Visual C++ change history 2003 - 2015

msdn.microsoft.com/en-us/library/bb531344.aspx

Visual C++ 2015 Conformance Changes

C Runtime Library (CRT)

General Changes

- Refactored binaries The CRT Library has been refactored into a two different binaries, a Universal CRT (ucrtbase), which contains most of the standard functionality, and a VC Runtime Library (vcruntime140), which contains the compiler-related functionality, such as exception handling, and intrinsics. If you are using the default project settings, then this change does not impact you since the linker will use the new default libraries automatically. If you have set the project's **Linker** property **Ignore All Default Libraries** to **Yes** or you are using the /NODEFAULTLIB linker option on the command line, then you must update your list of libraries (in the **Additional Dependencies** property) to include the new, refactored libraries. Replace the old CRT library (libcmt.lib, libcmtd.lib, msvcrt.lib, msvcrtd.lib) with the equivalent refactored libraries. For each of the two refactored libraries, there are static (.lib) and dynamic (.dll) versions, and release (with no suffix) and debug versions (with the "d" suffix). The dynamic versions have an import library that you link with. The two refactored libraries are Universal CRT, specifically ucrtbase.dll or .lib, ucrtbased.dll or .lib, and the VC runtime library, libvcruntime.lib, libvcruntime.dll, libvcruntimed.lib, and libvcruntimed.dll. See CRT Library Features.

- localeconv The localeconv function declared in locale.h now works correctly when per-thread locale is enabled. In previous versions of the library, this function would return the lconv data for the global locale, not the thread's locale.

  If you use per thread locale, you should check your use of localeconv to see if your code assumes that the lconv data returned is for the global locale and modify it appropriately.

- C++ overloads of math library functions In previous versions, <math.h> defined some, but not all, of the C++ overloads for the math library functions. <cmath> defined the remaining overloads, so to get all of the overloads, one needed to include the <cmath> header. This led to problems with function overload resolution in code that only included <math.h>. Now, all C++ overloads have been removed from <math.h> and are now present only in <cmath>.

  To resolve errors, include <cmath> to get the declarations of the functions that were removed from <math.h>. The following table lists the functions that were moved.

  Functions that were moved:

  1. double abs(double) and float abs(float)
  2. double pow(double, int), float pow(float, float), float pow(float, int), long double pow(long double, long double), long double pow(long double, int)
  3. float and long double versions of floating point functions acos, acosh, asin, asinh, atan, atanh, atan2, cbrt, ceil, copysign, cos, cosh, erf, erfc, exp, exp2, expm1, fabs, fdim, floor, fma, fmax, fmin, fmod, frexp, hypot, ilogb, ldexp, lgamma, llrint, llround, log, log10, log1p, log2, lrint, lround, modf, nearbyint, nextafter, nexttoward, remainder, remquo, rint, round, scalbln, scalbn, sin, sinh, sqrt, tan, tanh, tgamma, trunc

  If you have code that uses abs with a floating point type that only includes the math.h header, the floating point versions will no longer be available, so the call, even with a floating point argument, now resolves to abs(int). This produces the error:

  ```
  warning C4244: 'argument' : conversion from 'float' to 'int', possible loss of data
  ```

  The fix for this warning is to replace the call to abs with a floating point version of abs, such as fabs for a double argument or fabsf for a float argument, or include the cmath header and continue to use abs.

- Floating point conformance Many changes to the math library have been made to improve conformance to the IEEE-754 and C11 Annex F specifications with respect to special case inputs such as NaNs and infinities. For example, quiet NaN inputs, which were often treated as errors in previous versions of the library, are no longer treated as errors. See IEEE 754 Standard and Annex F of the C11 Standard.

  These changes won't cause compile-time errors, but might cause programs to behave differently and more correctly according to the standard.

- FLT_ROUNDS In Visual Studio 2013, the FLT_ROUNDS macro expanded to a constant expression, which was incorrect because the rounding mode is configurable at runtime, for example, by calling fesetround. The FLT_ROUNDS macro is now dynamic and correctly reflects the current rounding mode.

and

- new and delete In previous versions of the library, the implementation-defined operator new and delete functions were exported from the runtime library DLL (for example, msvcr120.dll). These operator functions are now always statically linked into your binaries, even when using the runtime library DLLs.

  This is not a breaking change for native or mixed code (/clr), however for code compiled as /clr:pure, this might cause your code to fail to compile. If you compile code as /clr:pure, you may need to add #include <new> or #include <new.h> to work around build errors due to this change. Note that /clr:pure is deprecated in Visual Studio 2015 and might be removed in future releases.

- _beginthread and _beginthreadex The _beginthread and _beginthreadex functions now hold a reference to the module in which the thread procedure is defined for the duration of the thread. This helps to ensure that modules are not unloaded until a thread has run to completion.

- va_start and reference types When compiling C++ code, va_start now validates at compile-time that the argument passed to it is not of reference type. Reference-type arguments are prohibited by the C++ Standard.

and

- The printf and scanf family of functions are now defined inline. The definitions of all of the printf and scanf functions have been moved inline into <stdio.h>, <conio.h>, and other CRT headers. This is a breaking change that leads to a linker error (LNK2019, unresolved external symbol) for any programs that declared these functions locally without including the appropriate CRT headers. If possible, you should update the code to include the CRT headers (that is, add #include <stdio.h>) and the inline functions, but if you do not want to modify your code to include these header files, an alternative solution is to add an additional library to your linker input, legacy_stdio_definitions.lib.

  To add this library to your linker input in the IDE, open the context menu for the project node, choose **Properties**, then in the **Project Properties** dialog box, choose **Linker**, and edit the **Linker Input** to add legacy_stdio_definitions.lib to the semi-colon-separated list.

  If your project links with static libraries that were compiled with a release of Visual C++ earlier than 2015, the linker might report an unresolved external symbol. These errors might reference internal stdio definitions for _iob, _iob_func, or related imports for certain stdio functions in the form of _imp_*. Microsoft recommends that you recompile all static libraries with the latest version of the Visual C++ compiler and libraries when you upgrade a project. If the library is a third-party library for which source is not available, you should either request an updated binary from the third party or encapsulate your usage of that library into a separate DLL that you compile with the older version of the Visual C++ compiler and libraries.

  > **Warning**
  >
  > If you are linking with Windows SDK 8.1 or earlier, you might encounter these unresolved external symbol errors. In that case, you should resolve the error by adding legacy_stdio_definitions.lib to the linker input as described previously.

  To troubleshoot unresolved symbol errors, you can try using dumpbin.exe to examine the symbols defined in a binary. Try the following command line to view symbols defined in a library.

  C++

  ```
  dumpbin.exe /LINKERMEMBER somelibrary.lib
  ```

- gets and _getws The gets and _getws functions have been removed. The gets function was removed from the C Standard Library in C11 because it cannot be used securely. The _getws function was a Microsoft extension that was equivalent to gets but for wide strings. As alternatives to these functions, consider use of fgets, fgetws, gets_s, and _getws_s.

