value is pushed.

pop
(Optional) Removes the record from the top of the internal compiler stack. If *n* is not specified with **pop**, then the packing value associated with the resulting record on the top of the stack is the new packing alignment value. If *n* is specified, for example, `#pragma pack(pop, 16)`, *n* becomes the new packing alignment value. If you pop with *identifier*, for example, `#pragma pack(pop, r1)`, then all records on the stack are popped until the record that has *identifier* is found. That record is popped and the packing value associated with the resulting record on the top of is the stack the new packing alignment value. If you pop with an *identifier* that is not found in any record on the stack, then the **pop** is ignored.

identifier
(Optional) When used with *push*, assigns a name to the record on the internal compiler stack. When used with **pop**, pops records off the internal stack until *identifier* is removed; if *identifier* is not found on the internal stack, nothing is popped.

n
(Optional) Specifies the value, in bytes, to be used for packing. If the compiler option /Zp is not set for the module, the default value for *n* is 8. Valid values are 1, 2, 4, 8, and 16. The alignment of a member will be on a boundary that is either a multiple of *n* or a multiple of the size of the member, whichever is smaller.

`#pragma pack(pop, identifier, n)` is undefined.

Remarks

To pack a class is to place its members directly after each other in memory, which can mean that some or all members can be aligned on a boundary smaller than the default alignment the target architecture. **pack** gives control at the data-declaration level. This differs from compiler option /Zp, which only provides module-level control. **pack** takes effect at the first **struct**, **union**, or **class** declaration after the pragma is seen. **pack** has no effect on definitions. Calling **pack** with no arguments sets *n* to the value set in the compiler option `/Zp`. If the compiler option is not set, the default value is 8.

If you change the alignment of a structure, it may not use as much space in memory, but you may see a decrease in performance or even get a hardware-generated exception for unaligned access. You can modify this exception behavior by using SetErrorMode.

For more information about how to modify alignment, see these topics:

- __alignof
- align
- __unaligned
- Examples of Structure Alignment (x64 specific)

> **WARNING**
>
> Note that in Visual Studio 2015 and later you can use the standard alignas and alignof operators which, unlike `__alignof` and `declspec( align )` are portable across compilers. The C++ standard does not address packing, so you must still use **pack** (or the corresponding extension on other compilers) to specify alignments smaller than the target architecture's word size.

Examples

The following sample shows how to use the **pack** pragma to change the alignment of a structure.

```
// pragma_directives_pack.cpp
#include <stddef.h>
#include <stdio.h>

struct S {
   int i;   // size 4
   short j;   // size 2
   double k;   // size 8
};

#pragma pack(2)
struct T {
   int i;
   short j;
   double k;
};

int main() {
   printf("%zu ", offsetof(S, i));
   printf("%zu ", offsetof(S, j));
   printf("%zu\n", offsetof(S, k));

   printf("%zu ", offsetof(T, i));
   printf("%zu ", offsetof(T, j));
   printf("%zu\n", offsetof(T, k));
}
```

```
0 4 8
0 4 6
```

The following sample shows how to use the *push*, *pop*, and *show* syntax.

```
// pragma_directives_pack_2.cpp
// compile with: /W1 /c
#pragma pack()   // n defaults to 8; equivalent to /Zp8
#pragma pack(show)   // C4810
#pragma pack(4)   // n = 4
#pragma pack(show)   // C4810
#pragma pack(push, r1, 16)   // n = 16, pushed to stack
#pragma pack(show)   // C4810
#pragma pack(pop, r1, 2)   // n = 2 , stack popped
#pragma pack(show)   // C4810
```

See also

Pragma Directives and the __Pragma Keyword

pointers_to_members

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Specifies whether a pointer to a class member can be declared before its associated class definition and is used to control the pointer size and the code required to interpret the pointer.

Syntax

```
#pragma pointers_to_members( pointer-declaration, [most-general-representation] )
```

Remarks

You can place a **pointers_to_members** pragma in your source file as an alternative to using the /vmx compiler options or the inheritance keywords.

