Contents

C++ in Visual Studio

5/23/2019 • 5 minutes to read • Edit Online

NOTE

This developer documentation applies to Visual Studio 2015 and later. Use the version selector in the upper left of the page to match your version of Visual Studio.

If you are looking for a Visual C++ redistributable package so that you can run a program, go to the Microsoft Download Center and enter **Visual C++** in the search box.

Microsoft Visual C++, usually shortened to Visual C++ or MSVC, is the name for the C++, C, and assembly language development tools and libraries available as part of Visual Studio on Windows. These tools and libraries let you create Universal Windows Platform (UWP) apps, native Windows desktop and server applications, cross-platform libraries and apps that run on Windows, Linux, Android, and iOS, as well as managed apps and libraries that use the .NET Framework. You can use Visual C++ to write anything from simple console apps to the most sophisticated and complex apps for Windows desktop, from device drivers and operating system components to cross-platform games for mobile devices, and from the smallest IoT devices to multi-server high performance computing in the Azure cloud.

Visual Studio 2015, 2017 and 2019 can be installed side-by-side. You can use Visual Studio 2019 (compiler toolset v142) to edit and build programs using the toolset from Visual Studio 2015 (v140) and Visual Studio 2017 (v141).

What's New and Conformance History

What's New for C++ in Visual Studio
Find out what's new in Visual Studio.

What's New for C++ in Visual Studio 2003 through 2015
Find out what was new in C++ for each version of Visual Studio from 2003 through 2015.

C++ conformance improvements in Visual Studio
Learn about C++ conformance improvements in Visual Studio.

Visual C++ language conformance
A list of conformance status by feature in the MSVC C++ compiler.

Visual C++ change history 2003 - 2015
Learn about the breaking changes in previous versions.

Install Visual Studio and upgrade from earlier versions

Install C++ support in Visual Studio
Download Visual Studio 2017 or Visual Studio 2019 and install the Visual C++ toolset.

Visual C++ Porting and Upgrading Guide
Guidance for porting code and upgrading projects to Visual Studio 2015 or later to take advantage of greater compiler conformance to the C++ standard as well as greatly improved compilation times and security features such as Spectre mitigation.

Visual C++ Tools and Features in Visual Studio Editions

Find out about different Visual Studio editions.

Supported Platforms
Find out which platforms are supported.

Learn C++

Welcome Back to C++
Learn more about modern C++ programming techniques based on C++11 and later that enable you to write fast, safe code and avoid many of the pitfalls of C-style programming.

Standard C++
Learn about C++, get an overview of Modern C++, and find links to books, articles, talks, and events

Learn Visual C++
Start learning C++.

Visual C++ Samples
Information about samples.

C++ development tools

Overview of C++ Development in Visual Studio
How to use the Visual Studio IDE to create projects, edit code, link to libraries, compile, debug, create unit tests, do static analysis, deploy, and more.

Projects and Build Systems
How to create and configure Visual Studio C++ projects, CMake projects, and other kinds of projects with MSVC compiler and linker options.

Writing and refactoring C++ code
How to use the productivity features in the C++ editor to refactor, navigate, understand and write code.

Debugging Native Code
Use the Visual Studio debugger with C++ projects.

Code analysis for C/C++ overview
Use SAL annotations or the C++ Core Guidelines checkers to perform static analysis.

Write unit tests for C/C++ in Visual Studio
Create unit tests using the Microsoft Unit Testing Framework for C++, Google Test, Boost.Test, or CTest.

Write applications in C++

Universal Windows Apps
Find guides and reference content on the Windows Developer Center. For information about developing UWP apps, see Intro to the Universal Windows Platform and Create your first UWP app using C++.

Desktop Applications (C++)
Learn how to create traditional native C++ desktop applications for Windows.

.NET Programming with C++/CLI
Learn how to create DLLs that enable interoperability between native C++ and .NET programs written in languages such as C# or Visual Basic.

Linux Programming
Use the Visual Studio IDE to code and deploy to a remote Linux machine for compilation with GCC.

Create C/C++ DLLs in Visual Studio
Find out how to use Win32, ATL, and MFC to create Windows desktop DLLs, and provides information about how to compile and register your DLL.

Parallel Programming
Learn how to use the Parallel Patterns Library, C++ AMP, OpenMP, and other features that are related to multithreading on Windows.

Security Best Practices
Learn how to protect applications from malicious code and unauthorized use.

Cloud and Web Programming
In C++, you have several options for connecting to the web and the cloud.

Data Access
Connect to databases using ODBC and OLE DB.

Text and Strings
Learn about working with different text and string formats and encodings for local and international development.

Languages reference

C++ Language Reference

C/C++ Preprocessor Reference

C Language Reference

Compiler Intrinsics and Assembly Language

C++ Libraries in Visual Studio

The following sections provide information about the different C and C++ libraries that are included in Visual Studio.

C Run-Time Library Reference
Includes security-enhanced alternatives to functions that are known to pose security issues.

C++ Standard Library
The C++ Standard Library.

Active Template Library (ATL)
Support for COM components and apps.

Microsoft Foundation Class (MFC) libraries
Support for creating desktop apps that have traditional or Office-style user interfaces.

Parallel Patterns Library (PPL)
Asynchronous and parallel algorithms that execute on the CPU.

C++ AMP (C++ Accelerated Massive Parallelism)
Massively parallel algorithms that execute on the GPU.

Windows Runtime Template Library (WRL)
Universal Windows Platform (UWP) apps and components.

.NET Programming with C++/CLI
Programming for the common language runtime (CLR).

Third-party open source C++ libraries

The cross-platform **vcpkg** command-line tool greatly simplifies the discovery and installation of over 900 C++ open source libraries. See vcpkg: C++ Package Manager for Windows.

Feedback and community

How to Report a Problem with the Visual C++ Toolset
Learn how to create effective error reports against the Visual C++ toolset (compiler, linker, and other tools), and ways to submit your report.

Microsoft C++ Team Blog
Learn more about new features and the latest information from the developers of the C++ tools in Visual Studio.

Visual Studio Developer Community
Find out how to get help, file bugs, and make suggestions for Visual Studio.

See also

- C Language Reference
- C Run-Time Library Reference
- Compiler Intrinsics and Assembly Language

Overview of C++ development in Visual Studio

5/23/2019 • 5 minutes to read • Edit Online

As part of the Visual Studio Integrated Development Environment (IDE), Microsoft C++ (MSVC) shares many windows and tools in common with other languages. Many of those, including **Solution Explorer**, the code editor, and the debugger, are documented under Visual Studio IDE. Often, a shared tool or window has a slightly different set of features for C++ than for other languages. A few windows or tools are only available in Visual Studio Professional or Visual Studio Enterprise editions.

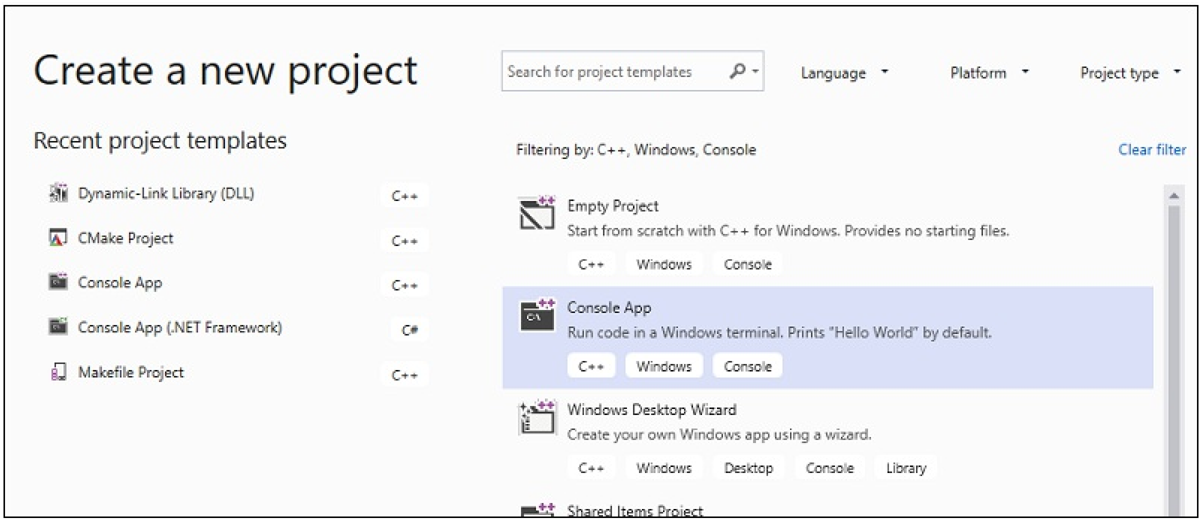
In addition to shared tools in the Visual Studio IDE, MSVC has several tools specifically for native code development. These tools are also listed in this article. For a list of which tools are available in each edition of Visual Studio, see C++ Tools and Features in Visual Studio Editions.

Create projects

A *project* is basically a set of source code files and resources such as images or data files that are built into an executable program or library.

Visual Studio provides support for any project system or custom build tools that you wish to use, with full support for IntelliSense, browsing and debugging:

- **MSBuild** is the native project system for Visual Studio. When you select **File** > **New** > **Project** from the main menu, you see many kinds of MSBuild *project templates* that get you started quickly developing different kinds of C++ applications.


Create a new project

Recent project templates

Dynamic-Link Library (DLL)	C++
CMake Project	C++
Console App	C++
Console App (.NET Framework)	C#
Makefile Project	C++

Search for project templates · Language · Platform · Project type

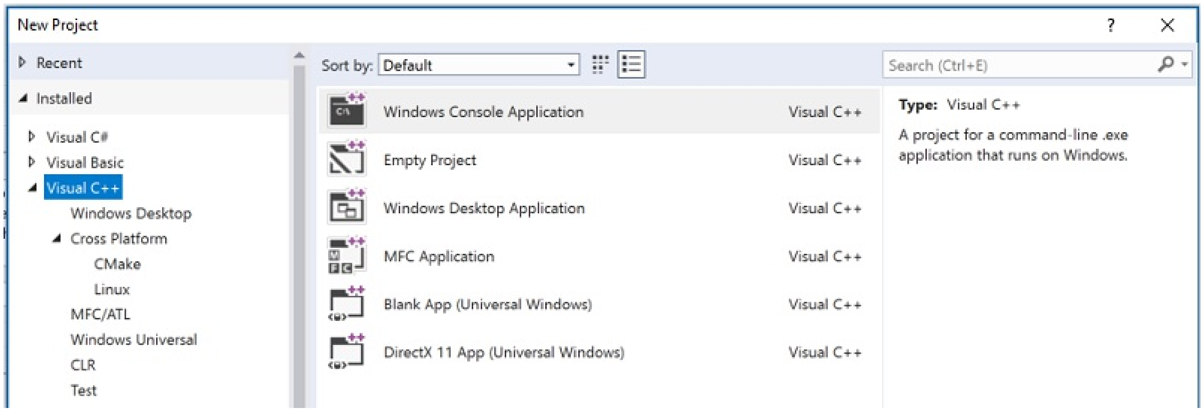
Filtering by: C++, Windows, Console · Clear filter

Empty Project
Start from scratch with C++ for Windows. Provides no starting files.
C++ · Windows · Console

Console App
Run code in a Windows terminal. Prints "Hello World" by default.
C++ · Windows · Console

Windows Desktop Wizard
Create your own Windows app using a wizard.
C++ · Windows · Desktop · Console · Library

Shared Items Project


New Project

Recent · Installed · Visual C# · Visual Basic · Visual C++ · Windows Desktop · Cross Platform · CMake · Linux · MFC/ATL · Windows Universal · CLR · Test

Sort by: Default · Search (Ctrl+E)

Windows Console Application	Visual C++
Empty Project	Visual C++
Windows Desktop Application	Visual C++
MFC Application	Visual C++
Blank App (Universal Windows)	Visual C++
DirectX 11 App (Universal Windows)	Visual C++

Type: Visual C++

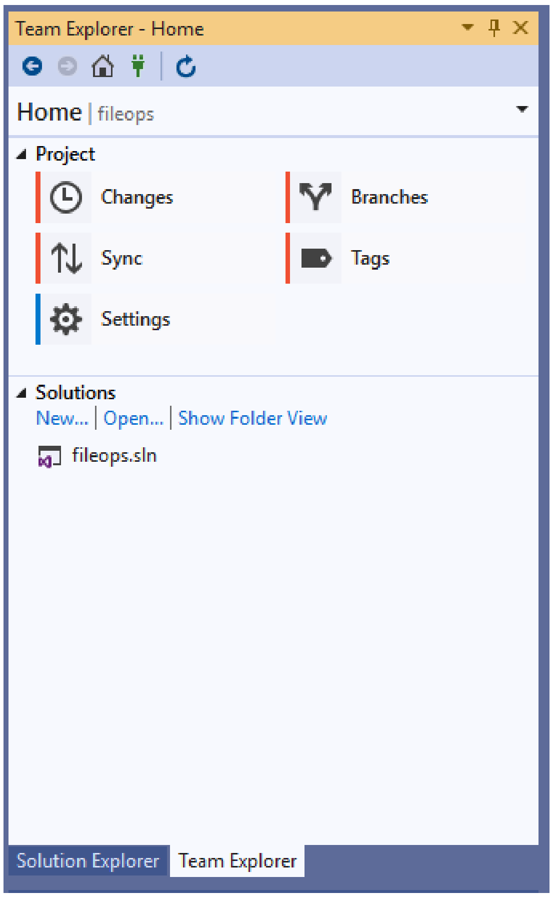
A project for a command-line .exe application that runs on Windows.

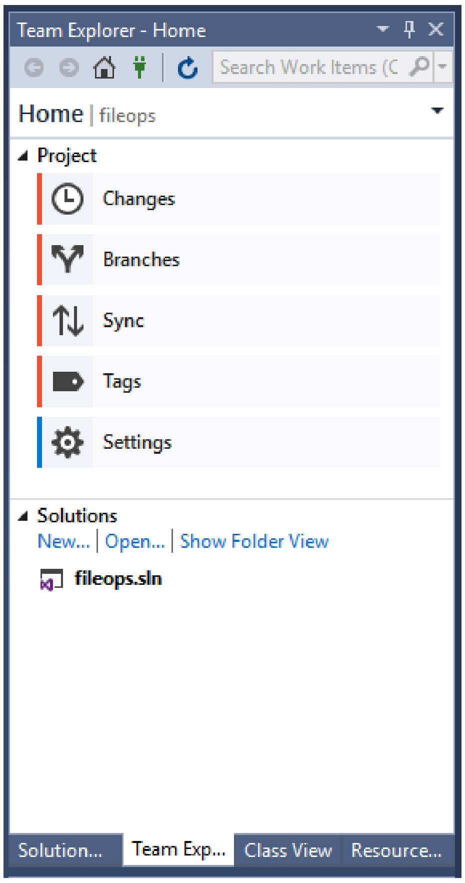
In general, you should use these templates for new projects unless you are using existing CMake projects, or you are using another project system. For more information, see Creating and managing MSBuild-based projects.

- **CMake** is a cross-platform build system that is integrated into the Visual Studio IDE when you install the Desktop development with C++ workload. You can use the CMake project template for new projects, or simply open a folder with a CMakeLists.txt file. For more information, see CMake projects in Visual Studio.
- Any other C++ build system, including a loose collection of files, is supported via the **Open Folder** feature. You create simple JSON files to invoke your build program and configure debugging sessions. For more information, see Open Folder projects for C++.

Add to source control

Source control enables you to coordinate work among multiple developers, isolate in-progress work from production code, and backup your source code. Visual Studio supports Git and Team Foundation Version Control (TFVC) through its **Team Explorer** window.





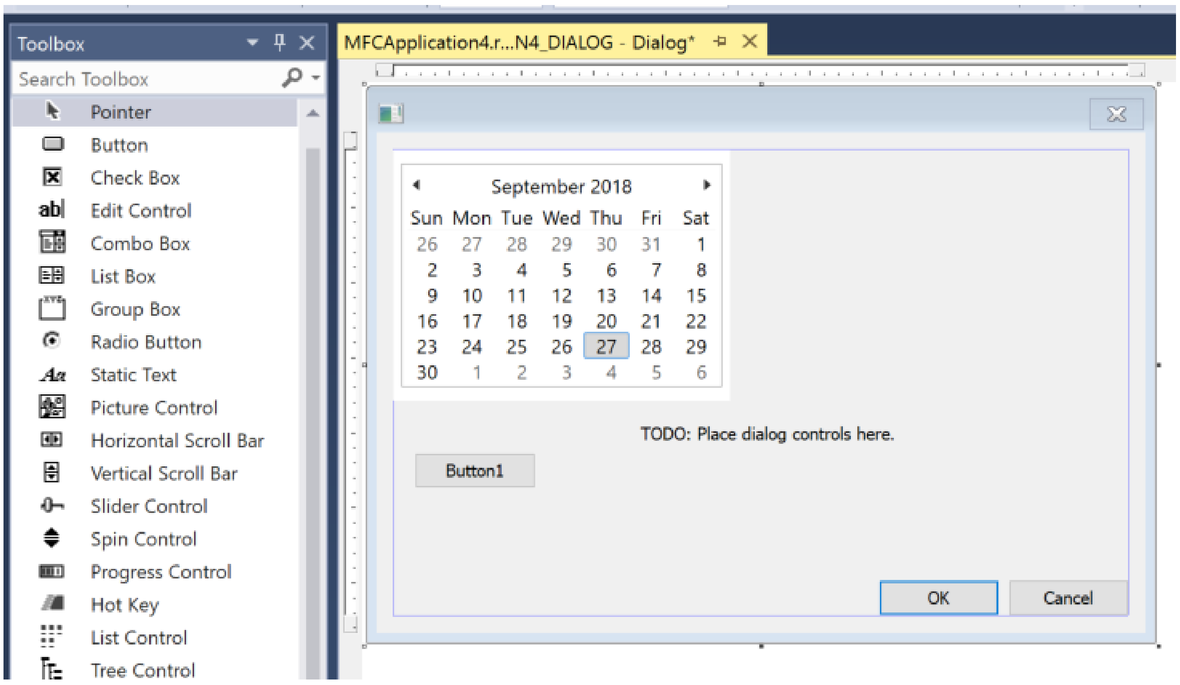
For more information about Git integration with repos in Azure, see Share your code with Visual Studio 2017 and Azure Repos Git. For information about Git integration with GitHub, see GitHub Extension for Visual Studio.

Obtain libraries

Use the vcpkg package manager to obtain and install third-party libraries. Over 900 open-source libraries are currently available in the catalog.

Create user interfaces with designers

If your program has a user interface, you can use a designer to quickly populate it with controls such as buttons, list boxes and so on. When you drag a control from the toolbox window and drop it onto the design surface, Visual Studio generates the resources and code required to make it all work. You then write the code to customize the appearance and behavior.

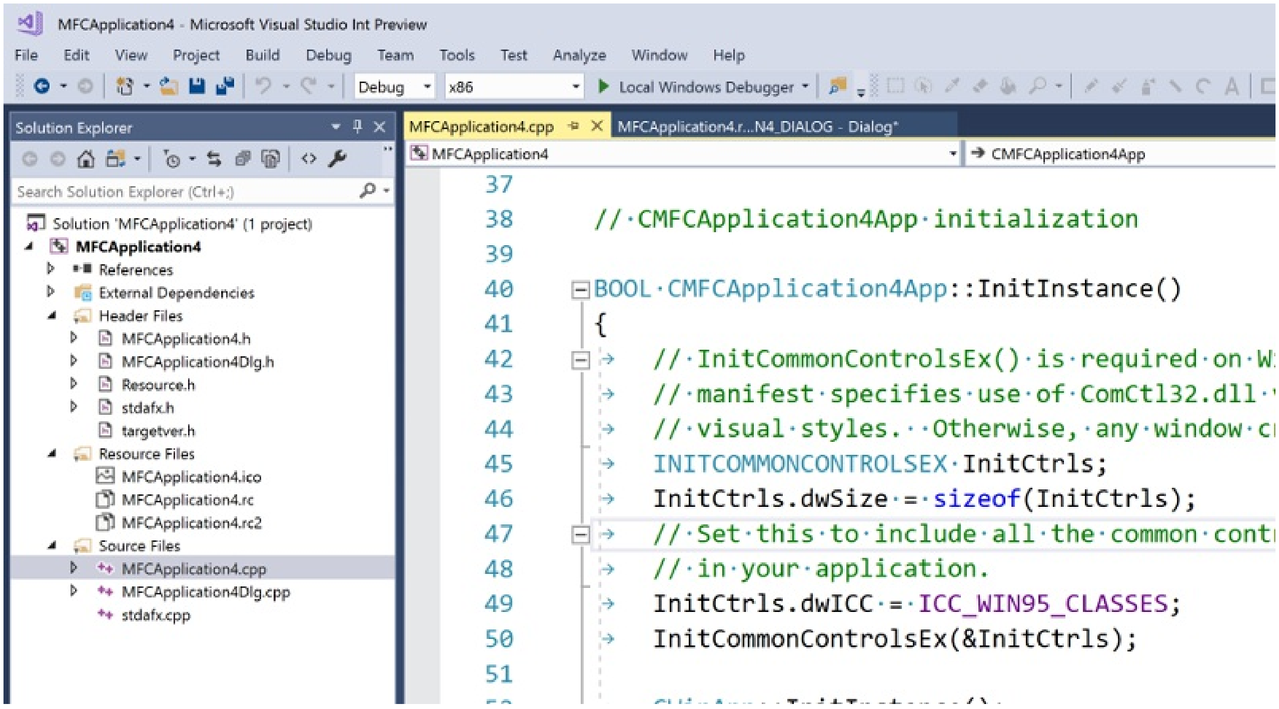


For more information about designing a user interface for a Universal Windows Platform app, see Design and UI.

For more information about creating a user interface for an MFC application, see MFC Desktop Applications. For information about Win32 Windows programs, see Windows Desktop Applications.

Write code

After you create a project, all the project files are displayed in the **Solution Explorer** window. (A *solution* is a logical container for one or more related projects.) When you click on a .h or .cpp file in **Solution Explorer**, the file opens up in the code editor.



```
    // CMFCApplication4App initialization

BOOL CMFCApplication4App::InitInstance()
{
    // InitCommonControlsEx() is required on W
    // manifest specifies use of ComCtl32.dll
    // visual styles.  Otherwise, any window c
    INITCOMMONCONTROLSEX InitCtrls;
    InitCtrls.dwSize = sizeof(InitCtrls);
    // Set this to include all the common cont
    // in your application.
    InitCtrls.dwICC = ICC_WIN95_CLASSES;
    InitCommonControlsEx(&InitCtrls);
```

The code editor is a specialized word processor for C++ source code. It color-codes language keywords, method and variable names, and other elements of your code to make the code more readable and easier to understand. It also provides tools for refactoring code, navigating between different files, and understanding how the code is structured. For more information, see Writing and refactoring code.

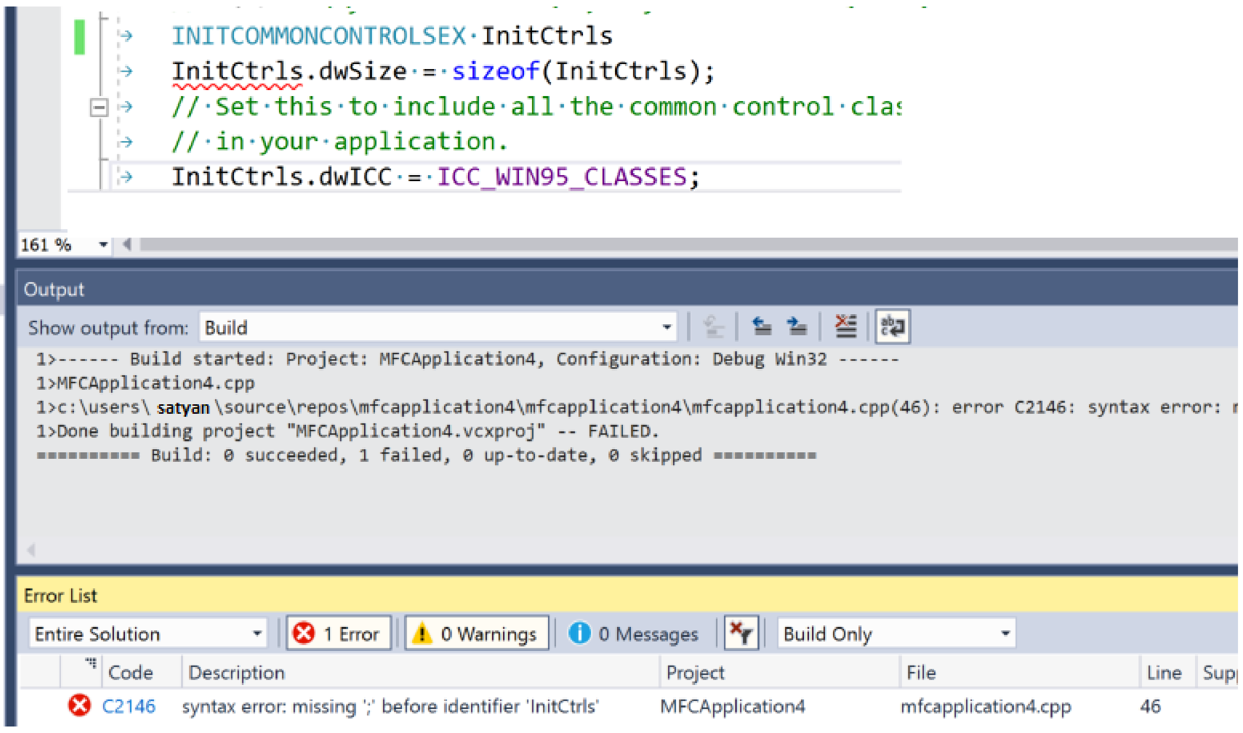
Add and edit resources

The term *resource* includes things such as dialogs, icons, images, localizable strings, splash screens, database connection strings, or any arbitrary data that you want to include in the executable file.

For more information on adding and editing resources in native desktop C++ projects, see Working with Resource Files.

Build (compile and link)

Choose **Build** > **Build Solution** on the menu bar, or enter the Ctrl+Shift+B key combination to compile and link a project. Build errors and warnings are reported in the Error List (Ctrl+\, E). The **Output** Window (Alt+2) shows information about the build process.



```
INITCOMMONCONTROLSEX·InitCtrls
InitCtrls.dwSize·=·sizeof(InitCtrls);
//·Set·this·to·include·all·the·common·control·clas
//·in·your·application.
InitCtrls.dwICC·=·ICC_WIN95_CLASSES;
```

Output

Show output from: Build

```
1>------ Build started: Project: MFCApplication4, Configuration: Debug Win32 ------
1>MFCApplication4.cpp
1>c:\users\ satyan \source\repos\mfcapplication4\mfcapplication4\mfcapplication4.cpp(46): error C2146: syntax error: r
1>Done building project "MFCApplication4.vcxproj" -- FAILED.
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

Error List

Entire Solution | 1 Error | 0 Warnings | 0 Messages | Build Only

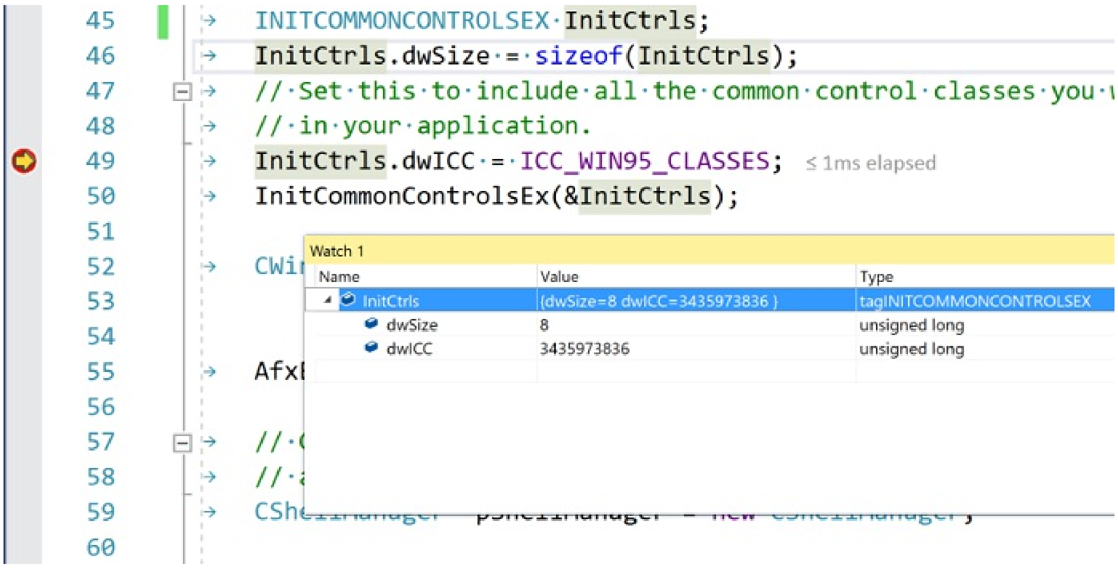
	Code	Description	Project	File	Line	Sup
⊗	C2146	syntax error: missing ';' before identifier 'InitCtrls'	MFCApplication4	mfcapplication4.cpp	46	

For more information about configuring builds, see Working with Project Properties and Projects and build systems.

You can also use the compiler (cl.exe) and many other build-related standalone tools such as NMAKE and LIB directly from the command line. For more information, see Build C/C++ code on the command line and C/C++ Building Reference.

Debug

You can start debugging by pressing **F5**. Execution pauses on any breakpoints you have set. You can also step through code one line at a time, view the values of variables or registers, and even in some cases make changes in code and continue debugging without re-compiling. The following illustration shows a debugging session in which execution is stopped on a breakpoint. The values of the data structure members are visible in the **Watch Window**.



```
INITCOMMONCONTROLSEX·InitCtrls;
InitCtrls.dwSize·=·sizeof(InitCtrls);
//·Set·this·to·include·all·the·common·control·classes·you·
//·in·your·application.
InitCtrls.dwICC·=·ICC_WIN95_CLASSES;  ≤ 1ms elapsed
InitCommonControlsEx(&InitCtrls);

CWi

AfxE

//·(
//·a
CShellManager   pShellManager = new CShellManager;
```

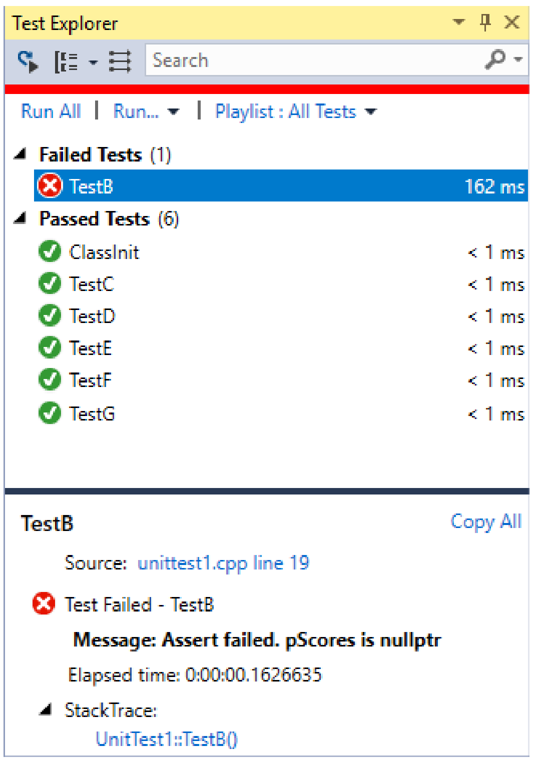
Watch 1

Name	Value	Type
InitCtrls	{dwSize=8 dwICC=3435973836 }	tagINITCOMMONCONTROLSEX
dwSize	8	unsigned long
dwICC	3435973836	unsigned long

For more information, see Debugging in Visual Studio.

Test

Visual Studio includes the Microsoft Unit Test Framework for C++, as well as support for Boost.Test, Google Test, and CTest. Run your tests from the **Test Explorer** window:



Failed Tests (1)

TestB	162 ms

Passed Tests (6)

ClassInit	< 1 ms
TestC	< 1 ms
TestD	< 1 ms
TestE	< 1 ms
TestF	< 1 ms
TestG	< 1 ms

TestB

Source: unittest1.cpp line 19

Test Failed - TestB

Message: Assert failed. pScores is nullptr

Elapsed time: 0:00:00.1626635

StackTrace:
UnitTest1::TestB()

For more information, see Verifying Code by Using Unit Tests and Write unit tests for C/C++ in Visual Studio.

Analyze

Visual Studio includes static code analysis tools that can detect potential problems in your source code. These tools include an implementation of the C++ Core Guidelines rules checkers. For more information, see Code analysis for C/C++ overview.

Deploy completed applications

You can deploy both traditional desktop applications and UWP apps to customers through the Microsoft Store. Deployment of the CRT is handled automatically behind the scenes. For more information, see Publish Windows apps and games.

You can also deploy a native C++ desktop to another computer For more information, see Deploying Desktop Applications.

For more information about deploying a C++/CLI program, see Deployment Guide for Developers,

Next steps

Explore Visual Studio further by following along with one of these introductory articles:

Learn to use the code editor

Learn about projects and solutions

4/3/2019 • 40 minutes to read • Edit Online

What's New for C++ in Visual Studio 2019

Visual Studio 2019 brings many updates and fixes to the Microsoft C++ environment. We've fixed many bugs and issues in the compiler and tools, many submitted by customers through the Report a Problem and Provide a Suggestion options under **Send Feedback**. Thank you for reporting bugs! For more information on what's new in all of Visual Studio, visit What's new in Visual Studio.

C++ compiler

- Enhanced support for C++17 features and correctness fixes, plus experimental support for C++20 features such as modules and coroutines. For detailed information, see C++ Conformance Improvements in Visual Studio 2019.
- The `/std:c++latest` option now includes C++20 features that aren't necessarily complete, including initial support for the C++20 operator <=> ("spaceship") for three-way comparison.
- The C++ compiler switch `/Gm` is nowdeprecated. Consider disabling the `/Gm` switch in your build scripts if it's explicitly defined. However, you can also safely ignore the deprecation warning for `/Gm`, because it's not treated as an error when using "Treat warnings as errors" (`/WX`).
- As MSVC begins implementing features from the C++20 standard draft under the `/std:c++latest` flag, `/std:c++latest` is now incompatible with `/clr` (all flavors), `/ZW`, and `/Gm`. In Visual Studio 2019, use `/std:c++17` or `/std:c++14` modes when compiling with `/clr`, `/ZW` or `/Gm` (but see previous bullet).
- Precompiled headers are no longer generated by default for C++ console and desktop apps.

Codegen, security, diagnostics, and versioning

Improved analysis with `/Qspectre` for providing mitigation assistance for Spectre Variant 1 (CVE-2017-5753). For more information, see Spectre Mitigations in MSVC.

C++ Standard Library improvements

- Implementation of additional C++17 and C++20 library features and correctness fixes. For detailed information, see C++ Conformance Improvements in Visual Studio 2019.
- Clang-Format has been applied to the C++ Standard Library headers for improved readability.
- Because Visual Studio now supports Just My Code for C++, the Standard Library no longer needs to provide custom machinery for `std::function` and `std::visit` to achieve the same effect. Removing that machinery largely has no user-visible effects, except that the compiler will no longer produce diagnostics that indicate issues on line 15732480 or 16707566 of <type_traits> or <variant>.

Performance/throughput improvements in the compiler and Standard Library

- Build throughput improvements, including the way the linker handles File I/O, and link time in PDB type merging and creation.
- Added basic support for OpenMP SIMD vectorization. You can enable it using the new compiler switch

`-openmp:experimental`. This option allows loops annotated with `#pragma omp simd` to potentially be vectorized. The vectorization isn't guaranteed, and loops annotated but not vectorized will get a warning reported. No SIMD clauses are supported, they're simply ignored with a warning reported.

- Added a new inlining command-line switch `-Ob3`, which is a more aggressive version of `-Ob2`. `-O2` (optimize the binary for speed) still implies `-Ob2` by default. If you find that the compiler doesn't inline aggressively enough, consider passing `-O2 -Ob3`.
- To support hand vectorization of loops with calls to math library functions, and certain other operations like integer division, we've added support for Short Vector Math Library (SVML) intrinsic functions. These functions compute the 128-bit, 256-bit, or 512-bit vector equivalents. See the Intel Intrinsic Guide for definitions of the supported functions.
- New and improved optimizations:
    - Constant-folding and arithmetic simplifications for expressions using SIMD vector intrinsics, for both float and integer forms.
    - A more powerful analysis for extracting information from control flow (if/else/switch statements) to remove branches always proven to be true or false.
    - Improved memset unrolling to use SSE2 vector instructions.
    - Improved removal of useless struct/class copies, especially for C++ programs that pass by value.
    - Improved optimization of code using `memmove`, such as `std::copy` or `std::vector` and `std::string` construction.
- Optimized the Standard Library physical design to avoid compiling parts of the Standard Library not #include'd, cutting in half the build time of an empty file that includes only <vector>. As a result of this change, you may need to add #include directives for headers that were previously indirectly included. For example, code that uses `std::out_of_range` may now need to #include <stdexcept>. Code that uses a stream insertion operator may now need to #include <ostream>. The benefit is that only translation units actually using <stdexcept> or <ostream> components pay the throughput cost to compile them.
- `if constexpr` was applied in more places in the Standard Library for improved throughput and reduced code size in copy operations, in permutations like reverse and rotate, and in the parallel algorithms library.
- The Standard Library now internally uses `if constexpr` to reduce compile times even in C++14 mode.
- The runtime dynamic linking detection for the parallel algorithms library no longer uses an entire page to store the function pointer array. Marking this memory read-only was deemed no longer relevant for security purposes.
- `std::thread`'s constructor no longer waits for the thread to start, and no longer inserts so many layers of function calls between the underlying C library `_beginthreadex` and the supplied callable object. Previously `std::thread` put 6 functions between `_beginthreadex` and the supplied callable object, which has been reduced to only 3 (2 of which are just `std::invoke`). This also resolves an obscure timing bug where `std::thread`'s constructor would hang if the system clock changed at the exact moment a `std::thread` was being created.
- Fixed a performance regression in `std::hash` that we introduced when implementing `std::hash<std::filesystem::path>`.
- In several places the Standard Library now uses destructors instead of catch blocks to achieve correctness. This results in better debugger interaction; exceptions you throw through the Standard Library in the affected locations will now show up as being thrown from their original throw site, rather than our rethrow. Not all Standard Library catch blocks have been eliminated; we expect the number of catch blocks to be

reduced in subsequent releases of MSVC.

- Suboptimal codegen in `std::bitset` caused by a conditional throw inside a noexcept function was fixed by factoring out the throwing path.
- The `std::list` and `std::unordered_*` family use non-debugging iterators internally in more places.
- Several `std::list` members were changed to reuse list nodes where possible rather than deallocating and reallocating them. For example, given a `list<int>` that already has a size of 3, a call to `assign(4, 1729)` will now overwrite the ints in the first 3 list nodes, and allocate one new list node with the value 1729, rather than deallocating all 3 list nodes and then allocating 4 new list nodes with the value 1729.
- All Standard Library calls to `erase(begin(), end())` were changed to `clear()`.
- `std::vector` now initializes and erases elements more efficiently in certain cases.
- Improvements to `std::variant` to make it more optimizer-friendly, resulting in better generated code. Code inlining is now much better with `std::visit`.

C++ IDE

Live Share C++ support

Live Share now supports C++, allowing developers using Visual Studio or Visual Studio Code to collaborate in real time. For more information, see Announcing Live Share for C++: Real-Time Sharing and Collaboration

IntelliCode for C++

IntelliCode is an optional extension (added as a workload component in 16.1) that uses its own extensive training and your code context to put what you're most likely to use at the top of your completion list. It can often eliminate the need to scroll down through the list. For C++, IntelliCode offers the most help when using popular libraries such as the Standard Library. For more information, see AI-Assisted Code Completion Suggestions Come to C++ via IntelliCode.

Template IntelliSense

The **Template Bar** now uses the **Peek Window** UI rather than a modal window, supports nested templates, and pre-populates any default arguments into the **Peek Window**. For more information, see Template IntelliSense Improvements for Visual Studio 2019 Preview 2. A **Most Recently Used** dropdown in the **Template Bar** enables you to quickly switch between previous sets of sample arguments.

New Start window experience

When launching the IDE, a new Start window appears with options to open recent projects, clone code from source control, open local code as a solution or a folder, or create a new project. The New Project dialog has also been overhauled into a search-first, filterable experience.

New names for some project templates

We've modified several project template names and descriptions to fit with the updated New Project dialog.

Various productivity improvements

Visual Studio 2019 includes the following features that will help make coding easier and more intuitive:

- Quick fixes for:
    - Add missing #include
    - NULL to nullptr
    - Add missing semicolon
    - Resolve missing namespace or scope
    - Replace bad indirection operands (* to & and & to *)

- Quick Info for a block by hovering on closing brace
- Peek Header / Code File
- Go to Definition on #include opens the file

For more information, see C++ Productivity Improvements in Visual Studio 2019 Preview 2.

Visual Studio 2019 version 16.1

QuickInfo improvements

The Quick Info tooltip now respects the semantic colorization of your editor. It also has a new **Search Online** link that will search for online docs to learn more about the hovered code construct. For red-squiggled code, the link provided by Quick Info will search for the error online. This way you don't need to retype the message into your browser. For more information, see Quick Info Improvements in Visual Studio 2019: Colorization and Search Online.

IntelliCode available in C++ workload

IntelliCode now ships as an optional component in the **Desktop Development with C++** workload. For more information, see Improved C++ IntelliCode now Ships with Visual Studio 2019.

CMake support

- Support for CMake 3.14

- Visual Studio can now open existing CMake caches generated by external tools, such as CMakeGUI, customized meta-build systems or build scripts that invoke cmake.exe themselves.

- Improved IntelliSense performance.

- A new settings editor provides an alternative to manually editing the CMakeSettings.json file, and provides some parity with CMakeGUI.

- Visual Studio helps bootstrap your C++ development with CMake on Linux by detecting if you have a compatible version of CMake on your Linux machine, and if not offers to install it for you.

- Incompatible settings in CMakeSettings, such as mismatched architectures or incompatible CMake generator settings, show squiggles in the JSON editor and errors in the error list.

- The vcpkg toolchain is automatically detected and enabled for CMake projects that are opened in the IDE once `vcpkg integrate install` has been run. This behavior can be turned off by specifying an empty toolchain file in CMakeSettings.

- CMake projects now enable Just My Code debugging by default.

- Static analysis warnings can now be processed in the background and displayed in the editor for CMake projects.

- Clearer build and configure 'begin' and 'end' messages for CMake projects and support for Visual Studio's build progress UI. Additionally, there's now a CMake verbosity setting in **Tools > Options** to customize the detail level of CMake build and configuration messages in the Output Window.

- The `cmakeToolchain` setting is now supported in CMakeSettings.json to specify toolchains without manually modifying the CMake command line.

- A new **Build All** menu shortcut **Ctrl+Shift+B**.

Visual Studio 2019 version 16.1

- Integrated support for editing, building, and debugging CMake projects with Clang/LLVM. For more information, see Clang/LLVM Support in Visual Studio.

Linux and WSL

Visual Studio 2019 version 16.1

- Support for AddressSanitizer (ASan) in Linux and CMake cross-platform projects. For more information, see AddressSanitizer (ASan) for the Linux Workload in Visual Studio 2019.
- Integrated Visual Studio support for using C++ with the Windows Subsystem for Linux (WSL). For more information, see C++ with Visual Studio 2019 and Windows Subsystem for Linux (WSL).

IncrediBuild integration

IncrediBuild is included as an optional component in the **Desktop development with C++** workload. The IncrediBuild Build Monitor is fully integrated in the Visual Studio IDE. For more information, see Visualize your build with IncrediBuild's Build Monitor and Visual Studio 2019.

Debugging

- For C++ applications running on Windows, PDB files now load in a separate 64-bit process. This change addresses a range of crashes caused by the debugger running out of memory when debugging applications that contain a large number of modules and PDB files.
- Search is enabled in the **Watch**, **Autos**, and **Locals** windows.

Windows desktop development with C++

- These C++ ATL/MFC wizards are no longer available:
  - ATL COM+ 1.0 Component Wizard
  - ATL Active Server Pages Component Wizard
  - ATL OLE DB Provider Wizard
  - ATL Property Page Wizard
  - ATL OLE DB Consumer Wizard
  - MFC ODBC Consumer
  - MFC class from ActiveX control
  - MFC class from Type Lib.

  Sample code for these technologies is archived at Microsoft Docs and the VCSamples GitHub repository.
- The Windows 8.1 SDK is no longer available in the Visual Studio installer. We recommend you upgrade your C++ projects to the latest Windows 10 SDK. If you have a hard dependency on 8.1, you can download it from the Windows SDK archive.
- Windows XP targeting will no longer be available for the latest C++ toolset. XP targeting with VS 2017-level MSVC compiler & libraries is still supported and can be installed via "Individual components."
- Our documentation actively discourages usage of Merge Modules for Visual C++ Runtime deployment. We're taking the extra step this release of marking our MSMs as deprecated. Consider migrating your VCRuntime central deployment from MSMs to the redistributable package.

Mobile development with C++ (Android and iOS)

The C++ Android experience now defaults to Android SDK 25 and Android NDK 16b.

Clang/C2 platform toolset

The Clang/C2 experimental component has been removed. Use the MSVC toolset for full C++ standards conformance with `/permissive-` and `/std:c++17`, or the Clang/LLVM toolchain for Windows.

Code analysis

- Code analysis now runs automatically in the background. Warnings display as green squiggles in-editor as you type. For more information, see In-editor code analysis in Visual Studio 2019 Preview 2.
- New experimental ConcurrencyCheck rules for well-known Standard Library types from the <mutex> header. For more information, see Concurrency Code Analysis in Visual Studio 2019.
- An updated partial implementation of the Lifetime profile checker, which detects dangling pointers and references. For more information, see Lifetime Profile Update in Visual Studio 2019 Preview 2.
- More coroutine-related checks, including C26138, C26810, C26811, and the experimental rule C26800. For more information, see New Code Analysis Checks in Visual Studio 2019: use-after-move and coroutine.

Visual Studio 2019 version 16.1

New quick fixes for uninitialized variable checks. For more information, see New code analysis quick fixes for uninitialized memory (C6001) and use before init (C26494) warnings.

Unit testing

The Managed C++ Test Project template is no longer available. You can continue using the Managed C++ Test framework in your existing projects. For new unit tests, consider using one of the native test frameworks for which Visual Studio provides templates (MSTest, Google Test), or the Managed C# Test Project template.

What's New for C++ in Visual Studio 2017

Visual Studio 2017 brings many updates and fixes to the C++ environment. We've fixed over 250 bugs and reported issues in the compiler and tools, many submitted by customers through the Report a Problem and Provide a Suggestion options under **Send Feedback**. Thank you for reporting bugs! For more information on what's new in all of Visual Studio, please visit What's new in Visual Studio 2017.

C++ compiler

C++ conformance improvements

In this release, we've updated the C++ compiler and standard library with enhanced support for C++11 and C++14 features, as well as preliminary support for certain features expected to be in the C++17 standard. For detailed information, see C++ Conformance Improvements in Visual Studio 2017.

Visual Studio 2017 version 15.5: The compiler supports about 75% of the features that are new in C++17, including structured bindings, `constexpr` lambdas, `if constexpr`, inline variables, fold expressions, and adding `noexcept` to the type system. These are available under the **/std:c++17** option. For more information, see C++ Conformance Improvements in Visual Studio 2017

Visual Studio 2017 version 15.7: The MSVC compiler toolset in Visual Studio version 15.7 now conforms with the C++ Standard. For more information, see Announcing: MSVC Conforms to the C++ Standard and Microsoft C++ Language Conformance.

New compiler options

- /permissive-: Enable all strict standards conformance compiler options and disable most Microsoft-specific compiler extensions (but not `__declspec(dllimport)`, for example). This option is on by default in Visual Studio 2017 version 15.5. The **/permissive-** conformance mode includes support for two-phase name lookup. For more information, see C++ Conformance Improvements in Visual Studio.

- /diagnostics: Enable display of the line number, the line number and column, or the line number and column and a caret under the line of code where the diagnostic error or warning was found.

- /debug:fastlink: Enable up to 30% faster incremental link times (vs. Visual Studio 2015) by not copying all debug information into the PDB file. The PDB file instead points to the debug information for the object and library files used to create the executable. See Faster C++ build cycle in VS "15" with /Debug:fastlink and Recommendations to speed C++ builds in Visual Studio.

- Visual Studio 2017 allows using /sdl with /await. We removed the /RTC limitation with Coroutines.

  Visual Studio 2017 version 15.3:

- /std:c++14 and /std:c++latest: These compiler options enable you to opt-in to specific versions of the ISO C++ programming language in a project. Most of the new draft standard features are guarded by the **/std:c++latest** option.

- /std:c++17 enables the set of C++17 features implemented by the compiler. This option disables compiler and standard library support for features that are changed or new in versions of the Working Draft and defect updates of the C++ Standard after C++17. To enable those features, use **/std:c++latest**.

Codegen, security, diagnostics and versioning

This release brings several improvements in optimization, code generation, toolset versioning, and diagnostics. Some notable improvements include:

- Improved code generation of loops: Support for automatic vectorization of division of constant integers, better identification of memset patterns.
- Improved code security: Improved emission of buffer overrun compiler diagnostics, and /guard:cf now guards switch statements that generate jump tables.
- Versioning: The value of the built-in preprocessor macro **_MSC_VER** is now being monotonically updated at every Visual C++ toolset update. For more information, see Visual C++ Compiler Version.
- New toolset layout: The compiler and related build tools have a new location and directory structure on your development machine. The new layout enables side-by-side installations of multiple versions of the compiler. For more information, see Compiler Tools Layout in Visual Studio "15".
- Improved diagnostics: The output window now shows the column where an error occurs. For more information, see C++ compiler diagnostics improvements in VS "15" Preview 5.
- When using co-routines, the experimental keyword **yield** (available under the **/await** option) has been removed. Your code should be updated to use `co_yield` instead. For more information, see the Visual C++ Team blog.

Visual Studio 2017 version 15.3:

Additional improvements to diagnostics in the compiler. For more information, see Diagnostic Improvements in Visual Studio 2017 15.3.0.

Visual Studio 2017 version 15.5:

Visual C++ runtime performance continues to improve due to better generated code quality. This means that you can simply recompile your code, and your app runs faster. Some of the compiler optimizations are brand new, such as the vectorization of conditional scalar stores, the combining of calls `sin(x)` and `cos(x)` into a new `sincos(x)`, and the elimination of redundant instructions from the SSA Optimizer. Other compiler optimizations are improvements to existing functionality such as vectorizer heuristics for conditional expressions, better loop optimizations, and float min/max codegen. The linker has a new and faster **/OPT:ICF** implementation which can result in up to 9% link time speedups, and there are other perf fixes in incremental linking. For more information, see /OPT (Optimizations) and /INCREMENTAL (Link Incrementally).

The Microsoft C++ compiler supports Intel's AVX-512, including the Vector Length instructions that bring new functions in AVX-512 to 128- and 256-bit wide registers.

The /Zc:noexceptTypes- option can be used to revert to the C++14 version of `noexcept` while using C++17 mode in general. This enables you to update your source code to conform to C++17 without having to rewrite all your `throw()` code at the same time. For more information, see Dynamic exception specification removal and noexcept.

Visual Studio 2017 version 15.7:

- New compiler switch /Qspectre to help mitigate against speculative execution side-channel attacks. See Spectre mitigations in MSVC for more information.
- New diagnostic warning for Spectre mitigation. See Spectre diagnostic in Visual Studio 2017 Version 15.7 Preview 4 for more information.
- A new value for /Zc, **/Zc:__cplusplus**, enables correct reporting of the C++ standard support. For example, when the switch is set and the compiler is in /std:c++17 mode the value expands to **201703L**. See MSVC now correctly reports __cplusplus for more information.

C++ Standard Library improvements

- Minor `basic_string` `_ITERATOR_DEBUG_LEVEL != 0` diagnostics improvements. Tripping an IDL check in string machinery will now report the specific behavior that caused the trip. For example, instead of "string iterator not dereferencable" you'll get "cannot dereference string iterator because it is out of range (e.g. an end iterator)".
- Performance improvement: made `basic_string::find(char)` overloads only call `traits::find` once. Previously this was implemented as a general string search for a string of length 1.
- Performance improvement: `basic_string::operator==` now checks the string's size before comparing the strings' contents.
- Performance improvement: removed control coupling in `basic_string`, which was difficult for the compiler optimizer to analyze. Note that for all short strings, calling `reserve` still has a nonzero cost to do nothing.
- We added <any>, <string_view>, `apply()`, `make_from_tuple()`.
- `std::vector` has been overhauled for correctness and performance: aliasing during insertion and emplacement is now correctly handled as required by the Standard, the strong exception guarantee is now provided when required by the Standard via `move_if_noexcept()` and other logic, and insertion/emplacement perform fewer element operations.
- The C++ Standard Library now avoids dereferencing null fancy pointers.
- Added <optional>, <variant>, `shared_ptr::weak_type`, and <cstdalign>.
- Enabled C++14 `constexpr` in `min(initializer_list)`, `max(initializer_list)`, and `minmax(initializer_list)`, and `min_element()`, `max_element()`, and `minmax_element()`.
- Improved `weak_ptr::lock()` performance.
- Fixed the `std::promise` move assignment operator, which previously could cause code to block forever.
- Fixed compiler errors with the `atomic<T*>` implicit conversion to `T*`.
- `pointer_traits<Ptr>` now correctly detects `Ptr::rebind<U>`.
- Fixed a missing `const` qualifier in the `move_iterator` subtraction operator.
- Fixed silent bad codegen for stateful user-defined allocators requesting `propagate_on_container_copy_assignment` and `propagate_on_container_move_assignment`.
- `atomic<T>` now tolerates overloaded `operator&()`.
- To increase compiler throughput, C++ Standard Library headers now avoid including declarations for unnecessary compiler intrinsics.
- Slightly improved compiler diagnostics for incorrect `bind()` calls.
- Improved the performance of `std::string` and `std::wstring` move constructors by more than three times.

For a complete list of Standard Library improvements see the Standard Library Fixes In VS 2017 RTM.

Visual Studio 2017 version 15.3

C++17 features

Several additional C++17 features have been implemented. For more information, see Visual C++ Language Conformance.

Other new features

- Implemented P0602R0 "variant and optional should propagate copy/move triviality".
- The Standard Library now officially tolerates dynamic RTTI being disabled via the /GR- option. Both `dynamic_pointer_cast()` and `rethrow_if_nested()` inherently require `dynamic_cast`, so the Standard Library now marks them as `=delete` under **/GR-**.
- Even when dynamic RTTI has been disabled via **/GR-**, "static RTTI" (in the form of `typeid(SomeType)`) is still available and powers several Standard Library components. The Standard Library now supports disabling this too, via **/D_HAS_STATIC_RTTI=0**. Note that this will disable `std::any`, the `target()` and `target_type()` member functions of `std::function`, and the `get_deleter()` friend member function of `std::shared_ptr` and `std::weak_ptr`.

Correctness fixes in Visual Studio 2017 version 15.3

- Standard Library containers now clamp their `max_size()` to `numeric_limits<difference_type>::max()` rather than the `max()` of `size_type`. This ensures that the result of `distance()` on iterators from that container is representable in the return type of `distance()`.
- Fixed missing specialization `auto_ptr<void>`.
- The `for_each_n()`, `generate_n()`, and `search_n()` algorithms previously failed to compile if the length argument was not an integral type; they now attempt to convert non-integral lengths to the iterators' `difference_type`.
- `normal_distribution<float>` no longer emits warnings inside the Standard Library about narrowing from double to float.
- Fixed some `basic_string` operations which were comparing with `npos` instead of `max_size()` when checking for maximum size overflow.
- `condition_variable::wait_for(lock, relative_time, predicate)` would wait for the entire relative time in the event of a spurious wake. Now it will wait for only a single interval of the relative time.
- `future::get()` now invalidates the `future`, as the standard requires.
- `iterator_traits<void *>` used to be a hard error because it attempted to form `void&`; it now cleanly becomes an empty struct to allow use of `iterator_traits` in "is iterator" SFINAE conditions.
- Some warnings reported by Clang **-Wsystem-headers** were fixed.
- Also fixed "exception specification in declaration does not match previous declaration" reported by Clang **-Wmicrosoft-exception-spec**.
- Also fixed mem-initializer-list ordering warnings reported by Clang and C1XX.
- The unordered containers did not swap their hashers or predicates when the containers themselves were swapped. Now they do.
- Many container swap operations are now marked `noexcept` (as our Standard Library never intends to throw an exception when detecting the non-`propagate_on_container_swap` non-equal-allocator undefined behavior condition).
- Many `vector<bool>` operations are now marked `noexcept`.
- The Standard Library will now enforce matching allocator `value_type` (in C++17 mode) with an opt-out escape hatch.
- Fixed some conditions where self-range-insert into `basic_string` would scramble the strings contents. (Note: self-range-insert into vectors is still prohibited by the Standard.)
- `basic_string::shrink_to_fit()` is no longer affected by the allocator's `propagate_on_container_swap`.
- `std::decay` now handles abominable function types (i.e. function types that are cv-qualified and/or ref-qualified).

- Changed include directives to use proper case sensitivity and forward slashes, improving portability.
- Fixed warning C4061 "enumerator '*enumerator*' in switch of enum '*enumeration*' is not explicitly handled by a case label". This warning is off-by-default and was fixed as an exception to the Standard Library's general policy for warnings. (The Standard Library is **/W4** clean, but does not attempt to be **/Wall** clean. Many off-by-default warnings are extremely noisy and aren't intended to be used on a regular basis.)
- Improved `std::list` debug checks. List iterators now check `operator->()`, and `list::unique()` now marks iterators as invalidated.
- Fixed uses-allocator metaprogramming in `tuple`.

Performance/throughput fixes

- Worked around interactions with `noexcept` which prevented inlining the `std::atomic` implementation into functions that use Structured Exception Handling (SEH).
- Changed the Standard Library's internal `_Deallocate()` function to optimize into smaller code, allowing it to be inlined into more places.
- Changed `std::try_lock()` to use pack expansion instead of recursion.
- Improved the `std::lock()` deadlock avoidance algorithm to use `lock()` operations instead of spinning on `try_lock()` on all the locks.
- Enabled the Named Return Value Optimization in `system_category::message()`.
- `conjunction` and `disjunction` now instantiate N + 1 types, instead of 2N + 2 types.
- `std::function` no longer instantiates allocator support machinery for each type-erased callable, improving throughput and reducing .obj size in programs that pass many distinct lambdas to `std::function`.
- `allocator_traits<std::allocator>` contains manually inlined `std::allocator` operations, reducing code size in code that interacts with `std::allocator` through `allocator_traits` only (that is, in most code).
- The C++11 minimal allocator interface is now handled by the Standard Library calling `allocator_traits` directly, instead of wrapping the allocator in an internal class `_Wrap_alloc`. This reduces the code size generated for allocator support, improves the optimizer's ability to reason about Standard Library containers in some cases, and provides a better debugging experience (as now you see your allocator type, rather than `_Wrap_alloc<your_allocator_type>` in the debugger).
- Removed metaprogramming for customized `allocator::reference`, which allocators aren't actually allowed to customize. (Allocators can make containers use fancy pointers but not fancy references.)
- The compiler front-end was taught to unwrap debug iterators in range-based for-loops, improving the performance of debug builds.
- The `basic_string` internal shrink path for `shrink_to_fit()` and `reserve()` is no longer in the path of reallocating operations, reducing code size for all mutating members.
- The `basic_string` internal grow path is no longer in the path of `shrink_to_fit()`.
- The `basic_string` mutating operations are now factored into non-allocating fast path and allocating slow path functions, making it more likely for the common no-reallocate case to be inlined into callers.
- The `basic_string` mutating operations now construct reallocated buffers in the desired state rather than resizing in place. For example, inserting at the beginning of a string now moves the content after the insertion exactly once (either down or to the newly allocated buffer), instead of twice in the reallocating case (to the newly allocated buffer and then down).
- Operations calling the C standard library in <string> now cache the `errno` address to remove repeated interaction with TLS.
- Simplified the `is_pointer` implementation.
- Finished changing function-based Expression SFINAE to `struct` and `void_t`-based.
- Standard Library algorithms now avoid postincrementing iterators.
- Fixed truncation warnings when using 32-bit allocators on 64-bit systems.
- `std::vector` move assignment is now more efficient in the non-POCMA non-equal-allocator case, by reusing the buffer when possible.

Readability and other improvements

- The Standard Library now uses C++14 `constexpr` unconditionally, instead of conditionally-defined macros.
- The Standard Library now uses alias templates internally.
- The Standard Library now uses `nullptr` internally, instead of `nullptr_t{}`. (Internal usage of NULL has been eradicated. Internal usage of 0-as-null is being cleaned up gradually.)
- The Standard Library now uses `std::move()` internally, instead of stylistically misusing `std::forward()`.
- Changed `static_assert(false, "message")` to `#error message`. This improves compiler diagnostics because `#error` immediately stops compilation.
- The Standard Library no longer marks functions as `__declspec(dllimport)`. Modern linker technology no longer requires this.
- Extracted SFINAE to default template arguments, which reduces clutter compared to return types and function argument types.
- Debug checks in <random> now use the Standard Library's usual machinery, instead of the internal function `_Rng_abort()` which called `fputs()` to **stderr**. This function's implementation is being retained for binary compatibility, but has been removed in the next binary-incompatible version of the Standard Library.

Visual Studio 2017 version 15.5

Several Standard Library features have been added, deprecated or removed in accordance with the C++17 standard. For more information see C++ conformance improvements in Visual Studio.

New experimental features

- Experimental support for the following parallel algorithms:
  - `all_of`
  - `any_of`
  - `for_each`
  - `for_each_n`
  - `none_of`
  - `reduce`
  - `replace`
  - `replace_if`
  - `sort`
- The signatures for the following parallel algorithms are added but not parallelized at this time; profiling showed no benefit in parallelizing algorithms that only move or permute elements:
  - `copy`
  - `copy_n`
  - `fill`
  - `fill_n`
  - `move`
  - `reverse`
  - `reverse_copy`
  - `rotate`
  - `rotate_copy`
  - `swap_ranges`

Performance fixes and improvements

- `basic_string<char16_t>` now engages the same `memcmp`, `memcpy`, and similar optimizations that `basic_string<wchar_t>` engages.
- An optimizer limitation which prevented function pointers from being inlined exposed by our "avoid copying functions" work in Visual Studio 2015 Update 3 has been worked around, restoring performance of

`lower_bound(iter, iter, function pointer)`.

- The overhead of iterator debugging's order verification of inputs to `includes`, `set_difference`, `set_symmetric_difference`, and `set_union` was reduced by unwrapping iterators before checking order.
- `std::inplace_merge` now skips over elements that are already in position.
- Constructing `std::random_device` no longer constructs and then destroys a `std::string`.
- `std::equal` and `std::partition` had a jump-threading optimization pass which saves an iterator comparison.
- When `std::reverse` is passed pointers to trivially copyable `T`, it will now dispatch to a handwritten vectorized implementation.
- `std::fill`, `std::equal`, and `std::lexicographical_compare` were taught how to dispatch to `memset` and `memcmp` for `std::byte` and `gsl::byte` (and other char-like enums and enum classes). Note that `std::copy` dispatches using `is_trivially_copyable` and thus didn't need any changes.
- The Standard Library no longer contains empty-braces destructors whose only behavior was to make types non-trivially-destructible.

Correctness fixes in Visual Studio 2017 version 15.5

- `std::partition` now calls the predicate N times instead of N + 1 times, as the standard requires.
- Attempts to avoid magic statics in version 15.3 have been repaired in version 15.5.
- `std::atomic<T>` no longer requires `T` to be default constructible.
- Heap algorithms that take logarithmic time no longer do a linear time assertion that the input is in fact a heap when iterator debugging is enabled.
- `__declspec(allocator)` is now guarded for C1XX only, to prevent warnings from Clang which doesn't understand this declspec.
- `basic_string::npos` is now available as a compile time constant.
- `std::allocator` in C++17 mode now properly handles allocation of over-aligned types, that is, types whose alignment is greater than `max_align_t`, unless disabled by **/Zc:alignedNew-**. For example, vectors of objects with 16 or 32-byte alignment will now be properly aligned for SSE and AVX instructions.

Visual Studio 2017 version 15.6

- <memory_resource>
- Library Fundamentals V1
- Deleting polymorphic_allocator assignment
- Improving class template argument deduction

Visual Studio 2017 version 15.7

- support for parallel algorithms is no longer experimental
- a new implementation of <filesystem>
- elementary string conversions (partial)
- std::launder()
- std::byte
- hypot(x,y,z)
- avoiding unnecessary decay
- mathematical special functions
- constexpr char_traits
- deduction guides for STL

See Visual C++ language conformance for more information.

Other Libraries

Open source library support

Vcpkg is an open-source command line tool that greatly simplifies the process of acquiring and building open source C++ static libs and DLLS in Visual Studio. For more information, see vcpkg: A package manager for C++.

Visual Studio 2017 version 15.5:

CPPRest SDK 2.9.0

The CPPRestSDK, a cross-platform web API for C++, has been updated to version 2.9.0. For more information, see CppRestSDK 2.9.0 is available on GitHub.

ATL

- Yet another set of name-lookup conformance fixes
- Existing move constructors and move assignment operators are now properly marked as non-throwing
- Un-suppress valid warning C4640 about thread safe init of local statics in atlstr.h
- Thread Safe Initialization of local statics was automatically turned off in the XP toolset when [using ATL AND building a DLL]. This is no longer the case. You can add **/Zc:threadSafeInit-** in your Project settings if having thread safe initialization off is desired.

Visual C++ runtime

- New header "cfguard.h" for Control Flow Guard symbols.

C++ IDE

- Configuration change performance is now better for C++ native projects and much better for C++/CLI projects. When a solution configuration is activated for the first time it will now be faster and all subsequent activations of this solution configuration will be almost instantaneous.

Visual Studio 2017 version 15.3:

- Several project and code wizards have been rewritten in the signature dialog style.
- **Add Class** now launches the Add Class wizard directly. All of the other items that were previously here are now available under **Add > New Item**.
- Win32 projects are now under the **Windows Desktop** category in the **New Project** dialog.
- The **Windows Console** and **Desktop Application** templates now create the projects without displaying a wizard. There's a new **Windows Desktop Wizard** under the same category that displays the same options as the old **Win32 Console Application** wizard.

Visual Studio 2017 version 15.5:

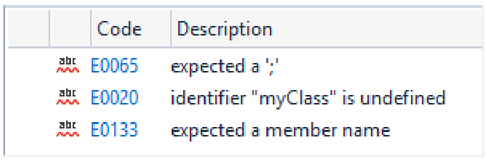
Several C++ operations that use the IntelliSense engine for refactoring and code navigation run much faster. The following numbers are based on the Visual Studio Chromium solution with 3500 projects:

Feature	Performance Improvement
Rename	5.3x
Change Signature	4.5x
Find All References	4.7x

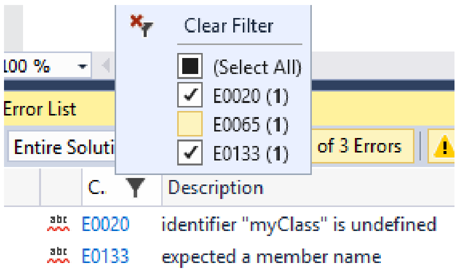
C++ now supports Ctrl+Click **Go To Definition**, making mouse navigation to definitions easy. The Structure Visualizer from the Productivity Power Tools pack is now also included in the product by default.

IntelliSense

- The new SQLite-based database engine is now being used by default. This will speed up database operations like **Go To Definition** and **Find All References**, and will significantly improve initial solution parse time. The setting has been moved to **Tools > Options > Text Editor > C/C++ > Advanced** (it was formerly under ...C/C++ | Experimental).
- We've improved IntelliSense performance on projects and files not using precompiled headers - an Automatic Precompiled Header will be created for headers in the current file.
- We've added error filtering and help for IntelliSense errors in the error list. Clicking on the error column now allows for filtering. Also, clicking on the specific errors or pressing F1 will launch an online search for the error message.

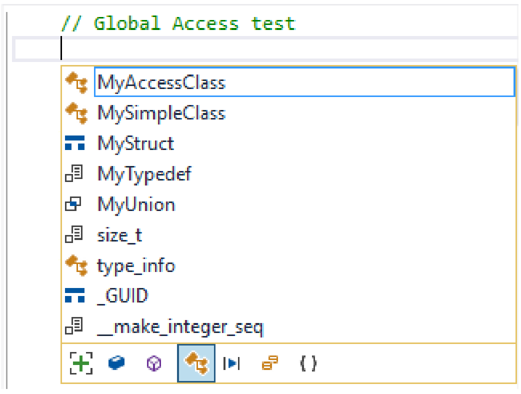


Code	Description
E0065	expected a ';'
E0020	identifier "myClass" is undefined
E0133	expected a member name



C.	Description
E0020	identifier "myClass" is undefined
E0133	expected a member name

- Added the ability to filter Member List items by kind.