- _cgets and _cgetws The _cgets and _cgetws functions have been removed. As alternatives to these functions, consider use of _cgets_s and _cgetws_s.

- Infinity and NaN Formatting In previous versions, infinities and NaNs would be formatted using a set of Visual C++-specific sentinel strings.
    - Infinity: 1.#INF
    - Quiet NaN: 1.#QNAN
    - Signaling NaN: 1.#SNAN
    - Indefinite NaN: 1.#IND

  Any of these may have been prefixed by a sign and may have been formatted slightly differently depending on field width and precision (sometimes with unusual effects, e.g. printf("%.2f\n", INFINITY) would print 1.#J because the #INF would be "rounded" to a precision of 2 digits). C99 introduced new requirements on how infinities and NaNs are to be formatted. The Visual C++ implementation now conforms to these requirements. The new strings are as follows:
    - Infinity: inf
    - Quiet NaN: nan
    - Signaling NaN: nan(snan)
    - Indefinite NaN:nan(ind)

  Any of these may be prefixed by a sign. If a capital format specifier is used (%F instead of %f) then the strings are printed in capital letters (INF instead of inf), as is required.

  The scanf functions have been modified to parse these new strings, so these strings will round-trip through printf and scanf.

- Floating point formatting and parsing New floating point formatting and parsing algorithms have been introduced to improve correctness. This change affects the printf and scanf families of functions, as well as functions like strtod.

  The old formatting algorithms would generate only a limited number of digits, then would fill the remaining decimal places with zero. This is usually good enough to generate strings that will round-trip back to the original floating point value, but it's not great if you want the exact value (or the closest decimal representation thereof). The new formatting algorithms generate as many digits as are required to represent the value (or to fill the specified precision). As an example of the improvement; consider the results when printing a large power of two:

  C++

  ```
  printf("%.0f\n", pow(2.0, 80))


  Old:  1208925819614629200000000    New:  120892581961462917470617 6
  ```

  The old parsing algorithms would consider only up to 17 significant digits from the input string and would discard the rest of the digits. This is sufficient to generate a very close approximation of the value represented by the string, and the result is usually very close to the correctly rounded result. The new implementation considers all present digits and produces the correctly rounded result for all inputs (up to 768 digits in length). In addition, these functions now respect the rounding mode (controllable via fesetround). This is a potentially breaking behavior change because these functions might output different results. The new results are always more correct than the old results.

- Hexadecimal and infinity/NaN floating point parsing The floating point parsing algorithms will now parse hexadecimal floating point strings (such as those generated by the %a and %A printf format specifiers) and all infinity and NaN strings that are generated by the printf functions, as described above.

- %A and %a zero padding The %a and %A format specifiers format a floating point number as a hexadecimal mantissa and binary exponent. In previous versions, the printf functions would incorrectly zero-pad strings. For example, printf("%07.0a\n", 1.0) would print 00x1p+0, where it should print 0x01p+0. This has been fixed.

- %A and %a precision The default precision of the %A and %a format specifiers was 6 in previous versions of the library. The default precision is now 13 for conformance with the C Standard.

  This is a runtime behavior change in the output of any function that uses a format string with %A or %a. In the old behavior, the output using the %A specifier might be "1.1A2B3Cp+111". Now the output for the same value is "1.1A2B3C4D5E6F7p+111". To get the old behavior, you can specify the precision, for example, %.6A. See Precision Specification.

- %F specifier The %F format/conversion specifier is now supported. It is functionally equivalent to the %f format specifier, except that infinities and NaNs are formatted using capital letters.

  In previous versions, the implementation used to parse F and N as length modifiers. This behavior dated back to the age of segmented address spaces: these length modifiers were used to indicate far and near pointers, respectively, as in %Fp or %Ns. This behavior has been removed. If %F is encountered, it is now treated as the %F format specifier; if %N is encountered, it is now treated as an invalid parameter.

- Exponent formatting The %e and %E format specifiers format a floating point number as a decimal mantissa and exponent. The %g and %G format specifiers also format numbers in this form in some cases. In previous versions, the CRT would always generate strings with three-digit exponents. For example, printf("%e\n", 1.0) would print 1.000000e+000. This was incorrect: C requires that if the exponent is representable using only one or two digits, then only two digits are to be printed.

  In Visual Studio 2005 a global conformance switch was added: _set_output_format. A program could call this function with the argument _TWO_DIGIT_EXPONENT, to enable conforming exponent printing. The default behavior has been changed to the standards-conforming exponent printing mode.

- Format string validation In previous versions, the printf and scanf functions would silently accept many invalid format strings, sometimes with unusual effects. For example, %hlhlhld would be treated as %d. All invalid format strings are now treated as invalid parameters.

- fopen mode string validation

  In previous versions, the fopen family of functions silently accepted some invalid mode strings (e.g. r+b+). Invalid mode strings are now detected and treated as invalid parameters.

- _O_U8TEXT mode

  The _setmode function now correctly reports the mode for streams opened in_O_U8TEXT mode. In previous versions of the library, it would report such streams as being opened in _O_WTEXT.

  This is a breaking change if your code interprets the _O_WTEXT mode for streams where the encoding is UTF-8. If your application doesn't support UTF_8, consider adding support for this increasingly common encoding.

- snprintf and vsnprintf The snprintf and vsnprintf functions are now implemented. Older code often provided definitions macro versions of these functions because they were not implemented by the CRT library, but these are no longer needed in newer versions. If snprintf or vsnprintf is defined as a macro before including <stdio.h>, compilation now fails with an error that indicates where the macro was defined.

  Normally, the fix to this problem is to delete any declarations of snprintf or vsnprintf in user code.

- tmpnam Generates Usable File Names In previous versions, the tmpnam and tmpnam_s functions generated file names in the root of the drive (such as \sd3c.). These functions now generate usable file name paths in a temporary directory.

- FILE Encapsulation In previous versions, the FILE type was completely defined in <stdio.h>, so it was possible for user code to reach into a FILE and modify its internals. The stdio library has been changed to hide implementation details. As part of this, FILE as defined in <stdio.h> is now an opaque type and its members are inaccessible from outside of the CRT itself.

- _outp and _inp The functions _outp, _outpw, _outpd, _inp, _inpw, and _inpd have been removed.

, , and

- strtof and wcstof The strtof and wcstof functions failed to set errno to ERANGE when the value was not representable as a float. This has been fixed. (Note that this error was specific to these two functions; the strtod, wcstod, strtold, and wcstold functions were unaffected.) This is a runtime breaking change.

- Aligned allocation functions In previous versions, the aligned allocation functions (_aligned_malloc, _aligned_offset_malloc, etc.) would silently accept requests for a block with an alignment of 0. The requested alignment must be a power of two, which zero is not. This has been fixed, and a requested alignment of 0 is now treated as an invalid parameter. This is a runtime breaking change.

- Heap functions The _heapadd, _heapset, and _heapused functions have been removed. These functions have been nonfunctional since the CRT was updated to use the Windows heap.