The *pointer-declaration* argument specifies whether you have declared a pointer to a member before or after the associated function definition. The *pointer-declaration* argument is one of the following two symbols:

ARGUMENT	COMMENTS
full_generality	Generates safe, sometimes nonoptimal code. You use *full_generality* if any pointer to a member is declared before the associated class definition. This argument always uses the pointer representation specified by the *most-general-representation* argument. Equivalent to /vmg.
best_case	Generates safe, optimal code using best-case representation for all pointers to members. Requires defining the class before declaring a pointer to a member of the class. The default is *best_case*.

The *most-general-representation* argument specifies the smallest pointer representation that the compiler can safely use to reference any pointer to a member of a class in a translation unit. The argument can be one of the following:

ARGUMENT	COMMENTS
single_inheritance	The most general representation is single-inheritance, pointer to a member function. Causes an error if the inheritance model of a class definition for which a pointer to a member is declared is ever either multiple or virtual.
multiple_inheritance	The most general representation is multiple-inheritance, pointer to a member function. Causes an error if the inheritance model of a class definition for which a pointer to a member is declared is virtual.

ARGUMENT	COMMENTS
virtual_inheritance	The most general representation is virtual-inheritance, pointer to a member function. Never causes an error. This is the default argument when `#pragma pointers_to_members(full_generality)` is used.

Caution

We advise you to put the **pointers_to_members** pragma only in the source code file that you want to affect, and only after any `#include` directives. This practice lessens the risk that the pragma will affect other files, and that you will accidently specify multiple definitions for the same variable, function, or class name.

Example

```
//   Specify single-inheritance only
#pragma pointers_to_members( full_generality, single_inheritance )
```

END C++ Specific

See also

Pragma Directives and the __Pragma Keyword

pop_macro

4/4/2019 • 2 minutes to read • Edit Online

Sets the value of the *macro_name* macro to the value on the top of the stack for this macro.

Syntax

```
#pragma pop_macro("
macro_name
")
```

Remarks

You must first issue a push_macro for *macro_name* before you can do a **pop_macro**.

Example

```
// pragma_directives_pop_macro.cpp
// compile with: /W1
#include <stdio.h>
#define X 1
#define Y 2

int main() {
   printf("%d",X);
   printf("\n%d",Y);
   #define Y 3   // C4005
   #pragma push_macro("Y")
   #pragma push_macro("X")
   printf("\n%d",X);
   #define X 2   // C4005
   printf("\n%d",X);
   #pragma pop_macro("X")
   printf("\n%d",X);
   #pragma pop_macro("Y")
   printf("\n%d",Y);
}
```

```
1
2
1
2
1
3
```

See also

Pragma Directives and the __Pragma Keyword

push_macro

4/4/2019 • 2 minutes to read • Edit Online

Saves the value of the *macro_name* macro on the top of the stack for this macro.

Syntax

```
#pragma push_macro("
macro_name
")
```

Remarks

You can retrieve the value for *macro_name* with `pop_macro`.

See pop_macro for a sample.

See also

Pragma Directives and the __Pragma Keyword

region, endregion

4/4/2019 • 2 minutes to read • Edit Online

`#pragma region` lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor.

Syntax

```
#pragma region name
#pragma endregion comment
```

Parameters

comment
(Optional) A comment that will display in the code editor.

name
(Optional) The name of the region. This name will display in the code editor.

Remarks

`#pragma endregion` marks the end of a `#pragma region` block.

A `#region` block must be terminated with `#pragma endregion`.

Example

```
// pragma_directives_region.cpp
#pragma region Region_1
void Test() {}
void Test2() {}
void Test3() {}
#pragma endregion Region_1

int main() {}
```

See also

Pragma Directives and the __Pragma Keyword

runtime_checks

4/4/2019 • 2 minutes to read • Edit Online

Disables or restores the /RTC settings.

Syntax

```
#pragma runtime_checks( "[runtime_checks]", {restore | off} )
```

Remarks

You cannot enable a run-time check that was not enabled with a compiler option. For example, if you do not specify `/RTCs`, specifying `#pragma runtime_checks( "s", restore)` will not enable stack frame verification.