```
// Global Access test
```

- Added a new experimental Predictive IntelliSense feature that provides contextually-aware filtering of what appears in the Member List. See C++ IntelliSense Improvements - Predictive IntelliSense & Filtering
- **Find All References** (Shift+F12) now helps you get around easily, even in complex codebases. It provides advanced grouping, filtering, sorting, searching within results, and (for some languages) colorization, so you can get a clear understanding of your references. For C++, the new UI includes information about whether we are reading from or writing to a variable.
- The Dot-to-Arrow IntelliSense feature has been moved from experimental to advanced, and is now enabled by default. The editor features **Expand Scopes** and **Expand Precedence** have also been moved from experimental to advanced.
- The experimental refactoring features **Change Signature** and **Extract Function** are now available by default.
- The experimental 'Faster project load' feature for C++ projects. The next time you open a C++ project it will load faster, and the time after that it will load really fast!
- Some of these features are common to other languages, and some are specific to C++. For more

information about these new features, see Announcing Visual Studio "15".

Visual Studio 1027 version 15.7: Support added for ClangFormat. For more information, see ClangFormat Support in Visual Studio 2017.

Non-MSBuild projects with Open Folder

Visual Studio 2017 introduces the **Open Folder** feature, which enables you to code, build and debug in a folder containing source code without the need to create any solutions or projects. This makes it much simpler to get started with Visual Studio even if your project is not an MSBuild-based project. With **Open Folder** you get access to the powerful code understanding, editing, building and debugging capabilities that Visual Studio already provides for MSBuild projects. For more information, see Open Folder projects for C++.

- Improvements to the Open Folder experience. You can customize the experience through these .json files:
  - CppProperties.json to customize the IntelliSense and browsing experience.
  - Tasks.json to customize the build steps.
  - Launch.json to customize the debugging experience.

Visual Studio 2017 version 15.3:

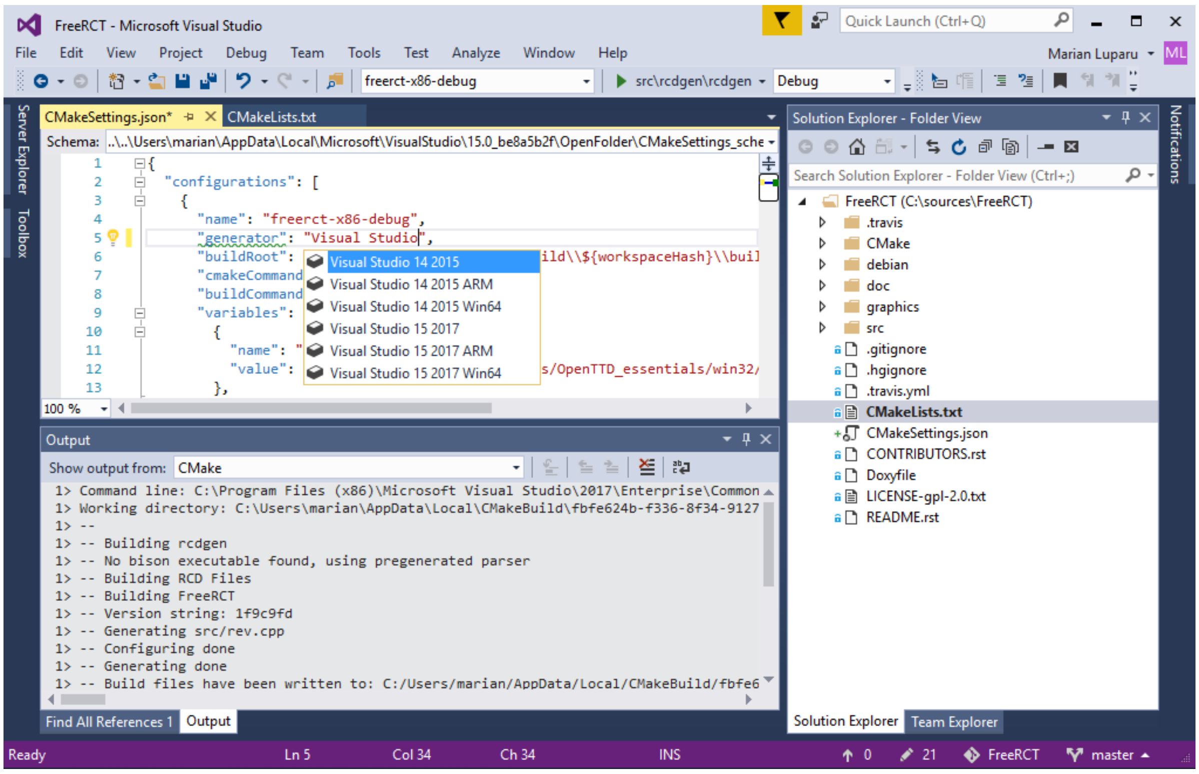
- Improved support for alternative compilers and build environments such as MinGW and Cygwin. For more information, see Using MinGW and Cygwin with Visual C++ and Open Folder.
- Added support to define global and configuration-specific environment variables in CppProperties.json and CMakeSettings.json. These environment variables can be consumed by debug configurations defined in launch.vs.json and tasks in tasks.vs.json. For more information, see Customizing your Environment with Visual C++ and Open Folder.
- Improved support for CMake's Ninja generator, including the ability to easily target 64-bit platforms.

CMake support via Open Folder

Visual Studio 2017 introduces support for using CMake projects without converting to MSBuild project files (.vcxproj). For more information, see CMake projects in Visual Studio. Opening CMake projects with **Open Folder** automatically configures the environment for C++ editing, building and debugging.

- C++ IntelliSense works without the need to create a CppProperties.json file in the root folder. Along with this, we've added a new dropdown to allow users to easily switch between configurations provided by CMake and CppProperties.json files.

- Further configuration is supported via a CMakeSettings.json file that sits in the same folder as the CMakeLists.txt file.


FreeRCT - Microsoft Visual Studio
File Edit View Project Debug Team Tools Test Analyze Window Help
freerct-x86-debug
src\rcdgen\rcdgen
Debug
Quick Launch (Ctrl+Q)
Marian Lupary
CMakeSettings.json*
CMakeLists.txt
Schema: ..\..\Users\marian\AppData\Local\Microsoft\VisualStudio\15.0_be8a5b2f\OpenFolder\CMakeSettings_sche
Visual Studio 14 2015
Visual Studio 14 2015 ARM
Visual Studio 14 2015 Win64
Visual Studio 15 2017
Visual Studio 15 2017 ARM
Visual Studio 15 2017 Win64
Solution Explorer - Folder View
Search Solution Explorer - Folder View (Ctrl+;)
FreeRCT (C:\sources\FreeRCT)
.travis
CMake
debian
doc
graphics
src
.gitignore
.hgignore
.travis.yml
CMakeLists.txt
CMakeSettings.json
CONTRIBUTORS.rst
Doxyfile
LICENSE-gpl-2.0.txt
README.rst
Solution Explorer
Team Explorer
Find All References 1
Output
Ready Ln 5 Col 34 Ch 34 INS
0
21
FreeRCT
master
Server Explorer
Toolbox
Notifications

```
{
  "configurations": [
    {
      "name": "freerct-x86-debug",
      "generator": "Visual Studio",
      "buildRoot":                                ild\\${workspaceHash}\\buil
      "cmakeCommand
      "buildCommand
      "variables":
        {
          "name": "
          "value":                                s/OpenTTD_essentials/win32/
        },
```

Show output from: CMake

```
1> Command line: C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise\Common
1> Working directory: C:\Users\marian\AppData\Local\CMakeBuild\fbfe624b-f336-8f34-9127
1> --
1> -- Building rcdgen
1> -- No bison executable found, using pregenerated parser
1> -- Building RCD Files
1> -- Building FreeRCT
1> -- Version string: 1f9c9fd
1> -- Generating src/rev.cpp
1> -- Configuring done
1> -- Generating done
1> -- Build files have been written to: C:/Users/marian/AppData/Local/CMakeBuild/fbfe6
```

Visual Studio 2017 version 15.3: Support added for the CMake Ninja generator.

Visual Studio 2017 version 15.5: Support added for importing existing CMake caches.

Visual Studio 2017 version 15.7: Support added for CMake 3.11, code analysis in CMake projects, Targets view in Solution Explorer, options for cache generation, and single file compilation. For more information, see CMake Support in Visual Studio and CMake projects in Visual Studio.

Windows desktop development with C++

We now provide a more granular installation experience for installing the original C++ workload. We have added selectable components that enable you to install just the tools that you need. Please note that the indicated installation sizes for the components listed in the installer UI are not accurate and underestimate the total size.

To successfully create Win32 projects in the C++ desktop workload, you must install both a toolset and a Windows SDK. Installing the recommended (selected) components **VC++ 2017 v141 toolset (x86, x64)** and **Windows 10 SDK (10.0.nnnnn)** ensures this will work. If the necessary tools are not installed, projects will not be created successfully and the wizard will hang.

Visual Studio 2017 version 15.5:

The Visual C++ Build tools (previously available as a standalone product) are now included as a workload in the Visual Studio Installer. This workload installs only the tools required to build C++ projects without installing the Visual Studio IDE. Both the v140 and v141 toolsets are included. The v141 toolset contains the latest improvements in Visual Studio 2017 version 15.5. For more information, see Visual Studio Build Tools now include the VS2017 and VS2015 MSVC Toolsets.

Linux development with C++

The popular extension Visual C++ for Linux Development is now part of Visual Studio. This installation provides everything you need to develop and debug C++ applications running on a Linux environment.

Visual Studio 2017 version 15.2:

Improvements have been made in cross-platform code sharing and type visualization. For more information, see

Linux C++ improvements for cross-platform code sharing and type visualization.

Visual Studio 2017 version 15.5:

- The Linux workload has added support for **rsync** as an alternative to **sftp** for synchronizing files to remote Linux machines.
- Support is added for cross compilation targeting ARM microcontrollers. To enable this in the installation, choose the **Linux development with C++** workload and select the option for **Embedded and IoT Development**. This adds the ARM GCC cross compilation tools and Make to your installation. For more information, see ARM GCC Cross Compilation in Visual Studio.
- Support added for CMake. You can now work on your existing CMake code base without having to convert it to a Visual Studio project. For more information, see Configure a Linux CMake Project.
- Support added for running remote tasks. This capability allows you to run any command on a remote system that is defined in Visual Studio's Connection Manager. Remote tasks also provide the capability to copy files to the remote system. For more information, see Configure a Linux CMake Project.

Visual Studio 2017 version 15.7:

- Various improvements to Linux workload scenarios. For more information, see Linux C++ Workload improvements to the Project System, Linux Console Window, rsync and Attach to Process.
- IntelliSense for headers on remote Linux connections. For more information, see IntelliSense for Remote Linux Headers and Configure a Linux CMake Project.

Game development with C++

Use the full power of C++ to build professional games powered by DirectX or Cocos2d.

Mobile development with C++ (Android and iOS)

You can now create and debug mobile apps using Visual Studio that can target Android and iOS.

Universal Windows Apps

C++ comes as an optional component for the Universal Windows App workload. Upgrading C++ projects currently must be done manually. If you open a v140-targeted UWP project in Visual Studio 2017, you need to select the v141 platform toolset in the project property pages if you do not have Visual Studio 2015 installed.

New options for C++ on Universal Windows Platform (UWP)

You now have new options for writing and packaging C++ applications for the Universal Windows Platform and the Windows Store: You can use the Desktop Bridge infrastructure to package your existing desktop application or COM object for deployment through the Windows Store or through your existing channels via side-loading. New capabilities in Windows 10 enable you to add UWP functionality to your desktop application in various ways. For more information, see Desktop Bridge.

Visual Studio 2017 version 15.5: A **Windows Application Packaging Project** project template is added which greatly simplifies the work of packaging desktop applications with using Desktop Bridge. It is available under **File | New | Project | Installed | Visual C++ | Universal Windows Platform**. For more information, see Package an app by using Visual Studio (Desktop Bridge).

When writing new code, you can now use C++/WinRT, a standard C++ language projection for the Windows Runtime implemented solely in header files. It allows you to both author and consume Windows Runtime APIs using any standards-compliant C++ compiler. C++/WinRT is designed to provide C++ developers with first-class access to the modern Windows API. For more information, see C++/WinRT Available on GitHub.

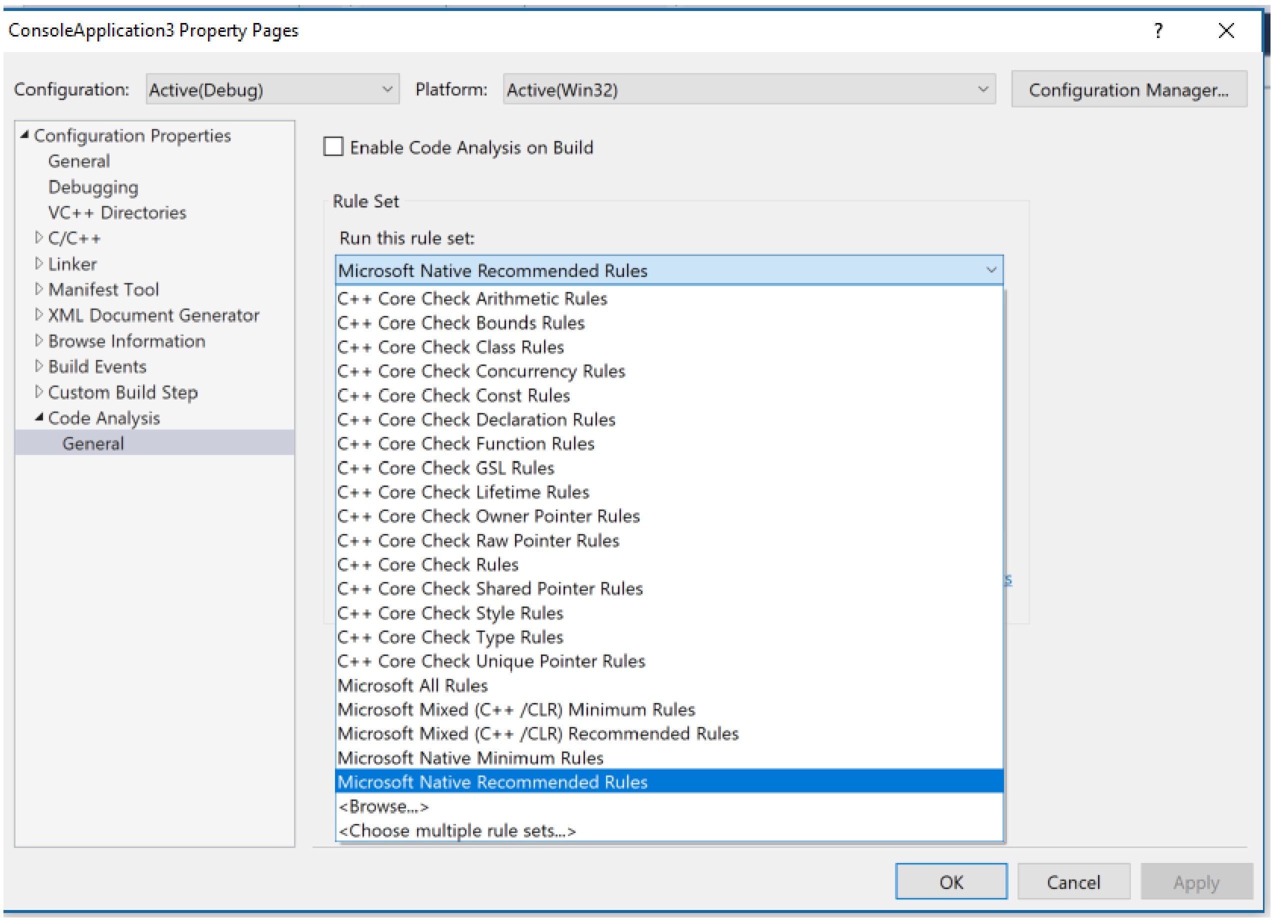
As of build 17025 of the Windows SDK Insider Preview, C++/WinRT is included in the Windows SDK. For more information, see C++/WinRT is now included the Windows SDK.

Clang/C2 platform toolset

The Clang/C2 toolset that ships with Visual Studio 2017 now supports the **/bigobj** switch, which is crucial for building large projects. It also includes several important bug fixes, both in the front-end and the back-end of the compiler.

C++ code analysis

The C++ Core Checkers for enforcing the C++ Core Guidelines are now distributed with Visual Studio. Simply enable the checkers in the **Code Analysis Extensions** page in the project's property pages and the extensions will be included when you run code analysis. For more information, see Using the C++ Core Guidelines checkers.


ConsoleApplication3 Property Pages

Configuration: Active(Debug)   Platform: Active(Win32)   Configuration Manager...

☐ Enable Code Analysis on Build

Rule Set

Run this rule set: Microsoft Native Recommended Rules

- C++ Core Check Arithmetic Rules
- C++ Core Check Bounds Rules
- C++ Core Check Class Rules
- C++ Core Check Concurrency Rules
- C++ Core Check Const Rules
- C++ Core Check Declaration Rules
- C++ Core Check Function Rules
- C++ Core Check GSL Rules
- C++ Core Check Lifetime Rules
- C++ Core Check Owner Pointer Rules
- C++ Core Check Raw Pointer Rules
- C++ Core Check Rules
- C++ Core Check Shared Pointer Rules
- C++ Core Check Style Rules
- C++ Core Check Type Rules
- C++ Core Check Unique Pointer Rules
- Microsoft All Rules
- Microsoft Mixed (C++ /CLR) Minimum Rules
- Microsoft Mixed (C++ /CLR) Recommended Rules
- Microsoft Native Minimum Rules
- Microsoft Native Recommended Rules
- <Browse...>
- <Choose multiple rule sets...>

Visual Studio 2017 version 15.3: Support added for rules related to resource management.

Visual Studio 2017 version 15.5: New C++ Core Guidelines checks cover smart pointer correctness, correct use of global initializers, and flagging uses of constructs like `goto` and bad casts.

Some warning numbers you may find in 15.3 are no longer available in 15.5. These warnings were replaced with more specific checks.

Visual Studio 2017 version 15.6:

- Support added for single-file analysis, and improvements in analysis run-time performance. For more information, see C++ Static Analysis Improvements for Visual Studio 2017 15.6 Preview 2

Visual Studio 2017 version 15.7:

- Support added for /analyze:ruleset which enables you to specify which code analysis rules to run.
- Support added for additional C++ Core Guidelines rules. For more information, see Using the C++ Core

Guidelines checkers.

Unit testing

Visual Studio 2017 version 15.5:

Google Test Adapter and Boost.Test Adapter are now available as components of the **Desktop Development with C++** workload, and are integrated with **Test Explorer**. CTest support is added for Cmake projects (using Open Folder) although full integration with **Test Explorer** is not yet available. For more information, see Writing unit tests for C/C++.

Visual Studio 2017 version 15.6:

- Support added for Boost.Test dynamic library support.
- A Boost.Test item template is now available in the IDE.

For more information, see Boost.Test Unit Testing: Dynamic Library support and New Item Template.

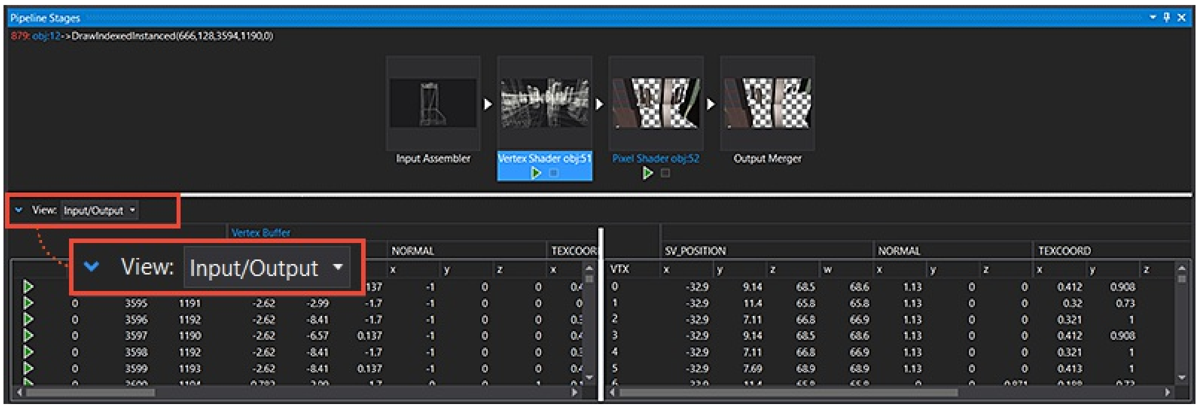
Visual Studio 2017 version 15.7:

CodeLens supported added for C++ unit test projects. For more information, see Announcing CodeLens for C++ Unit Testing.

Visual Studio graphics diagnostics

Visual Studio Graphics Diagnostics is a set of tools for recording and analyzing rendering and performance problems in Direct3D apps. Graphics Diagnostics features can be used with apps that are running locally on your Windows PC, in a Windows device emulator, or on a remote PC or device.

- **Input & Output for Vertex and Geometry shaders:** The ability to view input and output of vertex shaders and geometry shaders has been one of the most requested features, and it is now supported in the tools. Simply select the VS or GS stage in the Pipeline Stages view to start inspecting its input and output in the table below.

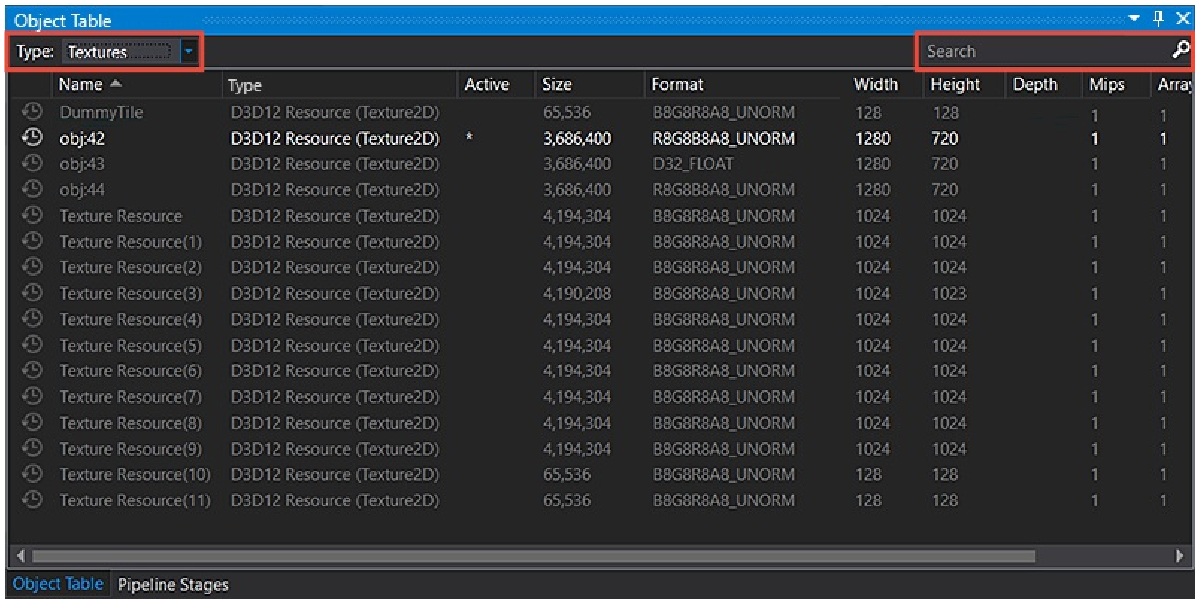


879: obj:12->DrawIndexedInstanced(666,128,3594,1190,0)

							NORMAL			TEXCOOR
							x	y	z	x
▷						137	-1	0	0	0.4
▷	0	3595	1191	-2.62	-2.99	-1.7	-1	0	0	0
▷	0	3596	1192	-2.62	-8.41	-1.7	-1	0	0	0.3
▷	0	3597	1190	-2.62	-6.57	0.137	-1	0	0	0.4
▷	0	3598	1192	-2.62	-8.41	-1.7	-1	0	0	0.3
▷	0	3599	1193	-2.62	-8.41	0.137	-1	0	0	0.4

	SV_POSITION				NORMAL			TEXCOORD		
VTX	x	y	z	w	x	y	z	x	y	z
0	-32.9	9.14	68.5	68.6	1.13	0	0	0.412	0.908	
1	-32.9	11.4	65.8	65.8	1.13	0	0	0.32	0.73	
2	-32.9	7.11	66.8	66.9	1.13	0	0	0.321	1	
3	-32.9	9.14	68.5	68.6	1.13	0	0	0.412	0.908	
4	-32.9	7.11	66.8	66.9	1.13	0	0	0.321	1	
5	-32.9	7.69	68.9	68.9	1.13	0	0	0.413	1	

- **Search and filter in the object table:** Provides a quick and easy way to find the resources you're looking for.

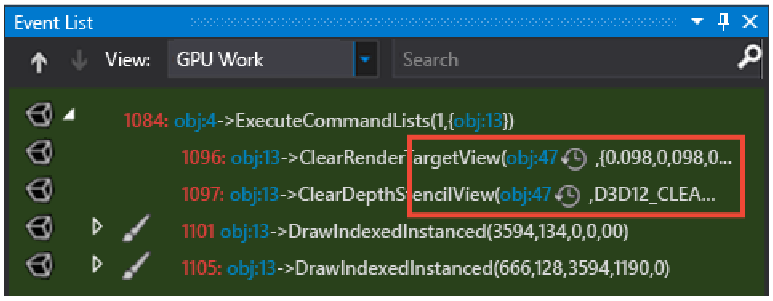

Object Table

Type: Textures

Search

Name	Type	Active	Size	Format	Width	Height	Depth	Mips	Array
DummyTile	D3D12 Resource (Texture2D)		65,536	B8G8R8A8_UNORM	128	128		1	1
obj:42	D3D12 Resource (Texture2D)	*	3,686,400	R8G8B8A8_UNORM	1280	720		1	1
obj:43	D3D12 Resource (Texture2D)		3,686,400	D32_FLOAT	1280	720		1	1
obj:44	D3D12 Resource (Texture2D)		3,686,400	R8G8B8A8_UNORM	1280	720		1	1
Texture Resource	D3D12 Resource (Texture2D)		4,194,304	B8G8R8A8_UNORM	1024	1024		1	1
Texture Resource(1)	D3D12 Resource (Texture2D)		4,194,304	B8G8R8A8_UNORM	1024	1024		1	1
Texture Resource(2)	D3D12 Resource (Texture2D)		4,194,304	B8G8R8A8_UNORM	1024	1024		1	1
Texture Resource(3)	D3D12 Resource (Texture2D)		4,190,208	B8G8R8A8_UNORM	1024	1023		1	1
Texture Resource(4)	D3D12 Resource (Texture2D)		4,194,304	B8G8R8A8_UNORM	1024	1024		1	1
Texture Resource(5)	D3D12 Resource (Texture2D)		4,194,304	B8G8R8A8_UNORM	1024	1024		1	1
Texture Resource(6)	D3D12 Resource (Texture2D)		4,194,304	B8G8R8A8_UNORM	1024	1024		1	1
Texture Resource(7)	D3D12 Resource (Texture2D)		4,194,304	B8G8R8A8_UNORM	1024	1024		1	1
Texture Resource(8)	D3D12 Resource (Texture2D)		4,194,304	B8G8R8A8_UNORM	1024	1024		1	1
Texture Resource(9)	D3D12 Resource (Texture2D)		4,194,304	B8G8R8A8_UNORM	1024	1024		1	1
Texture Resource(10)	D3D12 Resource (Texture2D)		65,536	B8G8R8A8_UNORM	128	128		1	1
Texture Resource(11)	D3D12 Resource (Texture2D)		65,536	B8G8R8A8_UNORM	128	128		1	1

- **Resource History:** This new view provides a streamlined way of seeing the entire modification history of a resource as it was used during the rendering of a captured frame. To invoke the history for any resource, simply click the clock icon next to any resource hyperlink.

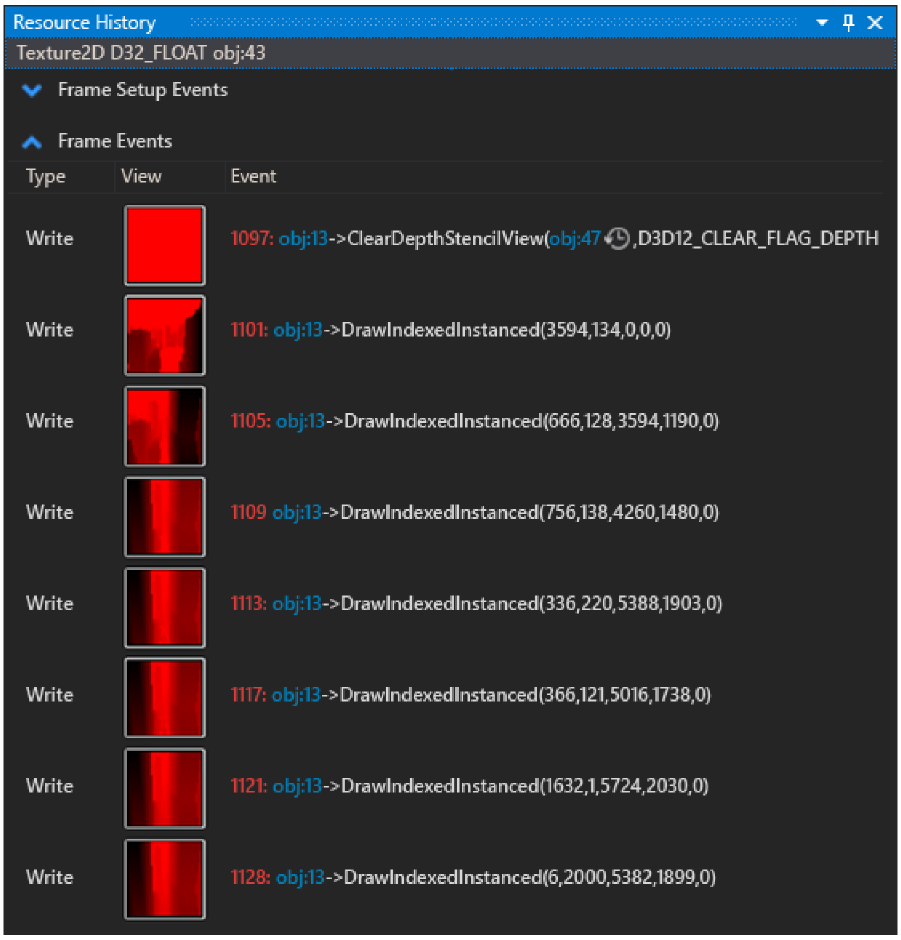

Event List

View: GPU Work

Search

```
1084: obj:4->ExecuteCommandLists(1,{obj:13})
    1096: obj:13->ClearRenderTargetView(obj:47 ,{0.098,0,098,0...
    1097: obj:13->ClearDepthStencilView(obj:47 ,D3D12_CLEA...
    1101 obj:13->DrawIndexedInstanced(3594,134,0,0,00)
    1105: obj:13->DrawIndexedInstanced(666,128,3594,1190,0)
```

This will display the new **Resource History** tool window, populated with the change history of the resource.



Resource History

Texture2D D32_FLOAT obj:43

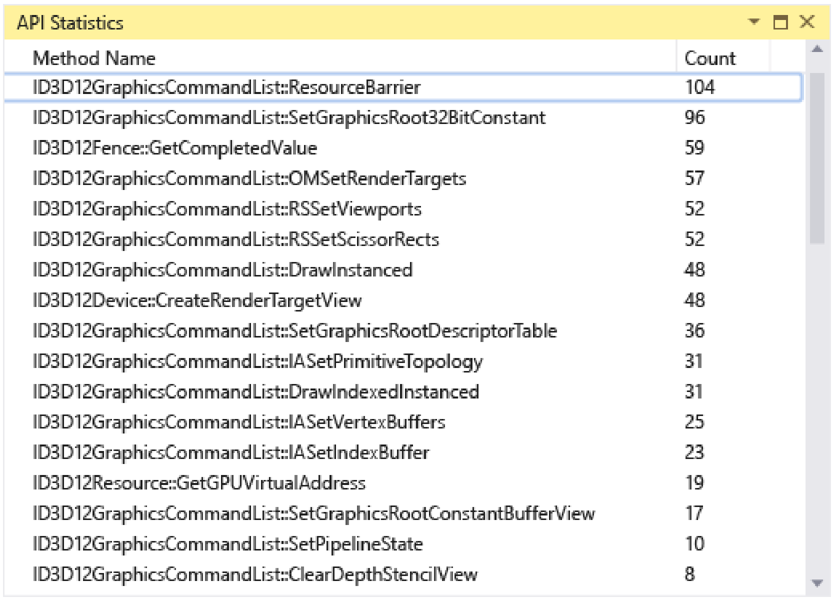
Frame Setup Events

Frame Events

Type	View	Event
Write		1097: obj:13->ClearDepthStencilView(obj:47 ,D3D12_CLEAR_FLAG_DEPTH
Write		1101: obj:13->DrawIndexedInstanced(3594,134,0,0,0)
Write		1105: obj:13->DrawIndexedInstanced(666,128,3594,1190,0)
Write		1109 obj:13->DrawIndexedInstanced(756,138,4260,1480,0)
Write		1113: obj:13->DrawIndexedInstanced(336,220,5388,1903,0)
Write		1117: obj:13->DrawIndexedInstanced(366,121,5016,1738,0)
Write		1121: obj:13->DrawIndexedInstanced(1632,1,5724,2030,0)
Write		1128: obj:13->DrawIndexedInstanced(6,2000,5382,1899,0)

Note that if your frame was captured with full call stack capturing enabled (**Visual Studio** > **Tools** > **Options** under **Graphics Diagnostics**), then the context of each change event can be quickly deduced and inspected within your Visual Studio project.

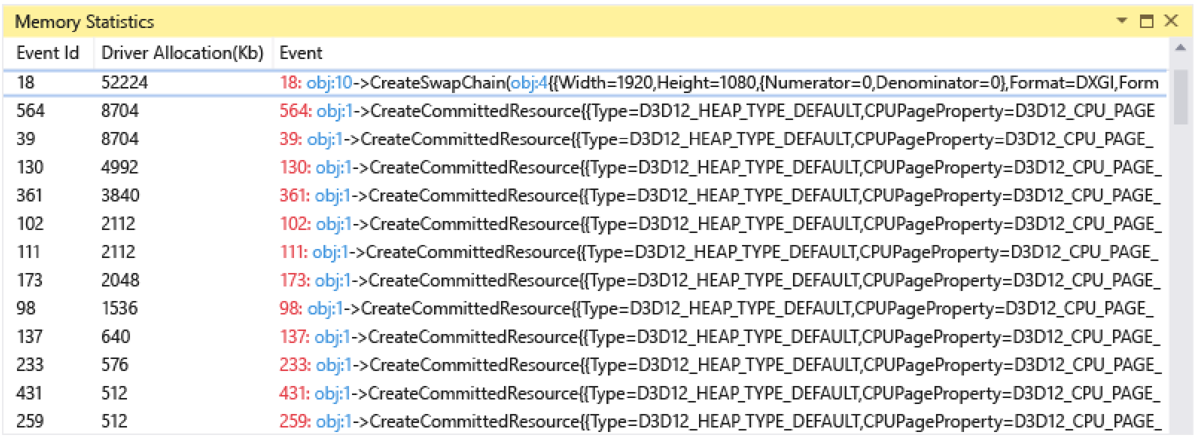
- **API Statistics:** View a high-level summary of API usage in your frame. This can be handy in discovering calls you may not realize you're making at all or calls you are making too much. This window is available via **View** > **API Statistics** in Visual Studio Graphics Analyzer.



API Statistics

Method Name	Count
ID3D12GraphicsCommandList::ResourceBarrier	104
ID3D12GraphicsCommandList::SetGraphicsRoot32BitConstant	96
ID3D12Fence::GetCompletedValue	59
ID3D12GraphicsCommandList::OMSetRenderTargets	57
ID3D12GraphicsCommandList::RSSetViewports	52
ID3D12GraphicsCommandList::RSSetScissorRects	52
ID3D12GraphicsCommandList::DrawInstanced	48
ID3D12Device::CreateRenderTargetView	48
ID3D12GraphicsCommandList::SetGraphicsRootDescriptorTable	36
ID3D12GraphicsCommandList::IASetPrimitiveTopology	31
ID3D12GraphicsCommandList::DrawIndexedInstanced	31
ID3D12GraphicsCommandList::IASetVertexBuffers	25
ID3D12GraphicsCommandList::IASetIndexBuffer	23
ID3D12Resource::GetGPUVirtualAddress	19
ID3D12GraphicsCommandList::SetGraphicsRootConstantBufferView	17
ID3D12GraphicsCommandList::SetPipelineState	10
ID3D12GraphicsCommandList::ClearDepthStencilView	8

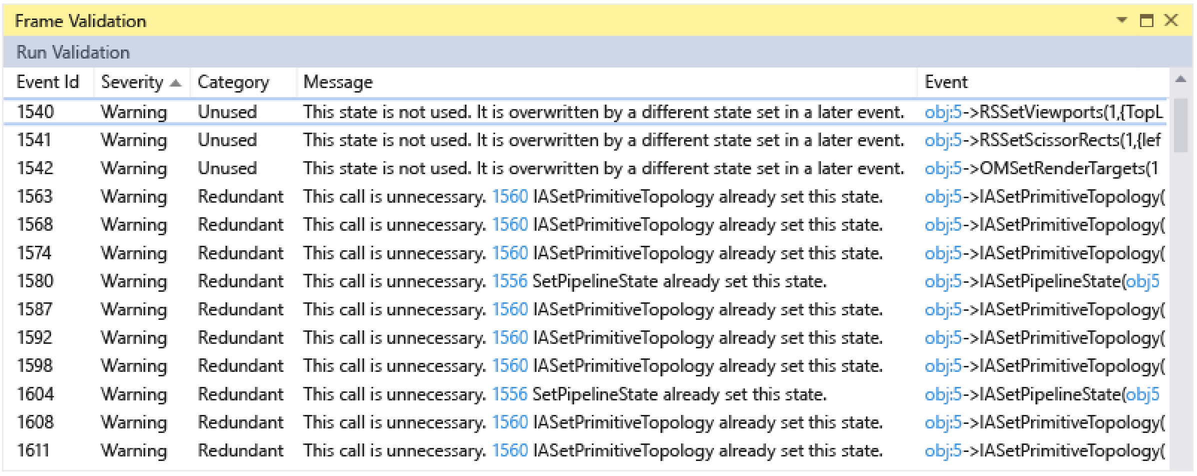
- **Memory Statistics:** View how much memory the driver is allocating for the resources you create in the frame. This window is available via **View** > **Memory Statistics** in **Visual Studio Graphics Analyzer**. Data can be copied to a CSV file for viewing in a spreadsheet by right-clicking and choosing **Copy All**.



Memory Statistics

Event Id	Driver Allocation(Kb)	Event
18	52224	18: obj:10->CreateSwapChain(obj:4{{Width=1920,Height=1080,{Numerator=0,Denominator=0},Format=DXGI,Form
564	8704	564: obj:1->CreateCommittedResource({Type=D3D12_HEAP_TYPE_DEFAULT,CPUPageProperty=D3D12_CPU_PAGE
39	8704	39: obj:1->CreateCommittedResource({Type=D3D12_HEAP_TYPE_DEFAULT,CPUPageProperty=D3D12_CPU_PAGE_
130	4992	130: obj:1->CreateCommittedResource({Type=D3D12_HEAP_TYPE_DEFAULT,CPUPageProperty=D3D12_CPU_PAGE_
361	3840	361: obj:1->CreateCommittedResource({Type=D3D12_HEAP_TYPE_DEFAULT,CPUPageProperty=D3D12_CPU_PAGE_
102	2112	102: obj:1->CreateCommittedResource({Type=D3D12_HEAP_TYPE_DEFAULT,CPUPageProperty=D3D12_CPU_PAGE_
111	2112	111: obj:1->CreateCommittedResource({Type=D3D12_HEAP_TYPE_DEFAULT,CPUPageProperty=D3D12_CPU_PAGE_
173	2048	173: obj:1->CreateCommittedResource({Type=D3D12_HEAP_TYPE_DEFAULT,CPUPageProperty=D3D12_CPU_PAGE_
98	1536	98: obj:1->CreateCommittedResource({Type=D3D12_HEAP_TYPE_DEFAULT,CPUPageProperty=D3D12_CPU_PAGE_
137	640	137: obj:1->CreateCommittedResource({Type=D3D12_HEAP_TYPE_DEFAULT,CPUPageProperty=D3D12_CPU_PAGE_
233	576	233: obj:1->CreateCommittedResource({Type=D3D12_HEAP_TYPE_DEFAULT,CPUPageProperty=D3D12_CPU_PAGE_
431	512	431: obj:1->CreateCommittedResource({Type=D3D12_HEAP_TYPE_DEFAULT,CPUPageProperty=D3D12_CPU_PAGE_
259	512	259: obj:1->CreateCommittedResource({Type=D3D12_HEAP_TYPE_DEFAULT,CPUPageProperty=D3D12_CPU_PAGE_

- **Frame Validation:** The new errors and warnings list provides an easy way to navigate your event list based on potential issues detected by the Direct3D debug layer. Click **View > Frame Validation** in Visual Studio Graphics Analyzer to open the window. Then click **Run Validation** to start the analysis. It can take several minutes to complete, depending on the frame's complexity.

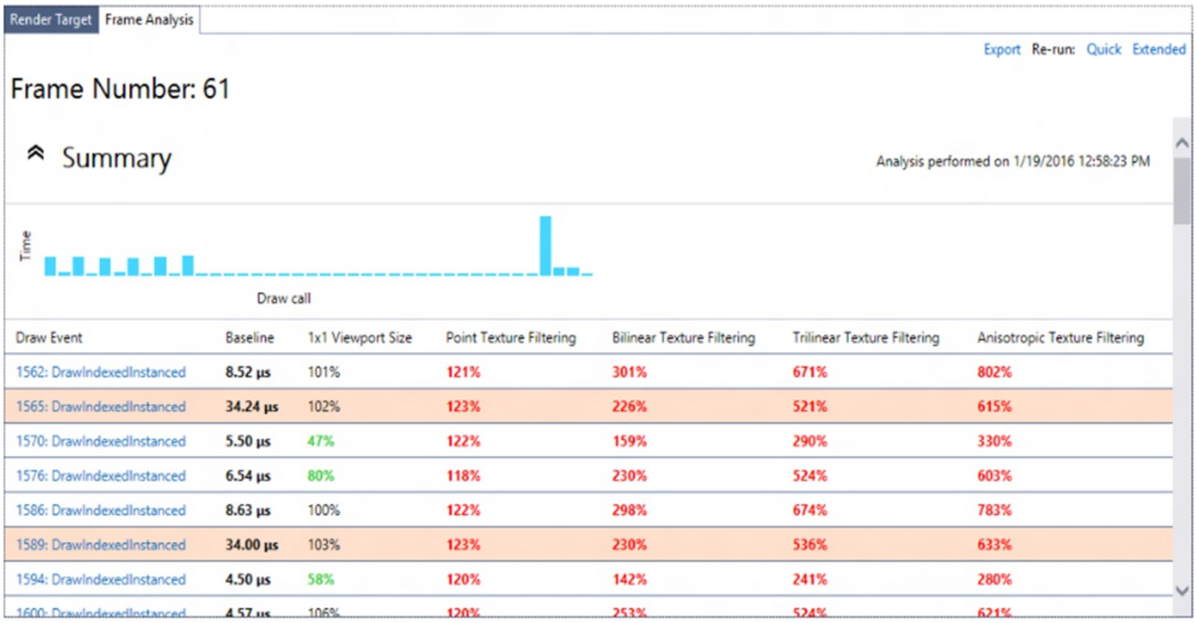


Frame Validation

Run Validation

Event Id	Severity	Category	Message	Event
1540	Warning	Unused	This state is not used. It is overwritten by a different state set in a later event.	obj:5->RSSetViewports(1,{TopL
1541	Warning	Unused	This state is not used. It is overwritten by a different state set in a later event.	obj:5->RSSetScissorRects(1,{lef
1542	Warning	Unused	This state is not used. It is overwritten by a different state set in a later event.	obj:5->OMSetRenderTargets(1
1563	Warning	Redundant	This call is unnecessary. 1560 IASetPrimitiveTopology already set this state.	obj:5->IASetPrimitiveTopology(
1568	Warning	Redundant	This call is unnecessary. 1560 IASetPrimitiveTopology already set this state.	obj:5->IASetPrimitiveTopology(
1574	Warning	Redundant	This call is unnecessary. 1560 IASetPrimitiveTopology already set this state.	obj:5->IASetPrimitiveTopology(
1580	Warning	Redundant	This call is unnecessary. 1556 SetPipelineState already set this state.	obj:5->IASetPipelineState(obj5
1587	Warning	Redundant	This call is unnecessary. 1560 IASetPrimitiveTopology already set this state.	obj:5->IASetPrimitiveTopology(
1592	Warning	Redundant	This call is unnecessary. 1560 IASetPrimitiveTopology already set this state.	obj:5->IASetPrimitiveTopology(
1598	Warning	Redundant	This call is unnecessary. 1560 IASetPrimitiveTopology already set this state.	obj:5->IASetPrimitiveTopology(
1604	Warning	Redundant	This call is unnecessary. 1556 SetPipelineState already set this state.	obj:5->IASetPipelineState(obj5
1608	Warning	Redundant	This call is unnecessary. 1560 IASetPrimitiveTopology already set this state.	obj:5->IASetPrimitiveTopology(
1611	Warning	Redundant	This call is unnecessary. 1560 IASetPrimitiveTopology already set this state.	obj:5->IASetPrimitiveTopology(

- **Frame Analysis for D3D12:** Use Frame Analysis to analyze draw call performance with directed "what-if" experiments. Switch to the Frame Analysis tab and run analysis to view the report. For more details, watch the GoingNative 25: Visual Studio Graphics Frame Analysis video.



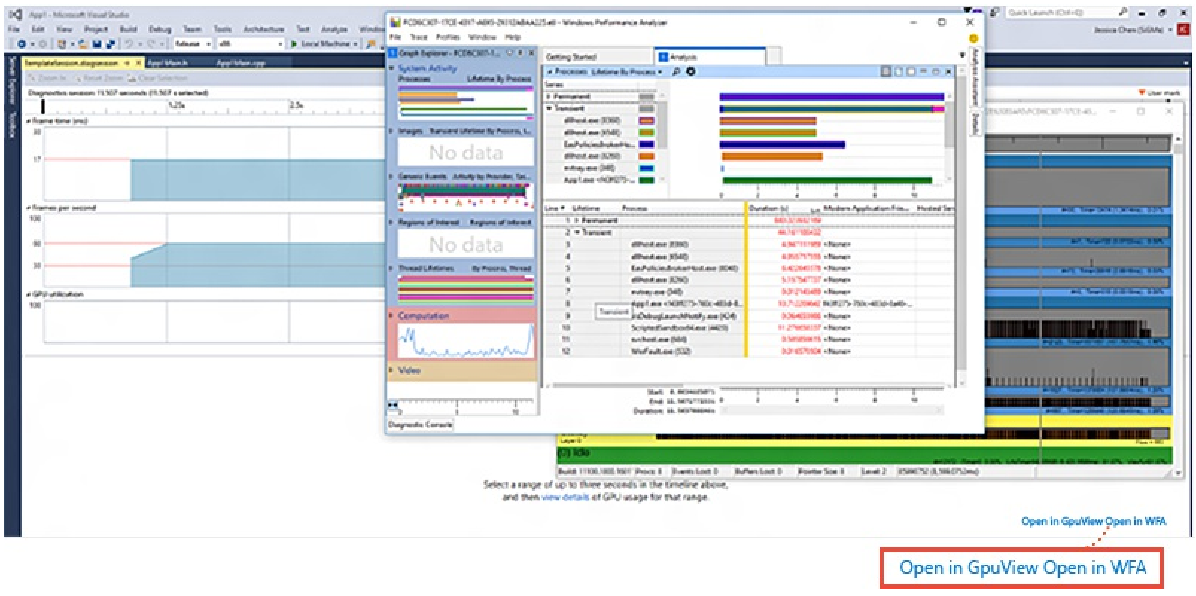
Frame Number: 61

Summary

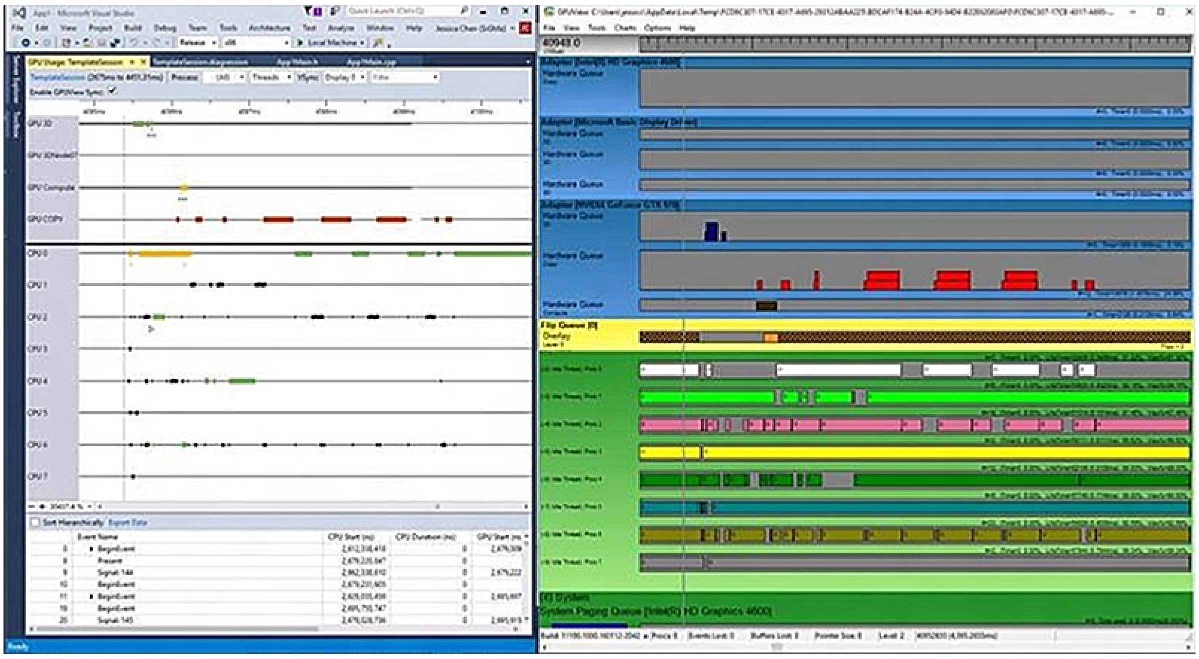
Analysis performed on 1/19/2016 12:58:23 PM

Draw Event	Baseline	1x1 Viewport Size	Point Texture Filtering	Bilinear Texture Filtering	Trilinear Texture Filtering	Anisotropic Texture Filtering
1562: DrawIndexedInstanced	8.52 µs	101%	121%	301%	671%	802%
1565: DrawIndexedInstanced	34.24 µs	102%	123%	226%	521%	615%
1570: DrawIndexedInstanced	5.50 µs	47%	122%	159%	290%	330%
1576: DrawIndexedInstanced	6.54 µs	80%	118%	230%	524%	603%
1586: DrawIndexedInstanced	8.63 µs	100%	122%	298%	674%	783%
1589: DrawIndexedInstanced	34.00 µs	103%	123%	230%	536%	633%
1594: DrawIndexedInstanced	4.50 µs	58%	120%	142%	241%	280%
1600: DrawIndexedInstanced	4.57 µs	106%	120%	253%	524%	621%

- **GPU Usage Improvements:** Open traces taken via the Visual Studio GPU Usage profiler with either GPU View or the Windows Performance Analyzer (WPA) tool for more detailed analysis. If you have the Windows Performance Toolkit installed there will be two hyperlinks, one for WPA and other for GPU View, at the bottom right of the session overview.



Traces opened in GPU View via this link support synchronized zooming and panning in the timeline between VS and GPU View. A checkbox in VS is used to control whether synchronization is enabled or not.



4/3/2019 • 68 minutes to read • Edit Online

C++ conformance improvements in Visual Studio 2019 RTW and version 16.1

Improvements in Visual Studio 2019 RTW

Visual Studio 2019 RTW contains the following conformance improvements, bug fixes and behavior changes in the Microsoft C++ compiler (MSVC).

Note: C++20 features will be made available in `/std:c++latest` mode until the C++20 implementation is complete for both the compiler and IntelliSense. At that time, the `/std:c++20` compiler mode will be introduced.

Improved modules support for templates and error detection

Modules are now officially in the C++20 standard. Improved support was added in Visual Studio 2017 version 15.9. For more information, see Better template support and error detection in C++ Modules with MSVC 2017 version 15.9.

Modified specification of aggregate type

The specification of an aggregate type has changed in C++20 (see Prohibit aggregates with user-declared constructors). In Visual Studio 2019, under `/std:c++latest`, a class with any user-declared constructor (e.g. including a constructor declared `= default` or `= delete`) is not an aggregate. Previously, only user-provided constructors would disqualify a class from being an aggregate. This change puts additional restrictions on how such types can be initialized.

The following code compiles without errors in Visual Studio 2017 but raises errors C2280 and C2440 in Visual Studio 2019 under `/std:c++latest`:

```
struct A
{
    A() = delete; // user-declared ctor
};

struct B
{
    B() = default; // user-declared ctor
    int i = 0;
};

A a{}; // ill-formed in C++20, previously well-formed
B b = { 1 }; // ill-formed in C++20, previously well-formed
```

Partial support for operator <=>

P0515R3 C++20 introduces the `<=>` three-way comparison operator, also known as the "spaceship operator". Visual Studio 2019 in `/std:c++latest` mode introduces partial support for the operator by raising errors for syntax that is now disallowed. For example, the following code compiles without errors in Visual Studio 2017 but raises multiple errors in Visual Studio 2019 under `/std:c++latest`:

```
struct S
{
 bool operator<=(const S&) const { return true; }
};

template <bool (S::*)(const S&) const>
struct U { };
int main(int argc, char** argv)
{
 U<&S::operator<=> u; // In Visual Studio 2019 raises C2039, 2065, 2146.
}
```

To avoid the errors, insert a space in the offending line before the final bracket: `U<&S::operator<= > u;`.

References to types with mismatched cv-qualifiers

MSVC previously allowed direct binding of a reference from a type with mismatched cv-qualifiers below the top level. This could allow modification of supposedly const data referred to by the reference, and the compiler now creates a temporary as required by the standard. In Visual Studio 2017, the following code compiles without warnings. In Visual Studio 2019, the compiler raises *warning C4172: <func:#1 "?PData@X@@QBEABQBXXZ"> returning address of local variable or temporary*.