- smallheap The smalheap link option has been removed. See Link Options.

- wcstok The signature of the wcstok function has been changed to match what is required by the C Standard. In previous versions of the library, the signature of this function was:

  C++

  ```
  wchar_t* wcstok(wchar_t*, wchar_t const*)
  ```

  It used an internal, per-thread context to track state across calls, as is done for strtok. The function now has the signature wchar_t* wcstok(wchar_t*, wchar_t const*, wchar_t**), and requires the caller to pass the context as a third argument to the function.

  A new _wcstok function has been added with the old signature to ease porting. When compiling C++ code, there is also an inline overload of wcstok that has the old signature. This overload is declared as deprecated. In C code, you may define_CRT_NON_CONFORMING_WCSTOK to cause _wcstok to be used in place of wcstok.

- clock In previous versions, the clock function was implemented using the Windows API GetSystemTimeAsFileTime. With this implementation, the clock function was sensitive to the system time, and was thus not necessarily monotonic. The clock function has been reimplemented in terms of QueryPerformanceCounter and is now monotonic.

- fstat and _utime In previous versions, the _stat, fstat, and _utime functions handle daylight savings time incorrectly. Prior to Visual Studio 2013, all of these functions incorrectly adjusted standard time times as if they were in daylight time.

  In Visual Studio 2013, the problem was fixed in the _stat family of functions, but the similar problems in the fstat and _utime families of functions were not fixed. This led to problems due to the inconsistency between the functions. The fstat and _utime families of functions have now been fixed, so all of these functions now handle daylight savings time correctly and consistently.

- asctime In previous versions, the asctime function would pad single-digit days with a leading zero, for example: Fri Jun 06 08:00:00 2014. The specification requires that such days be padded with a leading space, e.g. Fri Jun 6 08:00:00 2014. This has been fixed.

- strftime and wcsftime The strftime and wcsftime functions now support the %C, %D, %e, %F, %g, %G, %h, %n, %r, %R, %t, %T, %u, and %V format specifiers. Additionally, the E and O modifiers are parsed but ignored.

  The %c format specifier is specified as producing an "appropriate date and time representation" for the current locale. In the C locale, this representation is required to be the same as %a %b %e %T %Y. This is the same form as is produced by asctime. In previous versions, the %c format specifier incorrectly formatted times using a MM/DD/YY HH:MM:SS representation. This has been fixed.

- timespec and TIME_UTC The <time.h> header now defines the timespec type and the timespec_get function from the C11 Standard. In addition, the TIME_UTC macro, for use with the timespec_get function, is now defined. This is a breaking change for code that has a conflicting definition for any of these.
- CLOCKS_PER_SEC The CLOCKS_PER_SEC macro now expands to an integer of type clock_t, as required by the C language.

Standard Template Library

To enable new optimizations and debugging checks, the Visual Studio implementation of the C++ Standard Library intentionally breaks binary compatibility from one version to the next. Therefore, when the C++ Standard Library is used, object files and static libraries that are compiled by using different versions can't be mixed in one binary (EXE or DLL), and C++ Standard Library objects can't be passed between binaries that are compiled by using different versions. Such mixing emits linker errors about _MSC_VER mismatches. (_MSC_VER is the macro that contains the compiler's major version—for example, 1800 for Visual Studio 2013.) This check cannot detect DLL mixing, and cannot detect mixing that involves Visual C++ 2008 or earlier.

- STL include files Some changes have been made to the include structure in the STL headers. STL headers are allowed to include each other in unspecified ways. In general, you should write your code so that it carefully includes all of the headers that it needs according to the C++ standard and doesn't rely on which STL headers include which other STL headers. This makes code portable across versions and platforms. At least two header changes in Visual Studio 2015 affect user code. First, <string> no longer includes <iterator>. Second, <tuple> now declares std::array without including all of <array>, which can break code through the following combination of code constructs: your code has a variable named "array", and you have a using-directive "using namespace std;", and you include an STL header (such as <functional>) that includes <tuple>, which now declares std::array.
- steady_clock The <chrono> implementation of steady_clock has changed to meet the C++ Standard requirements for steadiness and monotonicity. steady_clock is now based on QueryPerformanceCounter and high_resolution_clock is now a typedef for steady_clock. As a result, in Visual C++ steady_clock::time_point is now a typedef for chrono::time_point<steady_clock>; however, this is not necessarily the case for other implementations.
- allocators and const We now require allocator equality/inequality comparisons to accept const arguments on both sides. If your allocators define these operators as follows:

  C++

  ```
  bool operator==(const MyAlloc& other)
  ```

  You should update these to declare them as const members.

  C++

  ```
  bool operator==(const MyAlloc& other) const
  ```

- const elements The C++ standard has always forbidden containers of const elements (such as vector<const T> or set<const T>). Visual C++ 2013 and earlier accepted such containers. In the current version, such containers fail to compile.

- std::allocator::deallocate In Visual C++ 2013 and earlier, std::allocator::deallocate(p, n) ignored the argument passed in for n. The C++ standard has always required that n be equal to the value passed as the first argument to the invocation of allocate which returned p. However, in the current version, the value of n is inspected. Code that passes arguments for n that differ from what the standard requires might crash at runtime.

- hash_map and hash_set The non-standard header files hash_map and hash_set are deprecated in Visual Studio 2015 and will be removed in a future release. Use unordered_map and unordered_set instead.

- comparators and operator() Associative containers (the <map> family) now require their comparators to have const-callable function call operators. The following code in a comparator class declaration now fails to compile:

  C++

  ```
  bool operator()(const X& a, const X& b)
  ```

  To resolve this error, change the function declaration to:

  C++

  ```
  bool operator()(const X& a, const X& b) const
  ```

- type traits The old names for type traits from an earlier version of the C++ draft standard have been removed. These were changed in C++11 and have been updated to the C++11 values in Visual Studio 2015. The following table shows the old and new names.

Old name	New name
add_reference	add_lvalue_reference
has_default_constructor	is_default_constructible
has_copy_constructor	is_copy_constructible
has_move_constructor	is_move_constructible
has_nothrow_constructor	is_nothrow_default_constructible
has_nothrow_default_constructor	is_nothrow_default_constructible
has_nothrow_copy	is_nothrow_copy_constructible
has_nothrow_copy_constructor	is_nothrow_copy_constructible
has_nothrow_move_constructor	is_nothrow_move_constructible
has_nothrow_assign	is_nothrow_copy_assignable
has_nothrow_copy_assign	is_nothrow_copy_assignable
has_nothrow_move_assign	is_nothrow_move_assignable
has_trivial_constructor	is_trivially_default_constructible
has_trivial_default_constructor	is_trivially_default_constructible
has_trivial_copy	is_trivially_copy_constructible
has_trivial_move_constructor	is_trivially_move_constructible
has_trivial_assign	is_trivially_copy_assignable
has_trivial_move_assign	is_trivially_move_assignable
has_trivial_destructor	is_trivially_destructible

- launch::any and launch::sync policies The nonstandard launch::any and launch::sync policies were removed. Instead, for launch::any, use launch:async | launch:deferred. For launch::sync, use launch::deferred. See launch Enumeration.