The **runtime_checks** pragma must appear outside a function and takes effect at the first function defined after the pragma is seen. The *restore* and *off* arguments turn options specified in the **runtime_checks** on or off.

The **runtime_checks** can be zero or more of the parameters shown in the following table.

Parameters of the runtime_checks Pragma

PARAMETER(S)	TYPE OF RUN-TIME CHECK
s	Enables stack (frame) verification.
c	Reports when a value is assigned to a smaller data type that results in a data loss.
u	Reports when a variable is used before it is defined.

These are the same letters used with the `/RTC` compiler option. For example:

```
#pragma runtime_checks( "sc", restore )
```

Using the **runtime_checks** pragma with the empty string (**""**) is a special form of the directive:

- When you use the *off* parameter, it turns the run-time error checks, listed in the table above, off.
- When you use the *restore* parameter, it resets the run-time error checks to those that you specified with the `/RTC` compiler option.

```
#pragma runtime_checks( "", off )
.
.
.
#pragma runtime_checks( "", restore )
```

See also

Pragma Directives and the __Pragma Keyword

section

4/4/2019 • 2 minutes to read • Edit Online

Creates a section in an .obj file.

Syntax

```
#pragma section( "section-name" [, attributes] )
```

Remarks

The meaning of the terms *segment* and *section* are interchangeable in this topic.

Once a section is defined, it remains valid for the remainder of the compilation. However, you must use __declspec(allocate) or nothing will be placed in the section.

section-name is a required parameter that will be the name of the section. The name must not conflict with any standard section names. See /SECTION for a list of names you should not use when creating a section.

attributes is an optional parameter consisting of one or more comma-separated attributes that you want to assign to the section. Possible *attributes* are:

ATTRIBUTE	DESCRIPTION
read	Allows read operations on data.
write	Allows write operations on data.
execute	Allows code to be executed.
shared	Shares the section among all processes that load the image.
nopage	Marks the section as not pageable; useful for Win32 device drivers.
nocache	Marks the section as not cacheable; useful for Win32 device drivers.
discard	Marks the section as discardable; useful for Win32 device drivers.
remove	Marks the section as not memory-resident; virtual device drivers (VxD) only.

If you do not specify attributes, the section will have read and write attributes.

Example

In the following example, the first instruction identifies the section and its attributes. The integer `j` is not put into `mysec` because it was not declared with `__declspec(allocate)`; `j` goes into the data section. The integer `i` does go into `mysec` as a result of its `__declspec(allocate)` storage-class attribute.

```
// pragma_section.cpp
#pragma section("mysec",read,write)
int j = 0;

__declspec(allocate("mysec"))
int i = 0;

int main(){}
```

See also

Pragma Directives and the __Pragma Keyword

setlocale

4/4/2019 • 2 minutes to read • Edit Online

Defines the locale (Country/Region and language) to be used when translating wide-character constants and string literals.

Syntax

```
#pragma setlocale( "[locale-string]" )
```

Remarks

Because the algorithm for converting multibyte characters to wide characters may vary by locale or the compilation may take place in a different locale from where an executable file will be run, this pragma provides a way to specify the target locale at compile time. This guarantees that the wide-character strings will be stored in the correct format.

The default *locale-string* is "".

The "C" locale maps each character in the string to its value as a **wchar_t** (unsigned short). Other values that are valid for `setlocale` are those entries that are found in the Language Strings list. For example, you could issue:

```
#pragma setlocale("dutch")
```

The ability to issue a language string depends on the code page and language ID support on your computer.

See also

Pragma Directives and the __Pragma Keyword

strict_gs_check

4/4/2019 • 2 minutes to read • Edit Online

This pragma provides enhanced security checking.

Syntax

```
#pragma strict_gs_check([push,] on )
#pragma strict_gs_check([push,] off )
#pragma strict_gs_check(pop)
```

Remarks

Instructs the compiler to insert a random cookie in the function stack to help detect some categories of stack-based buffer overrun. By default, the `/GS` (Buffer Security Check) compiler option does not insert a cookie for all functions. For more information, see /GS (Buffer Security Check).