```
struct X
{
    const void* const& PData() const
    {
        return _pv;
    }

    void* _pv;
};

int main()
{
    X x;
    auto p = x.PData(); // C4172
}
```

`reinterpret_cast` from an overloaded function

The argument to `reinterpret_cast` is not one of the contexts in which the address of an overloaded function is permitted. The following code compiles without errors in Visual Studio 2017, but in Visual Studio 2019 it raises *C2440: cannot convert from 'overloaded-function' to 'fp'*:

```
int f(int) { return 1; }
int f(float) { return .1f; }
using fp = int(*)(int);

int main()
{
    fp r = reinterpret_cast<fp>(&f);
}
```

To avoid the error, use an allowed cast for this scenario:

```
int f(int);
int f(float);
using fp = int(*)(int);

int main()
{
    fp r = static_cast<fp>(&f); // or just &f;
}
```

Lambda closures

In C++14, lambda closure types are not literal. The primary consequence of this rule is that a lambda may not be assigned to a `constexpr` variable. The following code compiles without errors in Visual Studio 2017 but in Visual Studio 2019 it raises *C2127: 'l': illegal initialization of 'constexpr' entity with a non-constant expression* :

```
int main()
{
    constexpr auto l = [] {}; // C2127 in VS2019
}
```

To avoid the error, either remove the `constexpr` qualifier, or else change the conformance mode to `/std:c++17`.

std::create_directory failure codes

Implemented P1164 from C++20 unconditionally. This changes `std::create_directory` to check whether the target was already a directory on failure. Previously, all ERROR_ALREADY_EXISTS type errors were turned into success-but-directory-not-created codes.

operator<<(std::ostream, nullptr_t)

Per LWG 2221, added `operator<<(std::ostream, nullptr_t)` for writing nullptrs to streams.

Additional parallel algorithms

New parallel versions of `is_sorted`, `is_sorted_until`, `is_partitioned`, `set_difference`, `set_intersection`, `is_heap`, and `is_heap_until`.

atomic initialization

P0883 "Fixing atomic initialization" changes `std::atomic` to value-initialize the contained T rather than default-initializing it. The fix is enabled when using Clang/LLVM with the Microsoft Standard Library. It is currently disabled for the Microsoft C++ compiler as a workaround for a bug in constexpr processing.

remove_cvref and remove_cvref_t

Implemented the `remove_cvref` and `remove_cvref_t` type traits from P0550. These remove reference-ness and cv-qualification from a type without decaying functions and arrays to pointers (unlike `std::decay` and `std::decay_t`).

Feature-test macros

P0941R2 - feature-test macros is complete, with support for `__has_cpp_attribute`. Feature-test macros are supported in all Standard modes.

Prohibit aggregates with user-declared constructors

C++20 P1008R1 - prohibiting aggregates with user-declared constructors is complete.

Improvements in Visual Studio 2019 version 16.1

char8_t

P0482r6. C++20 adds a new character type that is used to represent UTF-8 code units. u8 string literals in

C++20 have type `const char8_t[N]` instead of `const char[N]`, which was the case previously. Similar changes have been proposed for the C Standard in N2231. Suggestions for char8_t backward compatibility remediation are given in P1423r0. The Microsoft C++ compiler adds support for char8_t in Visual Studio 2019 version 16.1 when you specify the **/Zc:char8_t** compiler option. In the future, it will be supported with /std:c++latest, which can be reverted to C++17 behavior via **/Zc:char8_t-**. The EDG compiler which powers IntelliSense does not yet support it, so you will see spurious IntelliSense-only errors which do not impact the actual compilation.

Example

```
const char* s = u8"Hello"; // C++17
const char8_t* s = u8"Hello"; // C++20
```

std::type_identity metafunction and std::identity function object

P0887R1 type_identity. The deprecated `std::identity` class template extension has been removed, and replaced with the C++20 `std::type_identity` metafunction and `std::identity` function object. Both are available only under /std:c++latest.

The following example produces deprecation warning C4996 for `std::identity` (defined in <type_traits>) in Visual Studio 2017:

```
#include <type_traits>

using T = std::identity<int>::type;
T x, y = std::identity<T>{}(x);
int i = 42;
long j = std::identity<long>{}(i);
```

The following example shows how to use the new `std::identity` (defined in <functional>) together with the new `std::type_identity`:

```
#include <type_traits>
#include <functional>

using T = std::type_identity<int>::type;
T x, y = std::identity{}(x);
int i = 42;
long j = static_cast<long>(i);
```

Syntax checks for generic lambdas

The new lambda processor enables some conformance-mode syntactic checks in generic lambdas, under /std:c++latest or under any other language mode with **/experimental:newLambdaProcessor**.

In Visual Studio 2017, this code compiles without warnings, but in Visual Studio 2019 it produces error *C2760 syntax error: unexpected token '<id-expr>', expected 'id-expression'*:

```
void f() {
    auto a = [](auto arg) {
        decltype(arg)::Type t;
    };
}
```

The following example shows the correct syntax, now enforced by the compiler:

```
void f() {
    auto a = [](auto arg) {
        typename decltype(arg)::Type t;
    };
}
```

Argument-dependent lookup for function calls

P0846R0 (C++20) Increased ability to find function templates via argument-dependent lookup for function call expressions with explicit template arguments. Requires **/std:c++latest**.

Designated initialization

P0329R4 (C++20) Designated initialization allows specific members to be selected in aggregate initialization by using the `Type t { .member = expr }` syntax. Requires **/std:c++latest**.

New and updated Standard Library functions (C++20)

- `starts_with()` and `ends_with()` for `basic_string` and `basic_string_view`.
- `contains()` for associative containers.
- `remove()`, `remove_if()`, and `unique()` for `list` and `forward_list` now return `size_type`.
- `shift_left()` and `shift_right()` added to <algorithm>.

Bug fixes and behavior changes in Visual Studio 2019 RTW

Correct diagnostics for basic_string range constructor

In Visual Studio 2019, the `basic_string` range constructor no longer suppresses compiler diagnostics with `static_cast`. The following code compiles without warnings in Visual Studio 2017, despite the possible loss of data from `wchar_t` to `char` when initializing `out`:

```
std::wstring ws = /* … */;
std::string out(ws.begin(), ws.end());
```

Visual Studio 2019 correctly raises *C4244: 'argument': conversion from 'wchar_t' to 'const _Elem', possible loss of data*. To avoid the warning, you can initialize the std::string as shown in this example:

```
std::wstring ws = L"Hello world";
std::string out;
for (wchar_t ch : ws)
{
    out.push_back(static_cast<char>(ch));
}
```

Incorrect calls to += and -= under /clr or /ZW are now correctly detected

A bug was introduced in Visual Studio 2017 which caused the compiler to silently ignore errors and generate no code for the invalid calls to += and -= under `/clr` or `/ZW`. The following code compiles without errors in Visual Studio 2017 but in Visual Studio 2019 it correctly raises *error C2845: 'System::String ^': pointer arithmetic not allowed on this type*:

```
public enum class E { e };

void f(System::String ^s)
{
    s += E::e; // C2845 in VS2019
}
```

To avoid the error in this example, use the operator with the ToString() method: `s += E::e.ToString();`.

Initializers for inline static data members

Invalid member accesses within `inline` and `static constexpr` initializers are now correctly detected. The following example compiles without error in Visual Studio 2017, but in Visual Studio 2019 under `/std:c++17` mode it raises *error C2248: cannot access private member declared in class 'X'*.

```
struct X
{
    private:
        static inline const int c = 1000;
};

struct Y : X
{
    static inline int d = c; // C2248 in Visual Studio 2019
};
```

To avoid the error, declare the member `X::c` as protected:

```
struct X
{
    protected:
        static inline const int c = 1000;
};
```

C4800 reinstated

MSVC used to have a performance warning C4800 about implicit conversion to bool that was too noisy and insuppressible, leading us to remove it in Visual Studio 2017. However, over the lifecycle of Visual Studio 2017 we got a lot of feedback on the useful cases it was solving. We bring back in Visual Studio 2019 a carefully tailored C4800 along with its accompanying C4165, both of which can be easily suppressed with either an explicit cast or comparison to 0 of the appropriate type. C4800 is an off-by-default level 4 warning, and C4165 is an off-by-default level 3 warning. Both are discoverable by using the `/Wall` compiler option.

The following example raises C4800 and C4165 under `/Wall`:

```
bool test(IUnknown* p)
{
    bool valid = p; // warning C4800: Implicit conversion from 'IUnknown*' to bool. Possible information loss
    IDispatch* d = nullptr;
    HRESULT hr = p->QueryInterface(__uuidof(IDispatch), reinterpret_cast<void**>(&d));
    return hr; // warning C4165: 'HRESULT' is being converted to 'bool'; are you sure this is what you want?
}
```

To avoid the warnings in the previous example, you can write the code like this:

```
bool test(IUnknown* p)
{
    bool valid = p != nullptr; // OK
    IDispatch* d = nullptr;
    HRESULT hr = p->QueryInterface(__uuidof(IDispatch), reinterpret_cast<void**>(&d));
    return SUCCEEDED(hr);  // OK
}
```

Local class member function doesn't have a body

In Visual Studio 2017, *C4822: Local class member function doesn't have a body* is raised only when compiler option `/w14822` is explicitly set; it isn't shown with `/Wall`. In Visual Studio 2019, C4822 is an off-by-default warning, which makes it discoverable under `/Wall` without having to set `/w14822` explicitly.

```
void foo()
{
    struct A
        {
            int boo(); // warning C4822
        };
}
```

Function template bodies containing constexpr if statements

Template function bodies containing `if constexpr` statements have some `/permissive-` parsing-related checks enabled. For example, in Visual Studio 2017 the following code produces *C7510: 'Type': use of dependent type name must be prefixed with 'typename'* only if the `/permissive-` option is not set. In Visual Studio 2019 the same code raises errors whether or not the `/permissive-` option is set:

```
template <typename T>

int f()
{
    T::Type a; // error C7510

    if constexpr (T::val)
    {
        return 1;
    }
    else
    {
        return 2;
    }
}

struct X
{
    using Type = X;
    constexpr static int val = 1;
};

int main()
{
    return f<X>();
}
```

To avoid the error, add the `typename` keyword to the declaration of `a`: `typename T::Type a;`.

Inline assembly code is not supported in a lambda expression

The Visual C++ team was recently made aware of a security issue in which the use of inline-assembler within a lambda could lead to the corruption of 'ebp' (the return address register) at runtime. A malicious attacker might be able to take advantage of this scenario. Given the nature of the issue, the fact that inline assembler is only supported on x86, and the poor interaction of the inline assembler with the rest of the compiler it was felt that the safest solution to this problem was disallow inline assembler within a lambda expression.

Note: the only use of inline assembler within a lambda expression that we have encountered in the 'wild' was a use in which the aim was to capture the return address. In this scenario, you can capture the return address on all platforms simply by using a compiler intrinsic `_ReturnAddress()`.

The following code produces *C7552: inline assembler is not supported in a lambda* in both Visual Studio 2017

15.9 and in Visual Studio 2019:

```
#include <cstdio>

int f()
{
    int y = 1724;
    int x = 0xdeadbeef;

    auto lambda = [&]
    {
        __asm {

            mov eax, x
            mov y, eax
        }
    };

    lambda();
    return y;
}
```

To avoid the error, move the assembly code into a named function as shown in the following example:

```
#include <cstdio>

void g(int& x, int& y)
{
    __asm {
        mov eax, x
        mov y, eax
    }
}

int f()
{
    int y = 1724;
    int x = 0xdeadbeef;
    auto lambda = [&]
    {
        g(x, y);
    };
    lambda();
    return y;
}

int main()
{
    std::printf("%d\n", f());
}
```

Iterator debugging and std::move_iterator

The iterator debugging feature has been taught to properly unwrap `std::move_iterator`. For example, `std::copy(std::move_iterator<std::vector<int>::iterator>, std::move_iterator<std::vector<int>::iterator>, int*)` can now engage the `memcpy` fast path.

Fixes for <xkeycheck.h> keyword enforcement

The Standard Library's macro-ized keyword enforcement <xkeycheck.h> was fixed to emit the actual problem keyword detected rather than a generic message. It also supports C++20 keywords, and avoids tricking IntelliSense into saying random keywords are macros.

Allocator types un-deprecated

`std::allocator<void>`, `std::allocator::size_type`, and `std::allocator::difference_type` have been un-deprecated.

Correct warning for narrowing string conversions

A spurious `static_cast` not called for by the standard that accidentally suppressed C4244 narrowing warnings was removed from std::string. Attempting to call `std::string::string(const wchar_t*, const wchar_t*)` will now properly emit *C4244 "narrowing a wchar_t into a char."*

Various <filesystem> correctness fixes

- Fixed `std::filesystem::last_write_time` failing when attempting to change a directory's last write time.
- The `std::filesystem::directory_entry` constructor now stores a failed result, rather than throwing an exception, when supplied a nonexistent target path.
- The `std::filesystem::create_directory` 2-parameter version was changed to call the 1-parameter version, as the underlying `CreateDirectoryExW` function would perform `copy_symlink` when the existing_p was a symlink.
- `std::filesystem::directory_iterator` no longer fails when encountering a broken symlink.
- `std::filesystem::space` now accepts relative paths.
- `std::filesystem::path::lexically_relative` is no longer confused by trailing slashes, reported as LWG 3096.
- Worked around `CreateSymbolicLinkW` rejecting paths with forward slashes in `std::filesystem::create_symlink`.
- Worked around the POSIX deletion mode `delete` function existing on Windows 10 LTSB 1609 but not actually being capable of deleting files.
- `std::boyer_moore_searcher` and `std::boyer_moore_horspool_searcher`'s copy constructors and copy assignment operators now actually copy things.

Parallel algorithms on Windows 8 and later

The parallel algorithms library now properly uses the real `WaitOnAddress` family on Windows 8 and later, rather than always using the Windows 7 and earlier fake versions.

std::system_category::message() whitespace

`std::system_category::message()` now trims trailing whitespace from the returned message.

std::linear_congruential_engine divide by zero

Some conditions that would cause `std::linear_congruential_engine` to trigger divide by 0 have been fixed.

Fixes for iterator unwrapping

The iterator unwrapping machinery that was first exposed for programmer-user integration in Visual Studio 2017 15.8 (as described in https://devblogs.microsoft.com/cppblog/stl-features-and-fixes-in-vs-2017-15-8/) no longer unwraps iterators derived from standard library iterators. For example, a user that derives from `std::vector<int>::iterator` and tries to customize behavior now gets their customized behavior when calling standard library algorithms, rather than the behavior of a pointer. The unordered container reserve function now actually reserves for N elements, as described in LWG 2156.

Time handling

- Previously, some time values that were passed to the concurrency library would overflow, for example, `condition_variable::wait_for(seconds::max())`. These overflows, now fixed, changed behavior on a seemingly random 29-day cycle (when uint32_t milliseconds accepted by underlying Win32 APIs overflowed).

- The header now correctly declares timespec and timespec_get in namespace std in addition to declaring them in the global namespace.

Various fixes for containers

- Many Standard Library internal container functions have been made private for an improved IntelliSense experience. Additional fixes to mark members as private are expected in subsequent releases of MSVC.

- Exception safety correctness problems wherein the node-based containers like `list`, `map`, and `unordered_map` would become corrupted were fixed. During a `propagate_on_container_copy_assignment` or `propagate_on_container_move_assignment` reassignment operation, we would free the container's sentinel node with the old allocator, do the POCCA/POCMA assignment over the old allocator, and then try to acquire the sentinel node from the new allocator. If this allocation failed, the container is corrupted and can't even be destroyed, as owning a sentinel node is a hard data structure invariant. This was fixed to allocate the new sentinel node from the source container's allocator before destroying the existing sentinel node.

- The containers were fixed to always copy/move/swap allocators according to `propagate_on_container_copy_assignment`, `propagate_on_container_move_assignment`, and `propagate_on_container_swap`, even for allocators declared `is_always_equal`.

- Added the overloads for container merge and extract member functions that accept rvalue containers per P0083 "Splicing Maps And Sets"

std::basic_istream::read processing of \r\n => \n

`std::basic_istream::read` was fixed to not write into parts of the supplied buffer temporarily as part of \r\n => \n processing. This gives up some of the performance advantage that was gained in Visual Studio 2017 15.8 for reads larger than 4K in size, but efficiency improvements from avoiding 3 virtual calls per character are still present.

std::bitset constructor

`std::bitset`'s constructor no longer reads the ones and zeroes in reverse order for large bitsets.

std::pair::operator= regression

Fixed a regression in `std::pair`'s assignment operator introduced when implementing LWG 2729 "Missing SFINAE on std::pair::operator=";. It now correctly accepts types convertible to `std::pair` again.

Non-deduced contexts for add_const_t

Fixed a minor type traits bug, where `add_const_t` and related functions are supposed to be a non-deduced context. In other words, `add_const_t` should be an alias for `typename add_const<T>::type`, not `const T`.

See also

What's new in Visual Studio 2019

C++ conformance improvements in Visual Studio 2017 versions 15.0, 15.3, 15.5, 15.6, 15.7, 15.8, 15.9

With support for generalized constexpr and NSDMI for aggregates, the Microsoft C++ compiler is now complete for features added in the C++14 Standard. Note that the compiler still lacks a few features from the C++11 and C++98 Standards. See Visual C++ Language Conformance for a table that shows the current state of the compiler.

C++11

Expression SFINAE support in more libraries

The compiler continues to improve its support for expression SFINAE, which is required for template argument deduction and substitution where decltype and constexpr expressions may appear as template parameters. For more information, see Expression SFINAE improvements in Visual Studio 2017 RC.

C++ 14

NSDMI for Aggregates

An aggregate is an array or a class with no user-provided constructor, no private or protected non-static data members, no base classes, and no virtual functions. Beginning in C++14 aggregates may contain member initializers. For more information, see Member initializers and aggregates.

Extended constexpr

Expressions declared as constexpr are now allowed to contain certain kinds of declarations, if and switch statements, loop statements, and mutation of objects whose lifetime began within the constexpr expression evaluation. Also, there is no longer a requirement that a constexpr non-static member function be implicitly const. For more information, see Relaxing constraints on constexpr functions.

C++17

Terse static_assert

the message parameter for static_assert is optional. For more information, see Extending static_assert, v2.

[[fallthrough]] attribute

In **/std:c++17** mode, the [[fallthrough]] attribute can be used in the context of switch statements as a hint to the compiler that the fall-through behavior is intended. This prevents the compiler from issuing warnings in such cases. For more information, see Wording for [[fallthrough]] attribute.

Generalized range-based for loops

Range-based for loops no longer require that begin() and end() return objects of the same type. This enables end() to return a sentinel as used by ranges in range-v3 and the completed-but-not-quite-published Ranges Technical Specification. For more information, see Generalizing the Range-Based For Loop.

Improvements in Visual Studio 2017 version 15.3

constexpr lambdas

Lambda expressions may now be used in constant expressions. For more information, see constexpr lambda expressions in C++.

if constexpr in function templates

A function template may contain `if constexpr` statements to enable compile-time branching. For more information, see if constexpr statements.

Selection statements with initializers

An `if` statement may include an initializer that introduces a variable at block scope within the statement itself. For more information, see if statements with initializer.

[[maybe_unused]] and [[nodiscard]] attributes

New attributes to silence warnings when an entity is not used, or to create a warning if the return value of a function call is discarded. For more information, see Attributes in C++.

Using attribute namespaces without repetition

New syntax to enable only a single namespace identifier in an attribute list. For more information, see Attributes in C++.

Structured bindings

It is now possible in a single declaration to store a value with individual names for its components, when the value is an array, a `std::tuple` or `std::pair`, or has all public non-static data members. For more information, see Structured Bindings and Returning multiple values from a function.

Construction rules for enum class values

There is now an implicit/non-narrowing conversion from a scoped enumeration's underlying type to the enumeration itself, when its definition introduces no enumerator and the source uses a list-initialization syntax. For more information, see Construction Rules for enum class Values and Enumerations.

Capturing *this by value

The `*this` object in a lambda expression may now be captured by value. This enables scenarios in which the lambda is invoked in parallel and asynchronous operations, especially on newer machine architectures. For more information, see Lambda Capture of *this by Value as [=,*this].

Removing operator++ for bool

`operator++` is no longer supported on `bool` types. For more information, see Remove Deprecated operator++ (bool).

Removing deprecated "register" keyword

The `register` keyword, previously deprecated (and ignored by the compiler), is now removed from the language. For more information, see Remove Deprecated Use of the register Keyword.

For the complete list of conformance improvements up through Visual Studio 2015 Update 3, see Visual C++ What's New 2003 through 2015.

Improvements in Visual Studio 2017 version 15.5

Features marked with [14] are available unconditionally even in **/std:c++14** mode.

New compiler switch for extern constexpr

In earlier versions of Visual Studio, the compiler always gave a `constexpr` variable internal linkage even when the variable was marked `extern`. In Visual Studio 2017 version 15.5, a new compiler switch, /Zc:externConstexpr, enables correct standards-conforming behavior. For more information, see extern constexpr linkage.

Removing Dynamic Exception Specifications

P0003R5 Dynamic exception specifications were deprecated in C++11. the feature is removed from C++17, but the (still) deprecated `throw()` specification is retained strictly as an alias for `noexcept(true)`. For more information, see Dynamic exception specification removal and noexcept.

not_fn()

P0005R4 `not_fn` is a replacement of `not1` and `not2`.

Rewording enable_shared_from_this

P0033R1 `enable_shared_from_this` was added in C++11. The C++17 Standard updates the specification to better handle certain corner cases. [14]

Splicing Maps And Sets

P0083R3 This feature enables extraction of nodes from associative containers (e.g., map, set, unordered_map, unordered_set) which can then be modified and inserted back into the same container or a different container that uses the same node type. (A common use case is to extract a node from a `std::map`, change the key, and reinsert.)

Deprecating Vestigial Library Parts

P0174R2 Several features of the C++ Standard library have been superseded by newer features over the years, or else have been found to be not very useful or to be problematic. These features are officially deprecated in

C++17.

Removing Allocator Support In std::function

P0302R1 Prior to C++17 the class template `std::function` had several constructors that took an allocator argument. However, the use of allocators in this context was problematic, and the semantics were unclear. Therefore these contructors were removed.

Fixes for not_fn()

P0358R1 New wording for `std::not_fn` provides support of propagation of value category in case of wrapper invocation.

shared_ptr<T[]>, shared_ptr<T[N]>

P0414R2 Merging `shared_ptr` changes from Library Fundamentals to C++17. [14]

Fixing shared_ptr for Arrays

P0497R0 Fixes to shared_ptr support for arrays. [14]

Clarifying insert_return_type

P0508R0 The associative containers with unique keys, and the unordered containers with unique keys have a member function `insert` that returns a nested type `insert_return_type`. That return type is now defined as a specialization of a type that is parameterized on the Iterator and NodeType of the container.

Inline Variables For The STL

P0607R0

Annex D features deprecated

Annex D of the C++ standard contains all the features that have been deprecated, including `shared_ptr::unique()`, `<codecvt>`, and `namespace std::tr1`. When the **/std:c++17** compiler switch is set, almost all the Standard Library features in Annex D are marked as deprecated. For more information, see Standard Library features in Annex D are marked as deprecated.

The `std::tr2::sys` namespace in `<experimental/filesystem>` now emits a deprecation warning under **/std:c++14** by default, and is now removed under **/std:c++17** by default.

Improved conformance in iostreams by avoiding a non-Standard extension (in-class explicit specializations).

The Standard Library now uses variable templates internally.

The Standard Library has been updated in response to C++17 compiler changes, including the addition of noexcept in the type system and the removal of dynamic-exception-specifications.

Improvements in Visual Studio 2017 version 15.6

C++17 Library Fundamentals V1

P0220R1 incorporates Library Fundamentals Technical Specification for C++17 into the standard. Covers updates to <experimental/tuple>, <experimental/optional>, <experimental/functional>, <experimental/any>, <experimental/string_view> , <experimental/memory>, <experimental/memory_resource>,and <experimental/algorithm>.

C++17 Improving Class Template Argument Deduction For The STL

P0739R0 Move `adopt_lock_t` to front of parameter list for `scoped_lock` to enable consistent use of `scoped_lock`. Allow `std::variant` constructor to participate in overload resolution in more cases, in order to enable copy assignment.

Improvements in Visual Studio 2017 version 15.7

C++17 Rewording inheriting constructors

P0136R1 specifies that a **using** declaration that names a constructor now makes the corresponding base class constructors visible to initializations of the derived class rather than declaring additional derived class constructors. This is a change from C++14. In Visual Studio 2017 version 15.7 and later, in **/std:c++17** mode, code that is valid in C++14 and uses inheriting constructors may not be valid or may have different semantics.

The following example shows C++14 behavior:

```
struct A {
    template<typename T>
    A(T, typename T::type = 0);
    A(int);
};

struct B : A {
    using A::A;
    B(int n) = delete; // Error C2280
};

B b(42L); // Calls B<long>(long), which calls A(int)
          //  due to substitution failure in A<long>(long).
```

The following example shows **/std:c++17** behavior in Visual Studio 15.7:

```
struct A {
    template<typename T>
    A(T, typename T::type = 0);
    A(int);
};

struct B : A {
    using A::A;
    B(int n)
    {
        //do something
    }
};

B b(42L); // now calls B(int)
```

For more information, see Constructors.

C++17 Extended aggregate initialization

P0017R1

If the constructor of a base class is non-public, but accessible to a derived class, then under **/std:c++17** mode in Visual Studio version 15.7 you can no longer use empty braces to initialize an object of the derived type.

The following example shows C++14 conformant behavior:

```
struct Derived;

struct Base {
    friend struct Derived;
private:
    Base() {}
};

struct Derived : Base {};

Derived d1; // OK. No aggregate init involved.
Derived d2 {}; // OK in C++14: Calls Derived::Derived()
               // which can call Base ctor.
```

In C++17, `Derived` is now considered an aggregate type; therefore, the initialization of `Base` via the private default constructor happens directly as part of the extended aggregate initialization rule. Previously, the `Base` private constructor was called via the `Derived` constructor and it succeeded because of the friend declaration.

The following example shows C++17 behavior in Visual Studio version 15.7 in **/std:c++17** mode:

```
struct Derived;

struct Base {
    friend struct Derived;
private:
    Base() {}
};

struct Derived : Base {
    Derived() {} // add user-defined constructor
                 // to call with {} initialization
};

Derived d1; // OK. No aggregate init involved.

Derived d2 {}; // error C2248: 'Base::Base': cannot access
               // private member declared in class 'Base'
```

C++17 Declaring non-type template parameters with auto

P0127R2

In **/std:c++17** mode, the compiler can now deduce the type of a non-type template argument that is declared with **auto**:

```
template <auto x> constexpr auto constant = x;

auto v1 = constant<5>;      // v1 == 5, decltype(v1) is int
auto v2 = constant<true>;   // v2 == true, decltype(v2) is bool
auto v3 = constant<'a'>;    // v3 == 'a', decltype(v3) is char
```

One impact of this new feature is that valid C++14 code may not be valid or may have different semantics. For example, some overloads which were previously invalid are now valid. The following example shows C++14 code that compiles because the call to `foo(p)` is bound to `foo(void*);`. In Visual Studio 2017 version 15.7, in **/std:c++17** mode, the `foo` function template is the best match.

```
template <int N> struct A;
template <typename T, T N> int foo(A<N>*) = delete;

void foo(void *);

void bar(A<0> *p)
{
    foo(p); // OK in C++14
}
```

The following example shows C++17 code in Visual Studio 15.7 in **/std:c++17** mode:

```
template <int N> struct A;
template <typename T, T N> int foo(A<N>*);

void foo(void *);

void bar(A<0> *p)
{
    foo(p); // C2280: 'int foo<int,0>(A<0>*)': attempting to reference a deleted function
}
```

C++17 Elementary string conversions (partial)

P0067R5 Low-level, locale-independent functions for conversions between integers and strings and between floating-point numbers and strings. (As of Visual Studio 15.7 Preview 2, supported for integers only.)

C++20 Avoiding unnecessary decay (partial)

P0777R1 Adds differentiation between the concept of "decay" and that of simply removing const or reference qualifiers. New type trait `remove_reference_t` replaces `decay_t` in some contexts. Support for `remove_cvref_t` is not yet implemented as of Visual Studio 2017 version 15.7 Preview 2.

C++17 Parallel algorithms

P0024R2 The Parallelism TS is incorporated into the standard, with minor modifications.

C++17 hypot(x, y, z)

P0030R1 Adds three new overloads to `std::hypot`, for types **float**, **double**, and **long double**, each of which has three input parameters.

C++17 <filesystem>

P0218R1 Adopts the File System TS into the standard with a few wording modifications.

C++17 Mathematical special functions

P0226R1 Adopts previous technical specifications for Mathematical Special Functions into the standard <cmath> header.

C++17 Deduction guides for the STL

P0433R2 Updates to STL to take advantage of C++17 adoption of P0091R3, which adds support for class template argument deduction.

C++17 Repairing elementary string conversions

P0682R1 Move the new elementary string conversion functions from P0067R5 into a new header <charconv> and make other improvements, including changing error handling to use `std::errc` instead of `std::error_code`.

C++17 constexpr for char_traits (partial)

P0426R1 Changes to `std::traits_type` member functions `length`, `compare`, and `find` in order to make `std::string_view` usable in constant expressions. (In Visual Studio 2017 version 15.6, supported for

Clang/LLVM only. In version 15.7 Preview 2, support is nearly complete for CIXX as well.)

Improvements in Visual Studio 2017 version 15.9

Left-to-right evaluation order for operators ->*, [], >>, and <<

Starting in C++17, the operands of the operators ->*, [], >>, and << must be evaluated in left-to-right order. There are two cases in which the compiler is unable to guarantee this order:

- when one of the operand expressions is an object passed by value or contains an object passed by value, or
- when compiled by using **/clr**, and one of the operands is a field of an object or an array element.

The compiler emits warning C4866 when it can't guarantee left-to-right evaluation. This warning is only generated if **/std:c++17** or later is specified, as the left-to-right order requirement of these operators was introduced in C++17.

To resolve this warning, first consider whether left-to-right evaluation of the operands is necessary, such as when evaluation of the operands might produce order-dependent side-effects. In many cases, the order in which operands are evaluated does not have an observable effect. If the order of evaluation must be left-to-right, consider whether you can pass the operands by const reference instead. This change eliminates the warning in the following code sample.

```
// C4866.cpp
// compile with: /w14866 /std:c++17

class HasCopyConstructor
{
public:
    int x;

    HasCopyConstructor(int x) : x(x) {}
    HasCopyConstructor(const HasCopyConstructor& h) : x(h.x) { }
};

int operator>>(HasCopyConstructor a, HasCopyConstructor b) { return a.x >> b.x; }

// This version of operator>> does not trigger the warning:
// int operator>>(const HasCopyConstructor& a, const HasCopyConstructor& b) { return a.x >> b.x; }

int main()
{
    HasCopyConstructor a{ 1 };
    HasCopyConstructor b{ 2 };

    a>>b;        // C4866 for call to operator>>
};
```

Bug fixes in Visual Studio versions 15.0, 15.3, 15.5, 15.7, 15.8, and 15.9

Copy-list-initialization

Visual Studio 2017 correctly raises compiler errors related to object creation using initializer lists that were not caught in Visual Studio 2015 and could lead to crashes or undefined runtime behavior. As per N4594 13.3.1.7p1, in copy-list-initialization, the compiler is required to consider an explicit constructor for overload resolution, but must raise an error if that overload is actually chosen.

The following two examples compile in Visual Studio 2015 but not in Visual Studio 2017.

```
struct A
{
    explicit A(int) {}
    A(double) {}
};

int main()
{
    A a1 = { 1 }; // error C3445: copy-list-initialization of 'A' cannot use an explicit constructor
    const A& a2 = { 1 }; // error C2440: 'initializing': cannot convert from 'int' to 'const A &'

}
```

To correct the error, use direct initialization:

```
A a1{ 1 };
const A& a2{ 1 };
```

In Visual Studio 2015, the compiler erroneously treated copy-list-initialization in the same way as regular copy-initialization; it considered only converting constructors for overload resolution. In the following example, Visual Studio 2015 chooses MyInt(23) but Visual Studio 2017 correctly raises the error.

```
// From http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_closed.html#1228
struct MyStore {
    explicit MyStore(int initialCapacity);
};

struct MyInt {
    MyInt(int i);
};

struct Printer {
    void operator()(MyStore const& s);
    void operator()(MyInt const& i);
};

void f() {
    Printer p;
    p({ 23 }); // C3066: there are multiple ways that an object of this type can be called with these
arguments
}
```

This example is similar to the previous one but raises a different error. It succeeds in Visual Studio 2015 and fails in Visual Studio 2017 with C2668.

```
struct A {
    explicit A(int) {}
};

struct B {
    B(int) {}
};

void f(const A&) {}
void f(const B&) {}

int main()
{
    f({ 1 }); // error C2668: 'f': ambiguous call to overloaded function
}
```

Deprecated typedefs

Visual Studio 2017 now issues the correct warning for deprecated typedefs that are declared in a class or struct. The following example compiles without warnings in Visual Studio 2015 but produces C4996 in Visual Studio 2017.

```
struct A
{
    // also for __declspec(deprecated)
    [[deprecated]] typedef int inttype;
};

int main()
{
    A::inttype a = 0; // C4996 'A::inttype': was declared deprecated
}
```

constexpr

Visual Studio 2017 correctly raises an error when the left-hand operand of a conditionally evaluating operation is not valid in a constexpr context. The following code compiles in Visual Studio 2015 but not in Visual Studio 2017 (C3615 constexpr function 'f' cannot result in a constant expression):

```
template<int N>
struct array
{
    int size() const { return N; }
};

constexpr bool f(const array<1> &arr)
{
    return arr.size() == 10 || arr.size() == 11; // C3615
}
```

To correct the error, either declare the `array::size()` function as `constexpr` or remove the `constexpr` qualifier from `f`.

Class types passed to variadic functions

In Visual Studio 2017, classes or structs that are passed to a variadic function such as printf must be trivially copyable. When passing such objects, the compiler simply makes a bitwise copy and does not call the constructor or destructor.

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

int main()
{
    std::atomic<int> i(0);
    printf("%i\n", i); // error C4839: non-standard use of class 'std::atomic<int>'
                        // as an argument to a variadic function.
                        // note: the constructor and destructor will not be called;
                        // a bitwise copy of the class will be passed as the argument
                        // error C2280: 'std::atomic<int>::atomic(const std::atomic<int> &)':
                        // attempting to reference a deleted function

    struct S {
        S(int i) : i(i) {}
        S(const S& other) : i(other.i) {}
        operator int() { return i; }
    private:
        int i;
    } s(0);
    printf("%i\n", s); // warning C4840 : non-portable use of class 'main::S'
                      // as an argument to a variadic function
}
```

To correct the error, you can call a member function that returns a trivially copyable type,

```
    std::atomic<int> i(0);
    printf("%i\n", i.load());
```

or else perform a static cast to convert the object before passing it:

```
    struct S {/* as before */} s(0);
    printf("%i\n", static_cast<int>(s))
```

For strings built and managed using CString, the provided `operator LPCTSTR()` should be used to cast a CString object to the C pointer expected by the format string.

```
CString str1;
CString str2 = _T("hello!");
str1.Format(_T("%s"), static_cast<LPCTSTR>(str2));
```

cv-qualifiers in class construction

In Visual Studio 2015, the compiler sometimes incorrectly ignores the cv-qualifier when generating a class object via a constructor call. This can potentially cause a crash or unexpected runtime behavior. The following example compiles in Visual Studio 2015 but raises a compiler error in Visual Studio 2017:

```
struct S
{
    S(int);
    operator int();
};

int i = (const S)0; // error C2440
```

To correct the error, declare `operator int()` as `const`.

Access checking on qualified names in templates

Previous versions of the compiler did not perform access checking on qualified names in some template contexts. This can interfere with expected SFINAE behavior where the substitution is expected to fail due to the inaccessibility of a name. This could have potentially caused a crash or unexpected behavior at runtime due to the compiler incorrectly calling the wrong overload of the operator. In Visual Studio 2017, a compiler error is raised. The specific error might vary but typically it is "C2672 no matching overloaded function found". The following code compiles in Visual Studio 2015 but raises an error in Visual Studio 2017:

```
#include <type_traits>

template <class T> class S {
    typedef typename T type;
};

template <class T, std::enable_if<std::is_integral<typename S<T>::type>::value, T> * = 0>
bool f(T x);

int main()
{
    f(10); // C2672: No matching overloaded function found.
}
```

Missing template argument lists

In Visual Studio 2015 and earlier, the compiler did not diagnose missing template argument lists when the template appeared in a template parameter list (for example as part of a default template argument or a non-type template parameter). This can result in unpredictable behavior, including compiler crashes or unexpected runtime behavior. The following code compiles in Visual Studio 2015 but produces an error in Visual Studio 2017.

```
template <class T> class ListNode;
template <class T> using ListNodeMember = ListNode<T> T::*;
template <class T, ListNodeMember M> class ListHead; // C2955: 'ListNodeMember': use of alias
                                                     // template requires template argument list

// correct:  template <class T, ListNodeMember<T> M> class ListHead;
```

Expression-SFINAE

To support expression-SFINAE, the compiler now parses decltype arguments when the templates are declared rather than instantiated. Consequently, if a non-dependent specialization is found in the decltype argument, it is not deferred to instantiation-time and is processed immediately and any resulting errors are diagnosed at that time.

The following example shows such a compiler error that is raised at the point of declaration:

```
#include <utility>
template <class T, class ReturnT, class... ArgsT>
class IsCallable
{
public:
    struct BadType {};

    template <class U>
    static decltype(std::declval<T>()(std::declval<ArgsT>()...)) Test(int); //C2064. Should be declval<U>

    template <class U>
    static BadType Test(...);

    static constexpr bool value = std::is_convertible<decltype(Test<T>(0)), ReturnT>::value;
};

constexpr bool test1 = IsCallable<int(), int>::value;
static_assert(test1, "PASS1");
constexpr bool test2 = !IsCallable<int*, int>::value;
static_assert(test2, "PASS2");
```

Classes declared in anonymous namespaces

According to the C++ standard, a class declared inside an anonymous namespace has internal linkage, and therefore cannot be exported. In Visual Studio 2015 and earlier, this rule was not enforced. In Visual Studio 2017 the rule is partially enforced. The following example raises this error in Visual Studio 2017: "error C2201: const anonymous namespace::S1::vftable: must have external linkage in order to be exported/imported."

```
struct __declspec(dllexport) S1 { virtual void f() {} }; //C2201
```

Default initializers for value class members (C++/CLI)

In Visual Studio 2015 and earlier, the compiler permitted (but ignored) a default member initializer for a member of a value class. Default initialization of a value class always zero-initializes the members; a default constructor is not permitted. In Visual Studio 2017, default member initializers raise a compiler error, as shown in this example:

```
value struct V
{
    int i = 0; // error C3446: 'V::i': a default member initializer
               // is not allowed for a member of a value class
};
```

Default Indexers (C++/CLI)

In Visual Studio 2015 and earlier, the compiler in some cases misidentified a default property as a default indexer. It was possible to work around the issue by using the identifier `default` to access the property. The workaround itself became problematic after `default` was introduced as a keyword in C++11. Therefore, in Visual Studio 2017 the bugs that required the workaround were fixed, and the compiler now raises an error when `default` is used to access the default property for a class.

```
//class1.cs

using System.Reflection;
using System.Runtime.InteropServices;

namespace ClassLibrary1
{
    [DefaultMember("Value")]
    public class Class1
    {
        public int Value
        {
            // using attribute on the return type triggers the compiler bug
            [return: MarshalAs(UnmanagedType.I4)]
            get;
        }
    }
    [DefaultMember("Value")]
    public class Class2
    {
        public int Value
        {
            get;
        }
    }
}

// code.cpp
#using "class1.dll"

void f(ClassLibrary1::Class1 ^r1, ClassLibrary1::Class2 ^r2)
{
       r1->Value; // error
       r1->default;
       r2->Value;
       r2->default; // error
}
```

In Visual Studio 2017, you can access both Value properties by their name:

```
#using "class1.dll"

void f(ClassLibrary1::Class1 ^r1, ClassLibrary1::Class2 ^r2)
{
       r1->Value;
       r2->Value;
}
```

Bug fixes in Visual Studio 2017 version 15.3

Calls to deleted member templates

In previous versions of Visual Studio, the compiler in some cases would fail to emit an error for ill-formed calls to a deleted member template which would've potentially caused crashes at runtime. The following code now produces C2280, "'int S<int>::f<int>(void)': attempting to reference a deleted function":

```
template<typename T>
struct S {
   template<typename U> static int f() = delete;
};

void g()
{
   decltype(S<int>::f<int>()) i; // this should fail
}
```

To fix the error, declare i as `int`.

Pre-condition checks for type traits

Visual Studio 2017 version 15.3 improves pre-condition checks for type-traits to more strictly follow the standard. One such check is for assignable. The following code produces C2139 in Visual Studio 2017 version 15.3:

```
struct S;
enum E;

static_assert(!__is_assignable(S, S), "fail"); // C2139 in 15.3
static_assert(__is_convertible_to(E, E), "fail"); // C2139 in 15.3
```

New compiler warning and runtime checks on native-to-managed marshaling

Calling from managed functions to native functions requires marshalling. The CLR performs the marshaling but it doesn't understand C++ semantics. If you pass a native object by value, CLR either calls the object's copy-constructor or uses BitBlt, which may cause undefined behavior at runtime.

Now the compiler emits a warning if it can know at compile time that a native object with deleted copy ctor is passed between native and managed boundary by value. For those cases in which the compiler doesn't know at compile time, it injects a runtime check so that the program calls `std::terminate` immediately when an ill-formed marshalling occurs. In Visual Studio 2017 version 15.3, the following code produces warning C4606 "'A': passing argument by value across native and managed boundary requires valid copy constructor. Otherwise the runtime behavior is undefined".

```
class A
{
public:
   A() : p_(new int) {}
   ~A() { delete p_; }

   A(A const &) = delete;
   A(A &&rhs) {
   p_ = rhs.p_;
}

private:
   int *p_;
};

#pragma unmanaged

void f(A a)
{
}

#pragma managed

int main()
{
    f(A()); // This call from managed to native requires marshalling. The CLR doesn't understand C++ and uses
BitBlt, which results in a double-free later.
}
```

To fix the error, remove the `#pragma managed` directive to mark the caller as native and avoid marshalling.

Experimental API warning for WinRT

WinRT APIs that are released for experimentation and feedback are decorated with `Windows.Foundation.Metadata.ExperimentalAttribute`. In Visual Studio 2017 version 15.3, the compiler produces warning C4698 when it encounters the attribute. A few APIs in previous versions of the Windows SDK have already been decorated with the attribute, and calls to these APIs now trigger this compiler warning. Newer Windows SDKs have the attribute removed from all shipped types, but if you are using an older SDK, you'll need to suppress these warnings for all calls to shipped types.

The following code produces warning C4698: "'Windows::Storage::IApplicationDataStatics2::GetForUserAsync' is for evaluation purposes only and is subject to change or removal in future updates":

```
Windows::Storage::IApplicationDataStatics2::GetForUserAsync(); //C4698
```

To disable the warning, add a #pragma:

```
#pragma warning(push)
#pragma warning(disable:4698)

Windows::Storage::IApplicationDataStatics2::GetForUserAsync();

#pragma warning(pop)
```

Out-of-line definition of a template member function

Visual Studio 2017 version 15.3 produces an error when it encounters an out-of-line definition of a template member function that was not declared in the class. The following code now produces error C2039: 'f': is not a member of 'S':

```
struct S {};

template <typename T>
void S::f(T t) {} //C2039: 'f': is not a member of 'S'
```

To fix the error, add a declaration to the class:

```
struct S {
    template <typename T>
    void f(T t);
};
template <typename T>
void S::f(T t) {}
```

Attempting to take the address of "this" pointer

In C++ `this` is an prvalue of type pointer to X. You cannot take the address of `this` or bind it to an lvalue reference. In previous versions of Visual Studio, the compiler would allow you to circumvent this restriction by performing a cast. In Visual Studio 2017 version 15.3, the compiler produces error C2664.

Conversion to an inaccessible base class

Visual Studio 2017 version 15.3 produces an error when you attempt to convert a type to a base class which is inaccessible. The compiler now raises "error C2243: 'type cast': conversion from 'D *' to 'B *' exists, but is inaccessible". The following code is ill-formed and can potentially cause a crash at runtime. The compiler now produces C2243 when it encounters code like this:

```
#include <memory>

class B { };
class D : B { }; // C2243. should be public B { };

void f()
{
   std::unique_ptr<B>(new D());
}
```

Default arguments are not allowed on out of line definitions of member functions

Default arguments are not allowed on out-of-line definitions of member functions in template classes The compiler will issue a warning under **/permissive**, and a hard error under **/permissive-**.

In previous versions of Visual Studio, the following ill-formed code could potentially cause a runtime crash. Visual Studio 2017 version 15.3 produces warning C5034: 'A<T>::f': an out-of-line definition of a member of a class template cannot have default arguments:

```
template <typename T>
struct A {
    T f(T t, bool b = false);
};

template <typename T>
T A<T>::f(T t, bool b = false) // C5034
{
    // ...
}
```

To fix the error, remove the `= false` default argument.

Use of offsetof with compound member designator

In Visual Studio 2017 version 15.3, using `offsetof(T, m)` where *m* is a "compound member designator" results in a warning when you compile with the **/Wall** option. The following code is ill-formed and could potentially cause a crash at runtime. Visual Studio 2017 version 15.3 produces "warning C4841: non-standard extension used: compound member designator in offsetof":

```
struct A {
   int arr[10];
};

// warning C4841: non-standard extension used: compound member designator in offsetof
constexpr auto off = offsetof(A, arr[2]);
```

To fix the code, either disable the warning with a pragma or change the code to not use `offsetof`:

```
#pragma warning(push)
#pragma warning(disable: 4841)
constexpr auto off = offsetof(A, arr[2]);
#pragma warning(pop)
```

Using offsetof with static data member or member function

In Visual Studio 2017 version 15.3, using `offsetof(T, m)` where *m* refers to a static data member or a member function results in an error. The following code produces "error C4597: undefined behavior: offsetof applied to member function 'foo'" and "error C4597: undefined behavior: offsetof applied to static data member 'bar'":

```
#include <cstddef>

struct A {
   int foo() { return 10; }
   static constexpr int bar = 0;
};

constexpr auto off = offsetof(A, foo);
constexpr auto off2 = offsetof(A, bar);
```

This code is ill-formed and could potentially cause a crash at runtime. To fix the error, change the code to no longer invoke undefined behavior. This is non-portable code that is disallowed by the C++ standard.

New warning on declspec attributes

In Visual Studio 2017 version 15.3, the compiler no longer ignores attributes if `__declspec(...)` is applied before `extern "C"` linkage specification. Previously, the compiler would ignore the attribute, which could have runtime implications. When the **/Wall** and **/WX** options are set, the following code produces "warning C4768: __declspec attributes before linkage specification are ignored":

```
__declspec(noinline) extern "C" HRESULT __stdcall //C4768
```

To fix the warning, put extern "C" first:

```
extern "C" __declspec(noinline) HRESULT __stdcall
```

This warning is off by default in 15.3, but on by default in 15.5, and only impacts code compiled with **/Wall /WX**.

decltype and calls to deleted destructors

In previous versions of Visual Studio, the compiler did not detect when a call to a deleted destructor occurred in the context of the expression associated with 'decltype'. In Visual Studio 2017 version 15.3, the following code produces "error C2280: 'A<T>::~A(void)': attempting to reference a deleted function":

```
template<typename T>
struct A
{
   ~A() = delete;
};

template<typename T>
auto f() -> A<T>;

template<typename T>
auto g(T) -> decltype((f<T>()));

void h()
{
   g(42);
}
```

Uninitialized const variables

Visual Studio 2017 RTW release had a regression in which the C++ compiler would not issue a diagnostic if a 'const' variable was not initialized. This regression has been fixed in Visual Studio 2017 version 15.3. The following code now produces "warning C4132: 'Value': const object should be initialized":

```
const int Value; //C4132
```

To fix the error, assign a value to `Value`.

Empty declarations

Visual Studio 2017 version 15.3 now warns on empty declarations for all types, not just built-in types. The following code now produces a level 2 C4091 warning for all four declarations:

```
struct A {};
template <typename> struct B {};
enum C { c1, c2, c3 };

int;    // warning C4091 : '' : ignored on left of 'int' when no variable is declared
A;      // warning C4091 : '' : ignored on left of 'main::A' when no variable is declared
B<int>; // warning C4091 : '' : ignored on left of 'B<int>' when no variable is declared
C;      // warning C4091 : '' : ignored on left of 'C' when no variable is declared
```

To remove the warnings, simply comment-out or remove the empty declarations. In cases where the un-named object is intended to have a side effect (such as RAII) it should be given a name.

The warning is excluded under **/Wv:18** and is on by default under warning level W2.

std::is_convertible for array types

Previous versions of the compiler gave incorrect results for std::is_convertible for array types. This required library writers to special-case the Microsoft C++ compiler when using the `std::is_convertible<...>` type trait. In the following example, the static asserts pass in earlier versions of Visual Studio but fail in Visual Studio 2017 version 15.3:

```
#include <type_traits>

using Array = char[1];

static_assert(std::is_convertible<Array, Array>::value);
static_assert(std::is_convertible<const Array, const Array>::value, "");
static_assert(std::is_convertible<Array&, Array>::value, "");
static_assert(std::is_convertible<Array, Array&>::value, "");
```

`std::is_convertible<From, To>` is calculated by checking to see if an imaginary function definition is well formed:

```
   To test() { return std::declval<From>(); }
```

Private destructors and std::is_constructible

Previous versions of the compiler ignored whether a destructor was private when deciding the result of std::is_constructible. It now considers them. In the following example, the static asserts pass in earlier versions of Visual Studio but fail in Visual Studio 2017 version 15.3:

```
#include <type_traits>

class PrivateDtor {
   PrivateDtor(int) { }
private:
   ~PrivateDtor() { }
};

// This assertion used to succeed. It now correctly fails.
static_assert(std::is_constructible<PrivateDtor, int>::value);
```

Private destructors cause a type to be non-constructible. `std::is_constructible<T, Args...>` is calculated as if the following declaration were written:

```
   T obj(std::declval<Args>()...)
```

This call implies a destructor call.

C2668: Ambiguous overload resolution

Previous versions of the compiler sometimes failed to detect ambiguity when it found multiple candidates via both using declarations and argument dependent lookup. This can lead to wrong overload being chosen and unexpected runtime behavior. In the following example, Visual Studio 2017 version 15.3 correctly raises C2668 'f': ambiguous call to overloaded function:

```
namespace N {
   template<class T>
   void f(T&, T&);

   template<class T>
   void f();
}

template<class T>
void f(T&, T&);

struct S {};
void f()
{
   using N::f;

   S s1, s2;
   f(s1, s2); // C2668
}
```

To fix the code, remove the using `N::f` statement if you intended to call `::f()`.

C2660: local function declarations and argument dependent lookup

Local function declarations hide the function declaration in the enclosing scope and disable argument dependent lookup. However, previous versions of the compiler performed argument dependent lookup in this case, potentially leading to the wrong overload being chosen and unexpected runtime behavior. Typically, the error is due to an incorrect signature of the local function declaration. In the following example, Visual Studio 2017 version 15.3 correctly raises C2660 'f': function does not take 2 arguments:

```
struct S {};
void f(S, int);

void g()
{
   void f(S); // C2660 'f': function does not take 2 arguments:
   // or void f(S, int);
   S s;
   f(s, 0);
}
```

To fix the problem, either change the `f(S)` signature or remove it.

C5038: order of initialization in initializer lists

Class members are initialized in the order they are declared, not the order they appear in initializer lists. Previous versions of the compiler did not warn when the order of the initializer list differed from the order of declaration. This could lead to undefined runtime behavior if the initialization of one member depended on another member in the list already being initialized. In the following example, Visual Studio 2017 version 15.3 (with **/Wall**) raises "warning C5038: data member 'A::y' will be initialized after data member 'A::x'":

```
struct A
{
    A(int a) : y(a), x(y) {} // Initialized in reverse, y reused
    int x;
    int y;
};
```

To fix the problem, arrange the initializer list to have the same order as the declarations. A similar warning is raised when one or both initializers refer to base class members.

Note that the warning is off-by-default and only affects code compiled with **/Wall**.

Bug fixes and other behavior changes in Visual Studio 2017 version 15.5

Partial Ordering Change

The compiler now correctly rejects the following code and gives the correct error message:

```
template<typename... T>
int f(T* ...)
{
    return 1;
}

template<typename T>
int f(const T&)
{
    return 2;
}

int main()
{
    int i = 0;
    f(&i);    // C2668
}
```

```
t161.cpp
t161.cpp(16): error C2668: 'f': ambiguous call to overloaded function
t161.cpp(8): note: could be 'int f<int*>(const T &)'
        with
        [
            T=int*
        ]
t161.cpp(2): note: or       'int f<int>(int*)'
t161.cpp(16): note: while trying to match the argument list '(int*)'
```

The problem in the example above is that there are two differences in the types (const vs. non-const and pack vs. non-pack). To eliminate the compiler error, remove one of the differences. This enables the compiler to unambiguously order the functions.

```
template<typename... T>
int f(T* ...)
{
    return 1;
}

template<typename T>
int f(T&)
{
    return 2;
}

int main()
{
    int i = 0;
    f(&i);
}
```

Exception handlers

Handlers of reference to array or function type are never a match for any exception object. The compiler now correctly honors this rule and raises a level 4 warning. It also no longer matches a handler of `char*` or `wchar_t*` to a string literal when **/Zc:strictStrings** is used.

```
int main()
{
    try {
        throw "";
    }
    catch (int (&)[1]) {} // C4843 (This should always be dead code.)
    catch (void (&)()) {} // C4843 (This should always be dead code.)
    catch (char*) {} // This should not be a match under /Zc:strictStrings
}
```

```
warning C4843: 'int (&)[1]': An exception handler of reference to array or function type is unreachable, use
'int*' instead
warning C4843: 'void (__cdecl &)(void)': An exception handler of reference to array or function type is
unreachable, use 'void (__cdecl*)(void)' instead
```

The following code avoids the error:

```
catch (int (*)[1]) {}
```

std::tr1 namespace is deprecated

The non-Standard `std::tr1` namespace is now marked as deprecated in both C++14 and C++17 modes. In Visual Studio 2017 version 15.5, the following code raises C4996:

```
#include <functional>
#include <iostream>
using namespace std;

int main() {
    std::tr1::function<int (int, int)> f = std::plus<int>(); //C4996
    cout << f(3, 5) << std::endl;
    f = std::multiplies<int>();
    cout << f(3, 5) << std::endl;
}
```

```
warning C4996: 'std::tr1': warning STL4002: The non-Standard std::tr1 namespace and TR1-only machinery are
deprecated and will be REMOVED. You can define _SILENCE_TR1_NAMESPACE_DEPRECATION_WARNING to acknowledge that
you have received this warning.
```

To fix the error, remove the reference to the `tr1` namespace:

```
#include <functional>
#include <iostream>
using namespace std;

int main() {
    std::function<int (int, int)> f = std::plus<int>();
    cout << f(3, 5) << std::endl;
    f = std::multiplies<int>();
    cout << f(3, 5) << std::endl;
}
```

Standard Library features in Annex D are marked as deprecated

When the **/std:c++17** mode compiler switch is set, almost all Standard Library features in Annex D are marked as deprecated.

In Visual Studio 2017 version 15.5, the following code raises C4996:

```
#include <iterator>

class MyIter : public std::iterator<std::random_access_iterator_tag, int> {
public:
    // ... other members ...
};

#include <type_traits>

static_assert(std::is_same<MyIter::pointer, int*>::value, "BOOM");
```

```
warning C4996: 'std::iterator<std::random_access_iterator_tag,int,ptrdiff_t,_Ty*,_Ty &>::pointer': warning
STL4015: The std::iterator class template (used as a base class to provide typedefs) is deprecated in C++17.
(The <iterator> header is NOT deprecated.) The C++ Standard has never required user-defined iterators to
derive from std::iterator. To fix this warning, stop deriving from std::iterator and start providing publicly
accessible typedefs named iterator_category, value_type, difference_type, pointer, and reference. Note that
value_type is required to be non-const, even for constant iterators. You can define
_SILENCE_CXX17_ITERATOR_BASE_CLASS_DEPRECATION_WARNING or _SILENCE_ALL_CXX17_DEPRECATION_WARNINGS to
acknowledge that you have received this warning.
```

To fix the error, follow the instructions in the warning text, as demonstrated in the following code:

```
#include <iterator>

class MyIter {
public:
    typedef std::random_access_iterator_tag iterator_category;
    typedef int value_type;
    typedef ptrdiff_t difference_type;
    typedef int* pointer;
    typedef int& reference;

    // ... other members ...
};

#include <type_traits>

static_assert(std::is_same<MyIter::pointer, int*>::value, "BOOM");
```

Unreferenced local variables

In Visual Studio 15.5, warning C4189 is emitted in more cases, as shown in the following code:

```
void f() {
    char s[2] = {0}; // C4189. Either use the variable or remove it.
}
```

```
warning C4189: 's': local variable is initialized but not referenced
```

To fix the error, remove the unused variable.

Single line comments

In Visual Studio 2017 version 15.5, warnings C4001 and C4179 are no longer emitted by the C compiler.

Previously, they were only emitted under the **/Za** compiler switch. The warnings are no longer needed because single line comments have been part of the C standard since C99.

```
/* C only */
#pragma warning(disable:4001) //C4619
#pragma warning(disable:4179)
// single line comment
//* single line comment */
```

```
warning C4619: #pragma warning: there is no warning number '4001'
```

If the code does not need to be backwards compatible, you can avoid the warning by removing the C4001/C4179 suppression. If the code does need to be backward compatible, then suppress C4619 only.

```

/* C only */

#pragma warning(disable:4619)
#pragma warning(disable:4001)
#pragma warning(disable:4179)

// single line comment
/* single line comment */
```

__declspec attributes with extern "C" linkage

In earlier versions of Visual Studio, the compiler ignored `__declspec(...)` attributes when `__declspec(...)` was applied before the `extern "C"` linkage specification. This behavior caused code to be generated that user didn't intend, with possible runtime implications. The warning was added in Visual Studio version 15.3, but was off by default. In Visual Studio 2017 version 15.5, the warning is enabled by default.

```
__declspec(noinline) extern "C" HRESULT __stdcall //C4768
```

```
warning C4768: __declspec attributes before linkage specification are ignored
```

To fix the error, place the linkage specification before the __declspec attribute:

```
extern "C" __declspec(noinline) HRESULT __stdcall
```

This new warning C4768 is given on some Windows SDK headers that were shipped with Visual Studio 2017 15.3 or older (for example: version 10.0.15063.0, also known as RS2 SDK). However, later versions of Windows SDK headers (specifically, ShlObj.h and ShlObj_core.h) have been fixed so that they do not produce this warning. When you see this warning coming from Windows SDK headers, you can take these actions:

1. Switch to the latest Windows SDK that came with Visual Studio 2017 version 15.5 release.

2. Turn off the warning around the #include of the Windows SDK header statement:

```
   #pragma warning (push)
   #pragma warning(disable:4768)
   #include <shlobj.h>
   #pragma warning (pop)
```

Extern constexpr linkage

In earlier versions of Visual Studio, the compiler always gave a `constexpr` variable internal linkage even when the variable was marked `extern`. In Visual Studio 2017 version 15.5, a new compiler switch (**/Zc:externConstexpr**) enables correct standards-conforming behavior. Eventually this will become the default.

```
extern constexpr int x = 10;
```

```
error LNK2005: "int const x" already defined
```

If a header file contains a variable declared `extern constexpr`, it needs to be marked `__declspec(selectany)` in order to correctly have its duplicate declarations combined:

```
extern constexpr __declspec(selectany) int x = 10;
```

typeid can't be used on incomplete class type

In earlier versions of Visual Studio, the compiler incorrectly allowed the following code, resulting in potentially incorrect type information. In Visual Studio 2017 version 15.5, the compiler correctly raises an error:

```
#include <typeinfo>

struct S;

void f() { typeid(S); } //C2027 in 15.5
```

```
error C2027: use of undefined type 'S'
```

std::is_convertible target type

`std::is_convertible` requires the target type to be a valid return type. In earlier versions of Visual Studio, the compiler incorrectly allowed abstract types, which might lead to incorrect overload resolution and unintended runtime behavior. The following code now correctly raises C2338:

```
#include <type_traits>

struct B { virtual ~B() = 0; };
struct D : public B { virtual ~D(); };

static_assert(std::is_convertible<D, B>::value, "fail"); // C2338 in 15.5
```

To avoid the error, when using `is_convertible` you should compare pointer types because a non-pointer-type comparison might fail if one type is abstract:

```
#include <type_traits>

struct B { virtual ~B() = 0; };
struct D : public B { virtual ~D(); };

static_assert(std::is_convertible<D *, B *>::value, "fail");
```

Dynamic exception specification removal and noexcept

In C++17, `throw()` is an alias for `noexcept`, `throw(<type list>)` and `throw(...)` are removed, and certain types may include `noexcept`. This can cause source compatibility issues with code that conforms to C++14 or earlier. The **/Zc:noexceptTypes-** switch can be used to revert to the C++14 version of `noexcept` while using C++17 mode in general. This enables you to update your source code to conform to C++17 without having to rewrite all your `throw()` code at the same time.

The compiler also now diagnoses more mismatched exception specifications in declarations in C++17 mode or with **/permissive-** with the new warning C5043.

The following code generates C5043 and C5040 in Visual Studio 2017 version 15.5 when the **/std:c++17** switch is applied:

```
void f() throw(); // equivalent to void f() noexcept;
void f() {} // warning C5043
void g() throw(); // warning C5040

struct A {
    virtual void f() throw();
};

struct B : A {
    virtual void f() { } // error C2694
};
```

To remove the errors while still using **/std:c++17**, either add the **/Zc:noexceptTypes-** switch to the command line, or else update your code to use `noexcept`, as shown in the following example:

```
void f() noexcept;
void f() noexcept { }
void g() noexcept(false);

struct A {
    virtual void f() noexcept;
};

struct B : A {
    virtual void f() noexcept { }
};
```

Inline variables

Static constexpr data members are now implicitly inline, which means that their declaration within a class is now their definition. Using an out-of-line definition for a static constexpr data member is redundant, and now deprecated. In Visual Studio 2017 version 15.5 when the **/std:c++17** switch is applied, the following code now produces warning C5041 *'size': out-of-line definition for constexpr static data member is not needed and is deprecated in C++17*:

```
struct X {
    static constexpr int size = 3;
};
const int X::size; // C5041
```

extern "C" __declspec(...) warning C4768 now on by default

The warning was added in Visual Studio 2017 version 15.3 but was off by default. In Visual Studio 2017 version 15.5 it is on by default. See New warning on declspec attributes for more information.

Defaulted functions and __declspec(nothrow)

The compiler previously allowed defaulted functions to be declared with `__declspec(nothrow)` when the corresponding base/member functions permitted exceptions. This behavior is contrary to the C++ Standard and can cause undefined behavior at runtime. The standard requires such functions to be defined as deleted if there is an exception specification mismatch. Under **/std:c++17**, the following code raises C2280 *attempting to reference a deleted function. Function was implicitly deleted because the explicit exception specification is incompatible with that of the implicit declaration.*:

```
struct A {
    A& operator=(const A& other) { // No exception specification; this function may throw.
        ...
    }
};

struct B : public A {
    __declspec(nothrow) B& operator=(const B& other) = default;
};

int main()
{
    B b1, b2;
    b2 = b1; // error C2280
}
```

To correct this code, either remove __declspec(nothrow) from the defaulted function, or remove `= default` and provide a definition for the function along with any required exception handling:

```
struct A {
    A& operator=(const A& other) {
        // ...
    }
};

struct B : public A {
    B& operator=(const B& other) = default;
};

int main()
{
    B b1, b2;
    b2 = b1;
}
```

noexcept and partial specializations

With noexcept in the type system, partial specializations for matching particular "callable" types may fail to compile or choose the primary template due to a missing partial specialization for pointers-to-noexcept-functions.