MFC and ATL

- Microsoft Foundation Classes (MFC) is no longer included in a "Typical" install of Visual Studio because of its large size. To install MFC, choose the Custom install option in Visual Studio 2015 setup. If you already have Visual Studio 2015 installed, you can install MFC by re-running Visual Studio setup, choosing the Custom install option, and choosing Microsoft Foundation Classes. You can re-run Visual Studio setup from the Control Panel, Programs and Features, or from the installation media.

  The Visual C++ Redistributable Package still includes this library.

Concurrency Runtime

- Yield macro from Windows.h conflicting with concurrency::Context::Yield The Concurrency Runtime previously used #undef to undefine the Yield macro to avoid conflicts between the Yield macro defined in Windows.h h and the concurrency::Context::Yield function. This #undef has been removed and a new non-conflicting equivalent API call concurrency::Context::YieldExecution has been added. To work around conflicts with Yield, you can either update your code to call the YieldExecution function instead, or surround the Yield function name with parentheses at call sites, as in the following example:

  C++

  ```
  (concurrency::Context::Yield)();
  ```

When upgrading code from previous versions, you might also encounter compiler errors that are due to conformance improvements made in Visual C++ 2015. These improvements do not break binary compatibility from earlier versions of Visual C++, but they can produce compiler errors where none were emitted before. For more information, see Visual C++ What's New 2003 through 2015.

In Visual C++ 2015, ongoing improvements to compiler conformance can sometimes change how the compiler understands your existing source code. When this happens, you might encounter new or different errors during your build, or even behavioral differences in code that previously built and seemed to run correctly.

Fortunately, these differences have little or no impact on most of your source code and when source code or other changes are needed to address these differences, fixes are usually small and straight-forward. We've included many examples of previously-acceptable source code that might need to be changed *(before)* and the fixes to correct them *(after)*.

Although these differences can affect your source code or other build artifacts, they don't affect binary compatibility between updates to Visual C++ versions. A more-severe kind of change, the *breaking change* can affect binary compatibility, but these kinds of binary compatibility breaks only occur between major versions of Visual C++. For example, between Visual C++ 2013 and Visual C++ 2015. For information on the breaking changes that occurred between Visual C++ 2013 and Visual C++ 2015, see Breaking Changes in Visual C++ 2015.

Conformance Improvements in Visual C++ 2015

- /Zc:forScope- option

  The compiler option /Zc:forScope- is deprecated and will be removed in a future release.

  C++

  ```
  Command line warning  D9035: option 'Zc:forScope-' has been deprecated and will
  be removed in a future release
  ```

  The option was usually used in order to allow nonstandard code that uses loop variables after the point where, according to the standard, they should have gone out of scope. It was only necessary when you are compiling with the /Za option, since without /Za, using a for loop variable after the end of the loop is always allowed. If you don't care about standards conformance (for example, if your code isn't meant to portable to other compilers), you could turn off the /Za option (or set the Disable Language Extensions property to No). If you do care about writing portable, standards-compliant code, you should rewrite your code so that it conforms to the standard by moving the declaration of such variables to a point outside the loop.

  C++

  ```
  // C2065 expected
  int main() {
      // Uncomment the following line to resolve.
      // int i;
      for (int i = 0; i < 1; i++);
      i = 20;   // i has already gone out of scope under /Za
  }
  ```

- /Zg compiler option

  The /Zg compiler option (Generate Function Prototypes) is no longer available. This compiler option was previously deprecated.

- You can no longer run unit tests with C++/CLI from the command-line with mstest.exe. Instead, use vstest.console.exe. See VSTest.Console.exe command-line options.

- mutable keyword

  The **mutable** storage class specifier is no longer allowed in places where previously it compiled without error. Now, the compiler gives error C2071 (illegal storage class). According to the standard, the mutable specifier can be applied only to names of class data members, and cannot be applied to names declared const or static, and cannot be applied to reference members.

  For example, consider the following code:

  C++

  ```
  struct S
  {
      mutable int &r;
  };
  ```

  Previous versions of the Visual C++ compiler accepted this, but now the compiler gives the following error:

  ```
  error C2071: 'S::r': illegal storage class
  ```

  To fix the error, simply remove the redundant mutable keyword.

- char_16_t and char32_t

  You can no longer use **char16_t** or **char32_t** as aliases in a typedef, because these types are now treated as built-in. It was common for users and library authors to define char16_t and char32_t as aliases of uint16_t and uint32_t, respectively.

  C++

  ```
  #include <cstdint>

  typedef uint16_t char16_t; //C2628
  typedef uint32_t char32_t; //C2628

  int main(int argc, char* argv[])
  {
      uint16_t x = 1; uint32_t y = 2;
      char16_t a = x;
      char32_t b = y;
      return 0;
  }
  ```

  To update your code, remove the typedef declarations and rename any other identifiers that collide with these names.

- Non-type template parameters

  Certain code that involves non-type template parameters is now correctly checked for type compatibility when you provide explicit template arguments. For example, the following code compiled without error in previous versions of Visual C++.

  C++

  ```
  struct S1
  {
      void f(int);
      void f(int, int);
  };

  struct S2
  {
      template <class C, void (C::*Function)(int) const> void f() {}
  };

  void f()
  {
      S2 s2;
      s2.f<S1, &S1::f>();
  }
  ```

  The current compiler correctly gives an error, because the template parameter type doesn't match the template argument (the parameter is a pointer to a const member, but the function f is non-const):

  ```
  error C2893: Failed to specialize function template 'void S2::f(void)'note: With the following template arguments:note: 'C=S1'note: 'Function=S1::f'
  ```

  To address this error in your code, make sure that the type of the template argument you use matches the declared type of the template parameter.

- __declspec(align)

  The compiler no longer accepts **__declspec(align)** on functions. This was always ignored, but now it produces a compiler error.

  C++

  ```
  error C3323: 'alignas' and '__declspec(align)' are not allowed on function declarations
  ```

  To fix this problem, remove **__declspec(align)** from the function declaration. Since it had no effect, removing it does not change anything.

- Exception handling

  There are a couple of changes to exception handling. First, exception objects have to be either copyable or movable. The following code compiled in Visual C++ in Visual Studio 2013, but does not compile in Visual C++ in Visual Studio 2015:

C++

```
struct S
{
public:
    S();
private:
    S(const S &);
};

int main()
{
    throw S(); // error
}
```

The problem is that the copy constructor is private, so the object cannot be copied as happens in the normal course of handling an exception. The same applies when the copy constructor is declared **explicit**.

C++

```
struct S
{
    S();
    explicit S(const S &);
};

int main()
{
    throw S(); // error
}
```

To update your code, make sure that the copy constructor for your exception object is public and not marked **explicit**.