You must compile with `/GS` (Buffer Security Check) to enable **strict_gs_check**.

Use this pragma in code modules that are exposed to potentially harmful data. This pragma is very aggressive, and is applied to functions that might not need this defense, but is optimized to minimize its effect on the performance of the resulting application.

Even if you use this pragma, you should strive to write secure code. That is, make sure that your code has no buffer overruns. **strict_gs_check** might protect your application from buffer overruns that do remain in your code.

Example

In the following code a buffer overrun occurs when we copy an array to a local array. When you compile this code with `/GS`, no cookie is inserted in the stack, because the array data type is a pointer. Adding the **strict_gs_check** pragma forces the stack cookie into the function stack.

```
// pragma_strict_gs_check.cpp
// compile with: /c

#pragma strict_gs_check(on)

void ** ReverseArray(void **pData,
                     size_t cData)
{
    // *** This buffer is subject to being overrun!! ***
    void *pReversed[20];

    // Reverse the array into a temporary buffer
    for (size_t j = 0, i = cData; i ; --i, ++j)
        // *** Possible buffer overrun!! ***
            pReversed[j] = pData[i];

    // Copy temporary buffer back into input/output buffer
    for (size_t i = 0; i < cData ; ++i)
        pData[i] = pReversed[i];

    return pData;
}
```

See also

Pragma Directives and the __Pragma Keyword
/GS (Buffer Security Check)

vtordisp

4/4/2019 • 2 minutes to read • Edit Online

C++ Specific

Controls the addition of the hidden vtordisp construction/destruction displacement member.

Syntax

```
#pragma vtordisp([push,] n)
#pragma vtordisp(pop)
#pragma vtordisp()
#pragma vtordisp([push,] {on | off})
```

Parameters

push
Pushes the current vtordisp setting on the internal compiler stack and sets the new vtordisp setting to *n*. If *n* is not specified, the current vtordisp setting is not changed.

pop
Removes the top record from the internal compiler stack and restores the vtordisp setting to the removed value.

n
Specifies the new value for the vtordisp setting. Possible values are 0, 1 or 2, corresponding to the `/vd0`, `/vd1`, and `/vd2` compiler options. For more information, see /vd (Disable Construction Displacements).

on
Equivalent to `#pragma vtordisp(1)`.

off
Equivalent to `#pragma vtordisp(0)`.

Remarks

The **vtordisp** pragma is applicable only to code that uses virtual bases. If a derived class overrides a virtual function that it inherits from a virtual base class, and if a constructor or destructor for the derived class calls that function using a pointer to the virtual base class, the compiler might introduce additional hidden **vtordisp** fields into classes with virtual bases.

The **vtordisp** pragma affects the layout of classes that follow it. The `/vd0`, `/vd1`, and `/vd2` options specify the same behavior for complete modules. Specifying 0 or *off* suppresses the hidden **vtordisp** members. Turn off **vtordisp** only if there is no possibility that the class's constructors and destructors call virtual functions on the object pointed to by the **this** pointer.

Specifying 1 or *on*, the default, enables the hidden **vtordisp** members where they are necessary.

Specifying 2 enables the hidden **vtordisp** members for all virtual bases with virtual functions. `vtordisp(2)` might be necessary to ensure correct performance of **dynamic_cast** on a partially-constructed object. For more information, see Compiler Warning (level 1) C4436.

`#pragma vtordisp()`, with no arguments, restores the vtordisp setting to its initial setting.

```
#pragma vtordisp(push, 2)
class GetReal : virtual public VBase { ... };
#pragma vtordisp(pop)
```

END C++ Specific

See also

Pragma Directives and the __Pragma Keyword

warning Pragma

4/4/2019 • 3 minutes to read • Edit Online

Enables selective modification of the behavior of compiler warning messages.