In such cases, you may need to add additional partial specializations to handle the noexcept function pointers and noexcept pointers to member functions. These overloads are only legal in **/std:c++17** mode. If backwards-compatibility with C++14 must be maintained, and you are writing code that others consume, then you should guard these new overloads inside `#ifdef` directives. If you are working in a self-contained module, then instead of using `#ifdef` guards you can just compile with the **/Zc:noexceptTypes-** switch.

The following code compiles under **/std:c++14** but fails under **/std:c++17** with "error C2027:use of undefined type 'A<T>'":

```
template <typename T> struct A;

template <>
struct A<void(*)()>
{
    static const bool value = true;
};

template <typename T>
bool g(T t)
{
    return A<T>::value;
}

void f() noexcept {}

int main()
{
    return g(&f) ? 0 : 1; // C2027
}
```

The following code succeeds under **/std:c++17** because the compiler chooses the new partial specialization `A<void (*)() noexcept>`:

```
template <typename T> struct A;

template <>
struct A<void(*)()>
{
    static const bool value = true;
};

template <>
struct A<void(*)() noexcept>
{
    static const bool value = true;
};

template <typename T>
bool g(T t)
{
    return A<T>::value;
}

void f() noexcept {}

int main()
{
    return g(&f) ? 0 : 1; // OK
}
```

Bug fixes and other behavior changes in Visual Studio 2017 version 15.7

C++17 Default argument in the primary class template

This behavior change is a precondition for Template argument deduction for class templates - P0091R3, which is planned to be fully supported in a later preview of Visual Studio 2017 version 15.7.

Previously, the compiler ignored the default argument in the primary class template.

```
template<typename T>
struct S {
    void f(int = 0);
};

template<typename T>
void S<T>::f(int = 0) {} // Re-definition necessary
```

In **/std:c++17** mode in Visual Studio 2017 version 15.7, the default argument is not ignored:

```
template<typename T>
struct S {
    void f(int = 0);
};

template<typename T>
void S<T>::f(int) {} // Default argument is used
```

Dependent name resolution

This behavior change is a precondition for Template argument deduction for class templates - P0091R3, which is planned to be fully supported in a later preview of Visual Studio 2017 version 15.7.

In the following example, the compiler in Visual Studio 15.6 and earlier resolves `D::type` to `B<T>::type` in the primary class template.

```
template<typename T>
struct B {
    using type = T;
};

template<typename T>
struct D : B<T*> {
    using type = B<T*>::type;
};
```

Visual Studio 2017 version 15.7, in **/std:c++17** mode, requires the `typename` keyword in the `using` statement in D. Without `typename` the compiler raises warning C4346: '*B<T*>::type*': *dependent name is not a type* and error C2061: *syntax error: identifier 'type'*:

```
template<typename T>
struct B {
    using type = T;
};

template<typename T>
struct D : B<T*> {
    using type = typename B<T*>::type;
};
```

C++17 [[nodiscard]] attribute - warning level increase

In Visual Studio 2017 version 15.7 in **/std:c++17** mode, the warning level of C4834 ("discarding return value of function with 'nodiscard' attribute") is increased from W3 to W1. You can disable the warning with a cast to `void`, or by passing **/wd:4834** to the compiler

```
[[nodiscard]] int f() { return 0; }

int main() {
    f(); // warning: discarding return value
         // of function with 'nodiscard'
}
```

Variadic template constructor base class initialization list

In previous editions of Visual Studio, a variadic template constructor base class initialization list that was missing template arguments was erroneously allowed without error. In Visual Studio 2017 version 15.7, a compiler error is raised.

The following code example in Visual Studio 2017 version 15.7 raises *error C2614: D<int>: illegal member initialization: 'B' is not a base or member*

```
template<typename T>
struct B {};

template<typename T>
struct D : B<T>
{

    template<typename ...C>
    D() : B() {} // C2614. Missing template arguments to B.
};

D<int> d;
```

To fix the error, change the B() expression to B<T>().

constexpr aggregate initialization

Previous versions of the C++ compiler incorrectly handled constexpr aggregate initialization; it accepted invalid code in which the aggregate-init-list had too many elements, and produced bad codegen for it. The following code is an example of such code:

```
#include <array>
struct X {
    unsigned short a;
    unsigned char b;
};

int main() {
    constexpr std::array<X, 2> xs = {
        { 1, 2 },
        { 3, 4 }
    };
    return 0;
}
```

In Visual Studio 2017 version 15.7 update 3 and later, the previous example now raises *C2078 too many initializers*. The following example shows how to fix the code. When initializing a `std::array` with nested brace-init-lists, give the inner array a braced-list of its own:

```
#include <array>
struct X {
    unsigned short a;
    unsigned char b;
};

int main() {
    constexpr std::array<X, 2> xs = {{ // note double braces
        { 1, 2 },
        { 3, 4 }
    }}; // note double braces
    return 0;
}
```

Bug fixes and behavior changes in Visual Studio 2017 version 15.8

The compiler changes in Visual Studio 2017 version 15.8 all fall under the category of bug fixes and behavior changes, and are listed below:

typename on unqualified identifiers

In /permissive- mode, spurious `typename` keywords on unqualified identifiers in alias template definitions are no longer accepted by the compiler. The following code now produces C7511 *'T': 'typename' keyword must be followed by a qualified name*:

```
template <typename T>
using  X = typename T;
```

To fix the error, simply change the second line to `using X = T;`.

__declspec() on right side of alias template definitions

__declspec is no longer permitted on the right-hand-side of an alias template definition. This was previously accepted by the compiler but was completely ignored, and would never result in a deprecation warning when the alias was used.

The standard C++ attribute [[deprecated]] may be used instead, and will be respected as of Visual Studio 2017 version 15.6. The following code now produces C2760 *syntax error: unexpected token '__declspec', expected 'type specifier'*:

```
template <typename T>
using X = __declspec(deprecated("msg")) T;
```

To fix the error, change to code to the following (with the attribute placed before the '=' of the alias definition):

```
template <typename T>
using  X [[deprecated("msg")]] = T;
```

Two-phase name lookup diagnostics

Two-phase name lookup requires that non-dependent names used in template bodies must be visible to the template at definition time. Previously, the Microsoft C++ compiler would leave an unfound name un-looked-up until instantiation times. Now, it requires that non-dependent names are bound in the template body.

One way this can manifest is with lookup into dependent base classes. Previously, the compiler allowed the use of names that are defined in dependent base classes because they would be looked up during instantiation time when all the types are resolved. Now that code it is treated as an error. In these cases you can force the variable to be looked up at instantiation time by qualifying it with the base class type or otherwise making it dependent, for example by adding a `this->` pointer.

In **/permissive-** mode, the following code now raises C3861: '*base_value*': *identifier not found*:

```
template <class T>
struct Base {
    int base_value = 42;
};

template <class T>
struct S : Base<T> {
    int f() {
        return base_value;
    }
};
```

To fix the error, change the `return` statement to `return this->base_value;`.

Note: In the Boost python library, there has been for a long time an MSVC-specific workaround for a template forward declaration in unwind_type.hpp. Under /permissive- mode starting with Visual Studio 2017 version 15.8 (_MSC_VER=1915), the MSVC compiler does argument-dependent name lookup (ADL) correctly and is consistent with other compilers, making this workaround guard unnecessary. In order to avoid this error *C3861: 'unwind_type': identifier not found*, see PR 229 in the Boostorg repo to update the header file. We have already patched the vcpkg Boost package, so if you get or upgrade your Boost sources from vcpkg then you do not need to apply the patch separately.

forward declarations and definitions in namespace std

The C++ standard doesn't allow a user to add forward declarations or definitions into namespace `std`. Adding declarations or definitions to namespace `std` or to a namespace within namespace std now results in undefined behavior.

At some time in the future, Microsoft will move the location where some STL types are defined. When this happens, it will break existing code that adds forward declarations to namespace `std`. A new warning, C4643, helps identify such source issues. The warning is enabled in **/default** mode and is off by default. It will impact programs that are compiled with **/Wall** or **/WX**.

The following code now raises C4643: *Forward declaring 'vector' in namespace std is not permitted by the C++ Standard*.

```
namespace std {
    template<typename T> class vector;
}
```

To fix the error, use an **include** directive rather than a forward declaration:

```
#include <vector>
```

Constructors that delegate to themselves

The C++ Standard suggests that a compiler should emit a diagnostic when a delegating constructor delegates to itself. The Microsoft C++ compiler in /std:c++17 and /std:c++latest modes now raises C7535: '*X::X*': *delegating constructor calls itself*.

Without this error, the following program will compile but will generate an infinite loop:

```
class X {
public:
    X(int, int);
    X(int v) : X(v){}
};
```

To avoid the infinite loop, delegate to a different constructor:

```
class X {
public:

    X(int, int);
    X(int v) : X(v, 0) {}
};
```

offsetof with constant expressions

offsetof has traditionally been implemented using a macro that requires a reinterpret_cast. This is illegal in contexts that require a constant expression, but the Microsoft C++ compiler has traditionally allowed it. The offsetof macro that is shipped as part of the STL correctly uses a compiler intrinsic (**__builtin_offsetof**), but many people have used the macro trick to define their own **offsetof**.

In Visual Studio 2017 version 15.8, the compiler constrains the areas that these reinterpret_casts can appear in the default mode in order to help code conform to standard C++ behavior. Under /permissive-, the constraints are even stricter. Using the result of an offsetof in places that require constant expressions may result in code that issues warning C4644 *usage of the macro-based offsetof pattern in constant expressions is non-standard; use offsetof defined in the C++ standard library instead* or C2975 *invalid template argument, expected compile-time constant expression*.

The following code raises C4644 in **/default** and **/std:c++17** modes, and C2975 in **/permissive-** mode:

```
struct Data {
    int x;
};

// Common pattern of user-defined offsetof
#define MY_OFFSET(T, m) (unsigned long long)(&(((T*)nullptr)->m))

int main()

{
    switch (0) {
    case MY_OFFSET(Data, x): return 0;
    default: return 1;
    }
}
```

To fix the error, use **offsetof** as defined via <cstddef>:

```
#include <cstddef>

struct Data {
    int x;
};

int main()
{
    switch (0) {
    case offsetof(Data, x): return 0;
    default: return 1;
    }
}
```

cv-qualifiers on base classes subject to pack expansion

Previous versions of the Microsoft C++ compiler did not detect that a base-class had cv-qualifiers if it was also subject to pack expansion.

In Visual Studio 2017 version 15.8, in **/permissive-** mode the following code raises C3770 '*const S*'*: is not a valid base class*:

```
template<typename... T>
class X : public T... { };

class S { };

int main()
{
    X<const S> x;
}
```

template keyword and nested-name-specifiers

In **/permissive-** mode, the compiler now requires the `template` keyword to precede a template-name when it comes after a nested-name-specifier which is dependent.

The following code in **/permissive-** mode now raises C7510: '*foo*'*: use of dependent template name must be prefixed with* '*template*'*. note: see reference to class template instantiation* '*X*' *being compiled*:

```
template<typename T> struct Base
{
    template<class U> void foo() {}
};

template<typename T>
struct X : Base<T>
{
    void foo()
    {
        Base<T>::foo<int>();
    }
};
```

To fix the error, add the `template` keyword to the `Base<T>::foo<int>();` statement, as shown in the following example:

```
template<typename T> struct Base
{
    template<class U> void foo() {}
};

template<typename T>
struct X : Base<T>
{
    void foo()
    {
        // Add template keyword here:
        Base<T>::template foo<int>();
    }
};
```

Bug fixes and behavior changes in Visual Studio 2017 version 15.9

Identifiers in member alias templates

An identifier used in a member alias template definition must be declared before use.

In previous versions of the compiler, the following code was allowed:

```
template <typename... Ts>
struct A
{
  public:
    template <typename U>
    using from_template_t = decltype(from_template(A<U>{}));

  private:
    template <template <typename...> typename Type, typename... Args>
    static constexpr A<Args...> from_template(A<Type<Args...>>);
};

A<>::from_template_t<A<int>> a;
```

In Visual Studio 2017 version 15.9, in **/permissive-** mode, the compiler raises C3861: '*from_template*': *identifier not found*.

To fix the error, declare `from_template` before `from_template_t`.

Modules changes

In Visual Studio 2017, version 15.9, the compiler raises C5050 whenever the command line options for modules are not consistent between the module creation and module consumption sides. In the following example, there are two issues:

- on the consumption side (main.cpp) the option **/EHsc** is not specified.
- the C++ version is **/std:c++17** on the creation side and **/std:c++14** on the consumption side.

```
cl /EHsc /std:c++17 m.ixx /experimental:module
cl /experimental:module /module:reference m.ifc main.cpp /std:c++14
```

The compiler raises C5050 for both of these cases: *warning C5050: Possible incompatible environment while importing module 'm': mismatched C++ versions. Current "201402" module version "201703"*.

In addition, the compiler raises C7536 whenever the .ifc file has been tampered with. The header of the module interface contains an SHA2 hash of the contents below it. On import, the .ifc file is hashed in the same way and then checked against the hash provided in the header; if these do not match error C7536 is raised: *ifc failed* *integrity checks. Expected SHA2: '66d5c8154df0c71d4cab7665bab4a125c7ce5cb9a401a4d8b461b706ddd771c6'.*

Partial ordering involving aliases and non-deduced contexts

There is implementation divergence in the partial ordering rules involving aliases in non-deduced contexts. In the following example, GCC and the Microsoft C++ compiler (in **/permissive-** mode) raise an error, while Clang accepts the code.

```
#include <utility>
using size_t = std::size_t;

template <typename T>
struct A {};
template <size_t, size_t>
struct AlignedBuffer {};
template <size_t len>
using AlignedStorage = AlignedBuffer<len, 4>;

template <class T, class Alloc>
int f(Alloc &alloc, const AlignedStorage<T::size> &buffer)
{
    return 1;
}

template <class T, class Alloc>
int f(A<Alloc> &alloc, const AlignedStorage<T::size> &buffer)
{
    return 2;
}

struct Alloc
{
    static constexpr size_t size = 10;
};

int main()
{
    A<void> a;
    AlignedStorage<Alloc::size> buf;
    if (f<Alloc>(a, buf) != 2)
    {
        return 1;
    }

    return 0;
}
```

The previous example raises C2668:

```
partial_alias.cpp(32): error C2668: 'f': ambiguous call to overloaded function
partial_alias.cpp(18): note: could be 'int f<Alloc,void>(A<void> &,const AlignedBuffer<10,4> &)'
partial_alias.cpp(12): note: or       'int f<Alloc,A<void>>(Alloc &,const AlignedBuffer<10,4> &)'
        with
        [
            Alloc=A<void>
        ]
partial_alias.cpp(32): note: while trying to match the argument list '(A<void>, AlignedBuffer<10,4>)'
```

The implementation divergence is due to a regression in the Standard wording where the resolution to core issue 2235 removed some text that would allow these overloads to be ordered. The current C++ standard does not provide a mechanism to partially order these functions, so they are considered ambiguous.

As a workaround, we recommended that you not rely on partial ordering to resolve this problem, and instead use SFINAE to remove particular overloads. In the following example, we use a helper class `IsA` to remove the first overload when `Alloc` is a specialization of `A`:

```
#include <utility>
using size_t = std::size_t;

template <typename T>
struct A {};
template <size_t, size_t>
struct AlignedBuffer {};
template <size_t len>
using AlignedStorage = AlignedBuffer<len, 4>;

template <typename T> struct IsA : std::false_type {};
template <typename T> struct IsA<A<T>> : std::true_type {};

template <class T, class Alloc, typename = std::enable_if_t<!IsA<Alloc>::value>>
int f(Alloc &alloc, const AlignedStorage<T::size> &buffer)
{
    return 1;
}

template <class T, class Alloc>
int f(A<Alloc> &alloc, const AlignedStorage<T::size> &buffer)
{
    return 2;
}

struct Alloc
{
    static constexpr size_t size = 10;
};

int main()
{
    A<void> a;
    AlignedStorage<Alloc::size> buf;
    if (f<Alloc>(a, buf) != 2)
    {
        return 1;
    }

    return 0;
}
```

Illegal expressions and non-literal types in templated function definitions

Illegal expressions and non-literal types are now correctly diagnosed in the definitions of templated functions that are explicitly specialized. Previously, such errors were not emitted for the function definition. However, the illegal expression or non-literal type would still have been diagnosed if evaluated as part of a constant expression.

In previous versions of Visual Studio, the following code compiles without warning:

```
void g();

template<typename T>
struct S
{
    constexpr void f();
};

template<>
constexpr void S<int>::f()
{
    g(); // C3615 in 15.9
}
```

In Visual Studio 2017 version 15.9, the code raises this error: *error C3615: constexpr function 'S::f' cannot result in a constant expression. note: failure was caused by call of undefined function or one not declared 'constexpr' note: see usage of 'g'*. To avoid the error, remove the `constexpr` qualifier from the explicit instantiation of the function f().

See also

Visual C++ Language Conformance

Microsoft C++ Language Conformance Table

5/21/2019 • 16 minutes to read • Edit Online

This topic summarizes the ISO C++03, C++11, C++14, C++17, and C++20 language standards conformance of compiler features and Standard Library features for the Microsoft C++ compiler in Visual Studio 2019 and earlier versions. Each compiler and standard library feature name links to the ISO C++ Standard proposal paper that describes the feature, if one is available at publication time. The Supported column lists the Visual Studio version in which support for the feature first appeared.

For details on conformance improvements and other changes in Visual Studio 2017 or Visual Studio 2019, set the version selector in the upper left of this page, then see C++ conformance improvements in Visual Studio and What's New for Visual C++ in Visual Studio. For conformance changes in earlier versions, see Visual C++ change history and Visual C++ What's New 2003 through 2015. For current news from the C++ team, visit the C++ team blog.

NOTE

There are no binary breaking changes between Visual Studio 2015, Visual Studio 2017, and Visual Studio 2019.

Compiler Features

FEATURE AREA	
C++03/11 Core Language Features	**Supported**
Everything else	VS 2015 [A]
Two-phase name lookup	VS 2017 15.7 [B]
N2634 Expression SFINAE	VS 2017 15.7
N1653 C99 preprocessor	Partial [C]
C++14 Core Language Features	**Supported**
N3323 Tweaked wording for contextual conversions	VS 2013
N3472 Binary literals	VS 2015
N3638 auto and decltype(auto) return types	VS 2015
N3648 init-captures	VS 2015
N3649 Generic lambdas	VS 2015
N3760 [[deprecated]] attribute	VS 2015
N3778 Sized deallocation	VS 2015

FEATURE AREA	
N3781 Digit separators	VS 2015
N3651 Variable templates	VS 2015.2
N3652 Extended constexpr	VS 2017 15.0
N3653 Default member initializers for aggregates	VS 2017 15.0
C++17 Core Language Features	**Supported**
N4086 Removing trigraphs	VS 2010 [14]
N3922 New rules for auto with braced-init-lists	VS 2015 [14]
N4051 typename in template template-parameters	VS 2015 [14]
N4266 Attributes for namespaces and enumerators	VS 2015 [14]
N4267 u8 character literals	VS 2015 [14]
N4230 Nested namespace definitions	VS 2015.3 [17]
N3928 Terse static_assert	VS 2017 15.0 [17]
P0184R0 Generalized range-based for-loops	VS 2017 15.0 [14]
P0188R1 [[fallthrough]] attribute	VS 2017 15.0 [17]
P0001R1 Removing the register keyword	VS 2017 15.3 [17]
P0002R1 Removing operator++ for bool	VS 2017 15.3 [17]
P0018R3 Capturing *this by value	VS 2017 15.3 [17]
P0028R4 Using attribute namespaces without repetition	VS 2017 15.3 [17]
P0061R1 __has_include	VS 2017 15.3 [14]
P0138R2 Direct-list-init of fixed enums from integers	VS 2017 15.3 [17]
P0170R1 constexpr lambdas	VS 2017 15.3 [17]
P0189R1 [[nodiscard]] attribute	VS 2017 15.3 [17]
P0212R1 [[maybe_unused]] attribute	VS 2017 15.3 [17]
P0217R3 Structured bindings	VS 2017 15.3 [17]
P0292R2 constexpr if-statements	VS 2017 15.3 [D]

FEATURE AREA	
P0305R1 Selection statements with initializers	VS 2017 15.3 [17]
P0245R1 Hexfloat literals	VS 2017 15.5 [17]
N4268 Allowing more non-type template args	VS 2017 15.5 [17]
N4295 Fold expressions	VS 2017 15.5 [17]
P0003R5 Removing dynamic-exception-specifications	VS 2017 15.5 [17]
P0012R1 Adding noexcept to the type system	VS 2017 15.5 [17]
P0035R4 Over-aligned dynamic memory allocation	VS 2017 15.5 [17]
P0386R2 Inline variables	VS 2017 15.5 [17]
P0522R0 Matching template template-parameters to compatible arguments	VS 2017 15.5 [17]
P0036R0 Removing some empty unary folds	VS 2017 15.5 [17]
N4261 Fixing qualification conversions	VS 2017 15.7 [17]
P0017R1 Extended aggregate initialization	VS 2017 15.7 [17]
P0091R3 Template argument deduction for class templates P0512R0 Class template argument deduction issues	VS 2017 15.7 [17]
P0127R2 Declaring non-type template parameters with auto	VS 2017 15.7 [17]
P0135R1 Guaranteed copy elision	VS 2017 15.6
P0136R1 Rewording inheriting constructors	VS 2017 15.7 [17]
P0137R1 std::launder	VS 2017 15.7 [17]
P0145R3 Refining expression evaluation order P0400R0 Order of evaluation of function arguments	VS 2017 15.7 [17]
P0195R2 Pack expansions in using-declarations	VS 2017 15.7 [17]
P0283R2 Ignoring unrecognized attributes	VS 2015 [14]

FEATURE AREA	
C++17 Core Language Features (Defect Reports)	**Supported**
P0702R1 Fixing class template argument deduction for initializer-list ctors	VS 2017 15.7 [17]

FEATURE AREA	
P0961R1 Relaxing the structured bindings customization point finding rules	VS 2019 16.0 [17]
P0969R0 Allowing structured bindings to accessible members	VS 2019 16.0 [17]
P0588R1 Simplifying implicit lambda capture	No
P0962R2 Relaxing the range-for loop customization point finding rules	No
P0929R2 Checking for abstract class types	No
P1009R2 Array size deduction in new-expressions	No
P1286R2 Contra CWG DR1778	No
Feature Area	
----	---
C++20 Core Language Features	**Supported**
P0704R1 Fixing const lvalue ref-qualified pointers to members	VS 2015 [14]
P1041R4 Make char16_t/char32_t string literals be UTF-16/32	VS 2015 [14]
P1330R0 Changing the active member of a union inside constexpr	VS 2017 15.0 [14]
P0972R0 noexcept For <chrono> zero(), min(), max()	VS 2017 15.7 [14]
P0515R3 Three-way (spaceship) comparison operator <=>	VS 2019 16.0 [20]
P1008R1 Prohibiting aggregates with user-declared constructors	VS 2019 16.0 [20]
P0329R4 Designated initialization	VS 2019 16.1 [20]
P0409R2 Allowing lambda-capture [=, this]	VS 2019 16.1 [20]
P0515R3 Three-way (spaceship) comparison operator <=>	VS 2019 16.0 [20]
P0941R2 Feature-test macros	VS 2019 16.0 [14]
P1008R1 Prohibiting aggregates with user-declared constructors	VS 2019 16.0 [20]
P0846R0 ADL and function templates that are not visible	VS 2019 16.1 [20]

FEATURE AREA	
P0641R2 const mismatch with defaulted copy constructor	Partial
P0306R4 Adding __VA_OPT__ for comma omission and comma deletion	No
P0315R4 Allowing lambdas in unevaluated contexts	No
P0409R2 Allowing lambda-capture [=, this]	No
[P0428R2 Familiar template syntax for generic lambdas] (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0428r2.pdf)	No
P0479R5 [[likely]] and [[unlikely]] attributes	No
P0542R5 Contracts	No
P0614R1 Range-based for-loops with initializers	No
P0624R2 Default constructible and assignable stateless lambdas	No
P0634R3 Down with typename!	No
P0683R1 Default member initializers for bit-fields	No
P0692R1 Relaxing access checking on specializations	No
P0722R3 Efficient sized delete for variable sized classes	No
P0732R2 Class types in non-type template parameters	No
P0734R0 Concepts	No
P0780R2 Allowing pack expansion in lambda init-capture	No
P0806R2 Deprecate implicit capture of this via [=]	No
P0840R2 [[no_unique_address]] attribute	No
P0857R0 Fixing functionality gaps in constraints	No
P0892R2 Conditional explicit	No
P0912R5 Coroutines	No
P0960R3 Allow initializing aggregates from a parenthesized list of values	No
P1002R1 try-catch blocks in constexpr functions	No

FEATURE AREA	
P1064R0 Allowing virtual function calls in constant expressions	No
P1073R3 Immediate functions	No
P1084R2 Today's return-type-requirements are insufficient	No
P1091R3 Extending structured bindings to be more like variable declarations	No
P1094R2 Nested inline namespaces	No
P1103R3 Modules	No
P1120R0 Consistency improvements for <=> and other comparison operators	No
P1139R2 Address wording issues related to ISO 10646	No
P1141R2 Yet another approach for constrained declarations	No
P1185R2 <=> != ==	No
P1236R1 Signed integers are two's complement	No
P1289R1 Access control in contract conditions	No
P1323R2 Contract postconditions and return type deduction	No
P1327R1 Allowing dynamic_cast, polymorphic typeid in constant expressions	No
P1353R0 Missing feature-test macros	No
P1381R1 Reference capture of structured bindings	No

Standard Library Features

FEATURE AREA	
C++20 Standard Library Features	**Supported**
P0809R0 Comparing Unordered Containers	VS 2010 [14]
P0858R0 Constexpr Iterator Requirements	VS 2017 15.3 [17]
P0777R1 Avoiding Unnecessary Decay	VS 2017 15.7 [14]
P0550R2 remove_cvref	VS 2019 16.0 [20]

FEATURE AREA	
P0318R1 unwrap_reference, unwrap_ref_decay	VS 2019 16.1 [20]
P0457R2 starts_with()/ends_with() For basic_string/basic_string_view	VS 2019 16.1 [20]
P0458R2 contains() For Ordered And Unordered Associative Containers	VS 2019 16.1 [20]
P0646R1 list/forward_list remove()/remove_if()/unique() Return size_type	VS 2019 16.1 [20]
P0769R2 shift_left(), shift_right()	VS 2019 16.1 [20]
P0887R1 type_identity	VS 2019 16.1 [20]
P0019R8 atomic_ref	No
P0020R6 atomic<float>, atomic<double>, atomic<long double>	No
P0053R7 <syncstream> P0753R2 osyncstream Manipulators	No
P0122R7 <span>	No
P0202R3 constexpr For <algorithm> And exchange()	No
P0339R6 polymorphic_allocator<>	No
P0340R3 SFINAE-Friendly underlying_type	No
P0355R7 <chrono> Calendars And Time Zones	No
P0356R5 bind_front()	No
P0357R3 Supporting Incomplete Types In reference_wrapper	No
P0415R1 constexpr For <complex> (Again)	No
P0439R0 enum class memory_order	No
P0463R1 endian	No
P0475R1 Guaranteed Copy Elision For Piecewise Construction	No
P0476R2 bit_cast	No
P0482R6 char8_t: A type for UTF-8 characters and strings	No

FEATURE AREA	
P0487R1 Fixing operator>>(basic_istream&, CharT*)	No
P0528R3 Atomic Compare-And-Exchange With Padding Bits	No
P0556R3 ispow2(), ceil2(), floor2(), log2p1()	No
P0591R4 Utility Functions For Uses-Allocator Construction	No
P0600R1 [[nodiscard]] For The STL, Part 1	No
P0608R3 Improving variant's Converting Constructor/Assignment	No
P0616R0 Using move() In <numeric>	No
P0619R4 Removing C++17-Deprecated Features In C++20	No
P0653R2 to_address()	No
P0655R1 visit()	No
P0674R1 make_shared() For Arrays	No
P0718R2 atomic<shared_ptr<T>>, atomic<weak_ptr<T>>	No
P0738R2 istream_iterator Cleanup	No
P0754R2 <version>	No
P0758R1 is_nothrow_convertible	No
P0767R1 Deprecating is_pod	No
P0768R1 Library Support For The Spaceship Comparison Operator <=>	No
P0771R1 noexcept For std::function's Move Constructor	No
P0811R3 midpoint(), lerp()	No
P0879R0 constexpr For Swapping Functions	No
P0896R4 <ranges>	No
P0898R3 Standard Library Concepts	No
P0912R5 Library Support For Coroutines	No
P0919R3 Heterogeneous Lookup For Unordered Containers	No

FEATURE AREA	
P0920R2 Precalculated Hash Value Lookup	No
P0935R0 Eradicating Unnecessarily Explicit Default Constructors	No
P0966R1 string::reserve() Should Not Shrink	No
P1001R2 execution::unseq	No
P1006R1 constexpr For pointer_traits<T*>::pointer_to()	No
P1007R3 assume_aligned()	No
P1020R1 Smart Pointer Creation With Default Initialization	No
P1023R0 constexpr For std::array Comparisons	No
P1032R1 Miscellaneous constexpr	No
P1165R1 Consistently Propagating Stateful Allocators In basic_string's operator+()	No
P1209R0 erase_if(), erase()	No
P1227R2 Signed std::ssize(), Unsigned span::size()	No
P1285R0 Improving Completeness Requirements For Type Traits	No
P1357R1 is_bounded_array, is_unbounded_array	No
C++17 Standard Library Features	**Supported**
LWG 2221 Formatted output operator for nullptr	VS 2019 16.1
N3911 void_t	VS 2015 [14]
N4089 Safe Conversions In unique_ptr<T[]>	VS 2015 [14]
N4169 invoke()	VS 2015 [14]
N4190 Removing auto_ptr, random_shuffle(), And Old <functional> Stuff	VS 2015 [rem]
N4258 noexcept Cleanups	VS 2015 [14]
N4259 uncaught_exceptions()	VS 2015 [14]
N4277 Trivially Copyable reference_wrapper	VS 2015 [14]

FEATURE AREA	
N4279 insert_or_assign()/try_emplace() For map/unordered_map	VS 2015 [14]
N4280 size(), empty(), data()	VS 2015 [14]
N4366 Precisely Constraining unique_ptr Assignment	VS 2015 [14]
N4387 Improving pair And tuple	VS 2015.2 [14]
N4389 bool_constant	VS 2015 [14]
N4508 shared_mutex (Untimed)	VS 2015.2 [14]
N4510 Supporting Incomplete Types In vector/list/forward_list	VS 2013 [14]
N4562 Library Fundamentals: <algorithm> sample()	VS 2017 15.0
N4562 Library Fundamentals: <any>	VS 2017 15.0
N4562 Library Fundamentals: <memory_resource> P0337R0 Deleting polymorphic_allocator Assignment	VS 2017 15.6
N4562 Library Fundamentals: <optional>	VS 2017 15.0
N4562 Library Fundamentals: <string_view>	VS 2017 15.0
N4562 Library Fundamentals: <tuple> apply()	VS 2017 15.0
N4562 Library Fundamentals: Boyer-Moore search() P0253R1 Fixing Searcher Return Types	VS 2017 15.3 [17]
P0003R5 Removing Dynamic Exception Specifications	VS 2017 15.5 [17]
P0004R1 Removing Deprecated Iostreams Aliases	VS 2015.2 [rem]
P0005R4 not_fn() P0358R1 Fixes For not_fn()	VS 2017 15.5 [17]
P0006R0 Variable Templates For Type Traits (is_same_v, etc.)	VS 2015.2 [14]
P0007R1 as_const()	VS 2015.2 [14]
P0013R1 Logical Operator Type Traits (conjunction, etc.)	VS 2015.2 [14]
P0024R2 Parallel Algorithms P0336R1 Renaming Parallel Execution Policies P0394R4 Parallel Algorithms Should terminate() For Exceptions P0452R1 Unifying <numeric> Parallel Algorithms	VS 2017 15.7
P0025R1 clamp()	VS 2015.3

FEATURE AREA	
P0030R1 hypot(x, y, z)	VS 2017 15.7
P0031R0 constexpr For <array> (Again) And <iterator>	VS 2017 15.3 [17]
P0032R3 Homogeneous Interface For variant/any/optional	VS 2017 15.0
P0033R1 Rewording enable_shared_from_this	VS 2017 15.5 [14]
P0040R3 Extending Memory Management Tools	VS 2017 15.3 [17]
P0063R3 C11 Standard Library	VS 2015 [C11, 14]
P0067R5 Elementary String Conversions	VS 2017 15.7 [charconv]
P0074R0 owner_less<>	VS 2015.2 [14]
P0077R2 is_callable, is_nothrow_callable	VS 2017 15.0
P0083R3 Splicing Maps And Sets P0508R0 Clarifying insert_return_type	VS 2017 15.5 [17]
P0084R2 Emplace Return Type	VS 2017 15.3 [17]
P0088R3 <variant>	VS 2017 15.0
P0092R1 <chrono> floor(), ceil(), round(), abs()	VS 2015.2 [14]
P0152R1 atomic::is_always_lock_free	VS 2017 15.3 [17]
P0154R1 hardware_destructive_interference_size, etc.	VS 2017 15.3 [17]
P0156R0 Variadic lock_guard	VS 2015.2 [14]
P0156R2 Renaming Variadic lock_guard to scoped_lock	VS 2017 15.3 [17]
P0163R0 shared_ptr::weak_type	VS 2017 15.0
P0174R2 Deprecating Vestigial Library Parts	VS 2017 15.5 [17]
P0185R1 is_swappable, is_nothrow_swappable	VS 2015.3
P0209R2 make_from_tuple()	VS 2017 15.0
P0218R1 <filesystem> P0219R1 Relative Paths For Filesystem P0317R1 Directory Entry Caching For Filesystem P0392R0 Supporting string_view In Filesystem Paths P0430R2 Supporting Non-POSIX Filesystems P0492R2 Resolving NB Comments for Filesystem	VS 2017 15.7 [E]
P0220R1 Library Fundamentals V1	VS 2017 15.6

FEATURE AREA	
P0226R1 Mathematical Special Functions	VS 2017 15.7
P0254R2 Integrating string_view And std::string	VS 2017 15.0
P0258R2 has_unique_object_representations	VS 2017 15.3 [G]
P0272R1 Non-const basic_string::data()	VS 2015.3
P0295R0 gcd(), lcm()	VS 2017 15.3 [17]
P0298R3 std::byte	VS 2017 15.3 [17, byte]
P0302R1 Removing Allocator Support In std::function	VS 2017 15.5 [17]
P0307R2 Making Optional Greater Equal Again	VS 2017 15.0
P0393R3 Making Variant Greater Equal	VS 2017 15.0
P0403R1 UDLs For <string_view> ("meow"sv, etc.)	VS 2017 15.3 [17]
P0414R2 shared_ptr<T[]>, shared_ptr<T[N]> P0497R0 Fixing shared_ptr For Arrays	VS 2017 15.5 [14]
P0418R2 atomic compare_exchange memory_order Requirements	VS 2017 15.3 [14]
P0426R1 constexpr For char_traits	VS 2017 15.7
P0433R2 Integrating template deduction for class templates into the standard library P0739R0 Improving class template argument deduction integration into the standard library	VS 2017 15.7
P0435R1 Overhauling common_type P0548R1 Tweaking common_type and duration	VS 2017 15.3 [14]
P0504R0 Revisiting in_place_t/in_place_type_t<T>/in_place_index_t<I>	VS 2017 15.0
P0505R0 constexpr For <chrono> (Again)	VS 2017 15.3 [17]
P0510R0 Rejecting variants Of Nothing, Arrays, References, And Incomplete Types	VS 2017 15.0
P0513R0 Poisoning hash P0599R1 noexcept hash	VS 2017 15.3 [14]
P0516R0 Marking shared_future Copying As noexcept	VS 2017 15.3 [14]
P0517R0 Constructing future_error From future_errc	VS 2017 15.3 [14]
P0521R0 Deprecating shared_ptr::unique()	VS 2017 15.5 [17]

FEATURE AREA	
P0558R1 Resolving atomic<T> Named Base Class Inconsistencies	VS 2017 15.3 [14]
P0595R2 std::is_constant_evaluated()	No
P0602R4 Propagating Copy/Move Triviality In variant/optional	VS 2017 15.3[17]
P0604R0 Changing is_callable/result_of To invoke_result, is_invocable, is_nothrow_invocable	VS 2017 15.3 [17]
P0607R0 Inline Variables for the Standard Library	VS 2017 15.5 [17]
P0618R0 Deprecating <codecvt>	VS 2017 15.5 [17]
P0682R1 Repairing Elementary String Conversions	VS 2015 15.7 [17]
C++14 Standard Library Features	**Supported**
N3462 SFINAE-Friendly result_of	VS 2015.2
N3302 constexpr For <complex>	VS 2015
N3469 constexpr For <chrono>	VS 2015
N3470 constexpr For <array>	VS 2015
N3471 constexpr For <initializer_list>, <tuple>, <utility>	VS 2015
N3545 integral_constant::operator()()	VS 2015
N3642 UDLs For <chrono>, <string> (1729ms, "meow"s, etc.)	VS 2015
N3644 Null Forward Iterators	VS 2015
N3654 quoted()	VS 2015
N3657 Heterogeneous Associative Lookup	VS 2015
N3658 integer_sequence	VS 2015
N3659 shared_mutex (Timed)	VS 2015
N3668 exchange()	VS 2015
N3669 Fixing constexpr Member Functions Without const	VS 2015
N3670 get<T>()	VS 2015
N3671 Dual-Range equal(), is_permutation(), mismatch()	VS 2015

FEATURE AREA	
N3778 Sized Deallocation	VS 2015
N3779 UDLs For <complex> (3.14i, etc.)	VS 2015
N3789 constexpr For <functional>	VS 2015
N3887 tuple_element_t	VS 2015
N3891 Renaming shared_mutex (Timed) To shared_timed_mutex	VS 2015
N3346 Minimal Container Element Requirements	VS 2013
N3421 Transparent Operator Functors (less<>, etc.)	VS 2013
N3655 Alias Templates For <type_traits> (decay_t, etc.)	VS 2013
N3656 make_unique()	VS 2013

A group of papers listed together indicates that a feature was voted into the Standard, and then one or more papers to improve or expand that feature were also voted in. These features are implemented together.

Supported values

No means not yet implemented.
Partial means the implementation is incomplete. For more details, see the Notes section.
VS 2010 indicates features that are supported in Visual Studio 2010.
VS 2013 indicates features that are supported in Visual Studio 2013.
VS 2015 indicates features that are supported in Visual Studio 2015 RTW.
VS 2015.2 and **VS 2015.3** indicate features that are supported in Visual Studio 2015 Update 2 and Visual Studio 2015 Update 3, respectively.
VS 2017 15.0 indicates features that are supported in Visual Studio 2017 version 15.0 (RTW).
VS 2017 15.3 indicates features that are supported in Visual Studio 2017 version 15.3.
VS 2017 15.5 indicates features that are supported in Visual Studio 2017 version 15.5.
VS 2017 15.7 indicates features that are supported in Visual Studio 2017 version 15.7.
VS 2019 16.0 indicates features that are supported in Visual Studio 2019 version 16.0 (RTW).
VS 2019 16.1 indicates features that are supported in Visual Studio 2019 version 16.1.

Notes

A In /std:c++14 mode, dynamic exception specifications remain unimplemented, and `throw()` is still treated as a synonym for `__declspec(nothrow)`. In C++17, dynamic exception specifications were mostly removed by P0003R5, leaving one vestige: `throw()` is deprecated and required to behave as a synonym for `noexcept`. In /std:c++17 mode, MSVC now conforms to the Standard by giving `throw()` the same behavior as `noexcept`, i.e. enforcement via termination.

The compiler option /Zc:noexceptTypes requests our old behavior of `__declspec(nothrow)`. It's likely that `throw()` will be removed in C++20. To help with migrating code in response to these changes in the Standard and our implementation, new compiler warnings for exception specification issues have been added under /std:c++17 and /permissive-.

B Supported in /permissive- mode in Visual Studio 2017 version 15.7. see Two-phase name lookup support comes to MSVC for more information.

C The compiler's support for C99 Preprocessor rules is incomplete in Visual Studio 2017. Variadic macros are supported, but there are many bugs in the preprocessor's behavior. We are overhauling the preprocessor, and will experimentally ship those changes under the /permissive- mode soon.

D Supported under /std:c++14 with a suppressible warning, C4984.

E This is a completely new implementation, incompatible with the previous `std::experimental` version, necessitated by symlink support, bug fixes, and changes in standard-required behavior. Currently, including <filesystem> provides the new `std::filesystem` and the previous `std::experimental::filesystem`, and including <experimental/filesystem> provides only the old experimental implementation. The experimental implementation will be REMOVED in the next ABI-breaking release of the libraries.

G Supported by a compiler intrinsic.

14 These C++17/20 features are always enabled, even when /std:c++14 (the default) is specified. This is either because the feature was implemented before the introduction of the **/std** options, or because conditional implementation was undesirably complex.

17 These features are enabled by the /std:c++17 (or /std:c++latest) compiler option.

20 These features are enabled by the /std:c++latest compiler option. When the C++20 implementation is complete, a new **/std:c++20** compiler option will be added, under which these features will also be available.

byte `std::byte` is enabled by /std:c++17 (or /std:c++latest), but because it can conflict with the Windows SDK headers in some cases, it has a fine-grained opt-out macro. It can be disabled by defining `_HAS_STD_BYTE` as `0`.

C11 The Universal CRT implemented the parts of the C11 Standard Library that are required by C++17, with the exception of C99 `strftime()` E/O alternative conversion specifiers, C11 `fopen()` exclusive mode, and C11 `aligned_alloc()`. The latter is unlikely to be implemented, because C11 specified `aligned_alloc()` in a way that's incompatible with the Microsoft implementation of `free()`, namely, that `free()` must be able to handle highly aligned allocations.

rem Features removed when the /std:c++17 (or /std:c++latest) compiler option is specified. These features can be re-enabled to ease the transition to newer language modes by use of these macros: `_HAS_AUTO_PTR_ETC`, `_HAS_FUNCTION_ALLOCATOR_SUPPORT`, `_HAS_OLD_IOSTREAMS_MEMBERS`, and `_HAS_UNEXPECTED`.

charconv `from_chars()` and `to_chars()` are available for integers. The timeline for floating-point `from_chars()` and floating-point `to_chars()` is as follows:

- VS 2017 15.7: Integer `from_chars()` and `to_chars()`.
- VS 2017 15.8: Floating-point `from_chars()`.
- VS 2017 15.9: Floating-point `to_chars()` overloads for shortest decimal.
- VS 2019 16.0: Floating-point `to_chars()` overloads for shortest hex and precision hex.
- VS 2019 16.2: Floating-point `to_chars()` overloads for precision fixed and precision scientific.
- Not yet implemented: The floating-point `to_chars()` overload for precision general.

parallel C++17's parallel algorithms library is complete. This doesn't mean every algorithm is parallelized in every case; the most important algorithms have been parallelized and execution policy signatures are provided even where algorithms are not parallelized. Our implementation's central internal header, yvals_core.h, contains the following "Parallel Algorithms Notes": C++ allows an implementation to implement parallel algorithms as calls to the serial algorithms. This implementation parallelizes several common algorithm calls, but not all.

The following algorithms are parallelized:

- `adjacent_difference`, `adjacent_find`, `all_of`, `any_of`, `count`, `count_if`, `equal`, `exclusive_scan`, `find`, `find_end`, `find_first_of`, `find_if`, `find_if_not`, `for_each`, `for_each_n`, `inclusive_scan`, `is_heap`, `is_heap_until`, `is_partitioned`, `is_sorted`, `is_sorted_until`, `mismatch`, `none_of`, `partition`, `reduce`, `remove` , `remove_if` , `replace` , `replace_if` , `search` , `search_n` , `set_difference` , `set_intersection` , `sort` , `stable_sort` , `transform` , `transform_exclusive_scan` , `transform_inclusive_scan` , `transform_reduce`

The following are not presently parallelized:

- No apparent parallelism performance improvement on target hardware; all algorithms which merely copy or permute elements with no branches are typically memory bandwidth limited:
    - `copy` , `copy_n` , `fill` , `fill_n` , `move` , `reverse` , `reverse_copy` , `rotate` , `rotate_copy` , `shift_left` , `shift_right` , `swap_ranges`
- Confusion over user parallelism requirements exists; likely in the above category anyway:
    - `generate` , `generate_n`
- Effective parallelism suspected to be infeasible:
    - `partial_sort` , `partial_sort_copy`
- Not yet evaluated; parallelism may be implemented in a future release and is suspected to be beneficial:
    - `copy_if` , `includes` , `inplace_merge` , `lexicographical_compare` , `max_element` , `merge` , `min_element` , `minmax_element` , `nth_element` , `partition_copy` , `remove_copy` , `remove_copy_if` , `replace_copy` , `replace_copy_if` , `set_symmetric_difference` , `set_union` , `stable_partition` , `unique` , `unique_copy`

See also

C++ Language Reference
C++ Standard Library
C++ conformance improvements in Visual Studio
What's New for Visual C++ in Visual Studio
Visual C++ change history 2003 through 2015
Visual C++ What's New 2003 through 2015
C++ team blog

Supported Platforms (Visual C++)

5/23/2019 • 2 minutes to read • Edit Online

Apps built by using Visual Studio can be targeted to various platforms, as follows.

OPERATING SYSTEM	X86	X64	ARM	ARM64****
Windows XP	X*	X*		
Windows Server 2003	X*	X*		
Windows Vista	X	X		
Windows Server 2008	X	X		
Windows 7	X	X		
Windows Server 2012 R2	X	X		
Windows 8	X	X	X	
Windows 8.1	X	X	X	
Windows 10	X	X	X	X
Android **	X	X	X	X
iOS **	X	X	X	X
Linux ***	X	X	X	X

* You can use the Windows XP platform toolset included in Visual Studio 2017, Visual Studio 2015, Visual Studio 2013, and Visual Studio 2012 Update 1 or later to build Windows XP and Windows Server 2003 projects. For information on how to use this platform toolset, see Configuring Programs for Windows XP. For additional information on changing the platform toolset, see How to: Modify the Target Framework and Platform Toolset.

** You can install the **Mobile development with C++** workload in the installer for Visual Studio 2017 and later. In Visual Studio 2015 setup, choose the optional **Visual C++ for Cross Platform Mobile Development** component to target iOS or Android platforms. For instructions, see Install Visual C++ for Cross-Platform Mobile Development. To build iOS code, you must have a Mac computer and meet other requirements. For a list of prerequisites and installation instructions, see Install And Configure Tools to Build using iOS. You can build x86 or ARM code to match the target hardware. Use x86 configurations to build for the iOS simulator, Microsoft Visual Studio Emulator for Android, and some Android devices. Use ARM configurations to build for iOS devices and most Android devices.

*** You can install the **Linux development with C++** workload in the installer for Visual Studio 2017 and later to target Linux platforms. For instructions, see Download, Install and Setup the Linux Workload. This toolset compiles your executable on the target machine, so you can build for any supported architecture.

**** ARM64 support is available in Visual Studio 2017 and later.

For information about how to set the target platform configuration, see How to: Configure Visual C++ Projects to Target 64-Bit, x64 Platforms.

See also

- Visual C++ Tools and Features in Visual Studio Editions
- Getting Started

C++ Tools and Features in Visual Studio Editions

5/23/2019 • 9 minutes to read • Edit Online

The following C++ features are available in Visual Studio 2019. Unless stated otherwise, all features are available in all editions: Visual Studio Community, Visual Studio Professional, and Visual Studio Enterprise. Some features require specific workloads or optional components, which you can install with the Visual Studio Installer.

Platforms

- Windows Desktop
- Universal Windows Platform ((tablet, PC, Xbox, IoT, and HoloLens))
- Linux
- Android
- iOS

Compilers

- MSVC 32-bit compiler for x86, x64, ARM, and ARM64
- MSVC 64-bit compiler for x86, x64, ARM, and ARM64
- GCC cross-compiler for ARM
- Clang/LLVM
  - On Windows, Clang/LLVM 7.0, targeting x86 or x64 (CMake support only). Other Clang versions might work but are not officially supported.
  - On Linux, any Clang/LLVM installation supported by the distro.

C++ Workloads

Visual Studio includes the following workloads for C++ development. You can install any or all of these, along with other workloads such as .NET Desktop Development, Python Development, Azure Development, Visual Studio Extension Development, and others.

Desktop development with C++

Included:

- C++ core desktop features

Optional Components:

- MSVC v142 - VS 2019 C++ x64/x86 build tools (v14.21)
- Windows 10 SDK (10.0.17763.0)
- Just-In-Time debugger
- C++ profiling tools
- C++ CMake tools for Windows
- C++ ATL for v142 build tools (x86 & x64)
- Test Adapter for Boost.Test
- Test Adapter for Google Test
- Live Share
- IntelliCode

- IntelliTrace (Enterprise only)
- C++ MFC for v142 build tools (x86 & x64)
- C++/CLI support for v142 build tools (14.21)
- C++ Modules for v142 build tools (x64/x86 – experimental)
- Clang compiler for Windows
- IncrediBuild - Build Acceleration
- Windows 10 SDK (10.0.17134.0)
- Windows 10 SDK (10.0.16299.0)
- MSVC v141 - VS 2017 C++ x64/x86 build tools (v14.16)
- MSVC v140 - VS 2015 C++ build tools (v14.00)

Linux development with C++

Included:

- C++ core features
- Windows Universal C Runtime
- C++ for Linux Development

Optional Components:

- C++ CMake tools for Linux
- Embedded and IoT development tools

Universal Windows Platform development

Included:

- Blend for Visual Studio
- .NET Native and .NET Standard
- NuGet package manager
- Universal Windows Platform tools
- Windows 10 SDK (10.0.17763.0)

Optional Components:

- IntelliCode
- IntelliTrace (Enterprise only)
- USB Device Connectivity
- C++ (v142) Universal Windows Platform tools
- C++ (v141) Universal Windows Platform tools
- Graphics debugger and GPU profiler for DirectX
- Windows 10 SDK (10.0.18362.0)
- Windows 10 SDK (10.0.17134.0)
- Windows 10 SDK (10.0.16299.0)
- Architecture and analysis tools

C++ Game Development

Included:

- C++ core features
- Windows Universal C Runtime
- C++ 2019 Redistributable Update
- MSVC v142 - VS 2019 C++ x64/x86 build tools (v14.21)

Optional Components:

- C++ profiling tools
- Windows 10 SDK (10.0.17763.0)
- IntelliCode
- IntelliTrace (Enterprise only)
- Windows 10 SDK (10.0.17134.0)
- Windows 10 SDK (10.0.16299.0)
- IncrediBuild - Build Acceleration
- Cocos
- Unreal Engine installer
- Android IDE support for Unreal engine

Mobile development with C++

Included:

- C++ core features
- Android SDK setup (API level 25) (local install for Mobile development with C++)

Optional Components:

- Android NDK (R16B)
- Apache Ant (1.9.3)
- C++ Android development tools
- IntelliCode
- Google Android Emulator (API Level 25) (local install)
- Intel Hardware Accelerated Execution Manager (HAXM) (local install)
- Android NDK (R16B) (32bit)
- C++ iOS development tools
- IncrediBuild - Build Acceleration

Individual components

You can install these components independently from any workload.

- JavaScript diagnostics
- Live Share
- C++ Universal Windows Platform runtime for v142 build tools
- ClickOnce Publishing
- Microsoft Visual Studio Installer Projects

Libraries and Headers

- Windows headers and libraries
- Windows Universal C Runtime (CRT)
- C++ Standard Library
- ATL
- MFC
- .NET Framework class library
- C++ Support Library for .NET
- OpenMP 2.0

- Over 900 open-source libraries via vcpkg catalog

Build and Project Systems

- CMake
- Any build system via Open Folder
- Command line builds (msbuild.exe)
- Native Multi-targeting
- Managed Multi-targeting
- Parallel Builds
- Build Customizations
- Property Pages Extensibility

Project Templates

The following project templates are available depending on which workloads you have installed.

Windows Desktop:

- Empty Project
- Console App
- Windows Desktop Wizard
- Windows Desktop Application
- Shared Items Project
- MFC App
- Dynamic Link Library
- CLR Empty Project
- CLR Console App
- Static Library
- CMake Project
- ATL Project
- MFC Dynamic Link Library
- CLR Class Library
- Makefile Project (Windows)
- MFC ActiveXControl
- Native Unit Test Project
- Google Test

Universal Windows Platform (C++/CX):

- Blank App
- DirectX 11 and XAML App
- DirectX 11 App
- DirectX 12 App
- Unit Test App
- DLL
- Windows Runtime Component
- Static Library
- Windows Application Packaging Project

Linux:

- Console App (Linux)
- Empty Project (Linux)
- Raspberry Pi Blink
- Makefile Project (Linux)

Tools

- Incremental Linker (Link.exe)
- Microsoft Makefile Utility (Nmake.exe)
- Lib Generator (Lib.exe)
- Windows Resource Compiler (Rc.exe)
- Windows Resource to Object Converter (CvtRes.exe)
- Browse Information Maintenance Utility (BscMake.exe)
- C++ Name Undecorator (Undname.exe)
- COFF/PE Dumper (Dumpbin.exe)
- COFF/PE Editor (Editbin.exe)
- MASM (Ml.exe)
- Spy++
- ErrLook
- AtlTrace
- Inference Rules
- Profile Guided Optimizations

Debugging Features

- Native Debugging
- natvis (native type visualization)
- Graphics Debugging
- Managed Debugging
- GPU usage
- Memory usage
- Remote Debugging
- SQL Debugging
- Static Code Analysis

Designers and Editors

- XAML Designer
- CSS Style Designer/Editor
- HTML Designer/Editor
- XML Editor
- Source Code Editor
- Productivity Features: Refactoring, EDG IntelliSense engine, C++ Code Formatting
- Windows Forms Designer
- Data Designer
- Native Resource Editor (.rc files)
- Resource Editors
- Model editor

- Shader designer
- Live Dependency Validation (Enterprise Only)
- Architectural Layer Diagrams (Enterprise Only)
- Architecture Validation (Enterprise Only)
- Code Clone (Enterprise Only)

Data Features

- Data Designer
- Data Objects
- Web Services
- Server Explorer

Automation and Extensibility

- Extensibility Object Models
- Code Model
- Project Model
- Resource Editor Model
- Wizard Model
- Debugger Object Model

Application Lifecycle Management Tools

- Unit Testing (Microsoft Native C++, Boost.Test, Google Test, CTest)
- Code map and dependency graphs (Professional and Enterprise)
- Code coverage (Enterprise Only)
- Manual testing (Enterprise only)
- Exploratory testing (Enterprise only)
- Test case management (Enterprise only)
- Code map debugger integration (Enterprise only)
- Live Unit Testing (Enterprise only)
- IntelliTrace (Enterprise only)
- IntelliTest (Enterprise only)
- Microsoft Fakes (Unit Test Isolation) (Enterprise only)
- Code Coverage (Enterprise only)

See also

Install Visual Studio
What's New in Visual Studio
C++ project types in Visual Studio

The following tables show Visual C++ features that are available in Visual Studio 2017. An X in a cell indicates that the feature is available; an empty cell indicates that the feature is not available. Notes in parentheses indicate that a feature is available, but restricted.

Platforms

Platform	Visual Studio Express for Windows 10	Visual Studio Express for Windows Desktop	Visual Studio Community/Professional	Visual Studio Enterprise
Windows Desktop		X	X	X
Universal Windows Platform ((phone, tablet, PC, Xbox, IoT, and HoloLens))	X		X	X
Linux	X	X		
Microsoft Store 8.1			X	X
Windows Phone 8.0			X	X
Android			X	X
iOS			X	X

Compilers

COMPILER	VISUAL STUDIO EXPRESS FOR WINDOWS	VISUAL STUDIO EXPRESS FOR WINDOWS DESKTOP	VISUAL STUDIO PROFESSIONAL / COMMUNITY	VISUAL STUDIO ENTERPRISE
MSVC 32-bit X86 compiler	X	X	X	X
X86_arm cross-compiler	X		X	X
MSVC 64-bit x64 compiler			X	X
X86_ x64 cross-compiler	X	X	X	X

Libraries and Headers

LIBRARY OR HEADER	VISUAL STUDIO EXPRESS FOR WINDOWS	VISUAL STUDIO EXPRESS FOR WINDOWS DESKTOP	VISUAL STUDIO PROFESSIONAL / COMMUNITY	VISUAL STUDIO ENTERPRISE
Windows headers and libraries and CRT library	(X)	X	X	X
C++ Standard Library	X	X	X	X
ATL			X	X

LIBRARY OR HEADER	VISUAL STUDIO EXPRESS FOR WINDOWS	VISUAL STUDIO EXPRESS FOR WINDOWS DESKTOP	VISUAL STUDIO PROFESSIONAL / COMMUNITY	VISUAL STUDIO ENTERPRISE
MFC			X	X
.NET Framework class library		X	X	X
C++ Support Library for .NET		X	X	X
OpenMP 2.0	X	X	X	X

Project Templates

TEMPLATE	VISUAL STUDIO EXPRESS FOR WINDOWS	VISUAL STUDIO EXPRESS FOR WINDOWS DESKTOP	VISUAL STUDIO PROFESSIONAL / COMMUNITY	VISUAL STUDIO ENTERPRISE
XAML Templates for UWP, Windows 8.1, Windows Phone 8.0	X		X	X
Direct3D App	X		X	X
DLL (Universal Windows)	X		X	X
Static Library (Universal Windows)	X		X	X
Windows Runtime Component	X		X	X
Unit Test App (Universal Windows)	X		X	X
ATL Project			X	X
Class Library (CLR)		X	X	X
CLR Console Application		X	X	X
CLR Empty Project		X	X	X
Custom Wizard			X	X
Empty Project		X	X	X
Makefile Project		X	X	X
MFC ActiveX Control			X	X

TEMPLATE	VISUAL STUDIO EXPRESS FOR WINDOWS	VISUAL STUDIO EXPRESS FOR WINDOWS DESKTOP	VISUAL STUDIO PROFESSIONAL / COMMUNITY	VISUAL STUDIO ENTERPRISE
MFC Application			X	X
MFC DLL			X	X
Test Project	X	X	X	X
Win32 Console Application		X	X	X
Win32 Project		X	X	X

Tools

TOOL	VISUAL STUDIO EXPRESS FOR WINDOWS	VISUAL STUDIO EXPRESS FOR WINDOWS DESKTOP	VISUAL STUDIO PROFESSIONAL / COMMUNITY	VISUAL STUDIO ENTERPRISE
Incremental Linker (Link.exe)	X	X	X	X
Program Maintenance Utility (Nmake.exe)		X	X	X
Lib Generator (Lib.exe)	X	X	X	X
Windows Resource Compiler (Rc.exe)	X	X	X	X
Windows Resource to Object Converter (CvtRes.exe)		X	X	X
Browse Information Maintenance Utility (BscMake.exe)	X	X	X	X
C++ Name Undecorator (Undname.exe)	X	X	X	X
COFF/PE Dumper (Dumpbin.exe)	X	X	X	X
COFF/PE Editor (Editbin.exe)	X	X	X	X
MASM (Ml.exe)			X	X
Spy++			X	X

TOOL	VISUAL STUDIO EXPRESS FOR WINDOWS	VISUAL STUDIO EXPRESS FOR WINDOWS DESKTOP	VISUAL STUDIO PROFESSIONAL / COMMUNITY	VISUAL STUDIO ENTERPRISE
ErrLook			X	X
AtlTrace			X	X
Devenv.com			X	X
Inference Rules			X	X
Upgrade VCBuild .vcproj projects to MSBuild (VCUpgrade.exe)	X	X	X	X
Profile Guided Optimizations			X	X

Debugging Features

DEBUGGING FEATURE	VISUAL STUDIO EXPRESS FOR WINDOWS	VISUAL STUDIO EXPRESS FOR WINDOWS DESKTOP	VISUAL STUDIO PROFESSIONAL / COMMUNITY	VISUAL STUDIO ENTERPRISE
Native Debugging	X	X	X	X
natvis (native type visualization)	X	X	X	X
Graphics Debugging	X		X	X
Managed Debugging		X	X	X
GPU usage	X		X	X
Memory usage	X		X	X
Remote Debugging	X	X	X	X
SQL Debugging			X	X
Static Code Analysis	Limited	Limited	X	X

Designers and Editors

DESIGNER OR EDITOR	VISUAL STUDIO EXPRESS FOR WINDOWS	VISUAL STUDIO EXPRESS FOR WINDOWS DESKTOP	VISUAL STUDIO PROFESSIONAL / COMMUNITY	VISUAL STUDIO ENTERPRISE
XAML Designer	X		X	X

DESIGNER OR EDITOR	VISUAL STUDIO EXPRESS FOR WINDOWS	VISUAL STUDIO EXPRESS FOR WINDOWS DESKTOP	VISUAL STUDIO PROFESSIONAL / COMMUNITY	VISUAL STUDIO ENTERPRISE
CSS Style Designer/Editor	X	X	X	X
HTML Designer/Editor	X	X	X	X
XML Editor	X	X	X	X
Source Code Editor	X	X	X	X
Productivity Features: Refactoring, IntelliSense, C++ Code Formatting	X	X	X	X
Windows Forms Designer		X	X	X
Data Designer			X	X
Native Resource Editor (.rc files)			X	X
Resource Editors	X	X	X	X
Model editor	X		X	X
Shader designer	X		X	X

Data Features

DATA FEATURE	VISUAL STUDIO EXPRESS FOR WINDOWS	VISUAL STUDIO EXPRESS FOR WINDOWS DESKTOP	VISUAL STUDIO PROFESSIONAL / COMMUNITY	VISUAL STUDIO ENTERPRISE
Data Designer			X	X
Data Objects			X	X
Web Services			X	X
Server Explorer			X	X

Build and Project Systems

BUILD OR PROJECT FEATURE	VISUAL STUDIO EXPRESS FOR WINDOWS	VISUAL STUDIO EXPRESS FOR WINDOWS DESKTOP	VISUAL STUDIO PROFESSIONAL / COMMUNITY	VISUAL STUDIO ENTERPRISE
Command line builds (msbuild.exe)	X	X	X	X
Native Multi-targeting		X	X	X
Managed Multi-targeting		X	X	X
Parallel Builds	X	X	X	X
Build Customizations	X	X	X	X
Property Pages Extensibility	X	X	X	X

Automation and Extensibility

AUTOMATION AND EXTENSIBILITY	VISUAL STUDIO EXPRESS FOR WINDOWS	VISUAL STUDIO EXPRESS FOR WINDOWS DESKTOP	VISUAL STUDIO PROFESSIONAL / COMMUNITY	VISUAL STUDIO ENTERPRISE
Extensibility Object Models			X	X
Code Model			X	X
Project Model			X	X
Resource Editor Model			X	X
Wizard Model			X	X
Debugger Object Model			X	X

Application Lifecycle Management Tools

Tool	Visual Studio Express for Windows	Visual Studio Express for Windows Desktop	Visual Studio Professional / Community	Visual Studio Enterprise
Unit Testing (native framework)	X	X	X	X
Unit Testing (managed framework)		X	X	X

Code coverage				X
Manual testing				X
Exploratory testing				X
Test case management				X
Code map and dependency graphs			read-only	X
Code map debugging				X

See also

Install Visual Studio
What's New in Visual Studio
C++ project types in Visual Studio

Visual C++ Samples

4/1/2019 • 2 minutes to read • Edit Online

The Visual C++ samples listed below demonstrate different functionalities across multiple technologies.