Catching an exception by value also requires the exception object to be copyable. The following code compiled in Visual C++ in Visual Studio 2013, but does not compile in Visual C++ in Visual Studio 2015:

C++

```
struct B
{
public:
    B();
private:
    B(const B &);
};

struct D : public B {};

int main()
```

```
{
    try
    {
    }
    catch (D d) // error
    {
    }
}
```

You can fix this issue by changing the parameter type for the **catch** to a reference.

C++

```
catch (D& d)
{
}
```

- String literals followed by macros

  The compiler now supports user defined literals. As a consequence, string literals followed by macros without any intervening whitespace are interpreted as user-defined literals, which might produce errors or unexpected results. For example, in previous compilers the following code compiled successfully:

  C++

  ```
  #define _x "there"
  char* func() {
      return "hello"_x;
  }
  int main()
  {
      char * p = func();
      return 0;
  }
  ```

  The compiler interpreted this as a string literal "hello" followed by a macro, which is expanded "there", and then the two string literals were concatenated into one. In Visual C++ in Visual Studio 2015, the compiler interprets this as a user-defined literal, but since there is no matching user-defined literal _x defined, it gives an error.

  C++

  ```
  error C3688: invalid literal suffix '_x'; literal operator or literal operator
  template 'operator ""_x' not found
  note: Did you forget a space between the string literal and the prefix of the
  following string literal?
  ```

  To fix this problem, add a space between the string literal and the macro.

- Adjacent string literals

  Similarly to the previous, due to related changes in string parsing, adjacent string literals (either wide or narrow character string literals) without any whitespace were interpreted as a single concatenated string in previous releases of Visaul C++. In Visual C++ in Visual Studio 2015, you must now add whitespace between the two strings. For example, the following code must be changed:

  C++

  ```
  char * str = "abc""def";
  ```

  Simply add a space in between the two strings.

  C++

  ```
  char * str = "abc" "def";
  ```

- Placement new and delete

  A change has been made to the delete operator in order to bring it into conformance with C++14 standard. Details of the standards change can be found at C++ Sized Deallocation. The changes add a form of the global delete operator that takes a size parameter. The breaking change is that if you were previously using an operator delete with the same signature (to correspond with a placement new operator), you will receive a compiler error (C2956, which occurs at the point where the placement new is used, since that's the position in code where the compiler tries to identify an appropriate matching delete operator).

  The function void operator delete(void *, size_t) was a placement delete operator corresponding to the placement new function "void * operator new(size_t, size_t)" in C++11. With C++14 sized deallocation, this delete function is now a *usual deallocation function* (global delete operator). The standard requires that if the use of a placement new looks up a corresponding delete function and finds a usual deallocation function, the program is ill-formed.

  For example, suppose your code defines both a placement new and a placement delete:

  C++

  ```
  void * operator new(std::size_t, std::size_t);
  void operator delete(void*, std::size_t) noexcept;
  ```

  The problem occurs because of the match in function signatures between a placement delete operator you've defined, and the new global sized delete operator. Consider whether you can use a different type other than size_t for any placement new and delete operators. Note that the type of the size_t typedef is compiler-dependent; it is a typedef for unsigned int in Visual C++. A good solution is to use an enumerated type such as this:

  C++

  ```
  enum class my_type : size_t {};
  ```

  Then, change your definition of placement new and delete to use this type as the second argument instead of size_t. You'll also need to update the calls to placement new to pass the new type (for example, by using static_cast to convert from the integer value) and update the definition of new and delete to cast back to the integer type. You don't need to use an enum for this; a class type with a size_t member would also work.

  An alternative solution is that you might be able to eliminate the placement new altogether. If your code uses placement new to implement a memory pool where the placement argument is the size of the object being allocated or deleted, then sized deallocation feature might be suitable to replace your own custom memory pool code, and you can get rid of the placement functions and just use your own two-argument delete operator instead of the placement functions.

  If you don't want to update your code immediately, you can revert to the old behavior by using the compiler option /Zc:sizedDealloc-. If you use this option, the two-argument delete functions don't exist and won't cause a conflict with your placement delete operator.

- Union data members

  Data members of unions can no longer have reference types. The following code compiled successfully in Visual C++ in Visual Studio 2013, but produces an error in Visual C++ in Visual Studio 2015.

  C++

  ```
  union U1
  {
      const int i;
  };
  union U2
  {
      int & i;
  };
  union U3
  {
      struct { int & i; };
  };
  ```

  The preceding code produces the following errors:

  ```
  test.cpp(67): error C2625: 'U2::i': illegal union member; type 'int &' is
  reference type
  test.cpp(70): error C2625: 'U3::i': illegal union member; type 'int &' is
  reference type
  ```

  To address this issue, change reference types either to a pointer or a value. Changing the type to a pointer requires changes in the code that uses the union field. Changing the code to a value would change the data stored in the union, which affects other fields since fields in union types share the same memory. Depending on the size of the value, it might also change the size of the union.

- Anonymous unions are now more conformant to the standard. Previous versions of the compiler generated an explicit constructor and destructor for anonymous unions. These are deleted in Visual C++ in Visual Studio 2015.

  C++

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

  union
  {
      struct
      {
          S s;
      };
  } u; // C2280
  ```

  The preceding code generates the following error in Visual C++ in Visual Studio 2015:

  C++

  ```
  error C2280: '<unnamed-type-u>::<unnamed-type-u>(void)': attempting to reference
  a deleted function
  note: compiler has generated '<unnamed-type-u>::<unnamed-type-u>' here
  ```

  To resolve this issue, provide your own definitions of the constructor and/or destructor.

  C++

  ```
  struct S
  {
      // Provide a default constructor by adding an empty function body.
      S() {}
  };

  union
  {
      struct
      {
          S s;
      };
  } u;
  ```

- Unions with anonymous structs

  In order to conform with the standard, the runtime behavior has changed for members of anonymous structures in unions. The constructor for anonymous structure members in a union is no longer implicitly

called when such a union is created. Also, the destructor for anonymous structure members in a union is no longer implicitly called when the union goes out of scope. Consider the following code, in which a union U contains an anonymous structure that contains a member which is a named structure S that has a destructor.

C++

```
#include <stdio.h>
struct S
{
    S() { printf("Creating S\n"); }
    ~S() { printf("Destroying S\n"); }
};
union U
{
    struct {
        S s;
    };
    U() {}
    ~U() {}
};

void f()
{
    U u;
    // Destructor implicitly called here.
}

int main()
{
    f();

    char s[1024];
    printf("Press any key.\n");
    gets_s(s);
    return 0;
}
```

In Visual C++ in Visual Studio 2013, the constructor for S is called when the union is created, and the destructor for S is called when the stack for function f is cleaned up. But in Visual C++ in Visual Studio 2015, the constructor and destructor are not called. The compiler gives a warning about this behavior change.