Syntax

```
#pragma warning(
    warning-specifier : warning-number-list [; warning-specifier : warning-number-list...] )
#pragma warning( push[ ,n ] )
#pragma warning( pop )
```

Remarks

The following warning-specifier parameters are available.

WARNING-SPECIFIER	MEANING
1, 2, 3, 4	Apply the given level to the specified warning(s). This also turns on a specified warning that is off by default.
default	Reset warning behavior to its default value. This also turns on a specified warning that is off by default. The warning will be generated at its default, documented, level. For more information, see Compiler Warnings That Are Off by Default.
disable	Do not issue the specified warning message(s).
error	Report the specified warnings as errors.
once	Display the specified message(s) only one time.
suppress	Pushes the current state of the pragma on the stack, disables the specified warning for the next line, and then pops the warning stack so that the pragma state is reset.

The following code statement illustrates that a `warning-number-list` parameter can contain multiple warning numbers, and that multiple `warning-specifier` parameters can be specified in the same pragma directive.

```
#pragma warning( disable : 4507 34; once : 4385; error : 164 )
```

This is functionally equivalent to the following code.

```
// Disable warning messages 4507 and 4034.
#pragma warning( disable : 4507 34 )

// Issue warning 4385 only once.
#pragma warning( once : 4385 )

// Report warning 4164 as an error.
#pragma warning( error : 164 )
```

The compiler adds 4000 to any warning number that is between 0 and 999.

For warning numbers in the range 4700-4999, which are the ones associated with code generation, the state of the warning in effect when the compiler encounters the open curly brace of a function will be in effect for the rest of the function. Using the **warning** pragma in the function to change the state of a warning that has a number larger than 4699 will only take effect after the end of the function. The following example shows the correct placement of **warning** pragmas to disable a code-generation warning message, and then to restore it.

```
// pragma_warning.cpp
// compile with: /W1
#pragma warning(disable:4700)
void Test() {
   int x;
   int y = x;   // no C4700 here
   #pragma warning(default:4700)   // C4700 enabled after Test ends
}

int main() {
   int x;
   int y = x;   // C4700
}
```

Notice that throughout a function body, the last setting of the **warning** pragma will be in effect for the whole function.

Push and Pop

The **warning** pragma also supports the following syntax, where *n* represents a warning level (1 through 4).

`#pragma warning( push [ , n ] )`

`#pragma warning( pop )`

The pragma `warning( push )` stores the current warning state for every warning. The pragma `warning( push, n )` stores the current state for every warning and sets the global warning level to *n*.

The pragma `warning( pop )` pops the last warning state pushed onto the stack. Any changes that you made to the warning state between *push* and *pop* are undone. Consider this example:

```
#pragma warning( push )
#pragma warning( disable : 4705 )
#pragma warning( disable : 4706 )
#pragma warning( disable : 4707 )
// Some code
#pragma warning( pop )
```

At the end of this code, *pop* restores the state of every warning (includes 4705, 4706, and 4707) to what it was at the start of the code.

When you write header files, you can use *push* and *pop* to guarantee that warning-state changes made by a user do not prevent the headers from compiling correctly. Use *push* at the start of the header and *pop* at the end. For example, if you have a header that does not compile cleanly at warning level 4, the following code would change the warning level to 3 and then restore the original warning level at the end of the header.

```
#pragma warning( push, 3 )
// Declarations/definitions
#pragma warning( pop )
```

For more information about compiler options that help you suppress warnings, see /FI and /w.

See also

Pragma Directives and the __Pragma Keyword

Compiler warnings that are off by default

4/4/2019 • 10 minutes to read • Edit Online

The compiler supports warnings that are turned off by default, because most developers don't find them useful. In some cases, they warn about a stylistic choice, or about common idioms in older code. Other warnings are about use of a Microsoft extension to the language. In other cases, they indicate an area where programmers often make incorrect assumptions, which may lead to unexpected or undefined behavior. If enabled, some of these warnings may appear many times in library headers. The C runtime libraries and the C++ standard libraries are intended to emit no warnings only at warning level /W4.