Visual C++ samples

Visual Studio samples

Windows on GitHub

Universal Windows app samples

The All-In-One code framework

Windows Desktop code samples

MFC samples

CodePlex samples

ADO code samples

Windows Hardware development samples

IMPORTANT

This sample code is intended to illustrate a concept, and it shows only the code that is relevant to that concept. It may not meet the security requirements for a specific environment, and it should not be used exactly as shown. We recommend that you add security and error-handling code to make your projects more secure and robust. Microsoft provides this sample code "AS IS" with no warranties.

Visual C++ Help and Community

5/15/2019 • 2 minutes to read • Edit Online

Here's how to getting information about how to write C++ code and use the Visual Studio development tools.

Samples

TITLE	DESCRIPTION
Developer Code Samples	Contains downloadable sample code from Microsoft and community contributors.

Product Documentation

TITLE	DESCRIPTION
C++ in Visual Studio	Contains reference and conceptual documentation about Visual C++. Part of the MSDN Library.
Windows Developer Center	Contains information about how to use C++ and other languages to develop apps for Windows 10. Part of the Windows Developer Center; the C++ content is under the Docs > Language Reference node.

Online and Offline Documentation

You can view Microsoft developer content online. This content is updated regularly.

You can also download and view the content locally in the offline Help Viewer. The offline documentation is organized by books of related content, which are also updated periodically. You can download the books you are interested in as they become available. For more information, see Microsoft Help Viewer.

Many sections of the documentation are also available in PDF form. These sections have a **Download PDF** link on included pages on docs.microsoft.com.

Related Articles

TITLE	DESCRIPTION
Visual C++ Team Blog	Contains posts on various subjects by the experts on the Visual C++ product team.
Channel 9	Contains video interviews and lectures. Use the search box on the Channel 9 home page to find C++ content.
Visual Studio	Contains articles and news about Visual Studio and related development tools.
MSDN forums and Developer Community	Official Microsoft forums where you can post questions about C++ and get answers from Microsoft and from experts in the community.

How to report a problem with the Visual C++ toolset or documentation

5/8/2019 • 24 minutes to read • Edit Online

If you encounter problems with the Microsoft C++ compiler, linker, or other tools and libraries, we want to know about them. If the issue is in our documentation, we want to know about that, too.

How to report a C++ toolset issue

The best way to let us know about a problem is to send us a report that includes a description of the problem you've encountered, details about how you're building your program, and a *repro*, a complete test case we can use to reproduce the problem on our own machines. This information lets us quickly verify that the problem exists in our code and is not local to your environment, to determine whether it affects other versions of the compiler, and to diagnose its cause.

In the sections below, you'll read about what makes a good report, how to generate a repro for the kind of issue you've found, and how to send your report to the product team. Your reports are important to us and to other developers like you. Thank you for helping us improve Visual C++!

How to prepare your report

Creating a high-quality report is important because it is very difficult to reproduce the problem you encountered on our own machines without complete information. The better your report is, the more effectively we are able recreate and diagnose the problem.

At a minimum, your report should contain

- The full version information of the toolset you're using.
- The full cl.exe command line used to build your code.
- A detailed description of the problem you encountered.
- A repro: a complete, simplified, self-contained source code example that demonstrates the problem.

Read on to learn more about the specific information we need and where you can find it, and how to create a good repro.

The toolset version

We need the full version information and the target architecture of the toolset that causes the problem so that we can test your repro against the same toolset on our machines. If we can reproduce the problem, this information also gives us a starting point to investigate which other versions of the toolset exhibit the same problem.

To report the full version of the compiler you're using

1. Open the **Developer Command Prompt** that matches the Visual Studio version and configuration architecture used to build your project. For example, if you build by using Visual Studio 2017 on x64 for x64 targets, choose **x64 Native Tools Command Prompt for VS 2017**. For more information, see Developer command prompt shortcuts.
2. In the developer command prompt console window, enter the command **cl /Bv**.

The output should look similar to this:

```
C:\Users\username\Source>cl /Bv
Microsoft (R) C/C++ Optimizing Compiler Version 19.14.26428.1 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

Compiler Passes:
 C:\Program Files (x86)\Microsoft Visual
Studio\2017\Enterprise\VC\Tools\MSVC\14.14.26428\bin\HostX86\x86\cl.exe:        Version 19.14.26428.1
 C:\Program Files (x86)\Microsoft Visual
Studio\2017\Enterprise\VC\Tools\MSVC\14.14.26428\bin\HostX86\x86\c1.dll:        Version 19.14.26428.1
 C:\Program Files (x86)\Microsoft Visual
Studio\2017\Enterprise\VC\Tools\MSVC\14.14.26428\bin\HostX86\x86\c1xx.dll:      Version 19.14.26428.1
 C:\Program Files (x86)\Microsoft Visual
Studio\2017\Enterprise\VC\Tools\MSVC\14.14.26428\bin\HostX86\x86\c2.dll:        Version 19.14.26428.1
 C:\Program Files (x86)\Microsoft Visual
Studio\2017\Enterprise\VC\Tools\MSVC\14.14.26428\bin\HostX86\x86\link.exe:      Version 14.14.26428.1
 C:\Program Files (x86)\Microsoft Visual
Studio\2017\Enterprise\VC\Tools\MSVC\14.14.26428\bin\HostX86\x86\mspdb140.dll:  Version 14.14.26428.1
 C:\Program Files (x86)\Microsoft Visual
Studio\2017\Enterprise\VC\Tools\MSVC\14.14.26428\bin\HostX86\x86\1033\clui.dll: Version 19.14.26428.1

cl : Command line error D8003 : missing source filename
```

Copy and paste the entire output into your report.

The command line

We need the exact command line (cl.exe and all of its arguments) used to build your code, so that we can build it in exactly the same way on our machines. This is important because the problem you've encountered might only exist when building with a certain argument or combination of arguments.

The best place to find this information is in the build log immediately after experiencing the problem. This ensures that the command line contains exactly the same arguments that might be contributing to the problem.

To report the contents of the command line

1. Locate the **CL.command.1.tlog** file and open it. By default, this file is located in your Documents folder in \Visual Studio *version*\Projects*SolutionName**ProjectName**Configuration**ProjectName*.tlog\CL.command.1.tlog, or in your User folder under \Source\Repos\*SolutionName**ProjectName**Configuration**ProjectName*.tlog\CL.command.1.tlog. It may be in a different location if you use another build system or if you have changed the default location for your project.

   Inside this file, you'll find the names of source code files followed by the command line arguments used to compile them, each on separate lines.

2. Locate the line that contains the name of the source code file where the problem occurs; the line below it contains the corresponding cl.exe command arguments.

Copy and paste the entire command line into your report.

A description of the problem

We need a detailed description of the problem you've encountered so that we can verify that we see the same effect on our machines; its also sometimes useful for us to know what you were trying to accomplish, and what you expected to happen.

Please provide the **exact error messages** given by the toolset, or the exact runtime behavior you see. We need this information to verify that we've properly reproduced the issue. Please include **all** of the compiler output, not just the last error message. We need to see everything that led up to the issue you report. If you can duplicate the issue by using the command line compiler, that compiler output is preferred; the IDE and other build systems may filter the error messages you see, or only capture the first line of an error message.

If the issue is that the compiler accepts invalid code and does not generate a diagnostic, please note this in your report.

To report a runtime behavior problem, include an **exact copy** of what the program prints out, and what you expect to see. Ideally, this is embedded in the output statement itself, for example, `printf("This should be 5: %d\n", actual_result);`. If your program crashes or hangs, mention that as well.

Add any other details that might help us diagnose the problem you experienced, such as any work-arounds you may have found. Avoid repeating information found elsewhere in your report.

The repro

A repro is a complete, self-contained source code example that reproducibly demonstrates the problem you've encountered (hence the name). We need a repro so that we can reproduce the error on our machines. The code should be sufficient by itself to create a simple executable that compiles and runs, or that would compile and run if not for the problem you've found. A repro is not a code snippet; it should have complete functions and classes and contain all the necessary #include directives, even for the standard headers.

What makes a good repro

A good repro is:

- **Minimal.** Repros should be as small as possible yet still demonstrate exactly the problem you encountered. Repros do not need to be complex or realistic; they only need to show code that conforms to the Standard or the documented compiler implementation, or in the case of a missing diagnostic, the code that is not conformant. Simple, to-the-point repros that contain just enough code to demonstrate the problem are best. If you can eliminate or simplify the code and remain conformant and also leave the issue unchanged, please do so. You do not need to include counter-examples of code that works.

- **Self-Contained.** Repros should avoid unnecessary dependencies. If you can reproduce the problem without third-party libraries, please do so. If you can reproduce the problem without any library code besides simple output statements (for example, `puts("this shouldn't compile");`, `std::cout << value;`, and `printf("%d\n", value);` are okay), please do so. It's ideal if the example can be condensed to a single source code file, without reference to any user headers. Reducing the amount of code we have to consider as a possible contributor to the problem is enormously helpful to us.

- **Against the latest compiler version.** Repros should use the most recent update to the latest version of the toolset, or the most recent prerelease version of the next update or next major release, whenever possible. Problems you may encounter in older versions of the toolset have very often been fixed in newer versions. Fixes are backported to older versions only in exceptional circumstances.

- **Checked against other compilers** if relevant. Repros that involve portable C++ code should verify behavior against other compilers if possible. The Standard ultimately determines program correctness, and no compiler is perfect, but when Clang and GCC accept your code without a diagnostic and MSVC does not, it's likely you're looking at a bug in our compiler. (Other possibilities include differences in Unix and Windows behavior, or different levels of C++ standards implementation, and so on.) On the other hand, if all the compilers reject your code, then it's likely that your code is incorrect. Seeing different error messages may help you diagnose the issue yourself.

  You can find lists of online compilers to test your code against in Online C++ compilers on the ISO C++ website, or this curated List of Online C++ Compilers on GitHub. Some specific examples include Wandbox, Compiler Explorer, and Coliru.

NOTE

The online compiler websites are not affiliated with Microsoft. Many online compiler websites are run as personal projects, and some of these sites may not be available when you read this, but a search should find others you can use.

Problems in the compiler, linker, and in the libraries, tend to show themselves in particular ways. The kind of problem you encounter will determine what kind of repro you should include in your report. Without an appropriate repro, we have nothing to investigate. Here are a few of the kinds of issues that you may see, and instructions for generating the kinds of repros you should use to report each kind of problems.

Frontend (parser) crash

Frontend crashes occur during the parsing phase of the compiler. Typically, the compiler will emit Fatal Error C1001 and reference the source code file and line number on which the error occurred; it will often mention a file msc1.cpp, but you can ignore this detail.

For this kind of crash, please provide a Preprocessed Repro.

Here's example compiler output for this kind of crash:

```
SandBoxHost.cpp
d:\o\dev\search\foundation\common\tools\sandbox\managed\managed.h(929):
        fatal error C1001: An internal error has occurred in the compiler.
(compiler file 'msc1.cpp', line 1369)
To work around this problem, try simplifying or changing the program near the
        locations listed above.
Please choose the Technical Support command on the Visual C++
Help menu, or open the Technical Support help file for more information
d:\o\dev\search\foundation\common\tools\sandbox\managed\managed.h(929):
        note: This diagnostic occurred in the compiler generated function
        'void Microsoft::Ceres::Common::Tools::Sandbox::SandBoxedProcess::Dispose(bool)'
Internal Compiler Error in d:\o\dev\otools\bin\x64\cl.exe.  You will be prompted
        to send an error report to Microsoft later.
INTERNAL COMPILER ERROR in 'd:\o\dev\otools\bin\x64\cl.exe'
    Please choose the Technical Support command on the Visual C++
    Help menu, or open the Technical Support help file for more information
```

Backend (code generation) crash

Backend crashes occur during the code generation phase of the compiler. Typically, the compiler will emit Fatal Error C1001, and might not reference the source code file and line number associated with the problem; it will often mention the file compiler\utc\src\p2\main.c, but you can ignore this detail.

For this kind of crash, please provide a Link repro if you are using Link-Time Code Generation (LTCG), enabled by the **/GL** command-line argument to cl.exe. If not, please provide a Preprocessed repro instead.

Here's example compiler output for a backend crash in which LTCG is not used. If your compiler output looks like this you should provide a Preprocessed Repro.

```
repro.cpp
\\officefile\public\tadg\vc14\comperror\repro.cpp(13) : fatal error C1001:
        An internal error has occurred in the compiler.
(compiler file 'f:\dd\vctools\compiler\utc\src\p2\main.c', line 230)
To work around this problem, try simplifying or changing the program near the
        locations listed above.
Please choose the Technical Support command on the Visual C++
Help menu, or open the Technical Support help file for more information
INTERNAL COMPILER ERROR in
        'C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\BIN\cl.exe'
    Please choose the Technical Support command on the Visual C++
    Help menu, or open the Technical Support help file for more information
```

If the line that begins with **INTERNAL COMPILER ERROR** mentions link.exe, rather than cl.exe, LTCG was enabled and you should provide a Link Repro. If its not clear whether LTCG was enabled from the compiler error message, you may need to examine the command line arguments that you copied from your build log in a previous step for the **/GL** command-line argument.

Linker crash

Linker crashes occur during the linking phase, after the compiler has run. Typically, the linker will emit Linker Tools Error LNK1000.

NOTE

If the output mentions C1001 or involves Link-Time Code Generation, refer to Backend (code generation) crash instead for more information.

For this kind of crash, please provide a Link repro.

Here's example compiler output for this kind of crash.

```
z:\foo.obj : error LNK1000: Internal error during IMAGE::Pass2

  Version 14.00.22816.0

  ExceptionCode            = C0000005
  ExceptionFlags           = 00000000
  ExceptionAddress         = 00007FF73C9ED0E6 (00007FF73C9E0000)
        "z:\tools\bin\x64\link.exe"
  NumberParameters         = 00000002
  ExceptionInformation[ 0] = 0000000000000000
  ExceptionInformation[ 1] = FFFFFFFFFFFFFFFF

CONTEXT:

  Rax    = 0000000000000400  R8     = 0000000000000000
  Rbx    = 000000655DF82580  R9     = 00007FF840D2E490
  Rcx    = 005C006B006F006F  R10    = 000000655F97E690
  Rdx    = 000000655F97E270  R11    = 0000000000000400
  Rsp    = 000000655F97E248  R12    = 0000000000000000
  Rbp    = 000000655F97EFB0  E13    = 0000000000000000
  Rsi    = 000000655DF82580  R14    = 000000655F97F390
  Rdi    = 0000000000000000  R15    = 0000000000000000
  Rip    = 00007FF73C9ED0E6  EFlags = 0000000000010206
  SegCs  = 0000000000000033  SegDs  = 000000000000002B
  SegSs  = 000000000000002B  SegEs  = 000000000000002B
  SegFs  = 0000000000000053  SegGs  = 000000000000002B
  Dr0    = 0000000000000000  Dr3    = 0000000000000000
  Dr1    = 0000000000000000  Dr6    = 0000000000000000
  Dr2    = 0000000000000000  Dr7    = 0000000000000000
```

If incremental linking is enabled and the crash occurred only after a successful initial link, (that is, only after the first full linking on which subsequent incremental linking is based) please also provide a copy of the object (.obj) and library (.lib) files that correspond to source files that were modified after the initial link was completed.

Bad code generation

Bad code generation is rare, but occurs when the compiler mistakenly generates incorrect code that will cause your application to crash at runtime rather than detecting this problem at compile-time. If you believe the problem you are experiencing results in bad code generation, treat your report the same as a Backend (code generation) crash.

For this kind of crash please provide a Link repro if you are using Link-Time Code Generation (LTCG), enabled by the **/GL** command-line argument to cl.exe. Please provide a Preprocessed repro if not.

How to generate a repro

To help us track down the source of the problem, a good repro is vital. Before you do any of the steps outlined below for specific kinds of repros, try to condense the code that demonstrates the problem as much as possible. Try to eliminate or minimize dependencies, required headers, and libraries, and limit the compiler options and preprocessor definitions used if possible.

Below are instructions for generating the various kinds of repros you'll use to report different kinds of problems.

Preprocessed repros

A *preprocessed repro* is a single source file that demonstrates a problem, generated from the output of the C preprocessor by using the **/P** compiler option on the original repro source file. This inlines included headers to remove dependencies on additional source and header files, and also resolves macros, #ifdefs, and other preprocessor commands that could depend your local environment.

> **NOTE**
>
> Preprocessed repros are not as useful for problems that might be the result of bugs in our standard library implementation, because we will often want to substitute our latest, in-progress implementation to see whether we've already fixed the problem. In this case, don't preprocess the repro, and if you can't reduce the problem to a single source file, package your code into a .zip file or similar, or consider using an IDE project repro. For more information, see Other repros.

To preprocess a source code file

1. Capture the command line arguments used to build your repro, as described in To report the contents of the command line.

2. Open the **Developer Command Prompt** that matches the Visual Studio version and configuration architecture used to build your project.

3. Change to the directory that contains your repro project.

4. In the developer command prompt console window, enter the command **cl /P** *arguments filename.cpp*, where *arguments* is the list of arguments captured above, and *filename.cpp* is the name of your repro source file. This command replicates the command line used for the repro, but stops the compilation after the preprocessor pass, and outputs the preprocessed source code to *filename*.i.

If you are preprocessing a C++/CX source code file, or you are using the C++ Modules feature, some additional steps are required. For more information, see the sections below.

After you have generated the preprocessed file, its a good idea to make sure that the problem still repros using the preprocessed file.

To confirm that the error still repros with the preprocessed file

1. In the developer command prompt console window, enter the command **cl** *arguments* **/TP** *filename*.i to tell cl.exe to compile the preprocessed file as a C++ source file, where *arguments* is the list of arguments

captured above, but with any **/D** and **/I** arguments removed (because they have already been included in the preprocessed file); and where *filename*.i is the name of your preprocessed file.

2. Confirm that the problem is reproduced.

Finally, attach the preprocessed repro *filename*.i to your report.

Preprocessed C++/CX WinRT/UWP code repros

If you're using C++/CX to build your executable, there are some extra steps required to create and validate a preprocessed repro.

To preprocess C++/CX source code

1. Create a preprocessed source file as described in To preprocess a source code file.

2. Search the generated *filename*.i file for **#using** directives.

3. Make a list of all of the referenced files. Leave out any Windows*.winmd files, platform.winmd files, and mscorlib.dll.

To prepare to validate that the preprocessed file still reproduces the problem,

1. Create a new directory for the preprocessed file and copy it to the new directory.

2. Copy the .winmd files from your **#using** list to the new directory.

3. Create an empty vccorlib.h file in the new directory.

4. Edit the preprocessed file to remove any **#using** directives for mscorlib.dll.

5. Edit the preprocessed file to change any absolute paths to just the bare filenames for the copied .winmd files.

Confirm that the preprocessed file still reproduces the problem, as above.

Preprocessed C++ Modules repros

If you're using the Modules feature of the C++ compiler, there are some different steps required to create and validate a preprocessed repro.

To preprocess a source code file that uses a module

1. Capture the command line arguments used to build your repro, as described in To report the contents of the command line.

2. Open the **Developer Command Prompt** that matches the Visual Studio version and configuration architecture used to build your project.

3. Change to the directory that contains your repro project.

4. In the developer command prompt console window, enter the command **cl /P** *arguments filename.cpp*, where *arguments* is the list of arguments captured above, and *filename.cpp* is the name of the source file that consumes the module.

5. Change to the directory that contains the repro project that built the module interface (the .ifc output).

6. Capture the command line arguments used to build your module interface.

7. In the developer command prompt console window, enter the command **cl /P** *arguments modulename.ixx*, where *arguments* is the list of arguments captured above, and *modulename.ixx* is the name of the file that creates the module interface.

After you have generated the preprocessed files, its a good idea to make sure the problem still repros using the preprocessed file.

To confirm that the error still repros with the preprocessed file

1. In the developer console window, change back to the directory that contains your repro project.
2. Enter the command **cl** *arguments* **/TP** *filename*.i as above, to compile the preprocessed file as if it were a C++ source file.
3. Confirm that the problem is still reproduced by the preprocessed file.

Finally, attach the preprocessed repro files (*filename*.i and *modulename*.i) along with the .ifc output to your report.

Link repros

A *link repro* is the linker-generated contents of a directory specified by the **link_repro** environment variable. It contains build artifacts that collectively demonstrate a problem that occurs at link time, such as a backend crash involving Link-Time Code Generation (LTCG), or a linker crash. These build artifacts are the ones needed as linker input so that the problem can be reproduced. A link repro can be created easily by using this environment variable to enable the built-in repro generation capability of the linker.

To generate a link repro

1. Capture the command line arguments used to build your repro, as described in To report the contents of the command line.
2. Open the **Developer Command Prompt** that matches the Visual Studio version and configuration architecture used to build your project.
3. In the developer command prompt console window, change to the directory that contains your repro project.
4. Enter **mkdir linkrepro** to create a directory for the link repro.
5. Enter the command **set link_repro=linkrepro** to set the **link_repro** environment variable to the directory you just created. If your build is run from a different directory, as is often the case for more complex projects, then set **link_repro** to the full path to your linkrepro directory instead.
6. To build the repro project in Visual Studio, in the developer command prompt console window, enter the command **devenv**. This ensures that the value of the **link_repro** environment variable is visible to Visual Studio. To build the project at the command line, use the command line arguments captured above to duplicate the repro build.
7. Build your repro project, and confirm that the expected problem has occurred.
8. Close Visual Studio if you used it to perform the build.
9. In the developer command prompt console window, enter the command **set link_repro=** to clear the **link_repro** environment variable.

Finally, package the repro by compressing the entire linkrepro directory into a .zip file or similar and attach it to your report.

Other repros

If you can't reduce the problem to a single source file or preprocessed repro, and the problem does not require a link repro, we can investigate an IDE project. All the guidance on how to create a good repro still applies; the code should be minimal and self-contained, the problem should occur in our most recent tools, and if relevant, the problem should not be seen in other compilers.

Create your repro as a minimal IDE project, then package it by compressing the entire directory structure into a .zip file or similar and attach it to your report.

Ways to send your report

There are a couple of good ways to get your report to us. You can use Visual Studio's built-in Report a Problem Tool, or the Visual Studio Developer Community pages. You can also get directly to our Developer Community pages by choosing the **Product feedback** button at the bottom of this page. The choice depends on whether you want to use the tools built into the IDE for capturing screenshots and organizing your report for posting on the Developer Community pages, or if you'd prefer to use the website directly.

NOTE

Regardless of how you submit your report, Microsoft respects your privacy. Microsoft is committed to compliance with all data privacy laws and regulations. For information about how we treat the data that you send us, see the Microsoft Privacy Statement.

Use the Report a Problem tool

The **Report a Problem** tool in Visual Studio is a way for Visual Studio users to report a variety of problems with just a few clicks. It provides a simple form that you can use to specify detailed information about the problem you've encountered and then submit your report without ever leaving the IDE.

Reporting your problem through the **Report a Problem** tool is easy and convenient from the IDE. You can access it from the title bar by choosing the **Send Feedback** icon next to the **Quick Launch** search box, or you can find it on the menu bar in **Help** > **Send Feedback** > **Report a Problem**.

When you choose to report a problem, first search the Developer Community for similar problems. If your problem has been reported before, upvote the topic and add comments with additional specifics. If you don't see a similar problem, choose the **Report new problem** button at the bottom of the Visual Studio Feedback dialog and follow the steps to report your problem.

Use the Visual Studio Developer Community pages

The Visual Studio Developer Community pages are another convenient way to report problems and find solutions for Visual Studio and the C++ compiler, tools, and libraries. There are specific Developer Community pages for Visual Studio, Visual Studio for Mac, .NET, C++, Azure DevOps, and TFS. Beneath these tabs, near the top of each page, is a search box you can use to find posts or topics that report problems similar to yours. You may find that a solution or other useful information related to your problem is already available. If someone has reported the same problem before, please upvote and comment on that topic rather than create a new problem report. To comment, vote, or report a new problem, you may be asked to sign in to your Visual Studio account and to agree to give the Developer Community app access to your profile.

For issues with the C++ compiler, linker, and other tools and libraries, use theC++ page. If you search for your problem, and it hasn't been reported before, choose the **Report a problem** button next to the search box at the top of the page. You can include your repro code and command line, screen shots, links to related discussions, and any other information you think is relevant and useful.

TIP

For other kinds of problems you might encounter in Visual Studio that are not related to the C++ toolset (For example, UI issues, broken IDE functionality, or general crashes), use the **Report a Problem** tool in the IDE. This is the best choice, due to its screenshot capabilities and its ability to record UI actions that lead to the problem you've encountered. These kinds of errors can also be looked up on the Developer Community site. For more information, see How to report a problem with Visual Studio.

Reports and privacy

By default, **all information in reports and any comments and replies are publicly visible**. Normally, this is a benefit, because it allows the entire community to see the issues, solutions, and workarounds other users have found. However, if you're concerned about making your data or identity public, for privacy or intellectual property reasons, you have options.

If you are concerned about revealing your identity, create a new Microsoft account that does not disclose any details about you. Use this account to create your report.

Don't put anything you want to keep private in the title or content of the initial report, which is public. Instead, note that you will send details privately in a separate comment. To make sure that your report is directed to the right people, include **cppcompiler** in the topic list of your problem report. Once the problem report is created, it's now possible to specify who can see your replies and attachments.

To create a problem report for private information

1. In the report you created, choose **Add comment** to create your private description of the problem.

2. In the reply editor, use the dropdown control below the **Submit** and **Cancel** buttons to specify the audience for your reply. Only the people you specify can see these private replies and any images, links, or code you include in them. Choose **Viewable by moderators and the original poster** to limit visibility to Microsoft employees and yourself.

3. Add the description and any other information, images, and file attachments needed for your repro. Choose the **Submit** button to send this information privately.

   Note that there is a 2GB limit on attached files, and a maximum of 10 files. For any larger uploads, please request an upload URL in your private comment.

Any replies under this comment have the same restricted visibility you specified. This is true even if the dropdown control on replies does not show the restricted visibility status correctly.

To maintain your privacy and keep your sensitive information out of public view, please take care to keep all interaction with Microsoft to replies under this restricted comment. Replies to other comments may cause you to accidentally disclose sensitive information.

How to report a C++ documentation issue

We use GitHub issues to track problems reported in our documentation. You can now create GitHub issues directly from a content page, which enables you interact in a much richer way with writers and product teams. If you see an issue with a document, a bad code sample, a confusing explanation, a critical omission, or even just a typo, you can easily let us know. Scroll to the bottom of the page and select **Sign in to give documentation feedback**. You'll need to create a GitHub account if you don't have one already, but once you do, you can see all of our documentation issues, their status, and get notifications when changes are made for the issue you reported. For more information, see A New Feedback System Is Coming to docs.microsoft.com.

When you create a documentation issue on GitHub by using the documentation feedback button, the issue is automatically filled in with some information about the page you created the issue on, so we know where the problem is located. Please don't edit this information. Just append the details about what's wrong and, if you like, a suggested fix. Our documentation is open source, so if you'd like to actually make a fix and propose it yourself, you can do that. For more information about how you can contribute to our documentation, see our Contributing guide on GitHub.

Install C++ support in Visual Studio

4/2/2019 • 8 minutes to read • Edit Online

If you haven't downloaded and installed Visual Studio and the Visual C++ tools yet, here's how to get started.

Visual Studio 2019 Installation

Welcome to Visual Studio 2019! In this version, it's easy to choose and install just the features you need. And because of its reduced minimum footprint, it installs quickly and with less system impact.

NOTE

This topic applies to installation of Visual Studio on Windows. Visual Studio Code is a lightweight, cross-platform development environment that runs on Windows, Mac, and Linux systems. The Microsoft C/C++ for Visual Studio Code extension supports IntelliSense, debugging, code formatting, auto-completion. Visual Studio for Mac doesn't support Microsoft C++, but does support .NET languages and cross-platform development. For installation instructions, see Install Visual Studio for Mac.

Want to know more about what else is new in this version? See the Visual Studio release notes.

Ready to install? We'll walk you through it, step-by-step.

Step 1 - Make sure your computer is ready for Visual Studio

Before you begin installing Visual Studio:

1. Check the system requirements. These requirements help you know whether your computer supports Visual Studio 2019.

2. Apply the latest Windows updates. These updates ensure that your computer has both the latest security updates and the required system components for Visual Studio.

3. Reboot. The reboot ensures that any pending installs or updates don't hinder the Visual Studio install.

4. Free up space. Remove unneeded files and applications from your %SystemDrive% by, for example, running the Disk Cleanup app.

For questions about running previous versions of Visual Studio side by side with Visual Studio 2019, see the Visual Studio 2019 Platform Targeting and Compatibility page.

Step 2 - Download Visual Studio

Next, download the Visual Studio bootstrapper file. To do so, choose the following button, choose the edition of Visual Studio that you want, choose **Save**, and then choose **Open folder**.

DOWNLOAD VISUAL STUDIO

Step 3 - Install the Visual Studio installer

Run the bootstrapper file to install the Visual Studio Installer. This new lightweight installer includes everything you need to both install and customize Visual Studio.

1. From your **Downloads** folder, double-click the bootstrapper that matches or is similar to one of the following files:

- **vs_community.exe** for Visual Studio Community
- **vs_professional.exe** for Visual Studio Professional
- **vs_enterprise.exe** for Visual Studio Enterprise

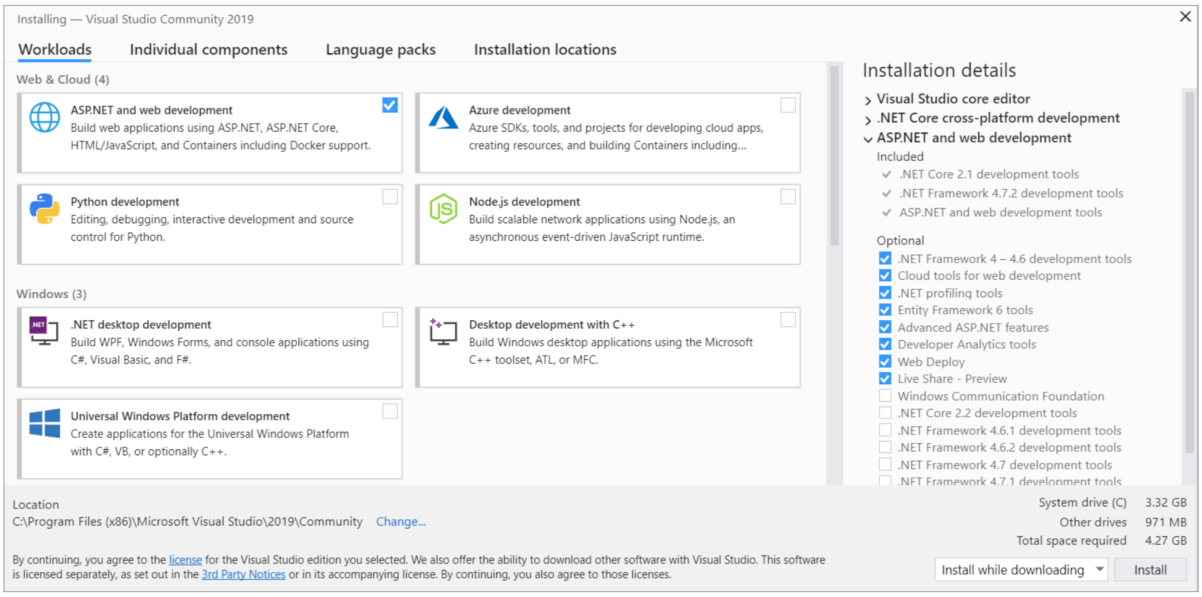
   If you receive a User Account Control notice, choose **Yes**.

2. We'll ask you to acknowledge the Microsoft License Terms and the Microsoft Privacy Statement. Choose **Continue**.

Step 4 - Choose workloads

After the installer is installed, you can use it to customize your installation by selecting the *workloads*, or feature sets, that you want. Here's how.

1. Find the workload you want in the **Installing Visual Studio** screen.



Installing — Visual Studio Community 2019

Workloads | Individual components | Language packs | Installation locations

Web & Cloud (4)

- [✓] ASP.NET and web development — Build web applications using ASP.NET, ASP.NET Core, HTML/JavaScript, and Containers including Docker support.
- [ ] Azure development — Azure SDKs, tools, and projects for developing cloud apps, creating resources, and building Containers including...
- [ ] Python development — Editing, debugging, interactive development and source control for Python.
- [ ] Node.js development — Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

Windows (3)

- [ ] .NET desktop development — Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F#.
- [ ] Desktop development with C++ — Build Windows desktop applications using the Microsoft C++ toolset, ATL, or MFC.
- [ ] Universal Windows Platform development — Create applications for the Universal Windows Platform with C#, VB, or optionally C++.

Installation details

> Visual Studio core editor
> .NET Core cross-platform development
∨ ASP.NET and web development

Included
- ✓ .NET Core 2.1 development tools
- ✓ .NET Framework 4.7.2 development tools
- ✓ ASP.NET and web development tools

Optional
- [✓] .NET Framework 4 – 4.6 development tools
- [✓] Cloud tools for web development
- [✓] .NET profiling tools
- [✓] Entity Framework 6 tools
- [✓] Advanced ASP.NET features
- [✓] Developer Analytics tools
- [✓] Web Deploy
- [✓] Live Share - Preview
- [ ] Windows Communication Foundation
- [ ] .NET Core 2.2 development tools
- [ ] .NET Framework 4.6.1 development tools
- [ ] .NET Framework 4.6.2 development tools
- [ ] .NET Framework 4.7 development tools
- [ ] .NET Framework 4.7.1 development tools

Location
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community Change...

System drive (C)	3.32 GB
Other drives	971 MB
Total space required	4.27 GB

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

Install while downloading | Install

For core C++ support, choose the "Desktop development with C++" workload. It comes with the default core editor, which includes basic code editing support for over 20 languages, the ability to open and edit code from any folder without requiring a project, and integrated source code control.

Additional workloads support other kinds of C++ development. For example, choose the "Universal Windows Platform development" workload to create apps that use the Windows Runtime for the Microsoft Store. Choose "Game development with C++" to create games that use DirectX, Unreal, and Cocos2d. Choose "Linux development with C++" to target Linux platforms, including IoT development.

The **Installation details** pane lists the included and optional components installed by each workload. You can select or deselect optional components in this list. For example, to support development by using the Visual Studio 2017 or 2015 compiler toolsets, choose the MSVC v141 or MSVC v140 optional components. You can add support for MFC, the experimental Modules language extension, IncrediBuild, and more.

2. After you choose the workload(s) and optional components you want, choose **Install**.

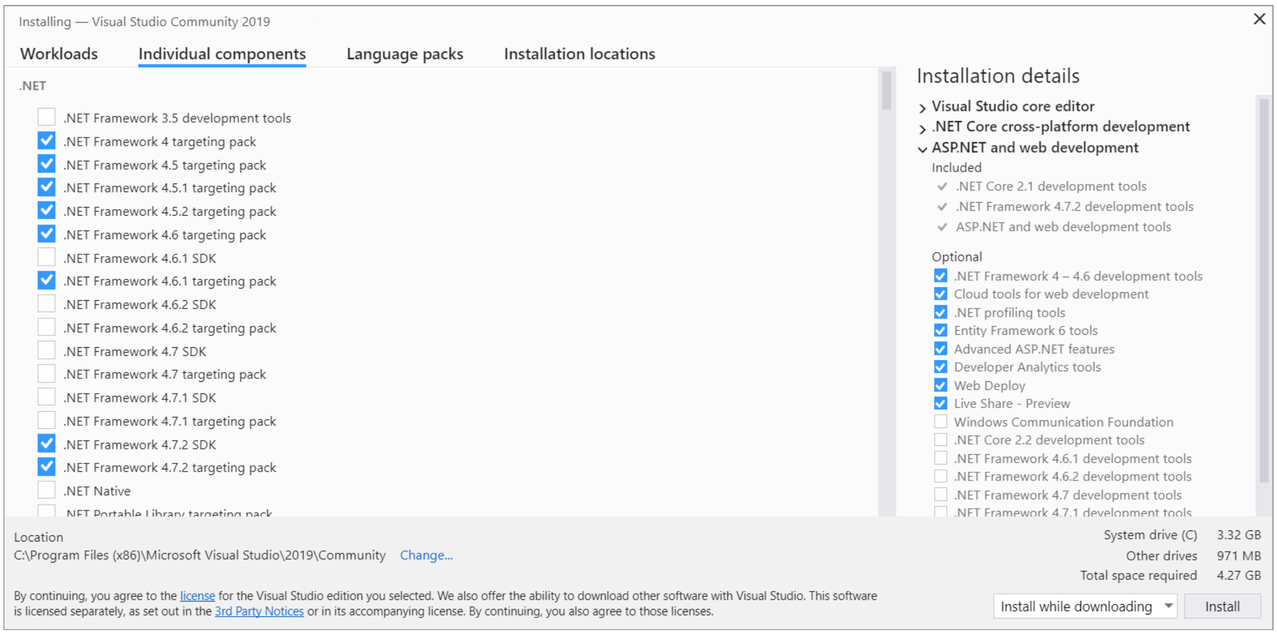
   Next, status screens appear that show the progress of your Visual Studio installation.

TIP

At any time after installation, you can install workloads or components that you didn't install initially. If you have Visual Studio open, go to **Tools** > **Get Tools and Features...** which opens the Visual Studio Installer. Or, open **Visual Studio Installer** from the Start menu. From there, you can choose the workloads or components that you wish to install. Then, choose **Modify**.

Step 5 - Choose individual components (Optional)

If you don't want to use the Workloads feature to customize your Visual Studio installation, or you want to add more components than a workload installs, you can do so by installing or adding individual components from the **Individual components** tab. Choose what you want, and then follow the prompts.



Installing — Visual Studio Community 2019

Workloads | Individual components | Language packs | Installation locations

.NET

- ☐ .NET Framework 3.5 development tools
- ☑ .NET Framework 4 targeting pack
- ☑ .NET Framework 4.5 targeting pack
- ☑ .NET Framework 4.5.1 targeting pack
- ☑ .NET Framework 4.5.2 targeting pack
- ☑ .NET Framework 4.6 targeting pack
- ☐ .NET Framework 4.6.1 SDK
- ☑ .NET Framework 4.6.1 targeting pack
- ☐ .NET Framework 4.6.2 SDK
- ☐ .NET Framework 4.6.2 targeting pack
- ☐ .NET Framework 4.7 SDK
- ☐ .NET Framework 4.7 targeting pack
- ☐ .NET Framework 4.7.1 SDK
- ☐ .NET Framework 4.7.1 targeting pack
- ☑ .NET Framework 4.7.2 SDK
- ☑ .NET Framework 4.7.2 targeting pack
- ☐ .NET Native
- ☐ .NET Portable Library targeting pack

Installation details

- › Visual Studio core editor
- › .NET Core cross-platform development
- ⌄ ASP.NET and web development

Included
- ✓ .NET Core 2.1 development tools
- ✓ .NET Framework 4.7.2 development tools
- ✓ ASP.NET and web development tools

Optional
- ☑ .NET Framework 4 – 4.6 development tools
- ☑ Cloud tools for web development
- ☑ .NET profiling tools
- ☑ Entity Framework 6 tools
- ☑ Advanced ASP.NET features
- ☑ Developer Analytics tools
- ☑ Web Deploy
- ☑ Live Share - Preview
- ☐ Windows Communication Foundation
- ☐ .NET Core 2.2 development tools
- ☐ .NET Framework 4.6.1 development tools
- ☐ .NET Framework 4.6.2 development tools
- ☐ .NET Framework 4.7 development tools
- ☐ .NET Framework 4.7.1 development tools

Location
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community Change...

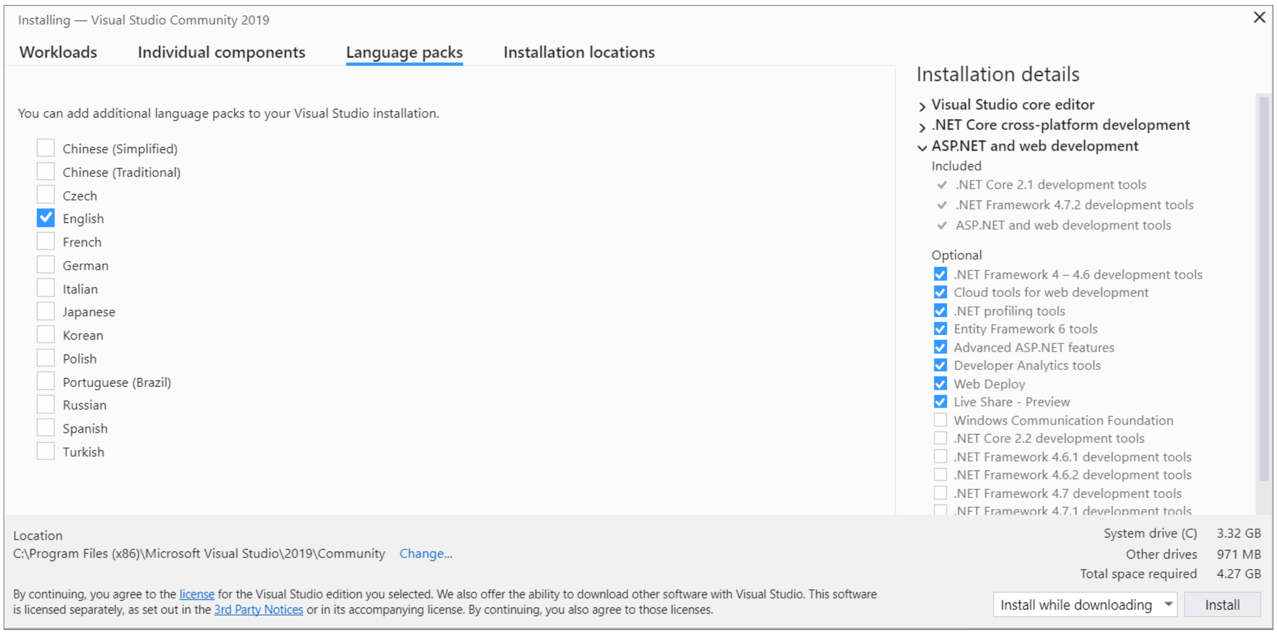
System drive (C)	3.32 GB
Other drives	971 MB
Total space required	4.27 GB

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

Install while downloading | Install

Step 6 - Install language packs (Optional)

By default, the installer program tries to match the language of the operating system when it runs for the first time. To install Visual Studio in a language of your choosing, choose the **Language packs** tab from the Visual Studio Installer, and then follow the prompts.



Installing — Visual Studio Community 2019

Workloads | Individual components | Language packs | Installation locations

You can add additional language packs to your Visual Studio installation.

- ☐ Chinese (Simplified)
- ☐ Chinese (Traditional)
- ☐ Czech
- ☑ English
- ☐ French
- ☐ German
- ☐ Italian
- ☐ Japanese
- ☐ Korean
- ☐ Polish
- ☐ Portuguese (Brazil)
- ☐ Russian
- ☐ Spanish
- ☐ Turkish

Installation details

- › Visual Studio core editor
- › .NET Core cross-platform development
- ⌄ ASP.NET and web development

Included
- ✓ .NET Core 2.1 development tools
- ✓ .NET Framework 4.7.2 development tools
- ✓ ASP.NET and web development tools

Optional
- ☑ .NET Framework 4 – 4.6 development tools
- ☑ Cloud tools for web development
- ☑ .NET profiling tools
- ☑ Entity Framework 6 tools
- ☑ Advanced ASP.NET features
- ☑ Developer Analytics tools
- ☑ Web Deploy
- ☑ Live Share - Preview
- ☐ Windows Communication Foundation
- ☐ .NET Core 2.2 development tools
- ☐ .NET Framework 4.6.1 development tools
- ☐ .NET Framework 4.6.2 development tools
- ☐ .NET Framework 4.7 development tools
- ☐ .NET Framework 4.7.1 development tools

Location
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community Change...

System drive (C)	3.32 GB
Other drives	971 MB
Total space required	4.27 GB

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

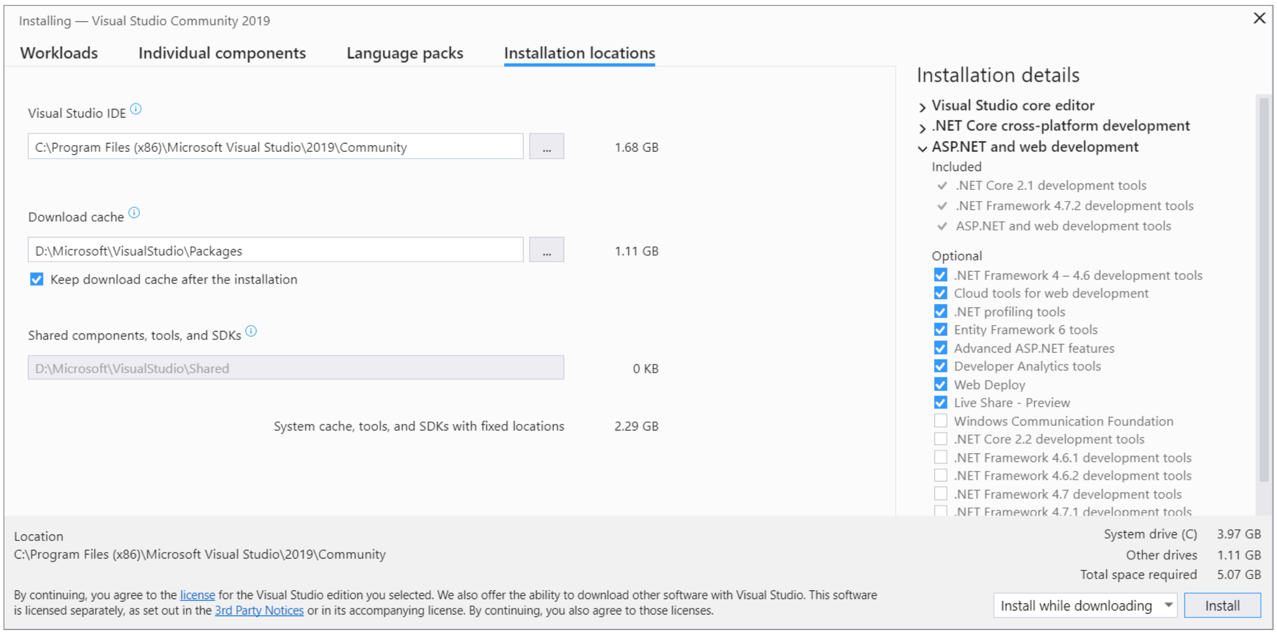
Install while downloading | Install

Change the installer language from the command line

Another way that you can change the default language is by running the installer from the command line. For example, you can force the installer to run in English by using the following command: `vs_installer.exe --locale en-US`. The installer will remember this setting when it's run the next time. The installer supports the following language tokens: zh-cn, zh-tw, cs-cz, en-us, es-es, fr-fr, de-de, it-it, ja-jp, ko-kr, pl-pl, pt-br, ru-ru, and tr-tr.

Step 7 - Change the installation location (Optional)

You can reduce the installation footprint of Visual Studio on your system drive. You can choose to move the download cache, shared components, SDKs, and tools to different drives, and keep Visual Studio on the drive that runs it the fastest.


Installing — Visual Studio Community 2019

Workloads | Individual components | Language packs | Installation locations

Visual Studio IDE
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community ... 1.68 GB

Download cache
D:\Microsoft\VisualStudio\Packages ... 1.11 GB
☑ Keep download cache after the installation

Shared components, tools, and SDKs
D:\Microsoft\VisualStudio\Shared 0 KB

System cache, tools, and SDKs with fixed locations 2.29 GB

Installation details
› Visual Studio core editor
› .NET Core cross-platform development
⌄ ASP.NET and web development
Included
✓ .NET Core 2.1 development tools
✓ .NET Framework 4.7.2 development tools
✓ ASP.NET and web development tools
Optional
☑ .NET Framework 4 – 4.6 development tools
☑ Cloud tools for web development
☑ .NET profiling tools
☑ Entity Framework 6 tools
☑ Advanced ASP.NET features
☑ Developer Analytics tools
☑ Web Deploy
☑ Live Share - Preview
☐ Windows Communication Foundation
☐ .NET Core 2.2 development tools
☐ .NET Framework 4.6.1 development tools
☐ .NET Framework 4.6.2 development tools
☐ .NET Framework 4.7 development tools
☐ .NET Framework 4.7.1 development tools

Location
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

System drive (C)	3.97 GB
Other drives	1.11 GB
Total space required	5.07 GB

Install while downloading | Install

IMPORTANT

You can select a different drive only when you first install Visual Studio. If you've already installed it and want to change drives, you must uninstall Visual Studio and then reinstall it.

Step 8 - Start developing

1. After Visual Studio installation is complete, choose the **Launch** button to get started developing with Visual Studio.
2. On the start window, choose **Create a new project**.
3. In the search box, enter the type of app you want to create to see a list of available templates. The list of templates depends on the workload(s) that you chose during installation. To see different templates, choose different workloads.

   You can also filter your search for a specific programming language by using the **Language** drop-down list. You can filter by using the **Platform** list and the **Project type** list, too.
4. Visual Studio opens your new project, and you're ready to code!

Visual Studio 2017 Installation

In Visual Studio 2017, it's easy to choose and install just the features you need. And because of its reduced minimum footprint, it installs quickly and with less system impact.

Prerequisites

- A broadband internet connection. The Visual Studio installer can download several gigabytes of data.
- A computer that runs Microsoft Windows 7 or later versions. We recommend Windows 10 for the best development experience. Make sure that the latest updates are applied to your system before you install Visual Studio.
- Enough free disk space. Visual Studio requires at least 7 GB of disk space, and can take 50 GB or more if

many common options are installed. We recommend you install it on your C: drive.

For details on the disk space and operating system requirements, see Visual Studio Product Family System Requirements. The installer reports how much disk space is required for the options you select.

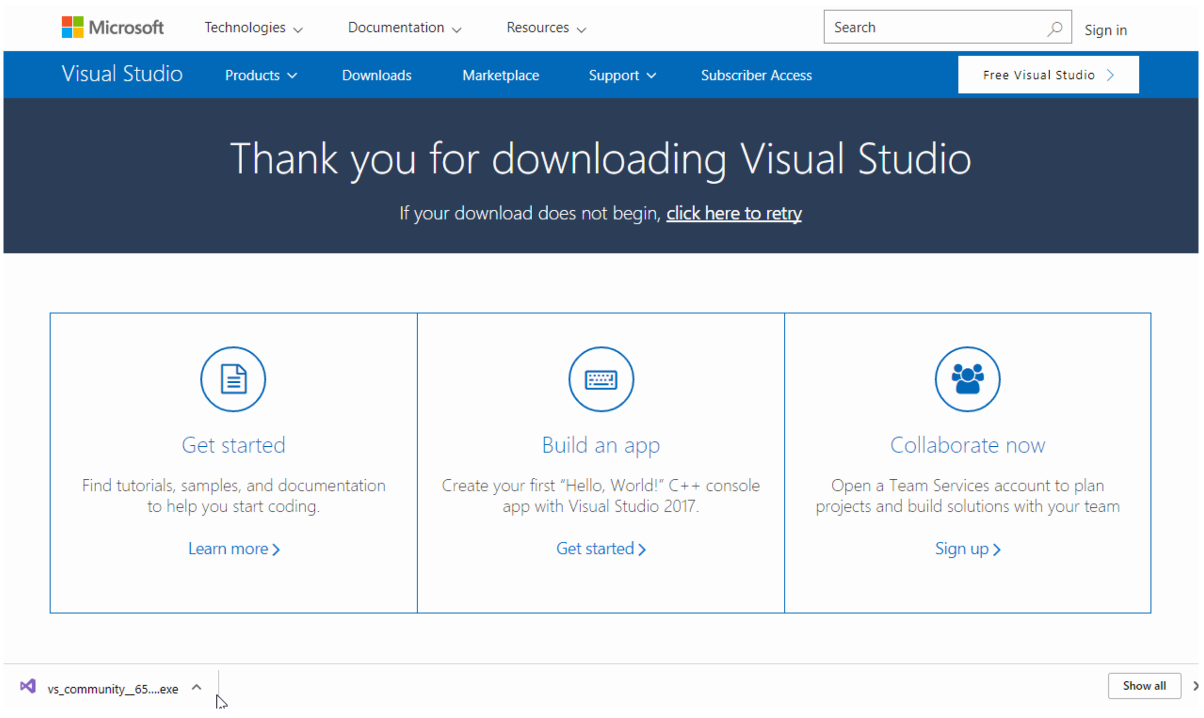
Download and install

1. Download the latest Visual Studio 2017 installer for Windows.

   Install Visual Studio 2017 Community

   > **TIP**
   >
   > The Community edition is for individual developers, classroom learning, academic research, and open source development. For other uses, install Visual Studio 2017 Professional or Visual Studio 2017 Enterprise.

2. Find the installer file you downloaded and run it. It may be displayed in your browser, or you may find it in your Downloads folder. The installer needs Administrator privileges to run. You may see a **User Account Control** dialog asking you to give permission to let the installer make changes to your system; choose **Yes**. If you're having trouble, find the downloaded file in File Explorer, right-click on the installer icon, and choose **Run as Administrator** from the context menu.



Thank you for downloading Visual Studio

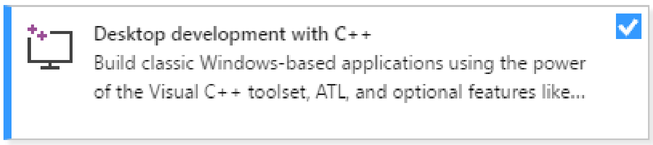
If your download does not begin, click here to retry

Get started
Find tutorials, samples, and documentation to help you start coding.
Learn more >

Build an app
Create your first "Hello, World!" C++ console app with Visual Studio 2017.
Get started >

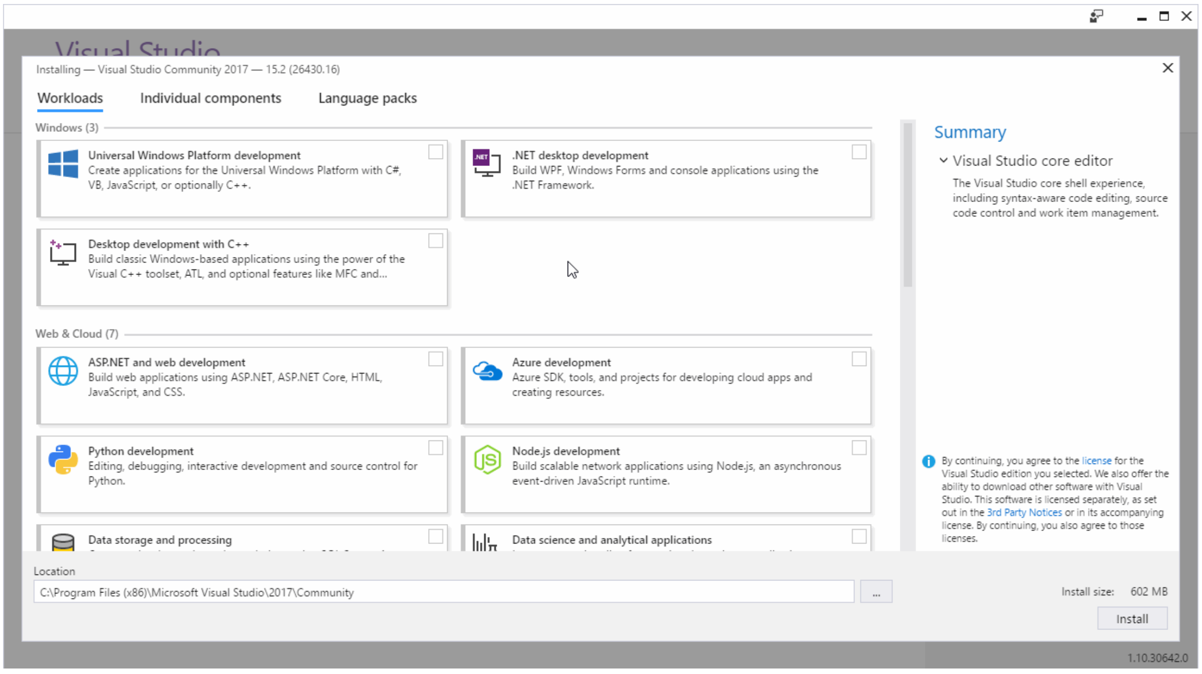
Collaborate now
Open a Team Services account to plan projects and build solutions with your team
Sign up >

3. The installer presents you with a list of workloads, which are groups of related options for specific development areas. Support for C++ is now part of optional workloads that aren't installed by default.



Desktop development with C++ ✓
Build classic Windows-based applications using the power of the Visual C++ toolset, ATL, and optional features like...

For C++, select the **Desktop development with C++** workload and then choose **Install**.



Installing — Visual Studio Community 2017 — 15.2 (26430.16)

Workloads | Individual components | Language packs

Windows (3)

Universal Windows Platform development
Create applications for the Universal Windows Platform with C#, VB, JavaScript, or optionally C++.

.NET desktop development
Build WPF, Windows Forms and console applications using the .NET Framework.

Desktop development with C++
Build classic Windows-based applications using the power of the Visual C++ toolset, ATL, and optional features like MFC and...

Web & Cloud (7)

ASP.NET and web development
Build web applications using ASP.NET, ASP.NET Core, HTML, JavaScript, and CSS.

Azure development
Azure SDK, tools, and projects for developing cloud apps and creating resources.

Python development
Editing, debugging, interactive development and source control for Python.

Node.js development
Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

Data storage and processing

Data science and analytical applications

Summary

Visual Studio core editor
The Visual Studio core shell experience, including syntax-aware code editing, source code control and work item management.

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

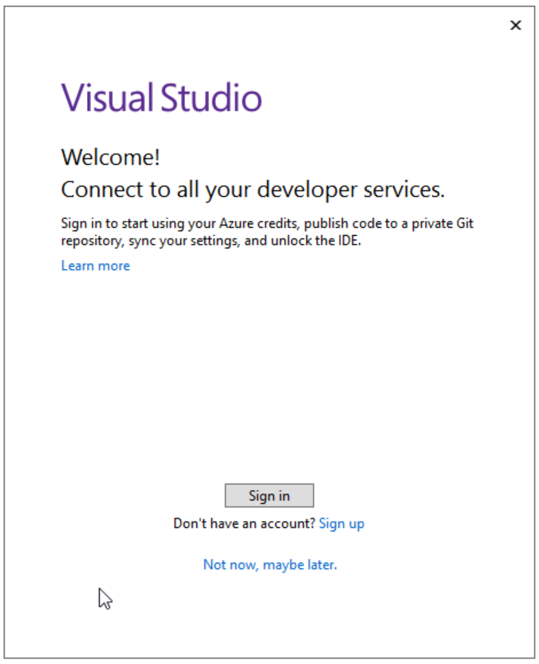
Location
C:\Program Files (x86)\Microsoft Visual Studio\2017\Community

Install size: 602 MB

4. When the installation completes, choose the **Launch** button to start Visual Studio.

   The first time you run Visual Studio, you're asked to sign in with a Microsoft Account. If you don't have one, you can create one for free. You must also choose a theme. Don't worry, you can change it later if you want to.

   It may take Visual Studio several minutes to get ready for use the first time you run it. Here's what it looks like in a quick time-lapse:



Visual Studio

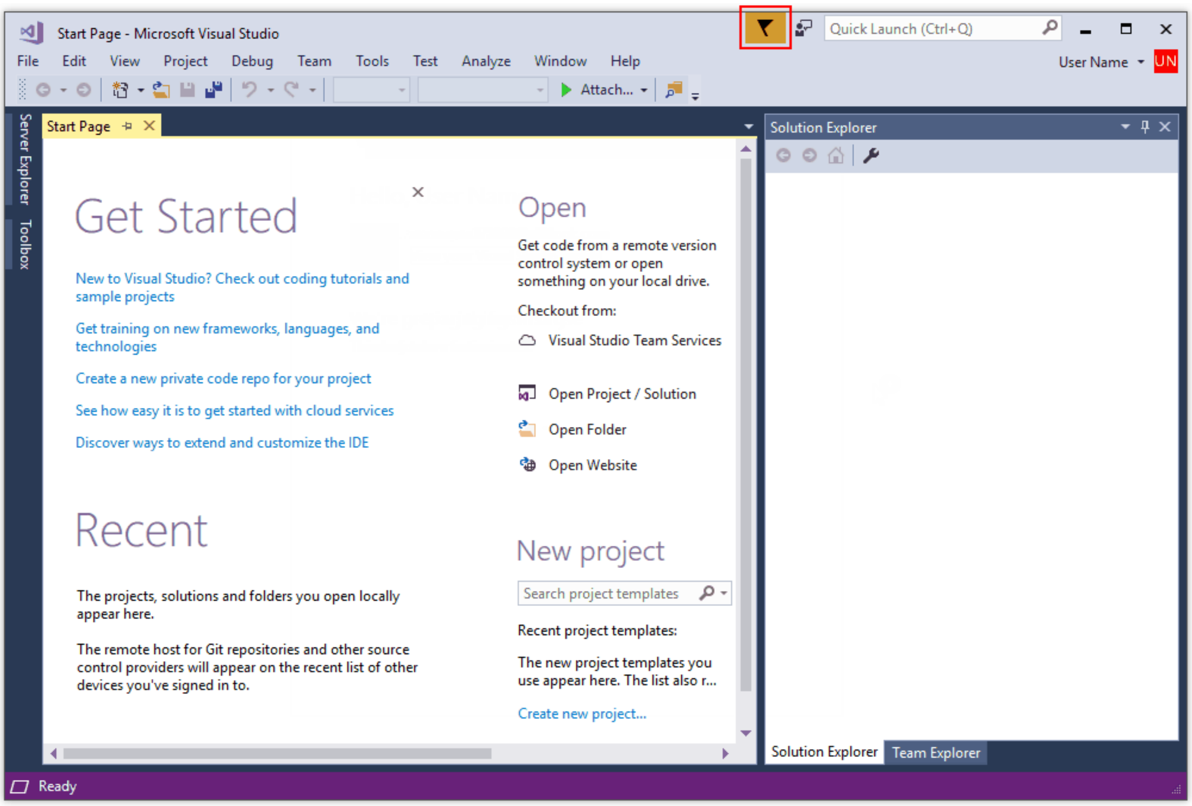
Welcome!