```
warning C4587: 'U::s': behavior change: constructor is no longer implicitly
calledwarning C4588: 'U::s': behavior change: destructor is no longer
implicitly called
```

To restore the original behavior, give the anonymous structure a name. The runtime behavior of non-anonymous structures is the same, regardless of the compiler version.

C++

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

```
struct S
{
    S() { printf("Creating S.\n"); }
    ~S() { printf("Destroying S\n"); }
};
union U
{
    struct
    {
        S s;
    } namedStruct;
    U() {}
    ~U() {}
};

void f()
{
    U u;
}

int main()
{
    f();

    char s[1024];
    printf("Press any key.\n");
    gets_s(s);
    return 0;
}
```

Alternatively, try moving the constructor and destructor code into new functions, and add calls to these functions from the constructor and destructor for the union.

C++

```
#include <stdio.h>

struct S
{
    void Create() { printf("Creating S.\n"); }
    void Destroy() { printf("Destroying S\n"); }
};
union U
{
    struct
    {
        S s;
    };
    U() { s.Create(); }
```

```
    ~U() { s.Destroy(); }
};


void f()
{
    U u;
}

int main()
{
    f();

    char s[1024];
    printf("Press any key.\n");
    gets_s(s);
    return 0;
}
```

- Template resolution

  Changes have been made to name resolution for templates. In C++, when considering candidates for the resolution of a name, it can be the case that one or more names under consideration as potential matches produces an invalid template instantiation. These invalid instantiations do not normally cause compiler errors, a principle which is known as SFINAE (Substitution Failure Is Not An Error).

  Now, if SFINAE requires the compiler to instantiate the specialization of a class template, then any errors that occur during this process are compiler errors. In previous versions, the compiler would ignore such errors. For example, consider the following code:

  C++

```
#include <type_traits>

template< typename T>
struct S
{
    S() = default;
    S(const S&);
    S(S& &);

    template< typename U, typename = typename std::enable_if< std::is_base_of<
T, U> ::value> ::type>
    S(S< U> & &);
};

struct D;

void f1()
{
```

```
    S< D> s1;
    S< D> s2(s1);
}

struct B
{
};

struct D : public B
{
};

void f2()
{
    S< D> s1;
    S< D> s2(s1);
}
```

If you compile with the current compiler, you get the following error:

```
type_traits(1110): error C2139: 'D': an undefined class is not allowed as an
argument to compiler intrinsic type trait '__is_base_of'
..\t331.cpp(14): note: see declaration of 'D'
..\t331.cpp(10): note: see reference to class template instantiation
'std::is_base_of<T,U>' being compiled
with
[
    T=D,
    U=D
]
```

This is because at the point of the first invocation of the is_base_of the class 'D' has not yet been defined.

In this case, the fix is not to use such type traits until the class has been defined. If you move the definitions of B and D to the beginning of the code file, the error is resolved. If the definitions are in header files, check the order of the include statements for the header files to make sure that any class definitions are compiled before the problematic templates are used.

- Copy constructors

  In both Visual Studio 2013 and Visual Studio 2015, the compiler generates a copy constructor for a class if that class has a user-defined move constructor but no user-defined copy constructor. In Dev14, this implicitly generated copy constructor is also marked "= delete".

Conformance Improvements in Update 1

- Private virtual base classes and indirect inheritance

  Previous versions of the compiler allowed a derived class to call member functions of its *indirectly-derived* **private virtual** base classes. This old behavior was incorrect and does not conform to the C++

standard. The compiler no longer accepts code written in this way and issues compiler error C2280 as a result.

```
error C2280: 'void *S3::__delDtor(unsigned int)': attempting to reference a
deleted function
```

Example (before)

C++

```
class base
{
protected:
    base();
    ~base();
};

class middle : private virtual base {};
class top : public virtual middle {};

void destroy(top *p)
{
    delete p;  //
}
```

Example (after)

C++

```
class base;  // as above

class middle : protected virtual base {};
class top : public virtual middle {};

void destroy(top *p)
{
    delete p;
}
```

-or -

C++

```
class base;  // as above

class middle : private virtual base {};
class top : public virtual middle, private virtual bottom {};

void destroy(top *p)
{
    delete p;
```

```
}
```

- Overloaded operator new and operator delete

  Previous versions of the compiler allowed non-member **operator new** and non-member **operator delete** to be declared static, and to be declared in namespaces other than the global namespace. This old behavior created a risk that the program would not call the **new** or **delete** operator implementation that the programmer intended, resulting in silent bad runtime behavior. The compiler no longer accepts code written in this way and issues compiler error C2323 instead.

  ```
  error C2323: 'operator new': non-member operator new or delete functions may
  not be declared static or in a namespace other than the global namespace.
  ```

  Example (before)

  C++

  ```
  static inline void * __cdecl operator new(size_t cb, const std::nothrow_t&)  //
  error C2323
  ```

  Example (after)

  C++

  ```
  void * __cdecl operator new(size_t cb, const std::nothrow_t&)  // removed
  'static inline'
  ```

  Additionally, although the compiler doesn't give a specific diagnostic, inline operator new is considered ill-formed.

- Calling 'operator *type*()' (user-defined conversion) on non-class types

  Previous versions of the compiler allowed 'operator *type*()' to be called on non-class types while silently ignoring it. This old behavior created a risk of silent bad code generation, resulting in unpredictable runtime behavior. The compiler no longer accepts code written in this way and issues compiler error C2228 instead.

  ```
  error C2228: left of '.operator type' must have class/struct/union
  ```

  Example (before)

  C++

  ```
  typedef int index_t;
  void bounds_check(index_t index);
  void login(int column)
  {
      bounds_check(column.operator index_t());  // error C2228
  }
  ```

  Example (after)

  C++

  ```
  typedef int index_t;
  void bounds_check(index_t index);
  void login(int column)
  {
      bounds_check(column);  // removed cast to 'index_t', 'index_t' is an alias
  of 'int'
  }
  ```

- Redundant typename in elaborated type specifiers

  Previous versions of the compiler allowed **typename** in an elaborated type specifiers; code written in this way is semantically incorrect. The compiler no longer accepts code written in this way and issues compiler error C3406 instead.

  ```
  error C3406: 'typename' cannot be used in an elaborated type specifier
  ```

  Example (before)

  C++

  ```
  template <typename class T>
  class container;
  ```

  Example (after)

  C++

  ```
  template <class T>  // alternatively, could be 'template <typename T>';
  'typename' is not elaborating a type specifier in this case
  class container;
  ```

- Type deduction of arrays from an initializer list

  Previous versions of the compiler did not support type deduction of arrays from an initializer list. The compiler now supports this form of type deduction and, as a result, calls to function templates using initializer lists might now be ambiguous or a different overload might be chosen than in previous versions of the compiler. To resolve these issues, the program must now explicitly specify the overload that the programmer intended.

  When this new behavior causes overload resolution to consider an additional candidate that is equally as good as the historic candidate, the call becomes ambiguous and the compiler issues compiler error C2668 as a result.