Enable warnings that are off by default

You can enable warnings that are normally off by default by using one of the following options:

- **#pragma warning(default :** *warning_number* **)**

  The specified warning (*warning_number*) is enabled at its default level. Documentation for the warning contains the default level of the warning.

- **#pragma warning(** *warning_level* **:** *warning_number* **)**

  The specified warning (*warning_number*) is enabled at the specified level (*warning_level*).

- /Wall

  `/Wall` enables all warnings that are off by default. If you use this option, you can turn off individual warnings by using the /wd option.

- */wLnnnn*

  This option enables warning *nnnn* at level *L*.

Warnings that are off by default

The following warnings are turned off by default in Visual Studio 2015 and later versions:

C4061 (level 4)	enumerator '*identifier*' in a switch of enum '*enumeration*' is not explicitly handled by a case label
C4062 (level 4)	enumerator '*identifier*' in a switch of enum '*enumeration*' is not handled
C4165 (level 1)	'HRESULT' is being converted to 'bool'; are you sure this is what you want?
C4191 (level 3)	'*operator*': unsafe conversion from '*type_of_expression*' to '*type_required*'
C4242 (level 4)	'*identifier*': conversion from '*type1*' to '*type2*', possible loss of data

C4254 (level 4)	'*operator*': conversion from '*type1*' to '*type2*', possible loss of data
C4255 (level 4)	'*function*': no function prototype given: converting '()' to '(void)'
C4263 (level 4)	'*function*': member function does not override any base class virtual member function
C4264 (level 1)	'*virtual_function*': no override available for virtual member function from base '*class*'; function is hidden
C4265 (level 3)	'*class*': class has virtual functions, but destructor is not virtual
C4266 (level 4)	'*function*': no override available for virtual member function from base '*type*'; function is hidden
C4287 (level 3)	'*operator*': unsigned/negative constant mismatch
C4289 (level 4)	nonstandard extension used : '*var*' : loop control variable declared in the for-loop is used outside the for-loop scope
C4296 (level 4)	'*operator*': expression is always false
C4339 (level 4)	'*type*' : use of undefined type detected in CLR meta-data - use of this type may lead to a runtime exception
C4342 (level 1)	behavior change: '*function*' called, but a member operator was called in previous versions
C4350 (level 1)	behavior change: '*member1*' called instead of '*member2*'
C4355	'this' : used in base member initializer list
C4365 (level 4)	'*action*': conversion from '*type_1*' to '*type_2*', signed/unsigned mismatch
C4370 (level 3)	layout of class has changed from a previous version of the compiler due to better packing
C4371 (level 3)	'*classname*': layout of class may have changed from a previous version of the compiler due to better packing of member '*member*'
C4388 (level 4)	signed/unsigned mismatch
C4412 (level 2)	'*function*': function signature contains type '*type*'; C++ objects are unsafe to pass between pure code and mixed or native
C4426 (level 1)	optimization flags changed after including header, may be due to #pragma optimize() $^{14.1}$
C4435 (level 4)	'*class1*' : Object layout under /vd2 will change due to virtual base '*class2*'

C4437 (level 4)	dynamic_cast from virtual base '*class1*' to '*class2*' could fail in some contexts
C4444 (level 3)	top level '__unaligned' is not implemented in this context
C4464 (level 4)	relative include path contains '..'
C4471 (level 4)	a forward declaration of an unscoped enumeration must have an underlying type (int assumed) $^{\text{Perm}}$
C4472 (level 1)	'*identifier*' is a native enum: add an access specifier (private/public) to declare a managed enum
C4514 (level 4)	'*function*': unreferenced inline function has been removed
C4536 (level 4)	'type name': type-name exceeds meta-data limit of '*limit*' characters
C4545 (level 1)	expression before comma evaluates to a function which is missing an argument list
C4546 (level 1)	function call before comma missing argument list
C4547 (level 1)	'*operator*': operator before comma has no effect; expected operator with side-effect
C4548 (level 1)	expression before comma has no effect; expected expression with side-effect
C4549 (level 1)	'*operator1*': operator before comma has no effect; did you intend '*operator2*'?
C4555 (level 1)	expression has no effect; expected expression with side-effect
C4557 (level 3)	'__assume' contains effect '*effect*'
C4571 (level 4)	informational: catch(...) semantics changed since Visual C++ 7.1; structured exceptions (SEH) are no longer caught
C4574 (level 4)	'*identifier*' is defined to be '0': did you mean to use '#if *identifier*'?
C4577 (level 1)	'noexcept' used with no exception handling mode specified; termination on exception is not guaranteed. Specify /EHsc
C4582 (level 4)	'*type*': constructor is not implicitly called
C4583 (level 4)	'*type*': destructor is not implicitly called