Connect to all your developer services.

Sign in to start using your Azure credits, publish code to a private Git repository, sync your settings, and unlock the IDE.

Visual Studio starts much faster when you run it again.

5. When Visual Studio opens, check to see if the flag icon in the title bar is highlighted:



Get Started

New to Visual Studio? Check out coding tutorials and sample projects

Get training on new frameworks, languages, and technologies

Create a new private code repo for your project

See how easy it is to get started with cloud services

Discover ways to extend and customize the IDE

Recent

The projects, solutions and folders you open locally appear here.

The remote host for Git repositories and other source control providers will appear on the recent list of other devices you've signed in to.

Open

Get code from a remote version control system or open something on your local drive.

Checkout from:

Visual Studio Team Services

New project

Recent project templates:

The new project templates you use appear here. The list also r...

If it's highlighted, select it to open the **Notifications** window. If there are any updates available for Visual Studio, we recommend you install them now. Once the installation is complete, restart Visual Studio.

Visual Studio 2015 Installation

To install Visual Studio 2015, go to Download older versions of Visual Studio. Run the setup program and choose **Custom installation** and then choose the C++ component. To add C++ support to an existing Visual Studio 2015 installation, click on the Windows Start button and type **Add Remove Programs**. Open the program from the results list and then find your Visual Studio 2015 installation in the list of installed programs. Double-click it, then choose **Modify** and select the Visual C++ components to install.

In general, we highly recommend that you use Visual Studio 2017 even if you need to compile your code using the Visual Studio 2015 compiler. For more information, see Use native multi-targeting in Visual Studio to build old projects.

When Visual Studio is running, you're ready to continue to the next step.

Next Steps

Create a C++ project

https://docs.microsoft.com/

Create a C++ console app project

4/3/2019 • 5 minutes to read • Edit Online

The usual starting point for a C++ programmer is a "Hello, world!" application that runs on the command line. That's what you'll create in Visual Studio in this step.

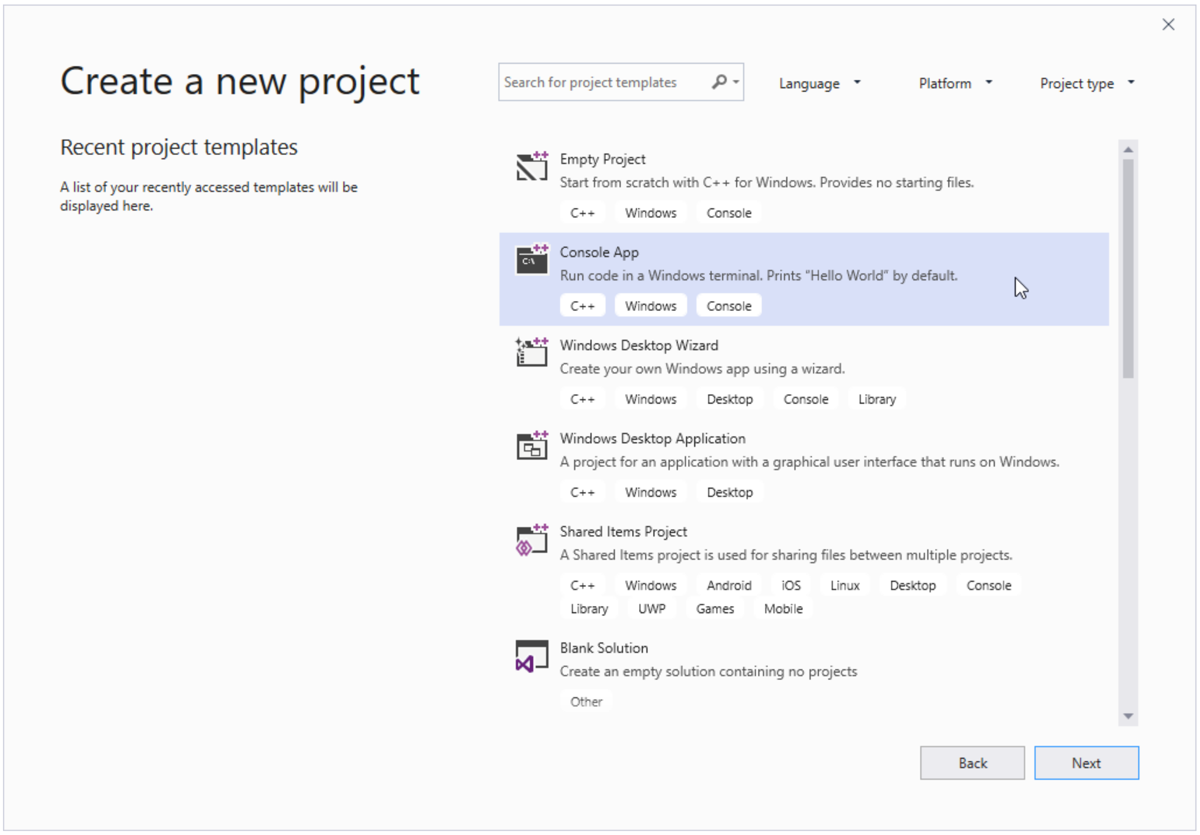
Prerequisites

- Have Visual Studio with the Desktop development with C++ workload installed and running on your computer. If it's not installed yet, see Install C++ support in Visual Studio.

Create your app project

Visual Studio uses *projects* to organize the code for an app, and *solutions* to organize your projects. A project contains all the options, configurations, and rules used to build your apps, and manages the relationship between all the project's files and any external files. To create your app, first, you'll create a new project and solution.

1. In Visual Studio, open the **File** menu and choose **New** > **Project** to open the **Create a new Project** dialog. Select the **Console App** template, and then choose **Next**.



Create a new project

Recent project templates

A list of your recently accessed templates will be displayed here.

Search for project templates | Language | Platform | Project type

Empty Project
Start from scratch with C++ for Windows. Provides no starting files.
C++ Windows Console

Console App
Run code in a Windows terminal. Prints "Hello World" by default.
C++ Windows Console

Windows Desktop Wizard
Create your own Windows app using a wizard.
C++ Windows Desktop Console Library

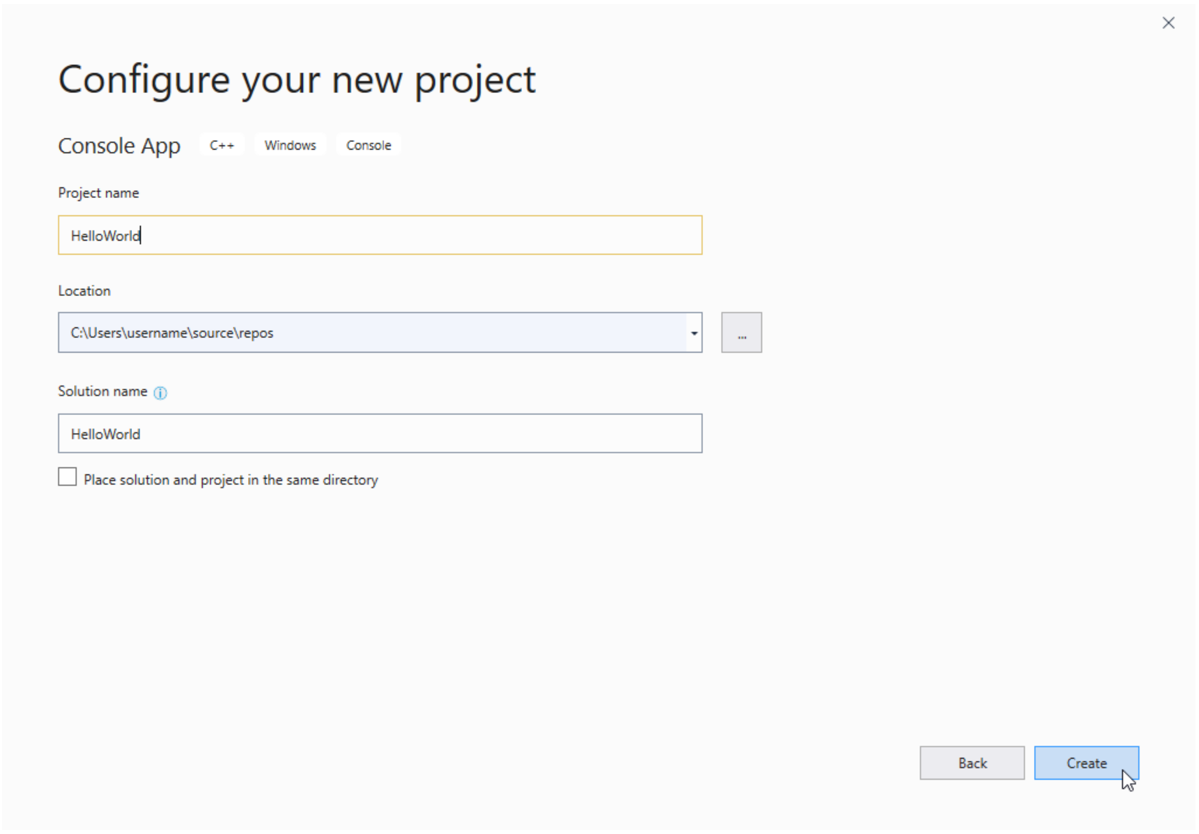
Windows Desktop Application
A project for an application with a graphical user interface that runs on Windows.
C++ Windows Desktop

Shared Items Project
A Shared Items project is used for sharing files between multiple projects.
C++ Windows Android iOS Linux Desktop Console Library UWP Games Mobile

Blank Solution
Create an empty solution containing no projects
Other

Back | Next

2. In the **Configure your new project** dialog, enter *HelloWorld* in the **Project name** edit box. Choose **Create** to create the project.



Configure your new project

Console App C++ Windows Console

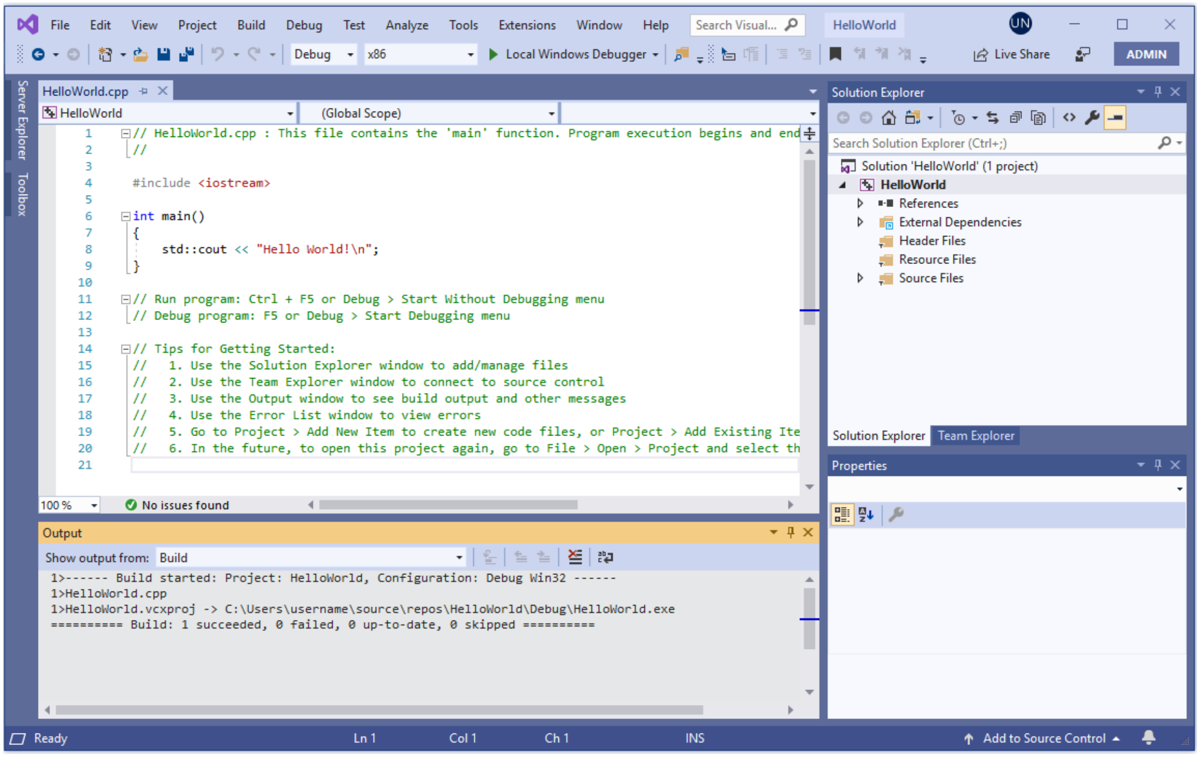
Project name: HelloWorld

Location: C:\Users\username\source\repos

Solution name: HelloWorld

☐ Place solution and project in the same directory

Visual Studio creates a new project, ready for you to add and edit your source code. By default, the Console App template fills in your source code with a "Hello World" app:

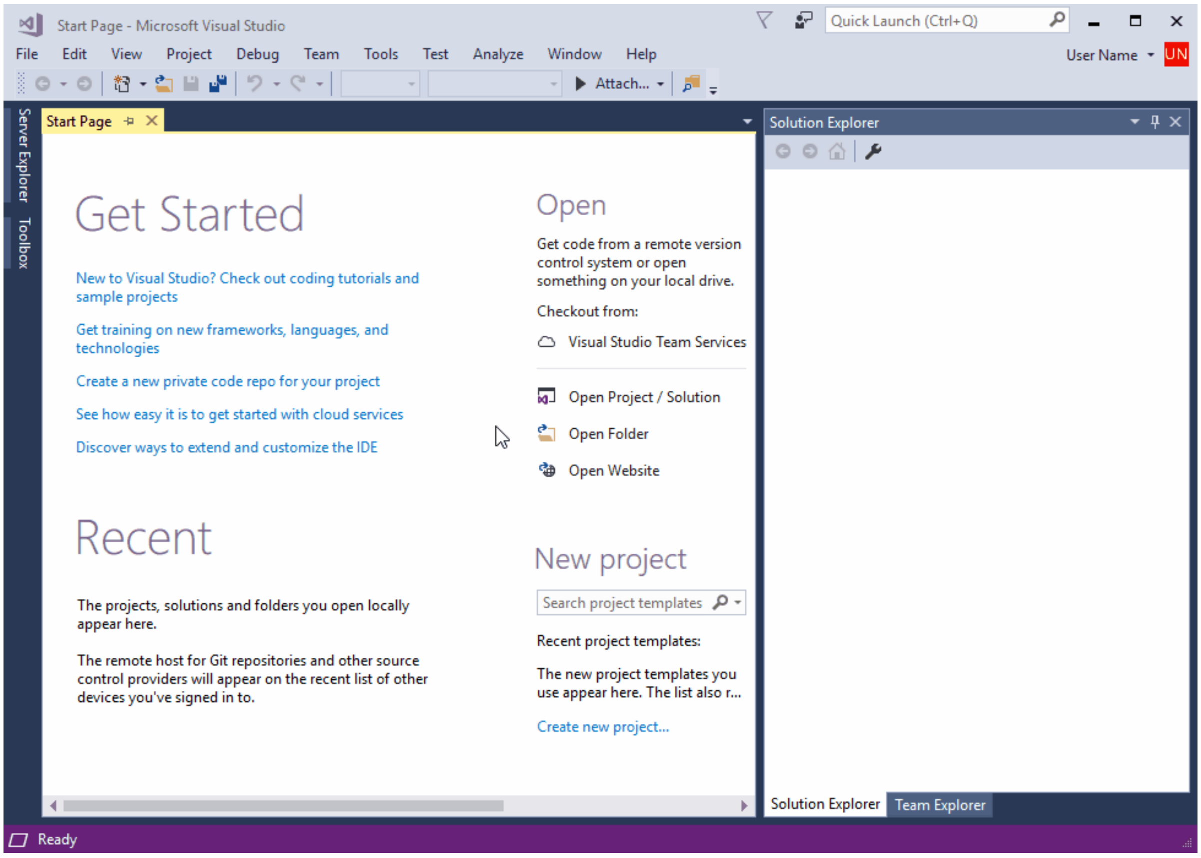


```cpp
// HelloWorld.cpp : This file contains the 'main' function. Program execution begins and end
//

#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu

// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or Project > Add Existing Ite
//   6. In the future, to open this project again, go to File > Open > Project and select th
```

```
1>------ Build started: Project: HelloWorld, Configuration: Debug Win32 ------
1>HelloWorld.cpp
1>HelloWorld.vcxproj -> C:\Users\username\source\repos\HelloWorld\Debug\HelloWorld.exe
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

When the code looks like this in the editor, you're ready to go on to the next step and build your app.

1. In Visual Studio, open the **File** menu and choose **New > Project** to open the **New Project** dialog.



Get Started

New to Visual Studio? Check out coding tutorials and sample projects

Get training on new frameworks, languages, and technologies

Create a new private code repo for your project

See how easy it is to get started with cloud services

Discover ways to extend and customize the IDE

Recent

The projects, solutions and folders you open locally appear here.

The remote host for Git repositories and other source control providers will appear on the recent list of other devices you've signed in to.

Open

Get code from a remote version control system or open something on your local drive.

Checkout from:

Visual Studio Team Services

Open Project / Solution

Open Folder

Open Website

New project

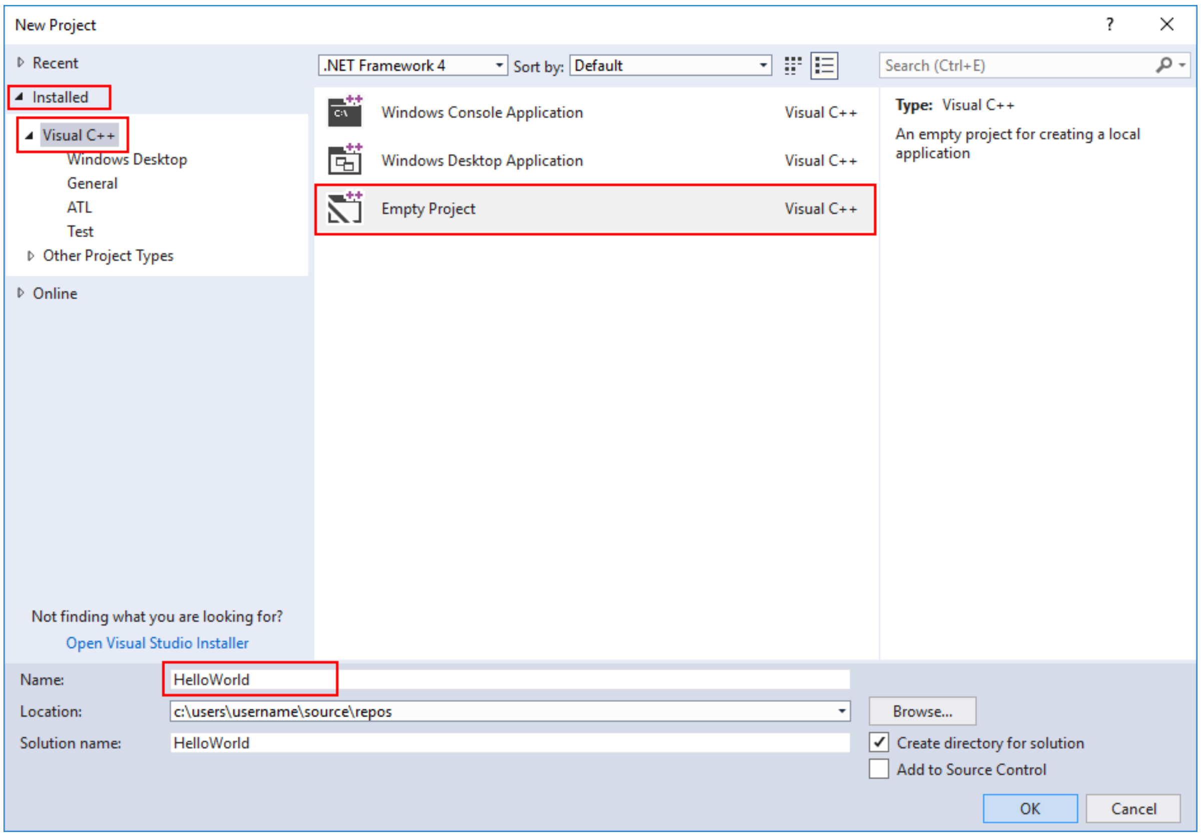
Search project templates

Recent project templates:

The new project templates you use appear here. The list also r...

Create new project...

2. In the **New Project** dialog, select **Installed**, **Visual C++** if it isn't selected already, and then choose the **Empty Project** template. In the **Name** field, enter *HelloWorld*. Choose **OK** to create the project.



Windows Console Application	Visual C++
Windows Desktop Application	Visual C++
Empty Project	Visual C++

Type: Visual C++

An empty project for creating a local application

Not finding what you are looking for?

Open Visual Studio Installer

Name: HelloWorld

Location: c:\users\username\source\repos

Solution name: HelloWorld

[✓] Create directory for solution

[ ] Add to Source Control

Visual Studio creates a new, empty project, ready for you to specialize for the kind of app you want to create and to add your source code files. You'll do that next.

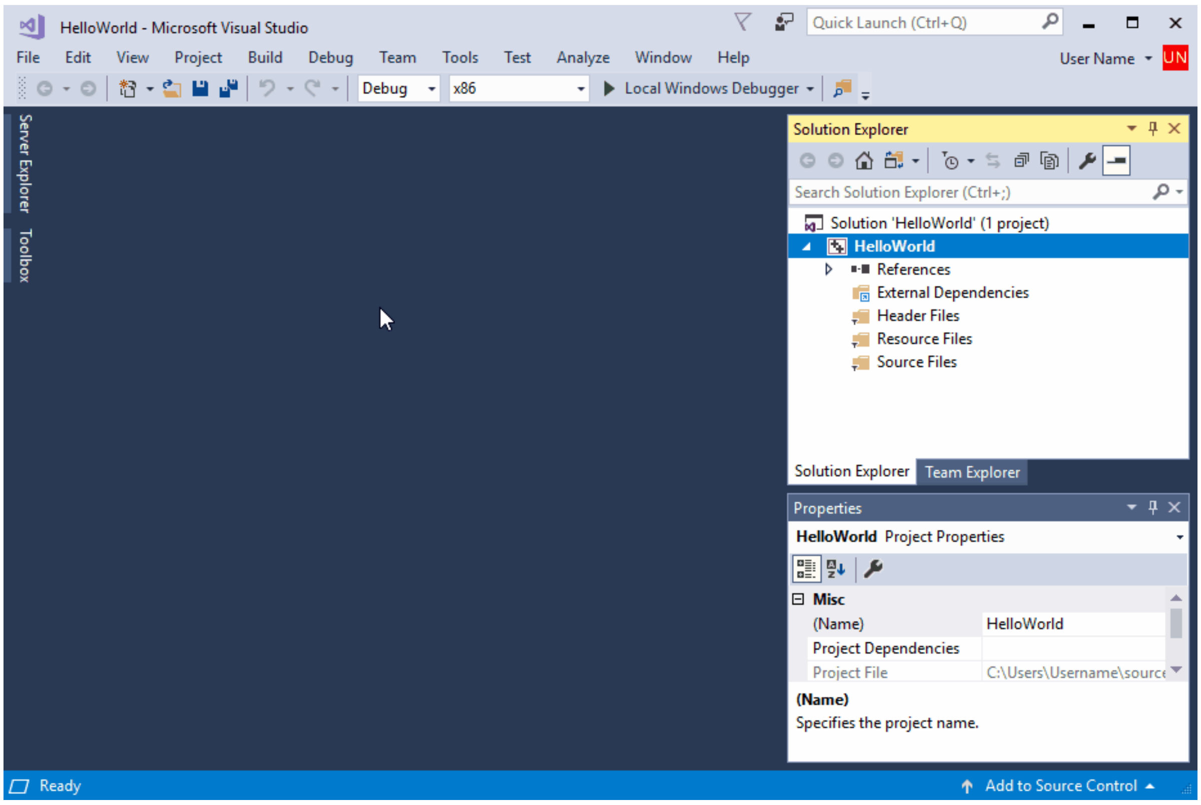
I ran into a problem.

Make your project a console app

Visual Studio can create all kinds of apps and components for Windows and other platforms. The **Empty Project** template isn't specific about what kind of app it creates. To create a *console app*, one that runs in a console or command prompt window, you must tell Visual Studio to build your app to use the console subsystem.

1. In Visual Studio, open the **Project** menu and choose **Properties** to open the **HelloWorld Property Pages** dialog.

2. In the **Property Pages** dialog, under **Configuration Properties**, select **Linker**, **System**, and then choose the edit box next to the **Subsystem** property. In the dropdown menu that appears, select **Console (/SUBSYSTEM:CONSOLE)**. Choose **OK** to save your changes.



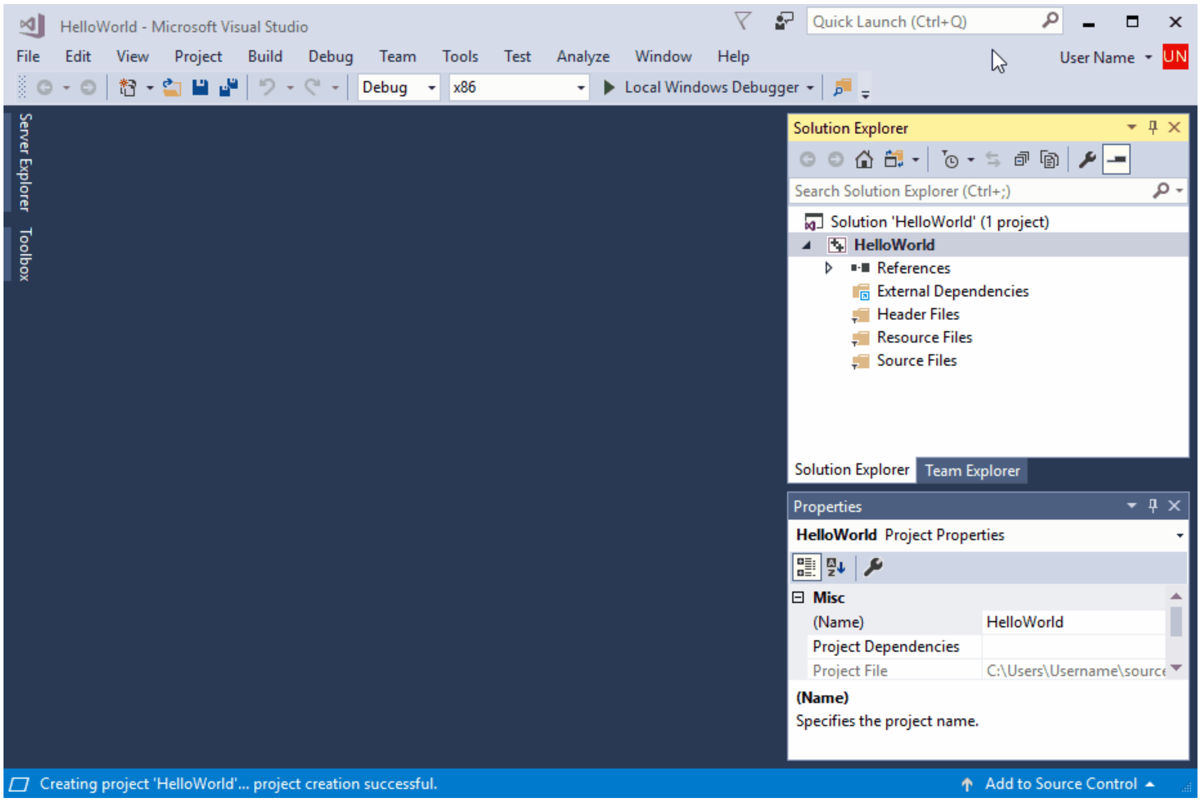
Visual Studio now knows to build your project to run in a console window. Next, you'll add a source code file and enter the code for your app.

I ran into a problem.

Add a source code file

1. In **Solution Explorer**, select the HelloWorld project. On the menu bar, choose **Project**, **Add New Item** to open the **Add New Item** dialog.

2. In the **Add New Item** dialog, select **Visual C++** under **Installed** if it isn't selected already. In the center pane, select **C++ file (.cpp)**. Change the **Name** to *HelloWorld.cpp*. Choose **Add** to close the dialog and create the file.


HelloWorld - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Debug
x86
Local Windows Debugger
Solution Explorer
Solution 'HelloWorld' (1 project)
HelloWorld
References
External Dependencies
Header Files
Resource Files
Source Files
Properties
HelloWorld Project Properties
Misc
(Name) HelloWorld
Project Dependencies
Project File C:\Users\Username\source
(Name)
Specifies the project name.
Creating project 'HelloWorld'... project creation successful.
Add to Source Control

Visual studio creates a new, empty source code file and opens it in an editor window, ready to enter your source code.

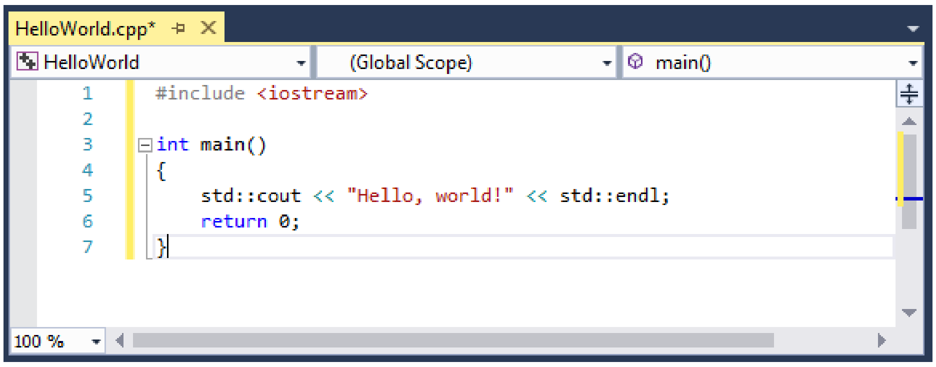
I ran into a problem.

Add code to the source file

1. Copy this code into the HelloWorld.cpp editor window.

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

The code should look like this in the editor window:



```
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

When the code looks like this in the editor, you're ready to go on to the next step and build your app.

I ran into a problem.

Next Steps

Build and run a C++ project

Troubleshooting guide

Come here for solutions to common issues when you create your first C++ project.

Create your app project issues

If the **New Project** dialog doesn't show a **Visual C++** entry under **Installed**, your copy of Visual Studio probably doesn't have the **Desktop development with C++** workload installed. You can run the installer right from the **New Project** dialog. Choose the **Open Visual Studio Installer** link to start the installer again. If the **User Account Control** dialog requests permissions, choose **Yes**. In the installer, make sure the **Desktop development with C++** workload is checked, and choose **OK** to update your Visual Studio installation.

If another project with the same name already exists, choose another name for your project, or delete the existing project and try again. To delete an existing project, delete the solution folder (the folder that contains the helloworld.sln file) in File Explorer.

Go back.

Make your project a console app issues

If you don't see **Linker** listed under **Configuration Properties**, choose **Cancel** to close the **Property Pages** dialog and then make sure that the **HelloWorld** project is selected in **Solution Explorer**, not the solution or another file or folder, before you try again.

The dropdown control does not appear in the **SubSystem** property edit box until you select the property. You can select it by using the pointer, or you can press Tab to cycle through the dialog controls until **SubSystem** is highlighted. Choose the dropdown control or press Alt+Down to open it.

Go back

Add a source code file issues

It's okay if you give the source code file a different name. However, don't add more than one source code file that contains the same code to your project.

If you added the wrong kind of file to your project, for example, a header file, delete it and try again. To delete the file, select it in **Solution Explorer** and press the Delete key.

Go back.

Add code to the source file issues

If you accidentally closed the source code file editor window, to open it again, double-click on HelloWorld.cpp in the **Solution Explorer** window.

If red squiggles appear under anything in the source code editor, check that your code matches the example in spelling, punctuation, and case. Case is significant in C++ code.

Go back.

https://docs.microsoft.com/

Build and run a C++ console app project

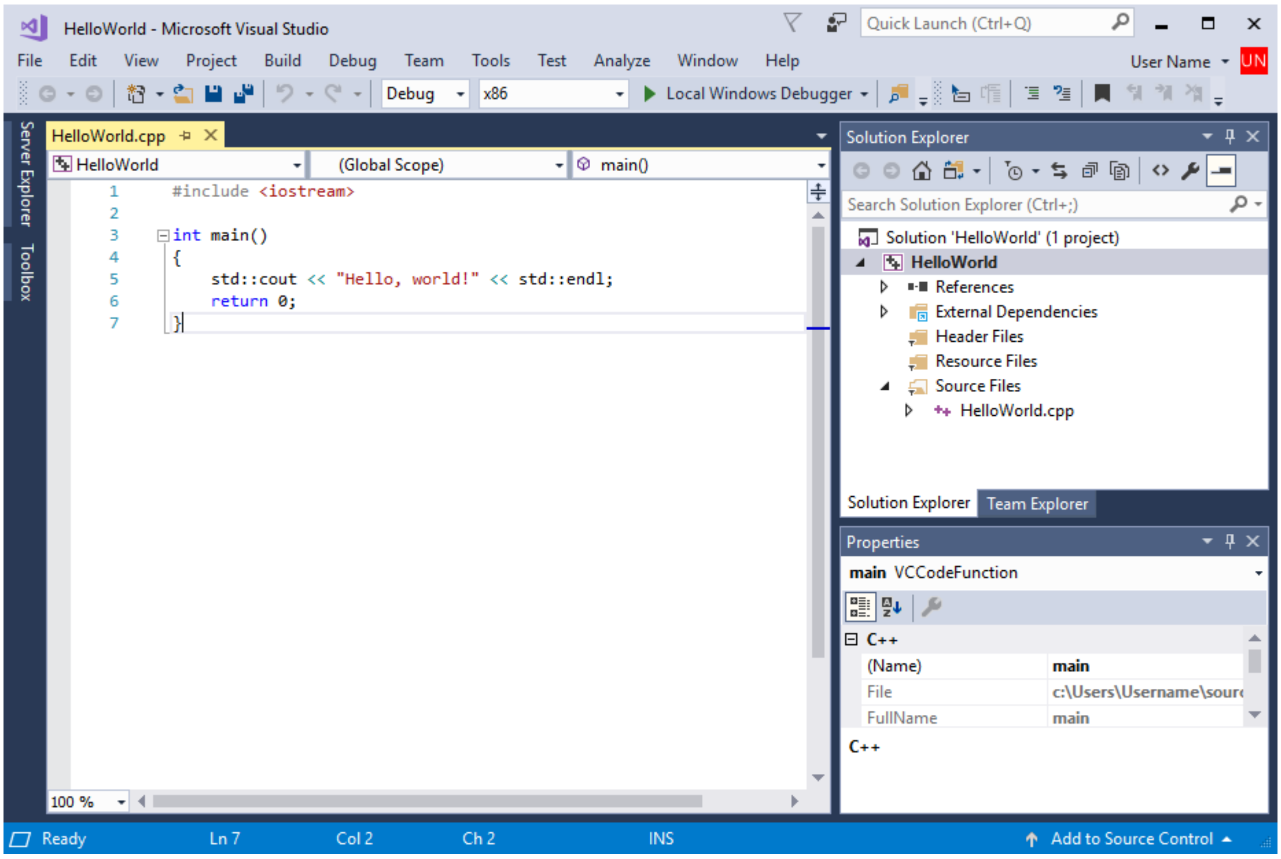
3/12/2019 • 3 minutes to read • Edit Online

When you've created a C++ console app project and entered your code, you can build and run it within Visual Studio, and then run it as a stand-alone app from the command line.

Prerequisites

- Have Visual Studio with the Desktop development with C++ workload installed and running on your computer. If it's not installed yet, follow the steps in Install C++ support in Visual Studio.
- Create a "Hello, World!" project and enter its source code. If you haven't done this yet, follow the steps in Create a C++ console app project.

If Visual Studio looks like this, you're ready to build and run your app:

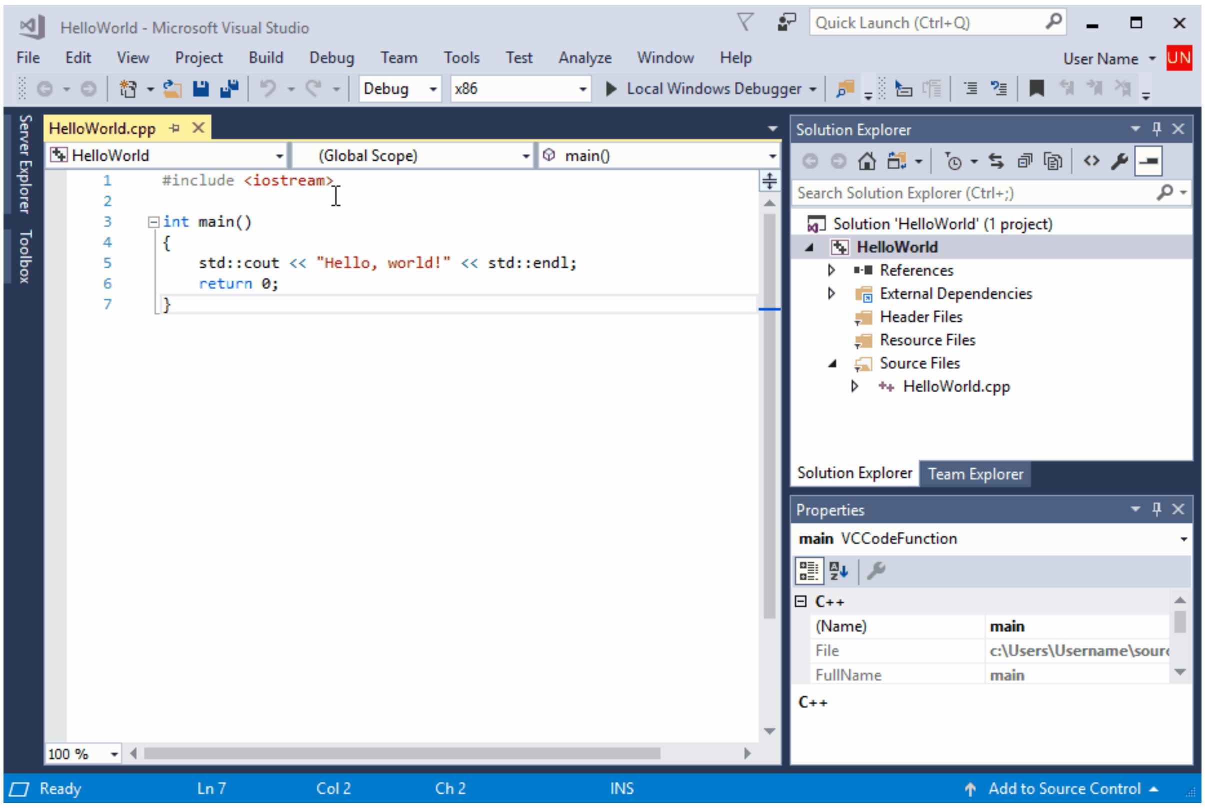


```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

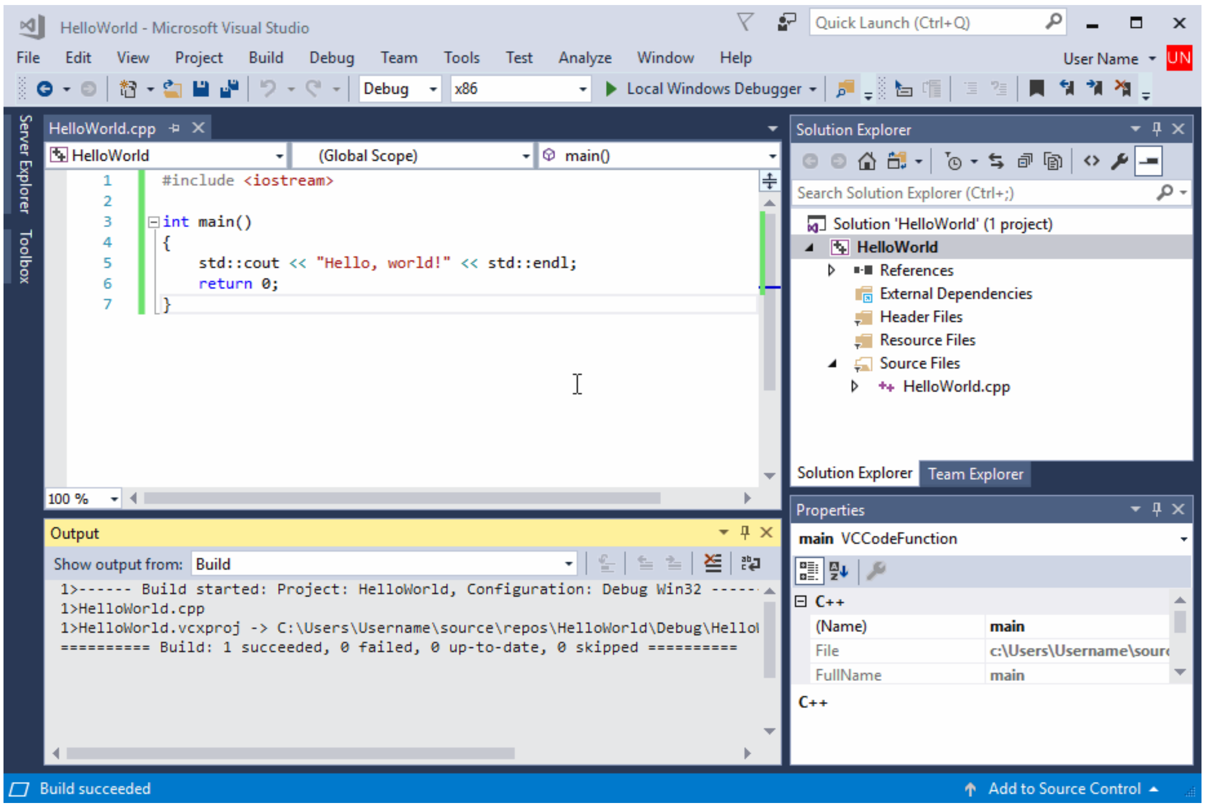
(Name)	main
File	c:\Users\Username\sourc
FullName	main

Build and run your code in Visual Studio

1. To build your project, choose **Build Solution** from the **Build** menu. The **Output** window shows the results of the build process.


```
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```
HelloWorld - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Test Analyze Window Help
User Name
Debug x86 Local Windows Debugger
HelloWorld.cpp
HelloWorld (Global Scope) main()
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'HelloWorld' (1 project)
HelloWorld
References
External Dependencies
Header Files
Resource Files
Source Files
HelloWorld.cpp
Solution Explorer Team Explorer
Properties
main VCCodeFunction
C++
(Name) main
File c:\Users\Username\sourc
FullName main
100 %
Ready Ln 7 Col 2 Ch 2 INS
Add to Source Control

2. To run the code, on the menu bar, choose **Debug**, **Start without debugging**.


```
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```
Output

Show output from: Build
```
1>------ Build started: Project: HelloWorld, Configuration: Debug Win32 -----
1>HelloWorld.cpp
1>HelloWorld.vcxproj -> C:\Users\Username\source\repos\HelloWorld\Debug\HelloW
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```
HelloWorld - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Test Analyze Window Help
User Name
Debug x86 Local Windows Debugger
HelloWorld.cpp
HelloWorld (Global Scope) main()
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'HelloWorld' (1 project)
HelloWorld
References
External Dependencies
Header Files
Resource Files
Source Files
HelloWorld.cpp
Solution Explorer Team Explorer
Properties
main VCCodeFunction
C++
(Name) main
File c:\Users\Username\sourc
FullName main
100 %
Build succeeded
Add to Source Control

A console window opens and then runs your app. When you start a console app in Visual Studio, it runs your code, then prints "Press any key to continue . . ." to give you a chance to see the output.

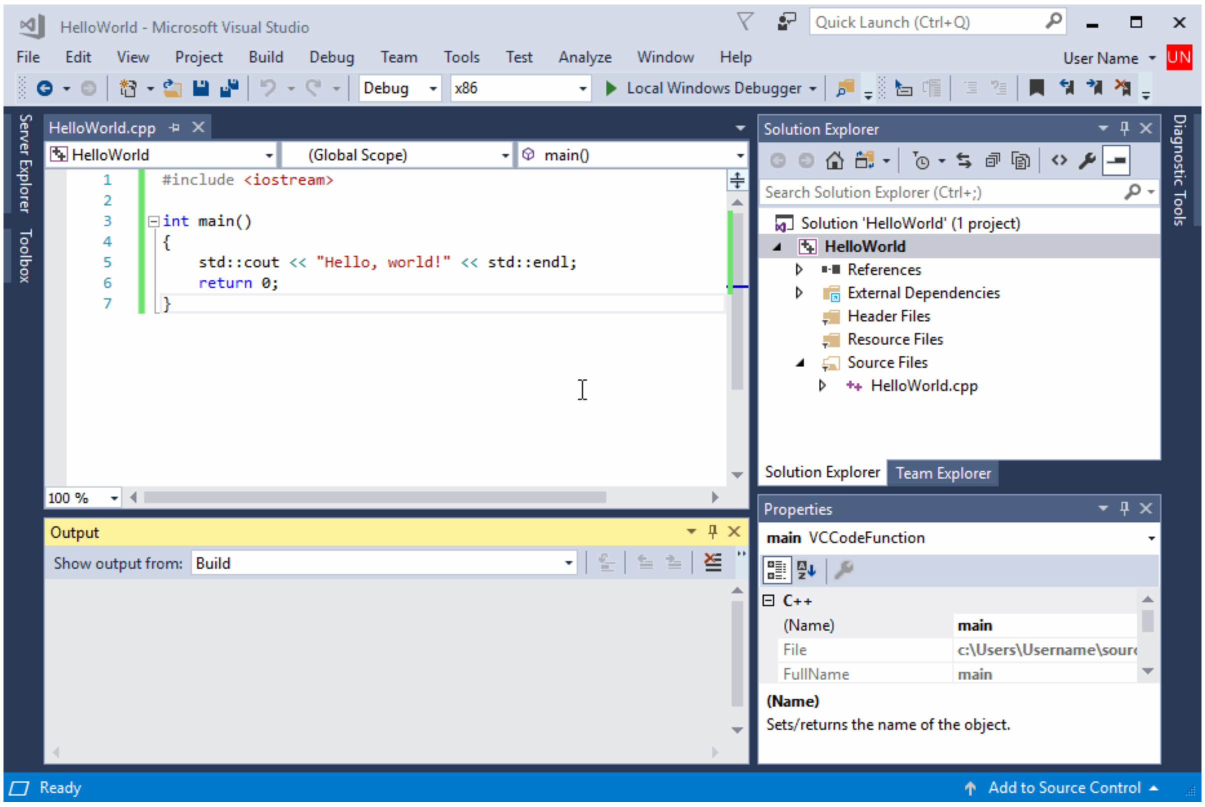
Congratulations! You've created your first "Hello, world!" console app in Visual Studio! Press a key to dismiss the console window and return to Visual Studio.

I ran into a problem.

Run your code in a command window

Normally, you run console apps at the command prompt, not in Visual Studio. Once your app is built by Visual Studio, you can run it from any command window. Here's how to find and run your new app in a command prompt window.

1. In **Solution Explorer**, select the HelloWorld solution and right-click to open the context menu. Choose **Open Folder in File Explorer** to open a **File Explorer** window in the HelloWorld solution folder.
2. In the **File Explorer** window, open the Debug folder. This contains your app, HelloWorld.exe, and a couple of other debugging files. Select HelloWorld.exe, hold down the Shift key and right-click to open the context menu. Choose **Copy as path** to copy the path to your app to the clipboard.
3. To open a command prompt window, press Windows-R to open the **Run** dialog. Enter *cmd.exe* in the **Open** textbox, then choose **OK** to run a command prompt window.
4. In the command prompt window, right-click to paste the path to your app into the command prompt. Press Enter to run your app.



```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Congratulations, you've built and run a console app in Visual Studio!

I ran into a problem.

Next Steps

Once you've built and run this simple app, you're ready for more complex projects. See Using the Visual Studio IDE for C++ Desktop Development for more detailed walkthroughs that explore the capabilities of Visual C++ in Visual Studio.

Troubleshooting guide

Come here for solutions to common issues when you create your first C++ project.

Build and run your code in Visual Studio issues

If red squiggles appear under anything in the source code editor, the build may have errors or warnings. Check that your code matches the example in spelling, punctuation, and case.

Go back.

Run your code in a command window issues

You can also navigate to the solution Debug folder at the command line to run your app. You can't run your app from other directories without specifying the path to the app. However, you can copy your app to another directory and run it from there.

If you don't see **Copy as path** in the shortcut menu, dismiss the menu, and then hold down the Shift key while you open it again. This is just for convenience. You can also copy the path to the folder from the File Explorer search bar, and paste it into the **Run** dialog, and then enter the name of your executable at the end. It's just a little more typing, but it has the same result.

Go back.

https://docs.microsoft.com/

C/C++ projects and build systems in Visual Studio

5/7/2019 • 4 minutes to read • Edit Online

You can use Visual Studio 2017 to edit, compile and build any C++ code base with full IntelliSense support without having to convert that code into a Visual Studio project or compile with the MSVC toolset. For example, you can edit a cross-platform CMake project in Visual Studio on a Windows machine, then compile it for Linux using g++ on a remote Linux machine.

C++ compilation

To *build* a C++ program means to compile source code from one or more files and then link those files into an executable file (.exe), a dynamic-load library (.dll) or a static library (.lib).

Basic C++ compilation involves three main steps:

- The C++ preprocessor transforms all the #directives and macro definitions in each source file. This creates a *translation unit*.
- The C++ compiler compiles each translation unit into object files (.obj), applying whatever compiler options have been set.
- The *linker* merges the object files into a single executable, applying the linker options that have been set.

The MSVC toolset

The Microsoft C++ compiler, linker, standard libraries, and related utilities comprise the MSVC compiler toolset (also called a toolchain or "build tools"). These are included in Visual Studio. You can also download and use the toolset as a standalone package for free from the Build Tools for Visual Studio 2017 download location.

You can build simple programs by invoking the MSVC compiler (cl.exe) directly from the command line. The following command accepts a single source code file, and invokes cl.exe to build an executable called *hello.exe*:

```
cl /EHsc hello.cpp
```

Note that here the compiler (cl.exe) automatically invokes the C++ preprocessor and the linker to produce the final output file. For more information, see Building on the command line.

Build systems and projects

Most real-world programs use some kind of *build system* to manage complexities of compiling multiple source files for multiple configurations (i.e. debug vs. release), multiple platforms (x86, x64, ARM, and so on), custom build steps, and even multiple executables that must be compiled in a certain order. You make settings in a build configuration file(s), and the build system accepts that file as input before it invoke the compiler. The set of source code files and build configuration files needed to build an executable file is called a *project*.

The following list shows various options for Visual Studio Projects - C++:

- create a Visual Studio project by using the Visual Studio IDE and configure it by using property pages. Visual Studio projects produce programs that run on Windows. For an overview, see Compiling and Building in the Visual Studio documentation.

- open a folder that contains a CMakeLists.txt file. CMake support is integrated into Visual Studio. You can use the IDE to edit, test and debug without modifying the CMake files in any way. This enables you to work in the same CMake project as others who might be using different editors. CMake is the recommended approach for cross-platform development. For more information, see CMake projects.

- open a loose folder of source files with no project file. Visual Studio will use heuristics to build the files. This is an easy way to compile and run small console applications. For more information, see Open Folder projects.
- open a folder that contains a makefile, or any other build system configuration file. You can configure Visual Studio to invoke any arbitrary build commands by adding JSON files to the folder. For more information, see Open Folder projects.
- Open a Windows makefile in Visual Studio. For more information, see NMAKE Reference.

MSBuild from the command line

You can invoke MSBuild from the command line by passing it a .vcxproj file along with command-line options. This approach requires a good understanding of MSBuild, and is recommended only when absolutely necessary. For more information, see MSBuild.

In This Section

Visual Studio projects How to create, configure, and build C++ projects in Visual Studio using its native build system (MSBuild).

CMake projects How to code, build, and deploy CMake projects in Visual Studio.

Open Folder projects How to use Visual Studio to code, build and deploy C++ projects based on any arbitrary build system, or no build system. at all.

Release builds How to create and troubleshoot optimized release builds for deployment to end users.

Use the MSVC toolset from the command line
Discusses how to use the C/C++ compiler and build tools directly from the command line rather than using the Visual Studio IDE.

Building DLLs in Visual Studio How to create, debug and deploy C/C++ DLLs (shared libraries) in Visual Studio.

Walkthrough: Creating and Using a Static Library How to create a .lib binary file.

Building C/C++ Isolated Applications and Side-by-side Assemblies Describes the deployment model for Windows Desktop applications, based on the idea of isolated applications and side-by-side assemblies.

Configure C++ projects for 64-bit, x64 targets How to target 64-bit x64 hardware with the MSVC build tools.

Configure C++ projects for ARM processors How to use the MSVC build tools to target ARM hardware.

Optimizing Your Code How to optimize your code in various ways including program guided optimizations.

Configuring Programs for Windows XP How to target Windows XP with the MSVC build tools.

C/C++ Building Reference
Provides links to reference articles about program building in C++, compiler and linker options, and various build tools.

Writing and refactoring code (C++)

5/15/2019 • 4 minutes to read • Edit Online

The C++ code editor and Visual Studio IDE provide many coding aids. Some are unique to C++, and some are essentially the same for all Visual Studio languages. For more information about the shared features, see Writing Code in the Code and Text Editor. Options for enabling and configuring C++-specific features are located under **Tools | Options | Text Editor | C/C++**. After choosing which option you want to set, you can get more help by pressing **F1** when the dialog is in focus. For general code formatting options, type `Editor C++` into **QuickLaunch**.

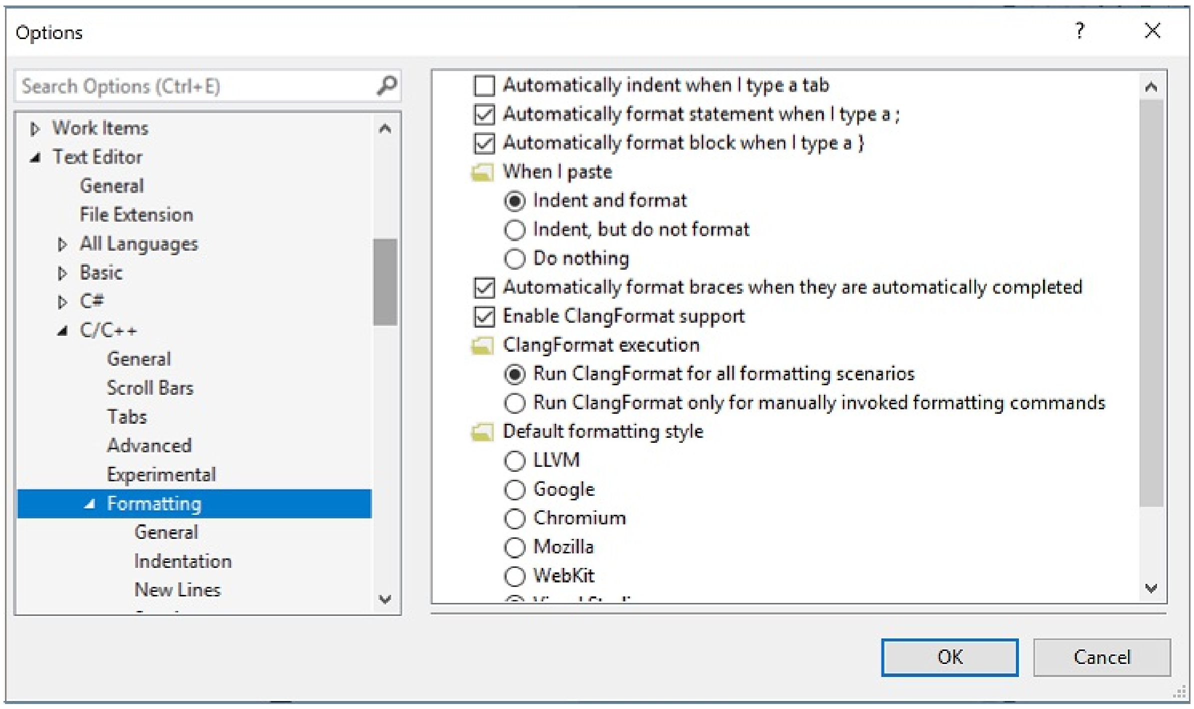
Experimental features, which may or may not be included in a future version of Visual Studio, are found in the Text Editor C++ Experimental dialog. In Visual Studio 2017 you can enable **Predictive IntelliSense** in this dialog.

Adding new files

To add new files to a project, right-click on the project node in Solution Explorer and choose **Add | New**.

Formatting options

To set formatting options such as indents, brace completion, and colorization, type "C++ Formatting" into the **QuickLaunch** window. Visual Studio 2017 version 15.7 and later supports ClangFormat. You can configure it in the C/C++ Formatting Property Page under **Tools | Options | Text Editor | C/C++ | Formatting**.



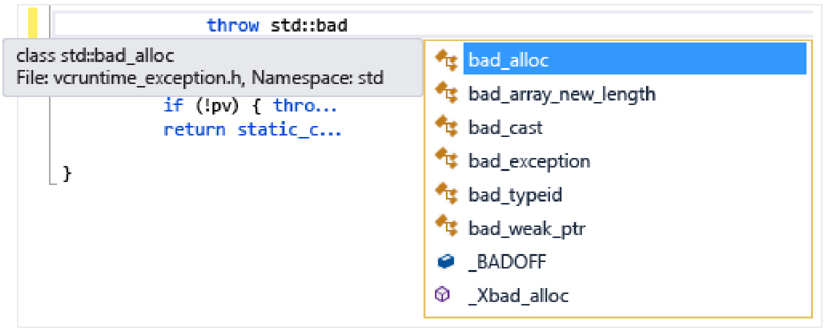
☐ Automatically indent when I type a tab
☑ Automatically format statement when I type a ;
☑ Automatically format block when I type a }
When I paste
- ● Indent and format
- ○ Indent, but do not format
- ○ Do nothing

☑ Automatically format braces when they are automatically completed
☑ Enable ClangFormat support
ClangFormat execution
- ● Run ClangFormat for all formatting scenarios
- ○ Run ClangFormat only for manually invoked formatting commands

Default formatting style
- ○ LLVM
- ○ Google
- ○ Chromium
- ○ Mozilla
- ○ WebKit

IntelliSense

IntelliSense is the name for a set of features that provide inline information about members, types, and function overloads. The following illustration shows the member list drop-down that appears as you type. You can press the tab key to enter the selected item text into your code file.



```
throw std::bad
```

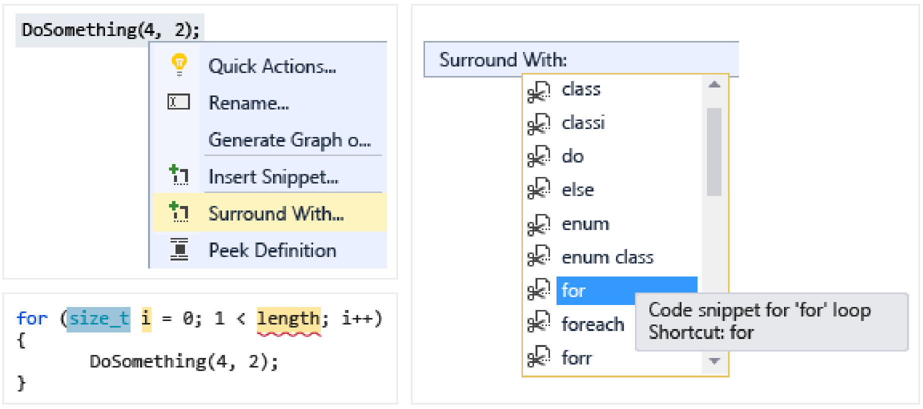
class std::bad_alloc
File: vcruntime_exception.h, Namespace: std

```
if (!pv) { thro...
return static_c...
}
```

For complete information see Visual C++ IntelliSense.

Insert Snippets

A snippet is a predefined piece of source code. Right-click on a single point or on selected text to either insert a snippet or surround the selected text with the snippet. The following illustration shows the three steps to surround a selected statement with a for loop. The yellow highlights in the final image are editable fields that you access with the tab key. For more information, see Code Snippets.



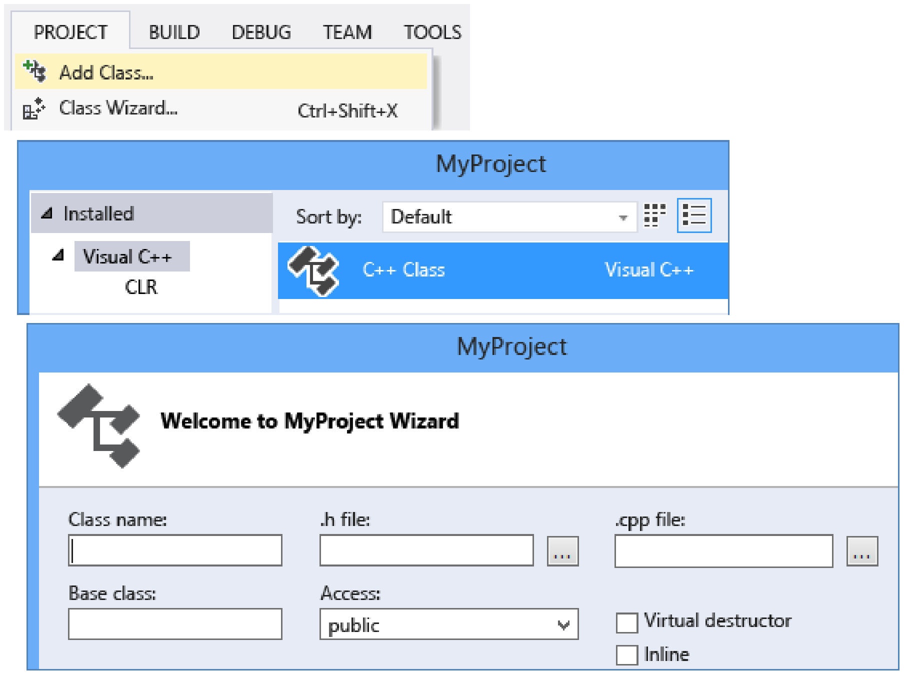
```
DoSomething(4, 2);
```

Code snippet for 'for' loop
Shortcut: for

```
for (size_t i = 0; i < length; i++)
{
        DoSomething(4, 2);
}
```

Add Class

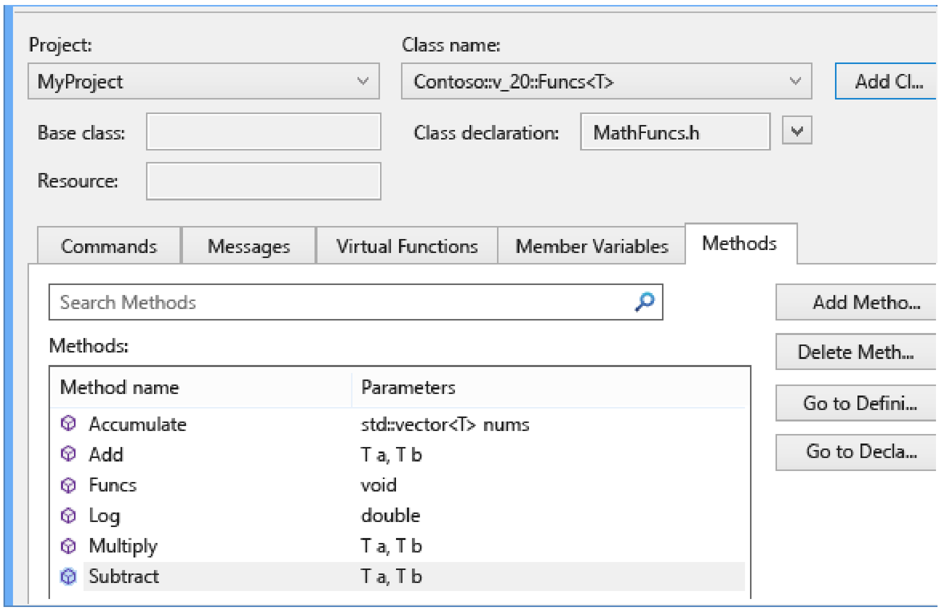
Add a new class from the **Project** menu by using the Class Wizard.



Welcome to MyProject Wizard

Class name:
.h file:
.cpp file:
Base class:
Access: public
Virtual destructor
Inline

You can also use Class Wizard to modify or examine an existing class.


Project: MyProject
Class name: Contoso::v_20::Funcs<T>
Add Cl...
Base class:
Class declaration: MathFuncs.h
Resource:

Commands | Messages | Virtual Functions | Member Variables | Methods

Search Methods

Add Metho...
Delete Meth...
Go to Defini...
Go to Decla...

Methods:

Method name	Parameters
Accumulate	std::vector<T> nums
Add	T a, T b
Funcs	void
Log	double
Multiply	T a, T b
Subtract	T a, T b

For more information, see Adding Functionality with Code Wizards (C++).

Refactoring

Refactorings are available under the Quick Action context menu, or by clicking on a light bulb in the editor. Some are also found in the **Edit > Refactor** menu. These features include:

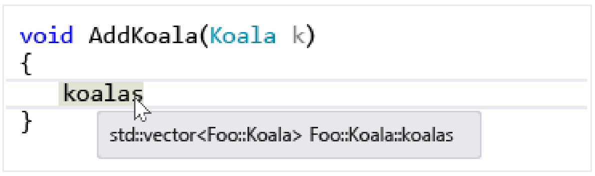
- Rename
- Extract Function
- Implement Pure Virtuals
- Create Declaration / Definition
- Move Function Definition
- Convert to Raw String Literal
- Change Signature

Navigate and understand

Visual C++ shares many code navigation features with other languages. For more information, see Navigating Code and Viewing the Structure of Code.

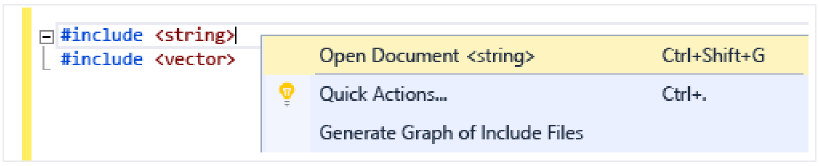
QuickInfo

Hover over a variable to see its type information.