  ```
  error C2668: 'function' : ambiguous call to overloaded function.
  ```

  Example 1: Ambiguous call to overloaded function (before)

  C++

  ```
  // In previous versions of the compiler, code written in this way would
  unambiguously call f(int, Args...)
  template < typename... Args>
  void f(int, Args...);  //

  template < int N, typename... Args>
  void f(const int(&)[N], Args...);

  int main()
  {
      // The compiler now considers this call ambiguous, and issues a compiler
  error
  ```

```
     f({ 3 });   error C2668 : 'f' ambiguous call to overloaded function
}
```

Example 1: ambiguous call to overloaded function (after)

C++

```
template < typename... Args>
void f(int, Args...);  //

template < int N, typename... Args>
void f(const int(&)[N], Args...);

int main()
{
    // To call f(int, Args...) when there is just one expression in the
initializer list, remove the braces from it.
    f(3);
}
```

When this new behavior causes overload resolution to consider an additional candidate that is a better match than the historic candidate, the call resolves unambiguously to the new candidate, causing a change in program behavior that is probably different than the programmer intended.

Example 2: change in overload resolution (before)

C++

```
// In previous versions of the compiler, code written in this way would
unambiguously call f(S, Args...)
struct S
{
    int i;
    int j;
};

template < typename... Args>
void f(S, Args...);

template < int N, typename... Args>
void f(const int *&)[N], Args...);

int main()
{
    // The compiler now resolves this call to f(const int (&)[N], Args...)
instead
     f({ 1, 2 });
}
```

Example 2: change in overload resolution (after)

C++

```
struct S;  // as before

template < typename... Args>
void f(S, Args...);

template < int N, typename... Args>
void f(const int *&)[N], Args...);

int main()
{
    // To call f(S, Args...), perform an explicit cast to S on the initializer
list.
    f(S{ 1, 2 });
}
```

- Restoration of switch statement warnings

  A Previous version of the compiler removed previously-existing warnings related to **switch** statements; these warnings have now been restored. The compiler now issues the restored warnings, and warnings related to specific cases (including the default case) are now issued on the line containing the offending case, rather than on the last line of the switch statement. As a result of now issuing those warnings on different lines than in the past, warnings previously suppressed by using #pragma warning(disable:####) may no longer be suppressed as intended. To suppress these warnings as intended, it might be necessary to move the #pragma warning(disable:####) directive to a line above the first potentially-offending case. The following are the restored warnings.

  ```
  warning C4060: switch statement contains no 'case' or 'default' labels

  warning C4061: enumerator 'bit1' in switch of enum 'flags' is not explicitly
  handled by a case label

  warning C4062: enumerator 'bit1' in switch of enum 'flags' is not handled

  warning C4063: case 'bit32' is not a valid value for switch of enum 'flags'

  warning C4064: switch of incomplete enum 'flags'

  warning C4065: switch statement contains 'default' but no 'case' labels

  warning C4808: case 'value' is not a valid value for switch condition of type
  'bool'

  Warning C4809: switch statement has redundant 'default' label; all possible
  'case' labels are given
  ```

  Example of C4063 (before)

  C++

```
class settings
{
public:
    enum flags
    {
        bit0 = 0x1,
        bit1 = 0x2,
        ...
    };
    ...
};

int main()
{
    auto val = settings::bit1;

    switch (val)
    {
    case settings::bit0:
        break;

    case settings::bit1:
        break;

         case settings::bit0 | settings::bit1:  // warning C4063
            break;
    }
};
```

Example of C4063 (after)

C++

```
class settings { ... };  // as above
int main()
{
    // since C++11, use std::underlying_type to determine the underlying type
of an enum
    typedef std::underlying_type< settings::flags> ::type flags_t;

        auto val = settings::bit1;

    switch (static_cast< flags_t> (val))
    {
    case settings::bit0:
        break;

    case settings::bit1:
        break;
```

```
    case settings::bit0 | settings::bit1:  // ok
        break;
    }
};
```

Examples of the other restored warnings are provided in their documentation.

- #include: use of parent-directory specifier '..' in pathname (only affects /Wall /WX)

  Previous versions of the compiler did not detect the use of the parent-directory specifier '..' in the pathname of **#include** directives. Code written in this way is usually intended to include headers that exist outside of the project by incorrectly using project-relative paths. This old behavior created a risk that the program could be compiled by including a different source file than the programmer intended, or that these relative paths would not be portable to other build environments. The compiler now detects and notifies the programmer of code written in this way and issues an optional compiler warning C4464, if enabled.

  ```
  warning C4464: relative include path contains '..'
  ```

  Example (before)

  C++

  ```
  #include "..\headers\C4426.h"  // emits warning C4464
  ```

  Example (after)

  C++

  ```
  #include "C4426.h"  // add absolute path to 'headers\' to your project's
  include directories
  ```

  Additionally, although the compiler does not give a specific diagnostic, we also recommend that the parent-directory specifier ".." should note be used to specify your project's include directories.

- #pragma optimize() extends past end of header file (only affects /Wall /WX)

  Previous versions of the compiler did not detect changes to optimization flag settings that escape a header file included within a translation unit. The compiler now detects and notifies the programmer of code written in this way and issues an optional compiler warning C4426 at the location of the offending **#include**, if enabled. This warning is only issued if the changes conflict with the optimization flags set by command-line arguments to the compiler.

  ```
  warning C4426: optimization flags changed after including header, may be due to
  #pragma optimize()
  ```

  Example (before)

  C++

  ```
  // C4426.h
  #pragma optimize("g", off)
  ...
  // C4426.h ends

  // C4426.cpp
  #include "C4426.h"  // warning C4426
  ```

  Example (after)

  C++

  ```
  // C4426.h
  #pragma optimize("g", off)
              ...
  #pragma optimize("", on)  // restores optimization flags set via command-line
  arguments
  // C4426.h ends

  // C4426.cpp
  #include "C4426.h"
  ```

- Mismatched #pragma warning(push) and #pragma warning(pop) (only affects /Wall /WX)

  Previous versions of the compiler did not detect #pragma warning(push) state changes being paired with #pragma warning(pop) state changes in a different source file, which is rarely intended. This old behavior created a risk that the program would be compiled with a different set of warnings enabled than the programmer intended, possibly resulting in silent bad runtime behavior. The compiler now detects and notifies the programmer of code written in this way and issues an optional compiler warning C5031 at the location of the matching #pragma warning(pop), if enabled. This warning includes a note referencing the location of the corresponding #pragma warning(push).