C4587 (level 1)	'*anonymous_structure*': behavior change: constructor is no longer implicitly called
C4588 (level 1)	'*anonymous_structure*': behavior change: destructor is no longer implicitly called
C4596 (level 4)	'*identifier*': illegal qualified name in member declaration $^{14.3}$ $^{\text{Perm}}$
C4598 (level 1 and level 3)	'#include "*header*"': header number *number* in the precompiled header does not match current compilation at that position $^{14.3}$
C4599 (level 3)	'*option path*': command-line argument number *number* does not match pre-compiled header $^{14.3}$
C4605 (level 1)	'/D*macro*' specified on current command line, but was not specified when precompiled header was built
C4608 (level 3)	'*union_member*' has already been initialized by another union member in the initializer list, '*union_member*' $^{\text{Perm}}$
C4619 (level 3)	#pragma warning: there is no warning number '*number*'
C4623 (level 4)	'derived class': default constructor could not be generated because a base class default constructor is inaccessible
C4625 (level 4)	'derived class': copy constructor could not be generated because a base class copy constructor is inaccessible
C4626 (level 4)	'derived class': assignment operator could not be generated because a base class assignment operator is inaccessible
C4628 (level 1)	digraphs not supported with -Ze. Character sequence '*digraph*' not interpreted as alternate token for '*char*'
C4640 (level 3)	'*instance*': construction of local static object is not thread-safe
C4643 (level 4)	Forward declaring '*identifier*' in namespace std is not permitted by the C++ Standard. $^{15.8}$
C4647 (level 3)	behavior change: __is_pod(*type*) has different value in previous versions
C4654 (level 4)	Code placed before include of precompiled header line will be ignored. Add code to precompiled header. $^{14.1}$
C4668 (level 4)	'*symbol*' is not defined as a preprocessor macro, replacing with '0' for '*directives*'
C4682 (level 4)	'*symbol*' : no directional parameter attribute specified, defaulting to [in]

C4686 (level 3)	'*user-defined type*': possible change in behavior, change in UDT return calling convention
C4692 (level 1)	'*function*': signature of non-private member contains assembly private native type '*native_type*'
C4710 (level 4)	'*function*': function not inlined
C4738 (level 3)	storing 32-bit float result in memory, possible loss of performance
C4746	volatile access of '*expression*' is subject to /volatile:<iso|ms> setting; consider using __iso_volatile_load/store intrinsic functions
C4749 (level 4)	conditionally supported: offsetof applied to non-standard-layout type '*type*'
C4767 (level 4)	section name '*symbol*' is longer than 8 characters and will be truncated by the linker
C4768 (level 3)	__declspec attributes before linkage specification are ignored
C4774 (level 4)	'*string*' : format string expected in argument *number* is not a string literal
C4777 (level 4)	'*function*' : format string '*string*' requires an argument of type '*type1*', but variadic argument *number* has type '*type2*'
C4786 (level 3)	'*symbol*' : object name was truncated to '*number*' characters in the debug information
C4800 (level 4)	Implicit conversion from '*type*' to bool. Possible information loss $^{16.0}$
C4820 (level 4)	'*bytes*' bytes padding added after construct '*member_name*'
C4822 (level 1)	'*member*': local class member function does not have a body
C4826 (level 2)	Conversion from '*type1*' to '*type2*' is sign-extended. This may cause unexpected runtime behavior.
C4837 (level 4)	trigraph detected: '??*character*' replaced by '*character*'
C4841 (level 4)	non-standard extension used: compound member designator used in offsetof
C4842 (level 4)	the result of 'offsetof' applied to a type using multiple inheritance is not guaranteed to be consistent between compiler releases
C4868 (level 4)	'*file*(*line_number*)' compiler may not enforce left-to-right evaluation order in braced initialization list