```
void AddKoala(Koala k)
{
    koalas
}
```
std::vector<Foo::Koala> Foo::Koala::koalas

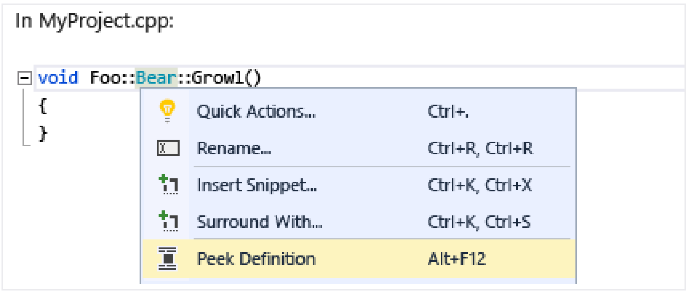
Open document (Navigate to header)

Right click on the header name in an `#include` directive and open the header file.

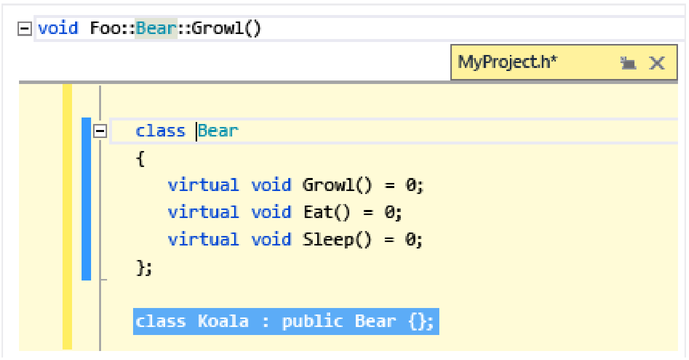

```
#include <string>
#include <vector>
```
Open Document <string>   Ctrl+Shift+G
Quick Actions...   Ctrl+.
Generate Graph of Include Files

Peek Definition

Hover over a variable or function declaration, right-click, then choose **Peek Definition** to see an inline view of its definition. For more information, see Peek Definition (Alt+F12).


In MyProject.cpp:

```
void Foo::Bear::Growl()
{
}
```



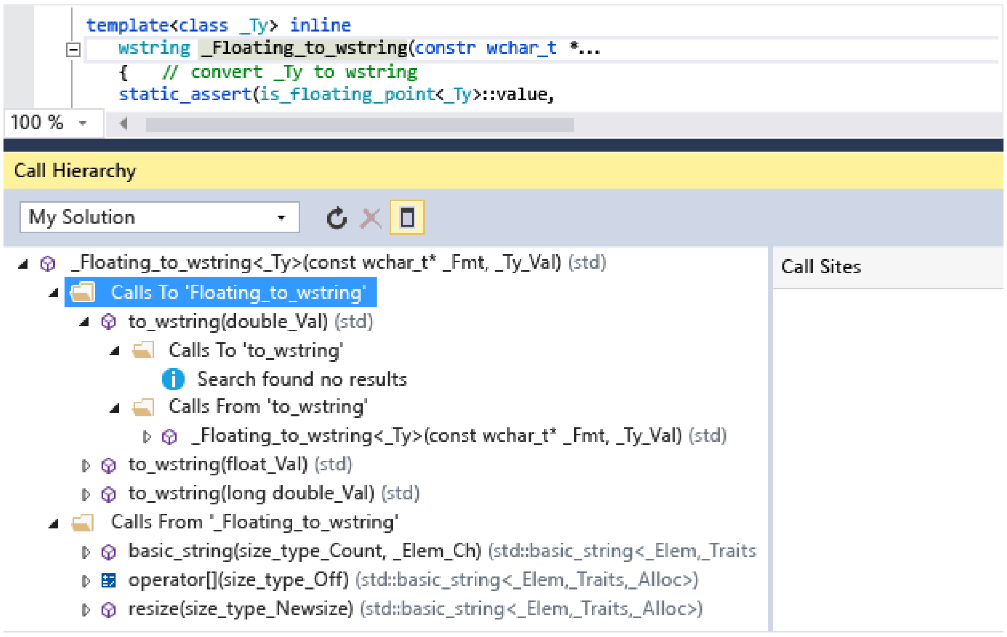
```
void Foo::Bear::Growl()

class Bear
{
    virtual void Growl() = 0;
    virtual void Eat() = 0;
    virtual void Sleep() = 0;
};

class Koala : public Bear {};
```

Go To Definition

Hover over a variable or function declaration, right-click, then choose **Go To Definition** to open the document where the object is defined.

View Call Hierarchy

Right click on any function call and view a recursive list of all the functions that it calls, and all the functions that call it. Each function in the list can be expanded in the same way. For more information, see Call Hierarchy.



```
template<class _Ty> inline
    wstring _Floating_to_wstring(constr wchar_t *...
    {   // convert _Ty to wstring
    static_assert(is_floating_point<_Ty>::value,
```

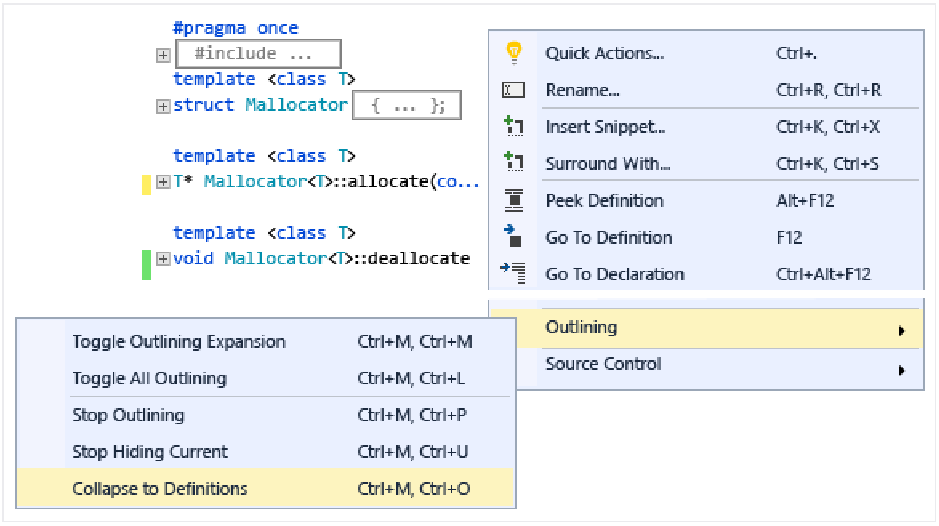
```
◢ _Floating_to_wstring<_Ty>(const wchar_t* _Fmt, _Ty_Val) (std)
  ◢ Calls To '_Floating_to_wstring'
    ◢ to_wstring(double_Val) (std)
      ◢ Calls To 'to_wstring'
          Search found no results
      ◢ Calls From 'to_wstring'
        ▷ _Floating_to_wstring<_Ty>(const wchar_t* _Fmt, _Ty_Val) (std)
    ▷ to_wstring(float_Val) (std)
    ▷ to_wstring(long double_Val) (std)
  ◢ Calls From '_Floating_to_wstring'
    ▷ basic_string(size_type_Count, _Elem_Ch) (std::basic_string<_Elem,_Traits
    ▷ operator[](size_type_Off) (std::basic_string<_Elem,_Traits,_Alloc>)
    ▷ resize(size_type_Newsize) (std::basic_string<_Elem,_Traits,_Alloc>)
```

Toggle Header / Code File

Right-click and choose **Toggle Header / Code File** to switch back and forth between a header file and its associated code file.

Outlining

Right-click anywhere in a source code file and choose **Outlining** to collapse or expand definitions and/or custom regions to make it easier to browse only the parts you are interested in. For more information, see Outlining.



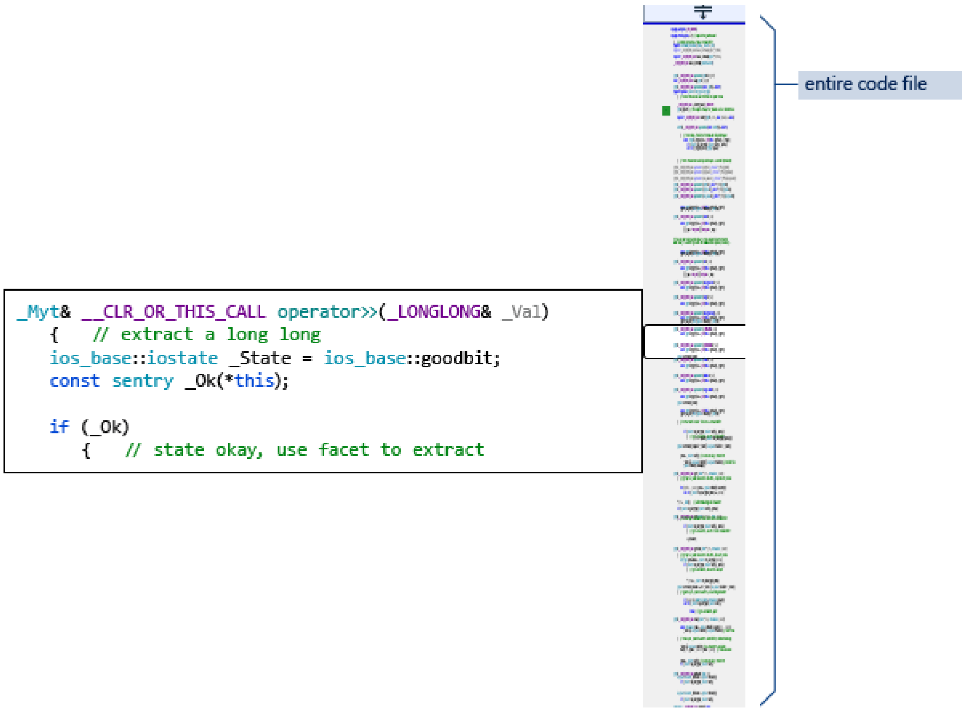
```
#pragma once
⊞ #include ...
template <class T>
⊞struct Mallocator { ... };

template <class T>
⊞T* Mallocator<T>::allocate(co...

template <class T>
⊞void Mallocator<T>::deallocate
```

Quick Actions...	Ctrl+.
Rename...	Ctrl+R, Ctrl+R
Insert Snippet...	Ctrl+K, Ctrl+X
Surround With...	Ctrl+K, Ctrl+S
Peek Definition	Alt+F12
Go To Definition	F12
Go To Declaration	Ctrl+Alt+F12
Outlining	▸
Source Control	▸

Toggle Outlining Expansion	Ctrl+M, Ctrl+M
Toggle All Outlining	Ctrl+M, Ctrl+L
Stop Outlining	Ctrl+M, Ctrl+P
Stop Hiding Current	Ctrl+M, Ctrl+U
Collapse to Definitions	Ctrl+M, Ctrl+O

Scrollbar map mode

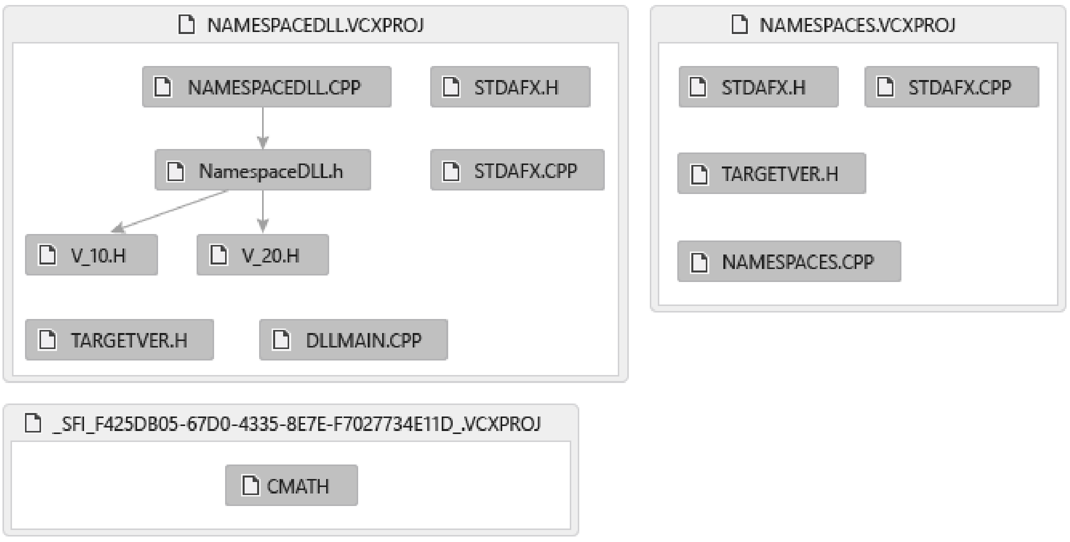
Scrollbar map mode enables you to quickly scroll and browse through a code file without actually leaving your current location. Or click anywhere on the code map to go directly to that location. For more information, see How to: Track your code by customizing the scrollbar.



```
_Myt& __CLR_OR_THIS_CALL operator>>(_LONGLONG& _Val)
    {   // extract a long long
    ios_base::iostate _State = ios_base::goodbit;
    const sentry _Ok(*this);

    if (_Ok)
        {   // state okay, use facet to extract
```

Generate graph of include files

Right click on a code file in your project and choose **Generate graph of include files** to see a graph of which files are included by other files.

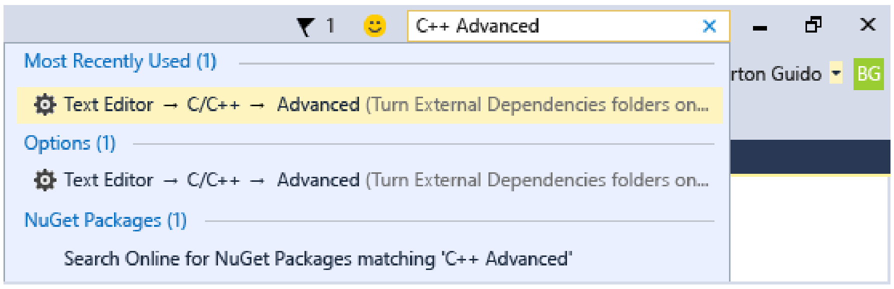


F1 Help

Place the cursor on or just after any type, keyword or function and press F1 to go directly to the relevant reference topic on docs.microsoft.com. F1 also works on items in the error list, and in many dialog boxes.

Quick Launch

To easily navigate to any window or tool in Visual Studio, simply type its name in the Quick Launch window in the upper right corner of the UI. The auto-completion list will filter as you type.


1
C++ Advanced
Most Recently Used (1)
Text Editor → C/C++ → Advanced (Turn External Dependencies folders on...
Options (1)
Text Editor → C/C++ → Advanced (Turn External Dependencies folders on...
NuGet Packages (1)
Search Online for NuGet Packages matching 'C++ Advanced'
rton Guido
BG

Overview of Windows Programming in C++

5/24/2019 • 7 minutes to read • Edit Online

There are several broad categories of Windows applications that you can create with C++. Each has its own programming model and set of Windows-specific libraries, but the C++ standard library and third-party C++ libraries can be used in any of them.

Command line (console) applications

C++ console applications run from the command line in a console window and can display text output only. For more information, see Console Applications.

Native desktop client applications

A *native desktop client application* is a C or C++ windowed application that uses the original native Windows C APIs or Component Object Model (COM) APIs to access the operating system. Those APIs are themselves written mostly in C. There's more than one way to create a native desktop app: You can program using the Win32 APIs directly, using a C-style message loop that processes operating system events. Or, you can program using *Microsoft Foundation Classes* (MFC), a lightly object-oriented C++ library that wraps Win32. Neither approach is considered "modern" compared to the Universal Windows Platform (UWP), but both are still fully supported and have millions of lines of code running in the world today. A Win32 application that runs in a window requires the developer to work explicitly with Windows messages inside a Windows procedure function. Despite the name, a Win32 application can be compiled as a 32-bit (x86) or 64-bit (x64) binary. In the Visual Studio IDE, the terms x86 and Win32 are synonymous.

To get started with traditional Windows C++ programming, see Get Started with Win32 and C++. After you gain some understanding of Win32, it will be easier to learn about MFC Desktop Applications. For an example of a traditional C++ desktop application that uses sophisticated graphics, see Hilo: Developing C++ Applications for Windows.

C++ or .NET?

In general, .NET programming in C# is less complex, less error-prone, and has a more modern object-oriented API than Win32 or MFC. In most cases, its performance is more than adequate. .NET features the Windows Presentation Foundation (WPF) for rich graphics, and you can consume both Win32 and the modern Windows Runtime API. As a general rule, we recommend using C++ for desktop applications when you require:

- precise control over memory usage
- the utmost economy in power consumption
- usage of the GPU for general computing
- access to DirectX
- heavy usage of standard C++ libraries

It's also possible to combine the power and efficiency of C++ with .NET programming. You can create a user interface in C# and use C++/CLI to enable the application to consume native C++ libraries. For more information, see .NET Programming with C++/CLI.

COM Components

The Component Object Model (COM) is a specification that enables programs written in different languages to communicate with one another. Many Windows components are implemented as COM objects and follow standard COM rules for object creation, interface discovery, and object destruction. Using COM objects from C++ desktop applications is relatively straightforward, but writing your own COM object is more advanced. The Active Template Library (ATL) provides macros and helper functions that simplify COM development. For more information, see ATL COM desktop components.

Universal Windows Platform apps

The Universal Windows Platform (UWP) is the modern Windows API. UWP apps run on any Windows 10 device, use XAML for the user-interface, and are fully touch-enabled. For more information about UWP, see What's a Universal Windows Platform (UWP) app? and Guide to Windows Universal Apps.

The original C++ support for UWP consisted of (1) C++/CX, a dialect of C++ with syntax extensions, or (2) the Windows Runtime Library (WRL), which is based on standard C++ and COM. Both C++/CX and WRL are still supported. For new projects, we recommend C++/WinRT, which is entirely based on standard C++ and provides faster performance.

Desktop Bridge

In Windows 10, you can package your existing desktop application or COM object as a UWP app, and add UWP features such as touch, or call APIs from the modern Windows API set. You can also add a UWP app to a desktop solution in Visual Studio, and package them together in a single package and use Windows APIs to communicate between them.

Visual Studio 2017 version 15.4 and later lets you create a Windows Application Package Project to greatly simplify the work of packaging your existing desktop application. A few restrictions apply to the registry calls or APIs your desktop application can use. However, in many cases you can create alternate code paths to achieve similar functionality while running in an app package. For more information, see Desktop Bridge.

Games

DirectX games can run on the PC or Xbox. For more information, see DirectX Graphics and Gaming.

SQL Server database clients

To access SQL Server databases from native code, use ODBC or OLE DB. For more information, see SQL Server Native Client.

Windows device drivers

Drivers are low-level components that make data from hardware devices accessible to applications and other operating system components. For more information, see Windows Driver Kit (WDK).

Windows services

A Windows *service* is a program that can run in the background with little or no user interaction. These programs are called *daemons* on UNIX systems. For more information, see Services.

SDKs, libraries, and header files

Visual Studio includes the C Runtime Library (CRT), the C++ Standard Library, and other Microsoft-specific libraries. Most of the include folders that contain header files for these libraries are located in the Visual Studio installation directory under the \VC\ folder. The Windows and CRT header files are found in the Windows SDK installation folder.

The Vcpkg package manager lets you conveniently install hundreds of third-party open-source libraries for

Windows.

The Microsoft libraries include:

- Microsoft Foundation Classes (MFC): An object-oriented framework for creating traditional Windows programs—especially enterprise applications—that have rich user interfaces that feature buttons, list boxes, tree views, and other controls. For more information, see MFC Desktop Applications.
- Active Template Library (ATL): A powerful helper library for creating COM components. For more information, see ATL COM Desktop Components.
- C++ AMP (C++ Accelerated Massive Parallelism): A library that enables high-performance general computational work on the GPU. For more information, see C++ AMP (C++ Accelerated Massive Parallelism).
- Concurrency Runtime: A library that simplifies the work of parallel and asynchronous programming for multicore and many-core devices. For more information, see Concurrency Runtime.

Many Windows programming scenarios also require the Windows SDK, which includes the header files that enable access to the Windows operating system components. By default, Visual Studio installs the Windows SDK as a component of the C++ Desktop workload, which enables development of Universal Windows apps. To develop UWP apps, you need the Windows 10 version of the Windows SDK. For information, see Windows 10 SDK. (For more information about the Windows SDKs for earlier versions of Windows, see the Windows SDK archive).

Program Files (x86)\Windows Kits is the default location for all versions of the Windows SDK that you've installed.

Other platforms such as Xbox and Azure have their own SDKs that you may have to install. For more information, see the DirectX Developer Center and the Azure Developer Center.

Development Tools

Visual Studio includes a powerful debugger for native code, static analysis tools, graphics debugging tools, a full-featured code editor, support for unit tests, and many other tools and utilities. For more information, see Get started developing with Visual Studio, and Overview of C++ development in Visual Studio.

In this section

TITLE	DESCRIPTION
Walkthrough: Creating a Standard C++ Program	Create a Windows console application.
Walkthrough: Creating Windows Desktop Applications (C++)	Create a native Windows desktop application.
Windows Desktop Wizard	Use the wizard to create new Windows projects.
Active Template Library (ATL)	Use the ATL library to create COM components in C++.
Microsoft Foundation Classes (MFC)	Use MFC to create large or small Windows applications with dialogs and controls
ATL and MFC Shared Classes	Use classes such as CString that are shared in ATL and MFC.
Data Access	OLE DB and ODBC

TITLE	DESCRIPTION
Text and Strings	Various string types on Windows.
Resources for Creating a Game Using DirectX	
How to: Use the Windows 10 SDK in a Windows Desktop Application	Windows SDK
Working with Resource Files	How to add images, icons, string tables, and other resources to a desktop application.
Resources for Creating a Game Using DirectX (C++)	Links to content for creating games in C++.
How to: Use the Windows 10 SDK in a Windows Desktop Application	Contains steps for setting up your project to build using the Windows 10 SDK.
Deploying Native Desktop Applications	Deploy native applications on Windows.

Related Articles

TITLE	DESCRIPTION
C++ in Visual Studio	Parent topic for Visual C++ developer content.
.NET Development with C++/CLI	Create wrappers for native C++ libraries that enable it to communication with .NET applications and components.
Component Extensions for .NET and UWP	Reference for syntax elements shared by C++/CX and C++/CLI.
Universal Windows Apps (C++)	Write UWP applications using C++/CX or Windows Runtime Template Library (WRL).
C++ Attributes for COM and .NET	Non-standard attributes for Windows-only programming using .NET or COM.

Universal Windows Apps (C++)

4/1/2019 • 2 minutes to read • Edit Online

The Universal Windows Platform (UWP) is the modern programming interface for Windows. With UWP you write an application or component once and deploy it on any Windows 10 device. You can write a component in C++ and applications written in any other UWP-compatible language can use it.

Most of the UWP documentation is in the Windows content tree at Universal Windows Platform documentation. There you will find beginning tutorials as well as reference documentation.

For new UWP apps and components, we recommend that you use C++/WinRT, a new standard C++17 language projection for Windows Runtime APIs. C++/WinRT is available in the Windows 10 SDK from version 1803 onward. C++/WinRT is implemented entirely in header files, and is designed to provide you with first-class access to the modern Windows API. Unlike the C++/CX implementation. C++/WinRT doesn't use non-standard syntax or Microsoft language extensions, and it takes full advantage of the C++ compiler to create highly-optimized output. For more information, see Introduction to C++/WinRT.

You can use the Desktop Bridge app converter to package your existing desktop application for deployment through the Microsoft Store. For more information, see Using Visual C++ Runtime in Centennial project and Desktop Bridge.

UWP apps that use C++/CX

Visual C++ language reference (C++/CX)	Describes the set of extensions that simplify C++ consumption of Windows Runtime APIs and enable error handling that's based on exceptions.
Building apps and libraries (C++/CX)	Describes how to create DLLs and static libraries that can be accessed from a C++/CX app or component.
Tutorial: Create a UWP "Hello, World" app in C++/CX	A walkthrough that introduces the basic concepts of UWP app development in C++/CX.
Creating Windows Runtime Components in C++/CX	Describes how to create DLLs that other UWP apps and components can consume.
UWP game programming	Describes how to use DirectX and C++/CX to create games.

UWP Apps that Use the Windows Runtime C++ Template Library (WRL)

The Windows Runtime C++ Template Library provides the low-level COM interfaces by which ISO C++ code can access the Windows Runtime in an exception-free environment. In most cases, we recommend that you use C++/WinRT or C++/CX instead of the Windows Runtime C++ Template Library for UWP app development. For information about the Windows Runtime C++ Template Library, see Windows Runtime C++ Template Library (WRL).

See also

C++ in Visual Studio
Overview of Windows Programming in C++

Game Development with C++

4/1/2019 • 2 minutes to read • Edit Online

When you create a Windows 10 game, you have the opportunity to reach millions of players worldwide across phone, PC, and Xbox One. With Xbox on Windows, Xbox Live, cross-device multiplayer, an amazing gaming community, and powerful new features like the Universal Windows Platform (UWP) and DirectX 12, Windows 10 games thrill players of all ages and genres. The new Universal Windows Platform (UWP) delivers compatibility for your game across Windows 10 devices with a common API for phone, PC, and Xbox One, along with tools and options to tailor your game to each device experience.

Game development is documented on the Windows Dev Center.

Download, install, and set up the Linux workload

3/11/2019 • 3 minutes to read • Edit Online

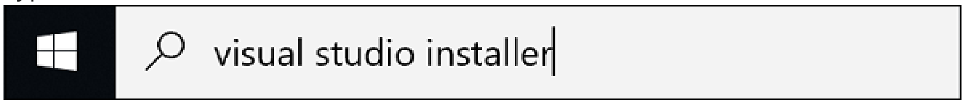
You can use the Visual Studio 2017 IDE in Windows to create, edit and debug C++ projects that execute on a Linux physical computer, virtual machine, or the Windows Subsystem for Linux.

You can work on your existing code base that uses CMake or any other build system without having to convert it to a Visual Studio project. If your code base is cross-platform, you can target both Windows and Linux from within Visual Studio. For example, you can edit, debug and profile your code on Windows using Visual Studio, then quickly retarget the project for Linux to do further testing. The Linux header files are automatically copied to your local machine where Visual Studio uses them to provide full IntelliSense support (Statement Completion, Go to Definition, and so on).

For any of these scenarios, the **Linux development with C++** workload is required.

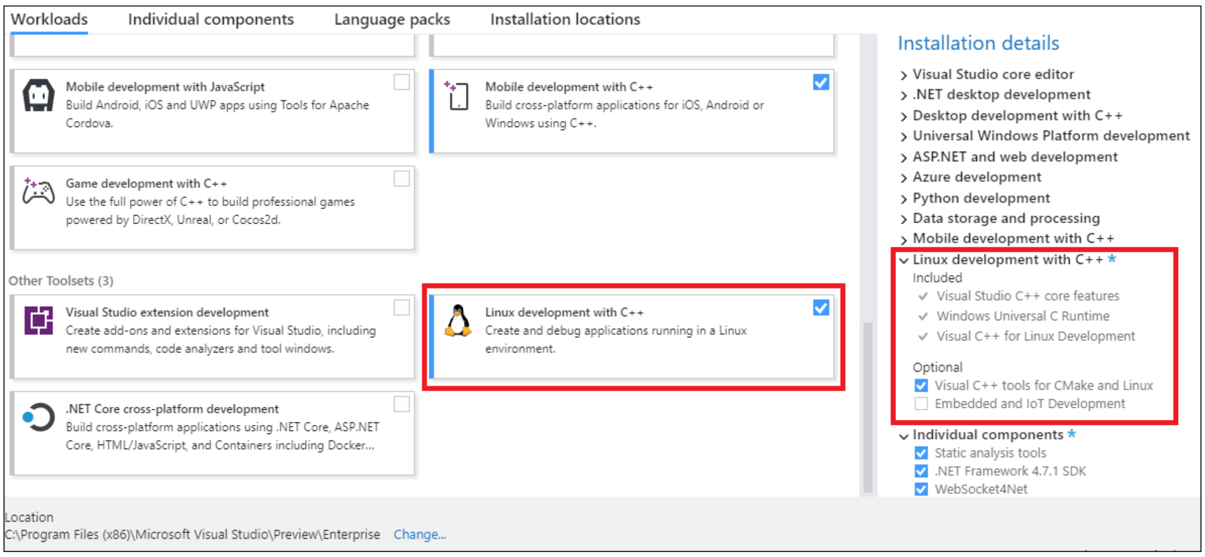
Visual Studio setup

1. Type "Visual Studio Installer" in the Windows search box:



visual studio installer

2. Look for the installer under the **Apps** results and double-click it. When the installer opens, choose **Modify**, and then click on the **Workloads** tab. Scroll down to **Other toolsets** and select the **Linux development with C++** workload.



Workloads | Individual components | Language packs | Installation locations

Mobile development with JavaScript
Build Android, iOS and UWP apps using Tools for Apache Cordova.

Mobile development with C++ ✓
Build cross-platform applications for iOS, Android or Windows using C++.

Game development with C++
Use the full power of C++ to build professional games powered by DirectX, Unreal, or Cocos2d.

Other Toolsets (3)

Visual Studio extension development
Create add-ons and extensions for Visual Studio, including new commands, code analyzers and tool windows.

Linux development with C++ ✓
Create and debug applications running in a Linux environment.

.NET Core cross-platform development
Build cross-platform applications using .NET Core, ASP.NET Core, HTML/JavaScript, and Containers including Docker...

Location
C:\Program Files (x86)\Microsoft Visual Studio\Preview\Enterprise Change...

Installation details
- Visual Studio core editor
- .NET desktop development
- Desktop development with C++
- Universal Windows Platform development
- ASP.NET and web development
- Azure development
- Python development
- Data storage and processing
- Mobile development with C++
- Linux development with C++ *
  - Included
    - ✓ Visual Studio C++ core features
    - ✓ Windows Universal C Runtime
    - ✓ Visual C++ for Linux Development
  - Optional
    - ✓ Visual C++ tools for CMake and Linux
    - ☐ Embedded and IoT Development
- Individual components *
  - ✓ Static analysis tools
  - ✓ .NET Framework 4.7.1 SDK
  - ✓ WebSocket4Net

3. If you use CMake or you are targeting IoT or embedded platforms, go to the **Installation details** pane on the right, under **Linux development with C++**, expand **Optional Components** and choose the components you need.

   Visual Studio 2017 version 15.4 and later
   : When you install the Linux C++ workload for Visual Studio, CMake support for Linux is selected by default.

4. Click **Modify** to continue with the installation.

Options for creating a Linux environment

If you don't already have a Linux machine, you can create a Linux Virtual Machine on Azure. For more information, see Quickstart: Create a Linux virtual machine in the Azure portal.

Another option, on Windows 10, is to activate the Windows Subsystem for Linux. For more information, see Windows 10 Installation Guide.

Linux setup: Ubuntu

The target Linux computer must have **openssh-server**, **g++**, **gdb**, and **gdbserver** installed, and the ssh daemon must be running. **zip** is required for automatic syncing of remote headers with your local machine for Intellisense support. If these applications are not already present, you can install them as follows:

1. At a shell prompt on your Linux computer, run:

   ```
   sudo apt-get install openssh-server g++ gdb gdbserver zip
   ```

   You may be prompted for your root password due to the sudo command. If so, enter it and continue. Once complete, the required services and tools are installed.

2. Ensure the ssh service is running on your Linux computer by running:

   ```
   sudo service ssh start
   ```

   This starts the service and runs it in the background, ready to accept connections.

Linux setup: Fedora

The target machine running Fedora uses the **dnf** package installer. To download **openssh-server**, **g++**, **gdb**, **gdbserver** and **zip**, and restart the ssh daemon, follow these instructions:

1. At a shell prompt on your Linux computer, run:

   ```
   sudo dnf install openssh-server gcc-g++ gdb gdb-gdbserver zip
   ```

   You may be prompted for your root password due to the sudo command. If so, enter it and continue. Once complete, the required services and tools are installed.

2. Ensure the ssh service is running on your Linux computer by running:

   ```
   sudo systemctl start sshd
   ```

   This starts the service and runs it in the background, ready to accept connections.

Ensure you have CMake 3.8 on the remote Linux machine

Your Linux distro may have an older version of CMake. The CMake support in Visual Studio requires the server mode support that was introduced in CMake 3.8. For a Microsoft-provided CMake variant, download the latest prebuilt binaries to your Linux machine at https://github.com/Microsoft/CMake/releases.

.NET Programming with C++/CLI (Visual C++)

3/11/2019 • 2 minutes to read • Edit Online

Visual Studio 2015: By default, CLR projects created with Visual Studio 2015 target .NET Framework 4.5.2. To target .NET Framework 4.6 when you create a new project, in the **New Project** dialog, change the target framework in the dropdown at the top middle of the dialog. To change the target framework for an existing project, close the project, edit the project file (.vcxproj), and change the value of the Target Framework Version to 4.6. Next time you open the project, the settings will take effect.

Visual Studio 2017: In Visual Studio 2017, the default framework is 4.6.1 and the Framework version selector is at the bottom of the **New Project Dialog**. C++/CLI itself is not installed by default. To install the component, open the Visual Studio Installer and choose the C++/CLI component under Visual C++.

In This Section

C++/CLI Tasks

Native and .NET Interoperability

C++/CLI Migration Primer

Pure and Verifiable Code (C++/CLI)

Regular Expressions (C++/CLI)

File Handling and I/O (C++/CLI)

Graphics Operations (C++/CLI)

Windows Operations (C++/CLI)

Data Access Using ADO.NET (C++/CLI)

Interoperability with Other .NET Languages (C++/CLI)

Serialization (C++/CLI)

Managed Types (C++/CLI)

Reflection (C++/CLI)

Strong Name Assemblies (Assembly Signing) (C++/CLI)

Debug Class (C++/CLI)

STL/CLR Library Reference

C++ Support Library

Exceptions in C++/CLI

Boxing (C++/CLI)

See also

Native and .NET Interoperability

Cloud and Web Programming in Visual C++

5/16/2019 • 2 minutes to read • Edit Online

In C++, you have several options for connecting to the web and the cloud.

Microsoft Azure SDKs and REST services

- Microsoft Azure Storage Client Library for C++

  The Azure Storage Client Library for C++ provides a comprehensive API for working with Azure storage, including but not limited to the following abilities:

  - Create, read, delete, and list blob containers, tables, and queues.
  - Create, read, delete, list and copy blobs plus read and write blob ranges.
  - Insert, delete, replace, merge, and query entities in an Azure table.
  - Enqueue and dequeue messages in an Azure queue.
  - Lazily list containers, blobs, tables, and queues, and lazily query entities
- The ANSI C99 Azure IoT Hub SDKs for Internet of Things enable IoT applications to run on the device or on the backend.
- OneDrive and SharePoint in Microsoft Graph

  The OneDrive API provides a set of HTTP services to connect your application to files and folders in Office 365 and SharePoint Server 2016.

Windows and cross-platform networking APIs

- C++ REST SDK (Code name "Casablanca")

  Provides a modern, cross-platform, asynchronous API for interacting with REST services.

  - Perform REST calls against any HTTP server, with built-in support for JSON document parsing and serialization
  - Supports OAuth 1 and 2, including a local redirect listener
  - Make WebSockets connections against remote services
  - A fully asynchronous task API based on PPL, including a built-in thread pool

  Supports Windows Desktop (7+), Windows Server (2012+), Universal Windows Platform, Linux, OSX, Android, and iOS.

- Windows::Web::Http::HttpClient

  A Windows Runtime HTTP client class modeled on the .NET Framework class of the same name in the System.Web namespace. `HttpClient` fully supports asynchronous upload and download over HTTP, and pipeline filters that enable the insertion of custom HTTP handlers into the pipeline. The Windows SDK includes sample filters for metered networks, OAuth authentication, and more. For apps that target only Universal Windows Platform, we recommend that you use the `Windows::Web:HttpClient` class.

- IXMLHTTPRequest2 interface

  Provides a native COM interface that you can use in Windows Runtime apps or Windows desktop apps to connect to the Internet over HTTP and issue GET, PUT, and other HTTP commands. For more information, see Walkthrough: Connecting Using Tasks and XML HTTP Requests.

- Windows Internet (WinInet)

  Windows API that you can use in Windows desktop apps to connect to the Internet.

See also

C++ in Visual Studio
Microsoft Azure C and C++ Developer Center
Networks and web services (UWP)

Visual C++ Porting and Upgrading Guide

5/15/2019 • 6 minutes to read • Edit Online

This topic provides a guide for upgrading Visual C++ code. This includes getting the code to compile and run correctly on a newer release of the tools, as well as taking advantage of new language and Visual Studio features. This topic also includes information about migrating legacy apps to more modern platforms.

Reasons to Upgrade Visual C++ Code

You should consider upgrading your code for the following reasons:

- Faster code, due to improved compiler optimizations.
- Faster builds, due to performance improvements in the compiler itself.
- Improved standards conformance. Visual C++ now implements many features from the latest C++ standards.
- Better security. Security features such as guard checking.

Porting your Code

When upgrading, first consider your application's code and projects. Is your application built with Visual Studio? If so, identify the projects involved. Do you have custom build scripts? If you have custom build scripts instead of using Visual Studio's build system, you will have more work to do in upgrading, because you can't save time by having Visual Studio update your project files and build settings.

The build system and project file format in Visual Studio changed from vcbuild in versions up to Visual Studio 2008 to MSBuild in versions of Visual Studio from 2010 onwards. If your upgrade is from a version prior to 2010, and you have a highly customized build system, you might have to do more work to upgrade. If you are upgrading from Visual Studio 2010 or later, your projects are already using MSBuild, so upgrading the project and build for your application should be easier.

If you are not using Visual Studio's build system, you should consider upgrading to use MSBuild. If you upgrade to use MSBuild, you might have an easier time in future upgrades, and it will be easier to use services such as Visual Studio Online. MSBuild supports all the target platforms that Visual Studio supports.

Porting Visual Studio Projects

To start upgrading a project or solution, just open the solution in the new version of Visual Studio, and follow the prompts to start upgrading it. When you upgrade a project, you get an upgrade report, which is also saved in your project folder as UpgradeLog.htm. The upgrade report shows a summary of what problems were encountered during the upgrade process and some information about changes that were made, or problems that could not be addressed automatically.

1. Project properties
2. Include files
3. Code that no longer compiles cleanly due to compiler conformance imrovements or changes in the standard
4. Code that relies on Visual Studio or Windows features that are no longer available or header files that either aren't included in a default installation of Visual Studio, or were removed from the product
5. Code that no longer compiles due to changes in APIs such as renamed APIs, changed function signatures,

or deprecated functions

6. Code that no longer compiles due to changes in diagnostics, such as warning becoming an error
7. Linker errors due to libraries that were changed, especially when /NODEFAULTLIB is used.
8. Runtime errors or unexpected results due to behavior changes
9. Errors due to errors that were introduced in the tools. If you encounter an issue, report it to the Visual C++ team through your normal support channels or by using the Visual Studio Feedback Center.

In addition to changes that you can't avoid due to compiler errors, some changes are optional in an upgrade process, such as:

1. New warnings might mean you want to clean up your code. Depending on the specific diagnostics, this can improve the portability, standards conformance, and security of your code.
2. You might want to take advantage of newer compiler features such as the /guard:cf (Enable Flow Control Guard) compiler option, which adds checks for unauthorized code execution.
3. You might want to update some code to use new language features that simplify the code, improve the performance of your programs, or update the code to use modern libraries and conform to modern standards and best practices.

Once you've upgraded and tested your project, you might also want to consider improving your code further or plan the future direction of your code, or even reconsider the architecture of your project. Will it receive ongoing development work? Will it be important for your code to run on other platforms? If so, what platforms? C++ is a standardized language designed with portability and cross-platform development in mind, and yet the code for many Windows applications is strongly tied to the Windows platform. Do you want to refactor your code, to separate out those parts that are more tied to the Windows platform?

What about your user interface? If you are using MFC, you might want to update the UI. Are you using any of the newer MFC features that were introduced in 2008 as a Feature Pack? If you just want to give your app a newer look and feel, without rewriting the entire app, you might consider using the ribbon APIs in MFC, or using some of new features of MFC.

If you want to give your program a XAML user-interface but don't want to create a UWP app, you can use C# with WPF to create the UI layer and refactor your standard C++ logic into DLLs. Create an interoperability layer in C++/CLI to connect C# with your native code. Another option is to create a UWP app using C++/CX or C++/WinRT. In Windows 10, you can use the Desktop App Converter to package your existing desktop application as a UWP app without having to modify any code.

Alternatively, perhaps you now have new requirements, or you can foresee the need for targeting platforms other than Windows desktop, such as Windows Phone, or Android devices. You could port your user interface code to a cross-platform UI library. With these UI frameworks, you can target multiple devices and still use Visual Studio and the Visual Studio debugger as your development environment.

Related Topics

TITLE	DESCRIPTION
Upgrading Projects from Earlier Versions of Visual C++	Discusses how to use projects created in earlier versions of Visual Studio.
What's New for The C++ compiler in Visual Studio	Changes in the IDE and tools to the current version of Visual Studio

TITLE	DESCRIPTION
C++ conformance improvements in Visual Studio	Standards conformance improvements from Visual Studio 2015 to Visual Studio
Visual C++ change history 2003 - 2015	A list of all the changes in the Visual C++ libraries and build tools from Visual Studio 2003 through 2015 that might require changes in your code.
Visual C++ What's New 2003 through 2015	All the "what's new" information for Visual C++ from Visual Studio 2003 through Visual Studio 2015.
Porting 3rd-party libraries	How to use the **vcpkg** command line tool to port older open-source libraries to versions compiled with more recent Visual C++ toolsets.
Porting and Upgrading: Examples and Case Studies	For this section, we ported and upgrades several samples and applications and discussed the experiences and results. You might find that reading these gives you a sense of what is involved in the porting and upgrading process. Throughout the process, we discuss tips and tricks for upgrading and show how specific errors were fixed.
Porting to the Universal Windows Platform	Contains information about porting code to Windows 10
Introduction to Visual C++ for UNIX Users	Provides information for UNIX users who are new to Visual C++ and want to become productive with it.
Porting from UNIX to Win32	Discusses options for migrating UNIX applications to Windows.

See also

C++ in Visual Studio

Security Best Practices for C++

5/8/2019 • 3 minutes to read • Edit Online

This article contains information about security tools and practices. Using them does not make applications immune from attack, but it makes successful attacks less likely.

Visual C++ Security Features

These security features are built into the Microsoft C++ compiler and linker:

/guard (Enable Control Flow Guard)
Causes the compiler to analyze control flow for indirect call targets at compile time, and then to insert code to verify the targets at runtime.

/GS (Buffer Security Check)
Instructs the compiler to insert overrun detection code into functions that are at risk of being exploited. When an overrun is detected, execution is stopped. By default, this option is on.

/SAFESEH (Image has Safe Exception Handlers)
Instructs the linker to include in the output image a table that contains the address of each exception handler. At run time, the operating system uses this table to make sure that only legitimate exception handlers are executed. This helps prevent the execution of exception handlers that are introduced by a malicious attack at run time. By default, this option is off.

/NXCOMPAT, /NXCOMPAT (Compatible with Data Execution Prevention) These compiler and linker options enable Data Execution Prevention (DEP) compatibility. DEP guards the CPU against the execution of non-code pages.

/analyze (Code Analysis)
This compiler option activates code analysis that reports potential security issues such as buffer overrun, un-initialized memory, null pointer dereferencing, and memory leaks. By default, this option is off. For more information, see Code Analysis for C/C++ Overview.

/DYNAMICBASE (Use address space layout randomization)
This linker option enables the building of an executable image that can be loaded at different locations in memory at the beginning of execution. This option also makes the stack location in memory much less predictable.

Security-Enhanced CRT

The C Runtime Library (CRT) has been augmented to include secure versions of functions that pose security risks —for example, the unchecked `strcpy` string copy function. Because the older, nonsecure versions of these functions are deprecated, they cause compile-time warnings. We encourage you to use the secure versions of these CRT functions instead of suppressing the compilation warnings. For more information, see Security Features in the CRT.

SafeInt Library

SafeInt Library helps prevent integer overflows and other exploitable errors that might occur when the application performs mathematical operations. The `SafeInt` library includes the SafeInt Class, the SafeIntException Class, and several SafeInt Functions.

The `SafeInt` class protects against integer overflow and divide-by-zero exploits. You can use it to handle comparisons between values of different types. It provides two error handling policies. The default policy is for the `SafeInt` class to throw a `SafeIntException` class exception to report why a mathematical operation cannot be completed. The second policy is for the `SafeInt` class to stop program execution. You can also define a custom policy.

Each `SafeInt` function protects one mathematical operation from an exploitable error. You can use two different kinds of parameters without converting them to the same type. To protect multiple mathematical operations, use the `SafeInt` class.

Checked Iterators

A checked iterator enforces container boundaries. By default, when a checked iterator is out of bounds, it generates an exception and ends program execution. A checked iterator provides other levels of response that depend on values that are assigned to preprocessor defines such as **_SECURE_SCL_THROWS** and **_ITERATOR_DEBUG_LEVEL**. For example, at **_ITERATOR_DEBUG_LEVEL=2**, a checked iterator provides comprehensive correctness checks in debug mode, which are made available by using asserts. For more information, see Checked Iterators and _ITERATOR_DEBUG_LEVEL.

Code Analysis for Managed Code

Code Analysis for Managed Code, also known as FxCop, checks assemblies for conformance to the.NET Framework design guidelines. FxCop analyzes the code and metadata in each assembly to check for defects in the following areas:

- Library design
- Localization
- Naming conventions
- Performance
- Security

Windows Application Verifier

The Application Verifier (AppVerifier) can help you identify potential application compatibility, stability, and security issues.

The AppVerifier monitors how an application uses the operating system. It watches the file system, registry, memory, and APIs while the application is running, and recommends source-code fixes for issues that it uncovers.

You can use the AppVerifier to:

- Test for potential application compatibility errors that are caused by common programming mistakes.
- Examine an application for memory-related issues.
- Identify potential security issues in an application.

Windows User Accounts

Using Windows user accounts that belong to the Administrators group exposes developers and--by extension--customers to security risks. For more information, see Running as a Member of the Users Group and How User Account Control (UAC) Affects Your Application.

Guidance for Speculative Execution Side Channels

For information about how to indentify and mitigate against speculative execution side channel hardware vulnerabilities in C++ software, see C++ Developer Guidance for Speculative Execution Side Channels.

See also

System.Security
Security
How User Account Control (UAC) Affects Your Application

Running as a Member of the Users Group

3/11/2019 • 2 minutes to read • Edit Online

This topic explains how configuring Windows user accounts as a member of the Users Group (as opposed to the Administrators Group) increases security by reducing the chances of being infected with malicious code.

Security Risks

Running as an administrator makes your system vulnerable to several kinds of security attack, such as "Trojan horse" and "buffer overrun." Merely visiting an Internet site as an administrator can be damaging to the system, as malicious code that is downloaded from an Internet site may attack your computer. If successful, it inherits your administrator permissions and can then perform actions such as deleting all your files, reformatting your hard drive, and creating a new user accounts with administrative access.

Non Administrator User Groups

The Windows user accounts that developers use normally should be added to either the Users or Power Users Groups. Developers should also be added to the Debugging Group. Being a member of the Users group allows you to perform routine tasks including running programs and visiting Internet sites without exposing your computer to unnecessary risk. As a member of the Power Users group, you can also perform tasks such as application installation, printer installation, and most Control Panel operations. If you need to perform administrative tasks such as upgrading the operating system or configuring system parameters, you should log into an administrator account for just long enough to perform the administrative task. Alternatively, the Windows **runas** command can be used to launch specific applications with Administrative access.

Exposing Customers to Security Risks

Not being part of the Administrators group is particularly important for developers because, in addition to protecting development machines, it prevents developers from inadvertently writing code that requires customers to join the Administrators Group in order to execute the applications you develop. If code that requires administrator access is introduced during development, it will fail at runtime, alerting you to the fact that your application now requires customers to run as Administrators.

Code That Requires Administrator Privileges

Some code requires Administrator access in order to execute. If possible, alternatives to this code should be pursued. Examples of code operations that require Administrator access are:

- Writing to protected areas of the file system, such as the Windows or Program Files directories
- Writing to protected areas of the registry, such as HKEY_LOCAL_MACHINE
- Installing assemblies in the Global Assembly Cache (GAC)

Generally, these actions should be limited to application installation programs. This allows users to use administrator status only temporarily.

Debugging

You can debug any applications that you launch within Visual Studio (native and unmanaged) as a non-administrator by becoming part of the Debugging Group. This includes the ability to attach to a running application using the Attach to Process command. However, it is necessary to be part of the Administrator Group in order to debug native or managed applications that were launched by a different user.

See also

Security Best Practices

How User Account Control (UAC) Affects Your Application

5/8/2019 • 2 minutes to read • Edit Online

User Account Control (UAC) is a feature of Windows Vista in which user accounts have limited privileges. You can find detailed information about UAC at these sites:

- Developer Best Practices and Guidelines for Applications in a Least Privileged Environment

Building Projects after Enabling UAC

If you build a Visual Studio C++ project on Windows Vista with UAC disabled, and you later enable UAC, you must clean and rebuild the project for it to work correctly.

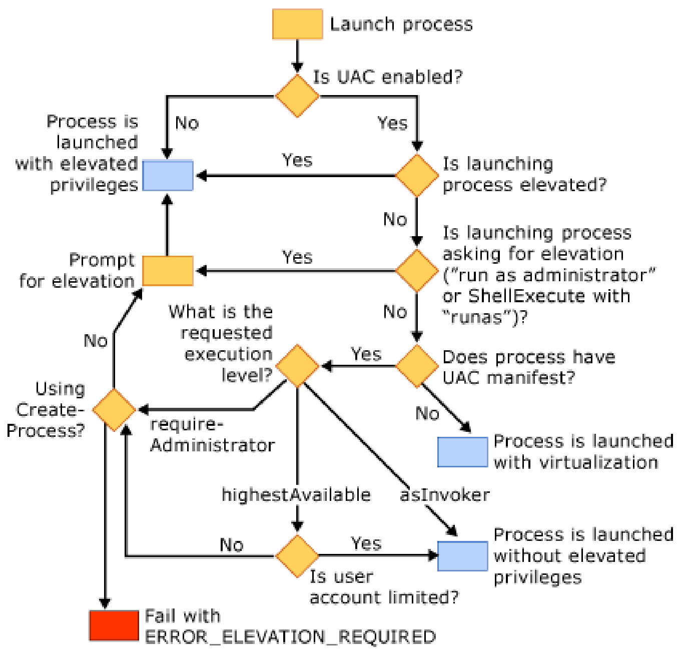
Applications that Require Administrative Privileges

By default, the Visual C++ linker embeds a UAC fragment into the manifest of an application with an execution level of `asInvoker`. If your application requires administrative privileges to run correctly (for example, if it modifies the HKLM node of the registry or if it writes to protected areas of the disk, such as the Windows directory), you must modify your application.

The first option is to modify the UAC fragment of the manifest to change the execution level to *requireAdministrator*. The application will then prompt the user for administrative credentials before it runs. For information about how to do this, see /MANIFESTUAC (Embeds UAC information in manifest).

The second option is to not embed a UAC fragment into the manifest by specifying the `/MANIFESTUAC:NO` linker option. In this case, your application will run virtualized. Any changes you make to the registry or to the file system will not persist after your application has ended.

The following flowchart describes how your application will run depending on whether UAC is enabled and whether the application has a UAC manifest:


Launch process
Is UAC enabled?
No
Yes
Process is launched with elevated privileges
Yes
Is launching process elevated?
No
Is launching process asking for elevation ("run as administrator" or ShellExecute with "runas")?
Prompt for elevation
Yes
No
What is the requested execution level?
Yes
Does process have UAC manifest?
No
Using Create-Process?
require-Administrator
No
Process is launched with virtualization
highestAvailable
asInvoker
No
Yes
Process is launched without elevated privileges
Is user account limited?
Fail with ERROR_ELEVATION_REQUIRED

See also

Security Best Practices

C++ Developer Guidance for Speculative Execution Side Channels

5/8/2019 • 20 minutes to read • Edit Online

This article contains guidance for developers to assist with identifying and mitigating speculative execution side channel hardware vulnerabilities in C++ software. These vulnerabilities can disclose sensitive information across trust boundaries and can affect software that runs on processors that support speculative, out-of-order execution of instructions. This class of vulnerabilities was first described in January, 2018 and additional background and guidance can be found in Microsoft's security advisory.

The guidance provided by this article is related to the classes of vulnerabilities represented by:

1. CVE-2017-5753, also known as Spectre variant 1. This hardware vulnerability class is related to side channels that can arise due to speculative execution that occurs as a result of a conditional branch misprediction. The Microsoft C++ compiler in Visual Studio 2017 (starting with version 15.5.5) includes support for the `/Qspectre` switch which provides a compile-time mitigation for a limited set of potentially vulnerable coding patterns related to CVE-2017-5753. The `/Qspectre` switch is also available in Visual Studio 2015 Update 3 through KB 4338871. The documentation for the /Qspectre flag provides more information on its effects and usage.

2. CVE-2018-3639, also known as Speculative Store Bypass (SSB). This hardware vulnerability class is related to side channels that can arise due to speculative execution of a load ahead of a dependent store as a result of a memory access misprediction.

An accessible introduction to speculative execution side channel vulnerabilities can be found in the presentation titled The Case of Spectre and Meltdown by one of the research teams that discovered these issues.

What are Speculative Execution Side Channel hardware vulnerabilities?

Modern CPUs provide higher degrees of performance by making use of speculative and out-of-order execution of instructions. For example, this is often accomplished by predicting the target of branches (conditional and indirect) which enables the CPU to begin speculatively executing instructions at the predicted branch target, thus avoiding a stall until the actual branch target is resolved. In the event that the CPU later discovers that a misprediction occurred, all of the machine state that was computed speculatively is discarded. This ensures that there are no architecturally visible effects of the mispredicted speculation.

While speculative execution does not affect the architecturally visible state, it can leave residual traces in non-architectural state, such as the various caches that are used by the CPU. It is these residual traces of speculative execution that can give rise to side channel vulnerabilities. To better understand this, consider the following code fragment which provides an example of CVE-2017-5753 (Bounds Check Bypass):

```
// A pointer to a shared memory region of size 1MB (256 * 4096)
unsigned char *shared_buffer;

unsigned char ReadByte(unsigned char *buffer, unsigned int buffer_size, unsigned int untrusted_index) {
    if (untrusted_index < buffer_size) {
        unsigned char value = buffer[untrusted_index];
        return shared_buffer[value * 4096];
    }
}
```

In this example, `ReadByte` is supplied a buffer, a buffer size, and an index into that buffer. The index parameter, as specified by `untrusted_index`, is supplied by a less privileged context, such as a non-administrative process. If `untrusted_index` is less than `buffer_size`, then the character at that index is read from `buffer` and used to index into a shared region of memory referred to by `shared_buffer`.

From an architectural perspective, this code sequence is perfectly safe as it is guaranteed that `untrusted_index` will always be less than `buffer_size`. However, in the presence of speculative execution, it is possible that the CPU will mispredict the conditional branch and execute the body of the if statement even when `untrusted_index` is greater than or equal to `buffer_size`. As a consequence of this, the CPU may speculatively read a byte from beyond the bounds of `buffer` (which could be a secret) and could then use that byte value to compute the address of a subsequent load through `shared_buffer`.

While the CPU will eventually detect this misprediction, residual side effects may be left in the CPU cache that reveal information about the byte value that was read out of bounds from `buffer`. These side effects can be detected by a less privileged context running on the system by probing how quickly each cache line in `shared_buffer` is accessed. The steps that can be taken to accomplish this are:

1. **Invoke `ReadByte` multiple times with `untrusted_index` being less than `buffer_size`**. The attacking context can cause the victim context to invoke `ReadByte` (e.g. via RPC) such that the branch predictor is trained to be not-taken as `untrusted_index` is less than `buffer_size`.

2. **Flush all cache lines in `shared_buffer`**. The attacking context must flush all of the cache lines in the shared region of memory referred to by `shared_buffer`. Since the memory region is shared, this is straightforward and can be accomplished using intrinsics such as `_mm_clflush`.

3. **Invoke `ReadByte` with `untrusted_index` being greater than `buffer_size`**. The attacking context causes the victim context to invoke `ReadByte` such that it incorrectly predicts that the branch will not be taken. This causes the processor to speculatively execute the body of the if block with `untrusted_index` being greater than `buffer_size`, thus leading to an out-of-bounds read of `buffer`. Consequently, `shared_buffer` is indexed using a potentially secret value that was read out-of-bounds, thus causing the respective cache line to be loaded by the CPU.

4. **Read each cache line in `shared_buffer` to see which is accessed most quickly**. The attacking context can read each cache line in `shared_buffer` and detect the cache line that loads significantly faster than the others. This is the cache line that is likely to have been brought in by step 3. Since there is a 1:1 relationship between byte value and cache line in this example, this allows the attacker to infer the actual value of the byte that was read out-of-bounds.

The above steps provide an example of using a technique known as FLUSH+RELOAD in conjunction with exploiting an instance of CVE-2017-5753.

What software scenarios can be impacted?

Developing secure software using a process like the Security Development Lifecycle (SDL) typically requires developers to identify the trust boundaries that exist in their application. A trust boundary exists in places where an application may interact with data provided by a less-trusted context, such as another process on the system or a non-administrative user mode process in the case of a kernel-mode device driver. The new class of vulnerabilities involving speculative execution side channels is relevant to many of the trust boundaries in existing software security models that isolate code and data on a device.

The following table provides a summary of the software security models where developers may need to be concerned about these vulnerabilities occurring:

TRUST BOUNDARY	DESCRIPTION
Virtual machine boundary	Applications that isolate workloads in separate virtual machines that receive untrusted data from another virtual machine may be at risk.
Kernel boundary	A kernel-mode device driver that receives untrusted data from a non-administrative user mode process may be at risk.
Process boundary	An application that receives untrusted data from another process running on the local system, such as through a Remote Procedure Call (RPC), shared memory, or other Inter-Process Communication (IPC) mechanisms may be at risk.
Enclave boundary	An application that executes within a secure enclave (such as Intel SGX) that receives untrusted data from outside of the enclave may be at risk.
Language boundary	An application that interprets or Just-In-Time (JIT) compiles and executes untrusted code written in a higher-level language may be at risk.

Applications that have attack surface exposed to any of the above trust boundaries should review code on the attack surface to identify and mitigate possible instances of speculative execution side channel vulnerabilities. It should be noted that trust boundaries exposed to remote attack surfaces, such as remote network protocols, have not been demonstrated to be at risk to speculative execution side channel vulnerabilities.

Potentially vulnerable coding patterns

Speculative execution side channel vulnerabilities can arise as a consequence of multiple coding patterns. This section describes potentially vulnerable coding patterns and provides examples for each, but it should be recognized that variations on these themes may exist. As such, developers are advised to take these patterns as examples and not as an exhaustive list of all potentially vulnerable coding patterns. The same classes of memory safety vulnerabilities that can exist in software today may also exist along speculative and out-of-order paths of execution, including but not limited to buffer overruns, out-of-bounds array accesses, uninitialized memory use, type confusion, and so on. The same primitives that attackers can use to exploit memory safety vulnerabilities along architectural paths may also apply to speculative paths.

In general, speculative execution side channels related to conditional branch misprediction can arise when a conditional expression operates on data that can be controlled or influenced by a less-trusted context. For example, this can include conditional expressions used in `if`, `for`, `while`, `switch`, or ternary statements. For each of these statements, the compiler may generate a conditional branch that the CPU may then predict the branch target for at runtime.

For each example, a comment with the phrase "SPECULATION BARRIER" is inserted where a developer could introduce a barrier as a mitigation. This is discussed in more detail in the section on mitigations.

Speculative out-of-bounds load

This category of coding patterns involves a conditional branch misprediction that leads to a speculative out-of-bounds memory access.

Array out-of-bounds load feeding a load

This coding pattern is the originally described vulnerable coding pattern for CVE-2017-5753 (Bounds Check Bypass). The background section of this article explains this pattern in detail.

```
// A pointer to a shared memory region of size 1MB (256 * 4096)
unsigned char *shared_buffer;

unsigned char ReadByte(unsigned char *buffer, unsigned int buffer_size, unsigned int untrusted_index) {
    if (untrusted_index < buffer_size) {
        // SPECULATION BARRIER
        unsigned char value = buffer[untrusted_index];
        return shared_buffer[value * 4096];
    }
}
```

Similarly, an array out-of-bounds load may occur in conjunction with a loop that exceeds its terminating condition due to a misprediction. In this example, the conditional branch associated with the `x < buffer_size` expression may mispredict and speculatively execute the body of the `for` loop when `x` is greater than or equal to `buffer_size`, thus resulting in a speculative out-of-bounds load.

```
// A pointer to a shared memory region of size 1MB (256 * 4096)
unsigned char *shared_buffer;

unsigned char ReadBytes(unsigned char *buffer, unsigned int buffer_size) {
    for (unsigned int x = 0; x < buffer_size; x++) {
        // SPECULATION BARRIER
        unsigned char value = buffer[x];
        return shared_buffer[value * 4096];
    }
}
```

Array out-of-bounds load feeding an indirect branch

This coding pattern involves the case where a conditional branch misprediction can lead to an out-of-bounds access to an array of function pointers which then leads to an indirect branch to the target address that was read out-of-bounds. The following snippet provides an example that demonstrates this.

In this example, an untrusted message identifier is provided to DispatchMessage through the `untrusted_message_id` parameter. If `untrusted_message_id` is less than `MAX_MESSAGE_ID`, then it is used to index into an array of function pointers and branch to the corresponding branch target. This code is safe architecturally, but if the CPU mispredicts the conditional branch, it could result in `DispatchTable` being indexed by `untrusted_message_id` when its value is greater than or equal to `MAX_MESSAGE_ID`, thus leading to an out-of-bounds access. This could result in speculative execution from a branch target address that is derived beyond the bounds of the array which could lead to information disclosure depending on the code that is executed speculatively.

```
#define MAX_MESSAGE_ID 16

typedef void (*MESSAGE_ROUTINE)(unsigned char *buffer, unsigned int buffer_size);

const MESSAGE_ROUTINE DispatchTable[MAX_MESSAGE_ID];

void DispatchMessage(unsigned int untrusted_message_id, unsigned char *buffer, unsigned int buffer_size) {
    if (untrusted_message_id < MAX_MESSAGE_ID) {
        // SPECULATION BARRIER
        DispatchTable[untrusted_message_id](buffer, buffer_size);
    }
}
```

As with the case of an array out-of-bounds load feeding another load, this condition may also arise in conjunction with a loop that exceeds its terminating condition due to a misprediction.

Array out-of-bounds store feeding an indirect branch

While the previous example showed how a speculative out-of-bounds load can influence an indirect branch target, it is also possible for an out-of-bounds store to modify an indirect branch target, such as a function pointer or a return address. This can potentially lead to speculative execution from an attacker-specified address.

In this example, an untrusted index is passed through the `untrusted_index` parameter. If `untrusted_index` is less than the element count of the `pointers` array (256 elements), then the provided pointer value in `ptr` is written to the `pointers` array. This code is safe architecturally, but if the CPU mispredicts the conditional branch, it could result in `ptr` being speculatively written beyond the bounds of the stack-allocated `pointers` array. This could lead to speculative corruption of the return address for `WriteSlot`. If an attacker can control the value of `ptr`, they may be able to cause speculative execution from an arbitrary address when `WriteSlot` returns along the speculative path.

```
unsigned char WriteSlot(unsigned int untrusted_index, void *ptr) {
    void *pointers[256];
    if (untrusted_index < 256) {
        // SPECULATION BARRIER
        pointers[untrusted_index] = ptr;
    }
}
```

Similarly, if a function pointer local variable named `func` were allocated on the stack, then it may be possible to speculatively modify the address that `func` refers to when the conditional branch misprediction occurs. This could result in speculative execution from an arbitrary address when the function pointer is called through.

```
unsigned char WriteSlot(unsigned int untrusted_index, void *ptr) {
    void *pointers[256];
    void (*func)() = &callback;
    if (untrusted_index < 256) {
        // SPECULATION BARRIER
        pointers[untrusted_index] = ptr;
    }
    func();
}
```

It should be noted that both of these examples involve speculative modification of stack-allocated indirect branch pointers. It is possible that speculative modification could also occur for global variables, heap-allocated memory, and even read-only memory on some CPUs. For stack-allocated memory, the Microsoft C++ compiler already takes steps to make it more difficult to speculatively modify stack-allocated indirect branch targets, such as by reordering local variables such that buffers are placed adjacent to a security cookie as part of the /GS compiler security feature.

Speculative type confusion

This category deals with coding patterns that can give rise to a speculative type confusion. This occurs when memory is accessed using an incorrect type along a non-architectural path during speculative execution. Both conditional branch misprediction and speculative store bypass can potentially lead to a speculative type confusion.

For speculative store bypass, this could occur in scenarios where a compiler reuses a stack location for variables of multiple types. This is because the architectural store of a variable of type `A` may be bypassed, thus allowing the load of type `A` to speculatively execute before the variable is assigned. If the previously stored variable is of a different type, then this can create the conditions for a speculative type confusion.

For conditional branch misprediction, the following code snippet will be used to describe different conditions that speculative type confusion can give rise to.

```
enum TypeName {
    Type1,
    Type2
};

class CBaseType {
public:
    CBaseType(TypeName type) : type(type) {}
    TypeName type;
};

class CType1 : public CBaseType {
public:
    CType1() : CBaseType(Type1) {}
    char field1[256];
    unsigned char field2;
};

class CType2 : public CBaseType {
public:
    CType2() : CBaseType(Type2) {}
    void (*dispatch_routine)();
    unsigned char field2;
};

// A pointer to a shared memory region of size 1MB (256 * 4096)
unsigned char *shared_buffer;

unsigned char ProcessType(CBaseType *obj)
{
    if (obj->type == Type1) {
        // SPECULATION BARRIER
        CType1 *obj1 = static_cast<CType1 *>(obj);

        unsigned char value = obj1->field2;

        return shared_buffer[value * 4096];
    }
    else if (obj->type == Type2) {
        // SPECULATION BARRIER
        CType2 *obj2 = static_cast<CType2 *>(obj);

        obj2->dispatch_routine();

        return obj2->field2;
    }
}
```

Speculative type confusion leading to an out-of-bounds load

This coding pattern involves the case where a speculative type confusion can result in an out-of-bounds or type-confused field access where the loaded value feeds a subsequent load address. This is similar to the array out-of-bounds coding pattern but it is manifested through an alternative coding sequence as shown above. In this example, an attacking context could cause the victim context to execute `ProcessType` multiple times with an object of type `CType1` (`type` field is equal to `Type1`). This will have the effect of training the conditional branch for the first `if` statement to predict not taken. The attacking context can then cause the victim context to execute `ProcessType` with an object of type `CType2`. This can result in a speculative type confusion if the conditional branch for the first `if` statement mispredicts and executes the body of the `if` statement, thus casting an object of type `CType2` to `CType1`. Since `CType2` is smaller than `CType1`, the memory access to `CType1::field2` will result in a speculative out-of-bounds load of data that may be secret. This value is then used in a load from `shared_buffer` which can create observable side effects, as with the array out-of-bounds example described previously.

Speculative type confusion leading to an indirect branch

This coding pattern involves the case where a speculative type confusion can result in an unsafe indirect branch during speculative execution. In this example, an attacking context could cause the victim context to execute `ProcessType` multiple times with an object of type `CType2` (`type` field is equal to `Type2`). This will have the effect of training the conditional branch for the first `if` statement to be taken and the `else if` statement to be not taken. The attacking context can then cause the victim context to execute `ProcessType` with an object of type `CType1`. This can result in a speculative type confusion if the conditional branch for the first `if` statement predicts taken and the `else if` statement predicts not taken, thus executing the body of the `else if` and casting an object of type `CType1` to `CType2`. Since the `CType2::dispatch_routine` field overlaps with the `char` array `CType1::field1`, this could result in a speculative indirect branch to an unintended branch target. If the attacking context can control the byte values in the `CType1::field1` array, they may be able to control the branch target address.

Speculative uninitialized use

This category of coding patterns involves scenarios where speculative execution may access uninitialized memory and use it to feed a subsequent load or indirect branch. For these coding patterns to be exploitable, an attacker needs to be able to control or meaningfully influence the contents of the memory that is used without being initialized by the context that it is being used in.

Speculative uninitialized use leading to an out-of-bounds load

A speculative uninitialized use can potentially lead to an out-of-bounds load using an attacker controlled value. In the example below, the value of `index` is assigned `trusted_index` on all architectural paths and `trusted_index` is assumed to be less than or equal to `buffer_size`. However, depending on the code produced by the compiler, it is possible that a speculative store bypass may occur that allows the load from `buffer[index]` and dependent expressions to execute ahead of the assignment to `index`. If this occurs, an uninitialized value for `index` will be used as the offset into `buffer` which could enable an attacker to read sensitive information out-of-bounds and convey this through a side channel through the dependent load of `shared_buffer`.

```
// A pointer to a shared memory region of size 1MB (256 * 4096)
unsigned char *shared_buffer;

void InitializeIndex(unsigned int trusted_index, unsigned int *index) {
    *index = trusted_index;
}

unsigned char ReadByte(unsigned char *buffer, unsigned int buffer_size, unsigned int trusted_index) {
    unsigned int index;

    InitializeIndex(trusted_index, &index); // not inlined

    // SPECULATION BARRIER
    unsigned char value = buffer[index];
    return shared_buffer[value * 4096];
}
```

Speculative uninitialized use leading to an indirect branch

A speculative uninitialized use can potentially lead to an indirect branch where the branch target is controlled by an attacker. In the example below, `routine` is assigned to either `DefaultMessageRoutine1` or `DefaultMessageRoutine` depending on the value of `mode`. On the architectural path, this will result in `routine` always being initialized ahead of the indirect branch. However, depending on the code produced by the compiler, a speculative store bypass may occur that allows the indirect branch through `routine` to be speculatively executed ahead of the assignment to `routine`. If this occurs, an attacker may be able to speculatively execute from an arbitrary address, assuming the attacker can influence or control the uninitialized value of `routine`.

```
#define MAX_MESSAGE_ID 16

typedef void (*MESSAGE_ROUTINE)(unsigned char *buffer, unsigned int buffer_size);

const MESSAGE_ROUTINE DispatchTable[MAX_MESSAGE_ID];
extern unsigned int mode;

void InitializeRoutine(MESSAGE_ROUTINE *routine) {
    if (mode == 1) {
        *routine = &DefaultMessageRoutine1;
    }
    else {
        *routine = &DefaultMessageRoutine;
    }
}

void DispatchMessage(unsigned int untrusted_message_id, unsigned char *buffer, unsigned int buffer_size) {
    MESSAGE_ROUTINE routine;

    InitializeRoutine(&routine); // not inlined

    // SPECULATION BARRIER
    routine(buffer, buffer_size);
}
```

Mitigation options

Speculative execution side channel vulnerabilities can be mitigated by making changes to source code. These changes can involve mitigating specific instances of a vulnerability, such as by adding a *speculation barrier*, or by making changes to the design of an application to make sensitive information inaccessible to speculative execution.

Speculation barrier via manual instrumentation

A *speculation barrier* can be manually inserted by a developer to prevent speculative execution from proceeding along a non-architectural path. For example, a developer can insert a speculation barrier before a dangerous coding pattern in the body of a conditional block, either at the beginning of the block (after the conditional branch) or before the first load that is of concern. This will prevent a conditional branch misprediction from executing the dangerous code on a non-architectural path by serializing execution. The speculation barrier sequence differs by hardware architecture as described by the following table:

ARCHITECTURE	SPECULATION BARRIER INTRINSIC FOR CVE-2017-5753	SPECULATION BARRIER INTRINSIC FOR CVE-2018-3639
x86/x64	_mm_lfence()	_mm_lfence()
ARM	not currently available	__dsb(0)
ARM64	not currently available	__dsb(0)

For example, the following code pattern can be mitigated by using the `_mm_lfence` intrinsic as shown below.

```
// A pointer to a shared memory region of size 1MB (256 * 4096)
unsigned char *shared_buffer;

unsigned char ReadByte(unsigned char *buffer, unsigned int buffer_size, unsigned int untrusted_index) {
    if (untrusted_index < buffer_size) {
        _mm_lfence();
        unsigned char value = buffer[untrusted_index];
        return shared_buffer[value * 4096];
    }
}
```

Speculation barrier via compiler-time instrumentation

The Microsoft C++ compiler in Visual Studio 2017 (starting with version 15.5.5) includes support for the `/Qspectre` switch which automatically inserts a speculation barrier for a limited set of potentially vulnerable coding patterns related to CVE-2017-5753. The documentation for the /Qspectre flag provides more information on its effects and usage. It is important to note that this flag does not cover all of the potentially vulnerable coding patterns and as such developers should not rely on it as a comprehensive mitigation for this class of vulnerabilities.

Masking array indices

In cases where a speculative out-of-bounds load may occur, the array index can be strongly bounded on both the architectural and non-architectural path by adding logic to explicitly bound the array index. For example, if an array can be allocated to a size that is aligned to a power of two, then a simple mask can be introduced. This is illustrated in the sample below where it is assumed that `buffer_size` is aligned to a power of two. This ensures that `untrusted_index` is always less than `buffer_size`, even if a conditional branch misprediction occurs and `untrusted_index` was passed in with a value greater than or equal to `buffer_size`.

It should be noted that the index masking performed here could be subject to speculative store bypass depending on the code that is generated by the compiler.

```
// A pointer to a shared memory region of size 1MB (256 * 4096)
unsigned char *shared_buffer;

unsigned char ReadByte(unsigned char *buffer, unsigned int buffer_size, unsigned int untrusted_index) {
    if (untrusted_index < buffer_size) {
        untrusted_index &= (buffer_size - 1);
        unsigned char value = buffer[untrusted_index];
        return shared_buffer[value * 4096];
    }
}
```

Removing sensitive information from memory

Another technique that can be used to mitigate speculative execution side channel vulnerabilities is to remove sensitive information from memory. Software developers can look for opportunities to refactor their application such that sensitive information is not accessible during speculative execution. This can be accomplished by refactoring the design of an application to isolate sensitive information into separate processes. For example, a web browser application can attempt to isolate the data associated with each web origin into separate processes, thus preventing one process from being able to access cross-origin data through speculative execution.

See also

Guidance to mitigate speculative execution side-channel vulnerabilities
Mitigating speculative execution side channel hardware vulnerabilities

Languages

4/1/2019 • 2 minutes to read • Edit Online

C Language
C++ Language
C/C++ Preprocessor
Compiler Intrinsics and Assembly Language

C Language Reference

2/12/2019 • 2 minutes to read • Edit Online

The *C Language Reference* describes the C programming language as implemented in Microsoft C. The book's organization is based on the ANSI C standard (sometimes referred to as C89) with additional material on the Microsoft extensions to the ANSI C standard.

- Organization of the C Language Reference

For additional reference material on C++ and the preprocessor, see:

- C++ Language Reference
- Preprocessor Reference

Compiler and linker options are documented in the C/C++ Building Reference.

See also

C++ Language Reference

C++ Language Reference

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

This reference explains the C++ programming language as implemented in the Microsoft C++ compiler. The organization is based on *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup and on the ANSI/ISO C++ International Standard (ISO/IEC FDIS 14882). Microsoft-specific implementations of C++ language features are included.

For an overview of Modern C++ programming practices, see Welcome Back to C++.

See the following tables to quickly find a keyword or operator:

- C++ Keywords
- C++ Operators

In This Section

Lexical Conventions
Fundamental lexical elements of a C++ program: tokens, comments, operators, keywords, punctuators, literals. Also, file translation, operator precedence/associativity.

Basic Concepts
Scope, linkage, program startup and termination, storage classes, and types.

Standard Conversions
Type conversions between built-in, or "fundamental," types. Also, arithmetic conversions and conversions among pointer, reference, and pointer-to-member types.

Operators, Precedence and Associativity
The operators in C++.

Expressions
Types of expressions, semantics of expressions, reference topics on operators, casting and casting operators, run-time type information.

Lambda Expressions
A programming technique that implicitly defines a function object class and constructs a function object of that class type.

Statements
Expression, null, compound, selection, iteration, jump, and declaration statements.

Declarations and Definitions
Storage-class specifiers, function definitions, initializations, enumerations, **class**, **struct**, and **union** declarations, and **typedef** declarations. Also, **inline** functions, **const** keyword, namespaces.

Classes, Structures, and Unions
Introduction to classes, structures, and unions. Also, member functions, special member functions, data members, bit fields, **this** pointer, nested classes.

Derived Classes
Single and multiple inheritance, **virtual** functions, multiple base classes, **abstract** classes, scope rules. Also, the **__super** and **__interface** keywords.

Member-Access Control
Controlling access to class members: **public**, **private**, and **protected** keywords. Friend functions and classes.

Overloading
Overloaded operators, rules for operator overloading.

Exception Handling
C++ exception handling, structured exception handling (SEH), keywords used in writing exception handling statements.

Assertion and User-Supplied Messages
`#error` directive, the **static_assert** keyword, the `assert` macro.

Templates
Template specifications, function templates, class templates, **typename** keyword, templates vs. macros, templates and smart pointers.

Event Handling
Declaring events and event handlers.

Microsoft-Specific Modifiers
Modifiers specific to Microsoft C++. Memory addressing, calling conventions, **naked** functions, extended storage-class attributes (**__declspec**), **__w64**.

Inline Assembler
Using assembly language and C++ in **__asm** blocks.

Compiler COM Support
A reference to Microsoft-specific classes and global functions used to support COM types.

Microsoft Extensions
Microsoft extensions to C++.

Nonstandard Behavior
Information about nonstandard behavior of the Microsoft C++ compiler.

Welcome Back to C++
An overview of modern C++ programming practices for writing safe, correct and efficient programs.

Related Sections

Component Extensions for Runtime Platforms
Reference material on using the Microsoft C++ compiler to target .NET.

C/C++ Building Reference
Compiler options, linker options, and other build tools.

C/C++ Preprocessor Reference
Reference material on pragmas, preprocessor directives, predefined macros, and the preprocessor.

Visual C++ Libraries
A list of links to the reference start pages for the various Microsoft C++ libraries.

See also

C Language Reference

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.

Compiler Intrinsics and Assembly Language

5/15/2019 • 2 minutes to read • Edit Online

This section of the documentation contains information about compiler intrinsics and the assembly language.

Related Articles

TITLE	DESCRIPTION
Compiler Intrinsics	Describes intrinsic functions that are available in Microsoft C and C++ for x86, ARM, and x64 architectures.
Inline Assembler	Explains how to use the Visual C/C++ inline assembler with x86 processors.
ARM Assembler Reference	Provides reference material for the Microsoft ARM assembler (armasm) and related tools.
Microsoft Macro Assembler Reference	Provides reference material for the Microsoft Macro assembler (masm).
C++ in Visual Studio	The top-level article for Visual C++ documentation.

C Run-Time Library Reference

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

The Microsoft run-time library provides routines for programming for the Microsoft Windows operating system. These routines automate many common programming tasks that are not provided by the C and C++ languages.

Sample programs are included in the individual reference topics for most routines in the library.

In This Section

C Run-Time Libraries
Discusses the .lib files that comprise the C run-time libraries.

Universal C runtime routines by category
Provides links to the run-time library by category.

Global Variables and Standard Types
Provides links to the global variables and standard types provided by the run-time library.

Global Constants
Provides links to the global constants defined by the run-time library.

Alphabetical Function Reference
Provides a table of contents entry point into an alphabetical listing of all C run-time library functions.

Generic-Text Mappings
Provides links to the generic-text mappings defined in Tchar.h.

Language and Country/Region Strings
Describes how to use the `setlocale` function to set the language and Country/Region strings.

Related Sections

Debug Routines
Provides links to the debug versions of the run-time library routines.

Run-Time Error Checking
Provides links to functions that support run-time error checks.

DLLs and Visual C++ run-time library behavior
Discusses the entry point and startup code used for a DLL.

Debugging
Provides links to using the Visual Studio debugger to correct logic errors in your application or stored procedures.

Component Extensions for .NET and UWP

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

The C++ standard allows compiler vendors to provide non-standard extensions to the language. Microsoft provides extensions to help you connect native C++ code to code that runs on the .NET Framework or the Universal Windows Platform (UWP). The .NET extensions are called C++/CLI and produce code that executes in the .NET managed execution environment that is called the Common Language Runtime (CLR). The UWP extensions are called C++/CX and they produce native machine code.

NOTE

For new applications, we recommend using C++/WinRT rather than C++/CX. C++/WinRT is a new, standard C++17 language projection for Windows Runtime APIs. We will continue to support C++/CX and WRL, but highly recommend that new applications use C++/WinRT. For more information, see C++/WinRT.

Two runtimes, one set of extensions

C++/CLI extends the ISO/ANSI C++ standard, and is defined under the Ecma C++/CLI Standard. For more information, see .NET Programming with C++/CLI (Visual C++).

The C++/CX extensions are a subset of C++/CLI. Although the extension syntax is identical in most cases, the code that is generated depends on whether you specify the `/ZW` compiler option to target UWP, or the `/clr` option to target .NET. These switches are set automatically when you use Visual Studio to create a project.

Data Type Keywords

The language extensions include *aggregate keywords*, which consist of two tokens separated by white space. The tokens might have one meaning when they are used separately, and another meaning when they are used together. For example, the word "ref" is an ordinary identifier, and the word "class" is a keyword that declares a native class. But when these words are combined to form **ref class**, the resulting aggregate keyword declares an entity that is known as a *runtime class*.

The extensions also include *context-sensitive* keywords. A keyword is treated as context-sensitive depending on the kind of statement that contains it, and its placement in that statement. For example, the token "property" can be an identifier, or it can declare a special kind of public class member.

The following table lists keywords in the C++ language extension.

KEYWORD	CONTEXT SENSITIVE	PURPOSE	REFERENCE
ref class **ref struct**	No	Declares a class.	Classes and Structs
value class **value struct**	No	Declares a value class.	Classes and Structs
interface class **interface struct**	No	Declares an interface.	interface class

KEYWORD	CONTEXT SENSITIVE	PURPOSE	REFERENCE
enum class **enum struct**	No	Declares an enumeration.	enum class
property	Yes	Declares a property.	property
delegate	Yes	Declares a delegate.	delegate (C++/CLI and C++/CX)
event	Yes	Declares an event.	event

Override Specifiers

You can use the following keywords to qualify override behavior for derivation. Although the **new** keyword is not an extension of C++, it is listed here because it can be used in an additional context. Some specifiers are also valid for native programming. For more information, see How to: Declare Override Specifiers in Native Compilations (C++/CLI).

KEYWORD	CONTEXT SENSITIVE	PURPOSE	REFERENCE
abstract	Yes	Indicates that functions or classes are abstract.	abstract
new	No	Indicates that a function is not an override of a base class version.	new (new slot in vtable)
override	Yes	Indicates that a method must be an override of a base-class version.	override
sealed	Yes	Prevents classes from being used as base classes.	sealed

Keywords for Generics

The following keywords have been added to support generic types. For more information, see Generics.

KEYWORD	CONTEXT SENSITIVE	PURPOSE
generic	No	Declares a generic type.
where	Yes	Specifies the constraints that are applied to a generic type parameter.

Miscellaneous Keywords

The following keywords have been added to the C++ extensions.

KEYWORD	CONTEXT SENSITIVE	PURPOSE	REFERENCE
finally	Yes	Indicates default exception handlings behavior.	Exception Handling
for each, in	No	Enumerates elements of a collection.	for each, in
gcnew	No	Allocates types on the garbage-collected heap. Use instead of **new** and **delete**.	ref new, gcnew
ref new	Yes	Allocates a Windows Runtime type. Use instead of **new** and **delete**.	ref new, gcnew
initonly	Yes	Indicates that a member can only be initialized at declaration or in a static constructor.	initonly (C++/CLI)
literal	Yes	Creates a literal variable.	literal
nullptr	No	Indicates that a handle or pointer does not point at an object.	nullptr

Template Constructs

The following language constructs are implemented as templates, instead of as keywords. If you specify the `/ZW` compiler option, they are defined in the `lang` namespace. If you specify the `/clr` compiler option, they are defined in the `cli` namespace.

KEYWORD	PURPOSE	REFERENCE
array	Declares an array.	Arrays
interior_ptr	(CLR only) Points to data in a reference type.	interior_ptr (C++/CLI)
pin_ptr	(CLR only) Points to CLR reference types to temporarily suppress the garbage-collection system.	pin_ptr (C++/CLI)
safe_cast	Determines and executes the optimal casting method for a runtime type.	safe_cast
typeid	(CLR only) Retrieves a System.Type object that describes the given type or object.	typeid

Declarators

The following type declarators instruct the runtime to automatically manage the lifetime and deletion of allocated objects.

OPERATOR	PURPOSE	REFERENCE
`^`	Declares a handle to an object; that is, a pointer to a Windows Runtime or CLR object that is automatically deleted when it is no longer usable.	Handle to Object Operator (^)
`%`	Declares a tracking reference; that is, a reference to a Windows Runtime or CLR object that is automatically deleted when it is no longer usable.	Tracking Reference Operator

Additional Constructs and Related Topics

This section lists additional programming constructs, and topics that pertain to the CLR.

TOPIC	DESCRIPTION
__identifier (C++/CLI)	(Windows Runtime and CLR) Enables the use of keywords as identifiers.
Variable Argument Lists (...) (C++/CLI)	(Windows Runtime and CLR) Enables a function to take a variable number of arguments.
.NET Framework Equivalents to C++ Native Types (C++/CLI)	Lists the CLR types that are used in place of C++ integral types.
appdomain **__declspec** modifier	**__declspec** modifier that mandates that static and global variables exist per appdomain.
C-Style Casts with /clr (C++/CLI)	Describes how C-style casts are interpreted.
__clrcall calling convention	Indicates the CLR-compliant calling convention.
`__cplusplus_cli`	Predefined Macros
Custom Attributes	Describes how to define your own CLR attributes.
Exception Handling	Provides an overview of exception handling.
Explicit Overrides	Demonstrates how member functions can override arbitrary members.
Friend Assemblies (C++)	Discusses how a client assembly can access all types in an assembly component.
Boxing	Demonstrates the conditions in which values types are boxed.
Compiler Support for Type Traits	Discusses how to detect characteristics of types at compile time.
managed, unmanaged pragmas	Demonstrates how managed and unmanaged functions can co-exist in the same module.

TOPIC	DESCRIPTION
process **__declspec** modifier	**__declspec** modifier that mandates that static and global variables exist per process.
Reflection (C++/CLI)	Demonstrates the CLR version of run-time type information.
String	Discusses compiler conversion of string literals to String.
Type Forwarding (C++/CLI)	Enables the movement of a type in a shipping assembly to another assembly so that client code does not have to be recompiled.
User-Defined Attributes	Demonstrates user-defined attributes.
#using Directive	Imports external assemblies.
XML Documentation	Explains XML-based code documentation by using /doc (Process Documentation Comments) (C/C++)

See also

.NET Programming with C++/CLI (Visual C++)
Native and .NET Interoperability

C++ Attributes for COM and .NET

5/8/2019 • 4 minutes to read • Edit Online

Microsoft defines a set of C++ attributes that simplify COM programming and .NET Framework common language runtime development. When you include attributes in your source files, the compiler works with provider DLLs to insert code or modify the code in the generated object files. These attributes aid in the creation of .idl files, interfaces, type libraries, and other COM elements. In the integrated development environment (IDE), attributes are supported by the wizards and by the Properties window.

While attributes eliminate some of the detailed coding needed to write COM objects, you need a background in COM fundamentals to best use them.

NOTE

If you are looking for C++ standard attributes, see Attributes.

Purpose of Attributes

Attributes extend C++ in directions not currently possible without breaking the classic structure of the language. Attributes allow providers (separate DLLs) to extend language functionality dynamically. The primary goal of attributes is to simplify the authoring of COM components, in addition to increasing the productivity level of the component developer. Attributes can be applied to nearly any C++ construct, such as classes, data members, or member functions. The following is a highlight of benefits provided by this new technology:

- Exposes a familiar and simple calling convention.
- Uses inserted code, which, unlike macros, is recognized by the debugger.
- Allows easy derivation from base classes without burdensome implementation details.
- Replaces the large amount of IDL code required by a COM component with a few concise attributes.

For example, to implement a simple event sink for a generic ATL class, you could apply the event_receiver attribute to a specific class such as `CMyReceiver`. The `event_receiver` attribute is then compiled by the Microsoft C++ compiler, which inserts the proper code into the object file.

```
[event_receiver(com)]
class CMyReceiver
{
   void handler1(int i) { ... }
   void handler2(int i, float j) { ... }
}
```

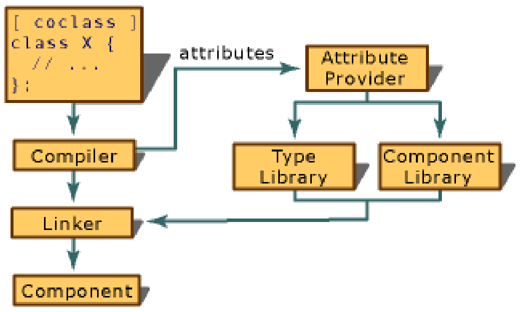
You can then set up the `CMyReceiver` methods `handler1` and `handler2` to handle events (using the intrinsic function __hook) from an event source, which you can create using event_source.

Basic Mechanics of Attributes

There are three ways to insert attributes into your project. First, you can insert them manually into your source code. Second, you can insert them using the property grid of an object in your project. Finally, you can insert them using the various wizards. For more information on using the **Properties** window and the various wizards, see Visual Studio Projects - C++.

As before, when the project is built, the compiler parses each C++ source file, producing an object file. However, when the compiler encounters an attribute, it is parsed and syntactically verified. The compiler then dynamically calls an attribute provider to insert code or make other modifications at compile time. The implementation of the provider differs depending on the type of attribute. For example, ATL-related attributes are implemented by Atlprov.dll.

The following figure demonstrates the relationship between the compiler and the attribute provider.



NOTE

Attribute usage does not alter the contents of the source file. The only time the generated attribute code is visible is during debugging sessions. In addition, for each source file in the project, you can generate a text file that displays the results of the attribute substitution. For more information on this procedure, see /Fx (Merge Injected Code) and Debugging Injected Code.

Like most C++ constructs, attributes have a set of characteristics that defines their proper usage. This is referred to as the context of the attribute and is addressed in the attribute context table for each attribute reference topic. For example, the coclass attribute can only be applied to an existing class or structure, as opposed to the cpp_quote attribute, which can be inserted anywhere within a C++ source file.

Building an Attributed Program

After you put Visual C++ attributes into your source code, you may want the Microsoft C++ compiler to produce a type library and .idl file for you. The following linker options help you build .tlb and .idl files:

- /IDLOUT
- /IGNOREIDL
- /MIDL
- /TLBOUT

Some projects contain multiple independent .idl files. These are used to produce two or more .tlb files and optionally bind them into the resource block. This scenario is not currently supported in Visual C++.

In addition, the Visual C++ linker will output all IDL-related attribute information to a single MIDL file. There will be no way to generate two type libraries from a single project.

Attribute Contexts

C++ attributes can be described using four basic fields: the target they can be applied to (**Applies To**), if they are repeatable or not (**Repeatable**), the required presence of other attributes (**Required Attributes**), and incompatibilities with other attributes (**Invalid Attributes**). These fields are listed in an accompanying table in each attribute's reference topic. Each of these fields is described below.

Applies To

This field describes the different C++ language elements that are legal targets for the specified attribute. For instance, if an attribute specifies "class" in the **Applies To** field, this indicates that the attribute can only be applied to a legal C++ class. If the attribute is applied to a member function of a class, a syntax error would result.

For more information, see Attributes by Usage.

Repeatable

This field states whether the attribute can be repeatedly applied to the same target. The majority of attributes are not repeatable.

Required Attributes

This field lists other attributes that need to be present (that is, applied to the same target) for the specified attribute to function properly. It is uncommon for an attribute to have any entries for this field.

Invalid Attributes

This field lists other attributes that are incompatible with the specified attribute. It is uncommon for an attribute to have any entries for this field.

In This Section

Attribute Programming FAQ
Attributes by Group
Attributes by Usage
Attributes Alphabetical Reference

Libraries

4/1/2019 • 2 minutes to read • Edit Online

Visual Studio includes the following libraries when you install one or more of the C++ workloads. For information about installing 3rd-party libraries, see vcpkg: A C++ package manager for Windows, Linux and MacOS.

Standard Libraries

C Runtime Library
C++ Standard Library
SafeInt Library
OpenMP

Libraries for Windows applications

MFC/ATL
Parallel Libraries
Data Access Libraries

C Run-Time Library Reference

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

The Microsoft run-time library provides routines for programming for the Microsoft Windows operating system. These routines automate many common programming tasks that are not provided by the C and C++ languages.

Sample programs are included in the individual reference topics for most routines in the library.

In This Section

C Run-Time Libraries
Discusses the .lib files that comprise the C run-time libraries.

Universal C runtime routines by category
Provides links to the run-time library by category.

Global Variables and Standard Types
Provides links to the global variables and standard types provided by the run-time library.

Global Constants
Provides links to the global constants defined by the run-time library.

Alphabetical Function Reference
Provides a table of contents entry point into an alphabetical listing of all C run-time library functions.

Generic-Text Mappings
Provides links to the generic-text mappings defined in Tchar.h.

Language and Country/Region Strings
Describes how to use the `setlocale` function to set the language and Country/Region strings.

Related Sections

Debug Routines
Provides links to the debug versions of the run-time library routines.

Run-Time Error Checking
Provides links to functions that support run-time error checks.

DLLs and Visual C++ run-time library behavior
Discusses the entry point and startup code used for a DLL.

Debugging
Provides links to using the Visual Studio debugger to correct logic errors in your application or stored procedures.

C++ Standard Library Reference

3/11/2019 • 2 minutes to read • Edit Online

A C++ program can call on a large number of functions from this conforming implementation of the C++ Standard Library. These functions perform essential services such as input and output and provide efficient implementations of frequently used operations.

For more information about Visual C++ run-time libraries, see CRT Library Features.

In This Section

C++ Standard Library Overview
Provides an overview of the Microsoft implementation of the C++ Standard Library.

iostream Programming
Provides an overview of iostream programming.

Header Files Reference
Provides links to reference topics discussing the C++ Standard Library header files, with code examples.

SafeInt Library

4/1/2019 • 2 minutes to read • Edit Online

SafeInt is a portable library that can be used with MSVC, GCC or Clang to help prevent integer overflows that might result when the application performs mathematical operations. The latest version of this library is located at https://github.com/dcleblanc/SafeInt.

In This Section

SECTION	DESCRIPTION
SafeInt Class	This class protects against integer overflows.
SafeInt Functions	Functions that can be used without creating a **SafeInt** object.
SafeIntException Class	A class of exceptions related to the **SafeInt** class.

Related Sections

SECTION	DESCRIPTION
C++ Language Reference	Reference and conceptual content for the C++ language.

SafeInt Class

4/1/2019 • 9 minutes to read • Edit Online

Extends the integer primitives to help prevent integer overflow and lets you compare different types of integers.

NOTE

The latest version of this library is located at https://github.com/dcleblanc/SafeInt.

Syntax

```
template<typename T, typename E = _SAFEINT_DEFAULT_ERROR_POLICY>
class SafeInt;
```

Parameters

TEMPLATE	DESCRIPTION
T	The type of integer or Boolean parameter that `SafeInt` replaces.
E	An enumerated data type that defines the error handling policy.
U	The type of integer or Boolean parameter for the secondary operand.

PARAMETER	DESCRIPTION
rhs	[in] An input parameter that represents the value on the right side of the operator in several stand-alone functions.
i	[in] An input parameter that represents the value on the right side of the operator in several stand-alone functions.
bits	[in] An input parameter that represents the value on the right side of the operator in several stand-alone functions.

Members

Public Constructors

NAME	DESCRIPTION
SafeInt::SafeInt	Default constructor.

Assignment Operators

NAME	SYNTAX
=	`template<typename U>` `SafeInt<T,E>& operator= (const U& rhs)`
=	`SafeInt<T,E>& operator= (const T& rhs) throw()`
=	`template<typename U>` `SafeInt<T,E>& operator= (const SafeInt<U, E>& rhs)`
=	`SafeInt<T,E>& operator= (const SafeInt<T,E>& rhs) throw()`

Casting Operators

NAME	SYNTAX
bool	`operator bool() throw()`
char	`operator char() const`
signed char	`operator signed char() const`
unsigned char	`operator unsigned char() const`
__int16	`operator __int16() const`
unsigned __int16	`operator unsigned __int16() const`
__int32	`operator __int32() const`
unsigned __int32	`operator unsigned __int32() const`
long	`operator long() const`
unsigned long	`operator unsigned long() const`
__int64	`operator __int64() const`
unsigned __int64	`operator unsigned __int64() const`
wchar_t	`operator wchar_t() const`

Comparison Operators

NAME	SYNTAX
<	`template<typename U>` `bool operator< (U rhs) const throw()`
<	`bool operator< (SafeInt<T,E> rhs) const throw()`

NAME	SYNTAX

>=	`template<typename U>`
	
	`bool operator>= (U rhs) const throw()`

>=	`Bool operator>= (SafeInt<T,E> rhs) const throw()`

>	`template<typename U>`
	
	`bool operator> (U rhs) const throw()`

>	`Bool operator> (SafeInt<T,E> rhs) const throw()`

<=	`template<typename U>`
	
	`bool operator<= (U rhs) const throw()`

<=	`bool operator<= (SafeInt<T,E> rhs) const throw()`

==	`template<typename U>`
	
	`bool operator== (U rhs) const throw()`

==	`bool operator== (bool rhs) const throw()`
==	`bool operator== (SafeInt<T,E> rhs) const throw()`

!=	`template<typename U>`
	
	`bool operator!= (U rhs) const throw()`

!=	`bool operator!= (bool b) const throw()`
!=	`bool operator!= (SafeInt<T,E> rhs) const throw()`

Arithmetic Operators

NAME	SYNTAX
+	`const SafeInt<T,E>& operator+ () const throw()`
-	`SafeInt<T,E> operator- () const`
++	`SafeInt<T,E>& operator++ ()`
--	`SafeInt<T,E>& operator-- ()`

%	`template<typename U>`
	
	`SafeInt<T,E> operator% (U rhs) const`

%	`SafeInt<T,E> operator% (SafeInt<T,E> rhs) const`

NAME	SYNTAX

%=	`template<typename U>`
	
	`SafeInt<T,E>& operator%= (U rhs)`

%=	`template<typename U>`
	
	`SafeInt<T,E>& operator%= (SafeInt<U, E> rhs)`

*	`template<typename U>`
	
	`SafeInt<T,E> operator* (U rhs) const`

*	`SafeInt<T,E> operator* (SafeInt<T,E> rhs) const`
*=	`SafeInt<T,E>& operator*= (SafeInt<T,E> rhs)`

*=	`template<typename U>`
	
	`SafeInt<T,E>& operator*= (U rhs)`

*=	`template<typename U>`
	
	`SafeInt<T,E>& operator*= (SafeInt<U, E> rhs)`

/	`template<typename U>`
	
	`SafeInt<T,E> operator/ (U rhs) const`

/	`SafeInt<T,E> operator/ (SafeInt<T,E> rhs ) const`
/=	`SafeInt<T,E>& operator/= (SafeInt<T,E> i)`

/=	`template<typename U>`
	
	`SafeInt<T,E>& operator/= (U i)`

/=	`template<typename U>`
	
	`SafeInt<T,E>& operator/= (SafeInt<U, E> i)`

+	`SafeInt<T,E> operator+ (SafeInt<T,E> rhs) const`

+	`template<typename U>`
	
	`SafeInt<T,E> operator+ (U rhs) const`

+=	`SafeInt<T,E>& operator+= (SafeInt<T,E> rhs)`

+=	`template<typename U>`
	
	`SafeInt<T,E>& operator+= (U rhs)`

NAME	SYNTAX

+=	`template<typename U>`
	
	`SafeInt<T,E>& operator+= (SafeInt<U, E> rhs)`

-	`template<typename U>`
	
	`SafeInt<T,E> operator- (U rhs) const`

-	`SafeInt<T,E> operator- (SafeInt<T,E> rhs) const`
-=	`SafeInt<T,E>& operator-= (SafeInt<T,E> rhs)`

-=	`template<typename U>`
	
	`SafeInt<T,E>& operator-= (U rhs)`

-=	`template<typename U>`
	
	`SafeInt<T,E>& operator-= (SafeInt<U, E> rhs)`

Logical Operators

NAME	SYNTAX
!	`bool operator !() const throw()`
~	`SafeInt<T,E> operator~ () const throw()`

<<	`template<typename U>`
	
	`SafeInt<T,E> operator<< (U bits) const throw()`

<<	`template<typename U>`
	
	`SafeInt<T,E> operator<< (SafeInt<U, E> bits) const throw()`

<<=	`template<typename U>`
	
	`SafeInt<T,E>& operator<<= (U bits) throw()`

<<=	`template<typename U>`
	
	`SafeInt<T,E>& operator<<= (SafeInt<U, E> bits) throw()`

>>	`template<typename U>`
	
	`SafeInt<T,E> operator>> (U bits) const throw()`

NAME	SYNTAX
>>	`template<typename U>` `SafeInt<T,E> operator>> (SafeInt<U, E> bits) const throw()`
>>=	`template<typename U>` `SafeInt<T,E>& operator>>= (U bits) throw()`
>>=	`template<typename U>` `SafeInt<T,E>& operator>>= (SafeInt<U, E> bits) throw()`
&	`SafeInt<T,E> operator& (SafeInt<T,E> rhs) const throw()`
&	`template<typename U>` `SafeInt<T,E> operator& (U rhs) const throw()`
&=	`SafeInt<T,E>& operator&= (SafeInt<T,E> rhs) throw()`
&=	`template<typename U>` `SafeInt<T,E>& operator&= (U rhs) throw()`
&=	`template<typename U>` `SafeInt<T,E>& operator&= (SafeInt<U, E> rhs) throw()`
^	`SafeInt<T,E> operator^ (SafeInt<T,E> rhs) const throw()`
^	`template<typename U>` `SafeInt<T,E> operator^ (U rhs) const throw()`
^=	`SafeInt<T,E>& operator^= (SafeInt<T,E> rhs) throw()`
^=	`template<typename U>` `SafeInt<T,E>& operator^= (U rhs) throw()`
^=	`template<typename U>` `SafeInt<T,E>& operator^= (SafeInt<U, E> rhs) throw()`
|	`SafeInt<T,E> operator&#124; (SafeInt<T,E> rhs) const throw()`
|	`template<typename U>` `SafeInt<T,E> operator&#124; (U rhs) const throw()`

NAME	SYNTAX
|=	`SafeInt<T,E>& operator&#124;= (SafeInt<T,E> rhs) throw()`
|=	`template<typename U>` `SafeInt<T,E>& operator&#124;= (U rhs) throw()`
|=	`template<typename U>` `SafeInt<T,E>& operator&#124;= (SafeInt<U, E> rhs) throw()`

Remarks

The `SafeInt` class protects against integer overflow in mathematical operations. For example, consider adding two 8-bit integers: one has a value of 200 and the second has a value of 100. The correct mathematical operation would be 200 + 100 = 300. However, because of the 8-bit integer limit, the upper bit will be lost and the compiler will return 44 (300 - 2^8) as the result. Any operation that depends on this mathematical equation will generate unexpected behavior.

The `SafeInt` class checks whether an arithmetic overflow occurs or whether the code tries to divide by zero. In both cases, the class calls the error handler to warn the program of the potential problem.

This class also lets you compare two different types of integers as long as they are `SafeInt` objects. Typically, when you perform a comparison, you must first convert the numbers to be the same type. Casting one number to another type often requires checks to make sure that there is no loss of data.

The Operators table in this topic lists the mathematical and comparison operators supported by the `SafeInt` class. Most mathematical operators return a `SafeInt` object of type `T`.

Comparison operations between a `SafeInt` and an integral type can be performed in either direction. For example, both `SafeInt<int>(x) < y` and `y> SafeInt<int>(x)` are valid and will return the same result.

Many binary operators do not support using two different `SafeInt` types. One example of this is the `&` operator. `SafeInt<T, E> & int` is supported, but `SafeInt<T, E> & SafeInt<U, E>` is not. In the latter example, the compiler does not know what type of parameter to return. One solution to this problem is to cast the second parameter back to the base type. By using the same parameters, this can be done with `SafeInt<T, E> & (U)SafeInt<U, E>`.

NOTE

For any bitwise operations, the two different parameters should be the same size. If the sizes differ, the compiler will throw an ASSERT exception. The results of this operation cannot be guaranteed to be accurate. To resolve this issue, cast the smaller parameter until it is the same size as the larger parameter.

For the shift operators, shifting more bits than exist for the template type will throw an ASSERT exception. This will have no effect in release mode. Mixing two types of SafeInt parameters is possible for the shift operators because the return type is the same as the original type. The number on the right side of the operator only indicates the number of bits to shift.

When you perform a logical comparison with a SafeInt object, the comparison is strictly arithmetic. For example, consider these expressions:

- `SafeInt<uint>((uint)~0) > -1`

- `((uint)~0) > -1`

The first statement resolves to **true**, but the second statement resolves to `false`. The bitwise negation of 0 is 0xFFFFFFFF. In the second statement, the default comparison operator compares 0xFFFFFFFF to 0xFFFFFFFF and considers them to be equal. The comparison operator for the `SafeInt` class realizes that the second parameter is negative whereas the first parameter is unsigned. Therefore, although the bit representation is identical, the `SafeInt` logical operator realizes that the unsigned integer is larger than -1.

Be careful when you use the `SafeInt` class together with the `?:` ternary operator. Consider the following line of code.

```
Int x = flag ? SafeInt<unsigned int>(y) : -1;
```

The compiler converts it to this:

```
Int x = flag ? SafeInt<unsigned int>(y) : SafeInt<unsigned int>(-1);
```

If `flag` is `false`, the compiler throws an exception instead of assigning the value of -1 to `x`. Therefore, to avoid this behavior, the correct code to use is the following line.

```
Int x = flag ? (int) SafeInt<unsigned int>(y) : -1;
```

`T` and `U` can be assigned a Boolean type, character type, or integer type. The integer types can be signed or unsigned and any size from 8 bits to 64 bits.

> **NOTE**
>
> Although the `SafeInt` class accepts any kind of integer, it performs more efficiently with unsigned types.

`E` is the error handling mechanism that `SafeInt` uses. Two error handling mechanisms are provided with the SafeInt library. The default policy is `SafeIntErrorPolicy_SafeIntException`, which throws a SafeIntException Class exception when an error occurs. The other policy is `SafeIntErrorPolicy_InvalidParameter`, which stops the program if an error occurs.

There are two options to customize the error policy. The first option is to set the parameter `E` when you create a `SafeInt`. Use this option when you want to change the error handling policy for just one `SafeInt`. The other option is to define _SAFEINT_DEFAULT_ERROR_POLICY to be your customized error-handling class before you include the `SafeInt` library. Use this option when you want to change the default error handling policy for all instances of the `SafeInt` class in your code.

> **NOTE**
>
> A customized class that handles errors from the SafeInt library should not return control to the code that called the error handler. After the error handler is called, the result of the `SafeInt` operation cannot be trusted.

Inheritance Hierarchy

`SafeInt`

Requirements

Header: safeint.h

Namespace: msl::utilities

SafeInt::SafeInt

Constructs a `SafeInt` object.

```
SafeInt() throw

SafeInt (
   const T& i
) throw ()

SafeInt (
   bool b
) throw ()

template <typename U>
SafeInt (
   const SafeInt <U, E>& u
)

I template <typename U>
SafeInt (
   const U& i
)
```

Parameters

i
[in] The value for the new `SafeInt` object. This must be a parameter of type T or U, depending on the constructor.

b
[in] The Boolean value for the new `SafeInt` object.

u
[in] A `SafeInt` of type U. The new `SafeInt` object will have the same value as *u*, but will be of type T.

U The type of data stored in the `SafeInt`. This can be either a Boolean, character, or integer type. If it is an integer type, it can be signed or unsigned and be between 8 and 64 bits.

Remarks

The input parameter for the constructor, *i* or *u*, must be a Boolean, character, or integer type. If it is another type of parameter, the `SafeInt` class calls static_assert to indicate an invalid input parameter.

The constructors that use the template type `U` automatically convert the input parameter to the type specified by `T`. The `SafeInt` class converts the data without any loss of data. It reports to the error handler `E` if it cannot convert the data to type `T` without data loss.

If you create a `SafeInt` from a Boolean parameter, you need to initialize the value immediately. You cannot construct a `SafeInt` using the code `SafeInt<bool> sb;`. This will generate a compile error.

SafeInt Functions

4/1/2019 • 5 minutes to read • Edit Online

The SafeInt library provides several functions that you can use without creating an instance of the SafeInt class. If you want to protect a single mathematical operation from integer overflow, you can use these functions. If you want to protect multiple mathematical operations, you should create `SafeInt` objects. It is more efficient to create `SafeInt` objects than to use these functions multiple times.

These functions enable you to compare or perform mathematical operations on two different types of parameters without having to convert them to the same type first.

Each of these functions has two template types: `T` and `U`. Each of these types can be a Boolean, character, or integral type. Integral types can be signed or unsigned and any size from 8 bits to 64 bits.

NOTE
The latest version of this library is located at https://github.com/dcleblanc/SafeInt.

In This Section

FUNCTION	DESCRIPTION
SafeAdd	Adds two numbers and protects against overflow.
SafeCast	Casts one type of parameter to another type.
SafeDivide	Divides two numbers and protects against dividing by zero.
SafeEquals, SafeGreaterThan, SafeGreaterThanEquals, SafeLessThan, SafeLessThanEquals, SafeNotEquals	Compares two numbers. These functions enable you to compare two different types of numbers without changing their types.
SafeModulus	Performs the modulus operation on two numbers.
SafeMultiply	Multiplies two numbers together and protects against overflow.
SafeSubtract	Subtracts two numbers and protects against overflow.

Related Sections

SECTION	DESCRIPTION
SafeInt	The `SafeInt` class.
SafeIntException	The exception class specific to the SafeInt library.

SafeAdd

Adds two numbers in a way that protects against overflow.

```
template<typename T, typename U>
inline bool SafeAdd (
   T t,
   U u,
   T& result
) throw ();
```

Parameters

t
[in] The first number to add. This must be of type T.

u
[in] The second number to add. This must be of type U.

result
[out] The parameter where `SafeAdd` stores the result.

Return Value

true if no error occurs; **false** if an error occurs.

SafeCast

Casts one type of number to another type.

```
template<typename T, typename U>
inline bool SafeCast (
   const T From,
   U& To
);
```

Parameters

From
[in] The source number to convert. This must be of type `T`.

To
[out] A reference to the new number type. This must be of type `U`.

Return Value

true if no error occurs; **false** if an error occurs.

SafeDivide

Divides two numbers in a way that protects against dividing by zero.

```
template<typename T, typename U>
inline bool SafeDivide (
   T t,
   U u,
   T& result
) throw ();
```

Parameters

t

[in] The divisor. This must be of type T.

u
[in] The dividend. This must be of type U.

result
[out] The parameter where `SafeDivide` stores the result.

Return Value

true if no error occurs; **false** if an error occurs.

SafeEquals

Compares two numbers to determine whether they are equal.

```
template<typename T, typename U>
inline bool SafeEquals (
   const T t,
   const U u
) throw ();
```

Parameters

t
[in] The first number to compare. This must be of type T.

u
[in] The second number to compare. This must be of type U.

Return Value

true if *t* and *u* are equal; otherwise **false**.

Remarks

The method enhances `==` because `SafeEquals` enables you to compare two different types of numbers.

SafeGreaterThan

Compares two numbers.

```
template<typename T, typename U>
inline bool SafeGreaterThan (
   const T t,
   const U u
) throw ();
```

Parameters

t
[in] The first number to compare. This must be of type `T`.

u
[in] The second number to compare. This must be of type `U`.

Return Value

true if *t* is greater than *u*; otherwise **false**.

Remarks

`SafeGreaterThan` extends the regular comparison operator by enabling you to compare two different types of numbers.

SafeGreaterThanEquals

Compares two numbers.

```
template <typename T, typename U>
inline bool SafeGreaterThanEquals (
   const T t,
   const U u
) throw ();
```

Parameters

t
[in] The first number to compare. This must be of type `T`.

u
[in] The second number to compare. This must be of type `U`.

Return Value

true if *t* is greater than or equal to *u*; otherwise **false**.

Remarks

`SafeGreaterThanEquals` enhances the standard comparison operator because it enables you to compare two different types of numbers.

SafeLessThan

Determines whether one number is less than another.

```
template<typename T, typename U>
inline bool SafeLessThan (
   const T t,
   const U u
) throw ();
```

Parameters

t
[in] The first number. This must be of type `T`.

u
[in] The second numer. This must be of type `U`.

Return Value

true if *t* is less than *u*; otherwise **false**.

Remarks

This method enhances the standard comparison operator because `SafeLessThan` enables you to compare two different types of number.

SafeLessThanEquals

Compares two numbers.

```
template <typename T, typename U>
inline bool SafeLessThanEquals (
   const T t,
   const U u
) throw ();
```

Parameters

t
[in] The first number to compare. This must be of type `T`.

u
[in] The second number to compare. This must be of type `U`.

Return Value

true if *t* is less than or equal to *u*; otherwise **false**.

Remarks

`SafeLessThanEquals` extends the regular comparison operator by enabling you to compare two different types of numbers.

SafeModulus

Performs the modulus operation on two numbers.

```
template<typename T, typename U>
inline bool SafeModulus (
   const T t,
   const U u,
   T& result
) throw ();
```

Parameters

t
[in] The divisor. This must be of type `T`.

u
[in] The dividend. This must be of type `U`.

result
[out] The parameter where `SafeModulus` stores the result.

Return Value

true if no error occurs; **false** if an error occurs.

SafeMultiply

Multiplies two numbers together in a way that protects against overflow.

```
template<typename T, typename U>
inline bool SafeMultiply (
   T t,
   U u,
   T& result
) throw ();
```

Parameters

t
[in] The first number to multiply. This must be of type `T`.

u
[in] The second number to multiply. This must be of type `U`.

result
[out] The parameter where `SafeMultiply` stores the result.

Return Value

`true` if no error occurs; `false` if an error occurs.

SafeNotEquals

Determines if two numbers are not equal.

```
template<typename T, typename U>
inline bool SafeNotEquals (
   const T t,
   const U u
) throw ();
```

Parameters

t
[in] The first number to compare. This must be of type `T`.

u
[in] The second number to compare. This must be of type `U`.

Return Value

true if *t* and *u* are not equal; otherwise **false**.

Remarks

The method enhances `!=` because `SafeNotEquals` enables you to compare two different types of numbers.

SafeSubtract

Subtracts two numbers in a way that protects against overflow.

```
template<typename T, typename U>
inline bool SafeSubtract (
   T t,
   U u,
   T& result
) throw ();
```

Parameters

t
[in] The first number in the subtraction. This must be of type `T`.

u
[in] The number to subtract from *t*. This must be of type `U`.

result

[out] The parameter where `SafeSubtract` stores the result.

Return Value

true if no error occurs; **false** if an error occurs.

SafeIntException Class

4/1/2019 • 2 minutes to read • Edit Online

The `SafeInt` class uses `SafeIntException` to identify why a mathematical operation cannot be completed.

NOTE

The latest version of this library is located at https://github.com/dcleblanc/SafeInt.

Syntax

```
class SafeIntException;
```

Members

Public Constructors

NAME	DESCRIPTION
SafeIntException::SafeIntException	Creates a `SafeIntException` object.

Remarks

The SafeInt class is the only class that uses the `SafeIntException` class.

Inheritance Hierarchy

`SafeIntException`

Requirements

Header: safeint.h

Namespace: msl::utilities

SafeIntException::SafeIntException

Creates a `SafeIntException` object.

```
SafeIntException();

SafeIntException(
   SafeIntError code
);
```

Parameters

code
[in] An enumerated data value that describes the error that occurred.

Remarks

The possible values for *code* are defined in the file Safeint.h. For convenience, the possible values are also listed here.

- `SafeIntNoError`
- `SafeIntArithmeticOverflow`
- `SafeIntDivideByZero`

OpenMP in Visual C++

4/22/2019 • 2 minutes to read • Edit Online

The OpenMP C and C++ application program interface lets you write applications that effectively use multiple processors. Visual C++ supports the OpenMP 2.0 standard.

In This Section

Library Reference
Provides links to constructs used in the OpenMP API.

C and C++ Application Program Interface
Discusses the OpenMP C and C++ API, as documented in the version 2.0 specification from the OpenMP Architecture Review Board.

Related Sections

/openmp (Enable OpenMP 2.0 Support)
Causes the compiler to process `#pragma omp`.

Predefined Macros
Names the predefined ANSI C and Microsoft C++ implementation macros. See the _OPENMP macro.

MFC and ATL

5/15/2019 • 2 minutes to read • Edit Online

The Microsoft Foundation Classes (MFC) provide a C++ object-oriented wrapper over Win32 for rapid development of native desktop applications. The Active Template Library (ATL) is a wrapper library that simplifies COM development and is used extensively for creating ActiveX controls.

You can create MFC or ATL programs with Visual Studio Community Edition or higher. The Express editions do not support MFC or ATL.

In Visual Studio 2015, Visual C++ is an optional component, and MFC and ATL components are optional sub-components under Visual C++. If you do not select these components when you first install Visual Studio, you will be prompted to install them the first time you attempt to create or open an MFC or ATL project.

In Visual Studio 2017 and later, MFC and ATL are optional sub-components under the **Desktop development with C++** workload in the Visual Studio Installer program. You can install ATL support without MFC, or combined MFC and ATL support (MFC depends on ATL). For more information about workloads and components, see Install Visual Studio.

Related Articles

TITLE	DESCRIPTION
MFC Desktop Applications	Microsoft Foundation Classes provide a thin object-oriented wrapper over Win32 to enable rapid development of GUI applications in C++.
ATL COM Desktop Components	ATL provides class templates and other use constructs to simplify creation of COM objects in C++.
ATL/MFC Shared Classes	References for CStringT Class and other classes that are shared by MFC and ATL.
Working with Resource Files	The resource editor lets you edit UI resources such as strings, images, and dialog boxes.
C++ in Visual Studio	Parent topic for all C++ content in the MSDN library.

Parallel Programming in Visual C++

5/15/2019 • 2 minutes to read • Edit Online

Visual C++ provides the following technologies to help you create multi-threaded and parallel programs that take advantage of multiple cores and use the GPU for general purpose programming.

Related Articles

TITLE	DESCRIPTION
Auto-Parallelization and Auto-Vectorization	Compiler optimizations that speed up code.
Concurrency Runtime	Classes that simplify the writing of programs that use data parallelism or task parallelism.
C++ AMP (C++ Accelerated Massive Parallelism)	Classes that enable the use of modern graphics processors for general purpose programming.
Multithreading Support for Older Code (Visual C++)	Older technologies that may be useful in older applications. For new apps, use the Concurrency Runtime or C++ AMP.
OpenMP	The Microsoft implementation of the OpenMP API.
C++ in Visual Studio	This section of the documentation contains information about most of the features of Visual C++.

Data Access in Visual C++

5/15/2019 • 2 minutes to read • Edit Online

Virtually all database products, SQL and NoSQL, provide an interface for native C++ applications. The industry standard interface is ODBC which is supported by all major SQL database products and many NoSQL products. For non-Microsoft products, consult the vendor for more information. Third-party libraries with various license terms are also available.

Since 2011 Microsoft has aligned on ODBC as the standard for native applications to connecting to Microsoft SQL Server databases, both on-premises and in the cloud. For more information, see Data Access Programming (MFC-ATL). C++/CLI libraries can use either the native ODBC drivers or ADO.NET. For more information, see Data Access Using ADO.NET (C++/CLI) and Accessing data in Visual Studio.

In This Section

Data Access Programming (MFC/ATL)
Describes legacy data access programming with Visual C++, where the preferred way is to use one of the class libraries such as the Active Template Class Library (ATL) or Microsoft Foundation Class (MFC) Library, which simplify working with the database APIs.

Open Database Connectivity (ODBC)
The Microsoft Foundation Classes (MFC) library supplies classes for programming with Open Database Connectivity (ODBC).

OLE DB Programming
A mostly legacy interface which is still required in some scenarios, specifically when you are programming against linked servers.

Related Topics

Connect to SQL Database using C and C++
Connect to Azure SQL Database from C or C++ applications.

Microsoft Azure Storage Client Library for C++
Azure Storage is a cloud storage solution for modern applications that rely on durability, availability, and scalability to meet the needs of their customers. Connect to Azure Storage from C++ by using the Azure Storage Client Library for C++.

ODBC Driver for SQL Server
The latest ODBC driver provides robust data access to Microsoft SQL Server and Microsoft Azure SQL Database for C/C++ based applications. Provides support for features including always encrypted, Azure Active Directory, and AlwaysOn Availability Groups. Also available for MacOS and Linux.

OLE DB Driver for SQL Server
The latest OLE DB driver is a stand-alone data access application programming interface (API) that supports Microsoft SQL Server and Microsoft Azure SQL Database.

Microsoft Azure C and C++ Developer Center
Azure makes it easy to build C++ applications with increased flexibility, scalability and reliability using tools you love.

How to use Blob Storage from C++

Azure Blob storage is a service that stores unstructured data in the cloud as objects/blobs. Blob storage can store any type of text or binary data, such as a document, media file, or application installer. Blob storage is also referred to as object storage.

ODBC Programmer's Reference
The ODBC interface is designed for use with the C programming language. Use of the ODBC interface spans three areas: SQL statements, ODBC function calls, and C programming.

See also

C++ in Visual Studio