C4905 (level 1)	wide string literal cast to 'LPSTR'
C4906 (level 1)	string literal cast to 'LPWSTR'
C4917 (level 1)	'*declarator*': a GUID can only be associated with a class, interface, or namespace
C4928 (level 1)	illegal copy-initialization; more than one user-defined conversion has been implicitly applied
C4931 (level 4)	we are assuming the type library was built for number-bit pointers
C4946 (level 1)	reinterpret_cast used between related classes: '*class1*' and '*class2*'
C4962	'*function*': profile-guided optimizations disabled because optimizations caused profile data to become inconsistent
C4986 (level 4)	'*symbol*': exception specification does not match previous declaration
C4987 (level 4)	nonstandard extension used: 'throw (...)'
C4988 (level 4)	'*symbol*': variable declared outside class/function scope
C5022	'*type*': multiple move constructors specified
C5023	'*type*': multiple move assignment operators specified
C5024 (level 4)	'*type*': move constructor was implicitly defined as deleted
C5025 (level 4)	'*type*': move assignment operator was implicitly defined as deleted
C5026 (level 1 and level 4)	'*type*': move constructor was implicitly defined as deleted
C5027 (level 1 and level 4)	'*type*': move assignment operator was implicitly defined as deleted
C5029 (level 4)	nonstandard extension used: alignment attributes in C++ apply to variables, data members and tag types only
C5031 (level 4)	#pragma warning(pop): likely mismatch, popping warning state pushed in different file $^{14.1}$
C5032 (level 4)	detected #pragma warning(push) with no corresponding #pragma warning(pop) $^{14.1}$
C5034	use of intrinsic '*intrinsic*' causes function *function* to be compiled as guest code $^{15.3}$

C5035	use of feature '*feature*' causes function *function* to be compiled as guest code [15.3]
C5036 (level 1)	varargs function pointer conversion when compiling with /hybrid:x86arm64 '*type1*' to '*type2*' [15.3]
C5038 (level 4)	data member '*member1*' will be initialized after data member '*member2*' [15.3]
C5039 (level 4)	'*function*': pointer or reference to potentially throwing function passed to extern C function under -EHc. Undefined behavior may occur if this function throws an exception. [15.5]
C5042 (level 3)	'*function*': function declarations at block scope cannot be specified 'inline' in standard C++; remove 'inline' specifier [15.5]
C5045	Compiler will insert Spectre mitigation for memory load if /Qspectre switch specified [15.7]

[14.1] This warning is available starting in Visual Studio 2015 Update 1.
[14.3] This warning is available starting in Visual Studio 2015 Update 3.
[15.3] This warning is available starting in Visual Studio 2017 version 15.3.
[15.5] This warning is available starting in Visual Studio 2017 version 15.5.
[15.7] This warning is available starting in Visual Studio 2017 version 15.7.
[15.8] This warning is available starting in Visual Studio 2017 version 15.8.

[16.0] This warning is available starting in Visual Studio 2019 RTM.

[Perm] This warning is off unless the /permissive- compiler option is set.

Warnings off by default in earlier versions

These warnings were off by default in versions of the compiler before Visual Studio 2015:

C4302 (level 2)	'*conversion*': truncation from '*type1*' to '*type2*'
C4311 (level 1)	'*variable*' : pointer truncation from '*type*' to '*type*'
C4312 (level 1)	'*operation*' : conversion from '*type1*' to '*type2*' of greater size
C4319 (level 1)	'*operator*': zero extending '*type1*' to '*type2*' of greater size

This warning was off by default in versions of the compiler before Visual Studio 2012:

C4431 (level 4)	missing type specifier - int assumed. Note: C no longer supports default-int

See also

warning