  ```
  warning C5031: #pragma warning(pop): likely mismatch, popping warning state
  pushed in different file
  ```

  Example (before)

C++

```
// C5031_part1.h
#pragma warning(push)
#pragma warning(disable:####)
...
// C5031_part1.h ends without #pragma warning(pop)

// C5031_part2.h
...
#pragma warning(pop)  // pops a warning state not pushed in this source file
...
// C5031_part1.h ends

// C5031.cpp
#include "C5031_part1.h" // leaves #pragma warning(push) 'dangling'
...
#include "C5031_part2.h" // matches 'dangling' #pragma warning(push), resulting
in warning C5031
...
```

Example (after)

C++

```
// C5031_part1.h
#pragma warning(push)
#pragma warning(disable:####)
...
#pragma warning(pop)  // pops the warning state pushed in this source file
// C5031_part1.h ends without #pragma warning(pop)

// C5031_part2.h
#pragma warning(push)  // pushes the warning state pushed in this source file
#pragma warning(disable:####)
...
#pragma warning(pop)
// C5031_part1.h ends

// C5031.cpp
#include "C5031_part1.h" // #pragma warning state changes are self-contained
and independent of other source files or their #include order.
...
#include "C5031_part2.h"
...
```

Though uncommon, code written in this way is sometimes intentional. Code written in this way is sensitive to changes in **#include** order; when possible, we recommend that source code files manage warning state in a self-contained way.

- Unmatched #pragma warning(push) (only affects /Wall /WX)

  Previous versions of the compiler did not detect unmatched #pragma warning(push) state changes at the end of a translation unit. The compiler now detects and notifies the programmer of code written in this way and issues an optional compiler warning C5032 at the location of the unmatched #pragma warning(push), if enabled. This warning is only issued if there are no compilation errors in the translation unit.

  ```
  warning C5032: detected #pragma warning(push) with no corresponding #pragma
  warning(pop)
  ```

  Example (before)

  C++

  ```
  // C5032.h
  #pragma warning(push)
  #pragma warning(disable:####)
  ...
  // C5032.h ends without #pragma warning(pop)

  // C5032.cpp
  #include "C5032.h"
  ...
  // C5032.cpp ends -- the translation unit is completed without #pragma
  warning(pop), resulting in warning C5032 on line 1 of C5032.h
  ```

  Example (after)

  C++

  ```
  // C5032.h
  #pragma warning(push)
  #pragma warning(disable:####)
  ...
  #pragma warning(pop) // matches #pragma warning (push) on line 1
  // C5032.h ends

  // C5032.cpp
  #include "C5032.h"
  ...
  // C5032.cpp ends -- the translation unit is completed without unmatched
  #pragma warning(push)
  ```

- Additional warnings might be issued as a result of improved #pragma warning state tracking

  Previous versions of the compiler tracked #pragma warning state changes insufficiently well to issue all intended warnings. This behavior created a risk that certain warnings would be effectively suppressed in circumstances different than the programmer intended. The compiler now tracks #pragma warning state more robustly -- especially related to #pragma warning state changes inside of templates -- and optionally issues new warnings C5031 and C5032 which are intended to help the programmer locate unintended uses of #pragma warning(push) and #pragma warning(pop).

  As a result of improved #pragma warning state change tracking, warnings formerly incorrectly suppressed or warnings related to issues formerly misdiagnosed might now be issued.

- Improved identification of unreachable code

  C++ Standard Library changes and improved ability to inline function calls over previous versions of the compiler might allow the compiler to prove that certain code is now unreachable. This new behavior can result in new and more-frequently issued instances of warning C4720.

  ```
  warning C4720: unreachable code
  ```

  In many cases, this warning might only be issued when compiling with optimizations enabled, since optimizations may inline more function calls, eliminate redundant code, or otherwise make it possible to determine that certain code is unreachable. We have observed that new instances of warning C4720 have frequently occurred in try/catch blocks, especially in relation to use of std::find.

  Example (before)

  C++

  ```
  try
  {
      auto iter = std::find(v.begin(), v.end(), 5);
  }
  catch (…)
  {
      do_something();    // ok
  }
  ```

  Example (after)

  C++

  ```
  try
  {
      auto iter = std::find(v.begin(), v.end(), 5);
  }
  catch (…)
  {
      do_something();    // warning C4702: unreachable code
  }
  ```

Conformance Improvements in Update 2

- Additional warnings and errors might be issued as a result of partial support for expression SFINAE

  Previous versions of the compiler did not parse certain kinds of expressions inside **decltype** specifiers due to lack of support for expression SFINAE. This old behavior was incorrect and does not conform to the C++ standard. The compiler now parses these expressions and has partial support for expression SFINAE due to ongoing conformance improvements. As a result, the compiler now issues warnings and errors found in expressions that previous versions of the compiler did not parse.

  When this new behavior parses a **decltype** expression that includes a type that has not yet been declared, the compiler issues compiler error C2039 as a result.

  ```
  error C2039: 'type': is not a member of ``global namespace''
  ```

  Example 1: use of an undeclared type (before)

  C++

  ```
  struct s1
  {
      template < typename T>
      auto f() - > decltype(s2< T> ::type::f());  // error C2039

      template< typename>
      struct s2 {};
  }
  ```

  Example 1 (after)

  C++

  ```
  struct s1
  {
      template < typename>  // forward declare s2
      struct s2;

          template < typename T>
      auto f() - > decltype(s2< T> ::type::f());

      template< typename>
      struct s2 {};
  }
  ```

  When this new behavior parses a **decltype** expression that is missing a necessary use of the **typename** keyword to specify that a dependent name is a type, the compiler issues compiler warning C4346 together with compiler error C2923.

  ```
  warning C4346: 'S2<T>::Type': dependent name is not a type

  error C2923: 's1': 'S2<T>::Type' is not a valid template type argument for
  ```

```
parameter 'T'
```

Example 2: dependent name is not a type (before)

C++

```
template < typename T>
struct s1
{
    typedef T type;
};

template < typename T>
struct s2
{
    typedef T type;
};

template < typename T>
T declval();

struct s
{
    template < typename T>
    auto f(T t) - > decltype(t(declval< S1< S2< T> ::type> ::type> ())); //
warning C4346, error C2923
};
```

Example 2 (after)

C++

```
template < typename T> struct s1 { ... };  // as above
template < typename T> struct s2 { ... };  // as above

template < typename T>
T declval();

struct s
{
    template < typename T>
    auto f(T t) - > decltype(t(declval< S1< typename S2< T> ::type> ::type>
()));
};
```

- **volatile**member variables prevent implicitly defined constructors and assignment operators

  Previous versions of the compiler allowed a class that has **volatile** member variables to have default copy/move constructors and default copy/move assignment operators automatically generated. This old behavior was incorrect and does not conform to the C++ standard. The compiler now considers a class that

has volatile member variables to have non-trivial construction and assignment operators which prevents default implementations of these operators from being automatically generated. When such a class is a member of a union (or an anonymous union inside of a class), the copy/move constructors and copy/move assignment operators of the union (or the class containing the unonymous union) will be implicitly defined as deleted. Attempting to construct or copy the union (or class containing the anonymous union) without explicitly defining them is an error and the compiler issues compiler error C2280 as a result.

```
error C2280: 'B::B(const B &)': attempting to reference a deleted function
```

Example (before)

C++

```
struct A
{
    volatile int i;
    volatile int j;
};

extern A* pa;

struct B
{
    union
    {
        A a;
        int i;
    };
};

B b1{ *pa };
B b2(b1);  // error C2280
```

Example (after)

C++

```
struct A
{
    int i;
        int j;
};

extern volatile A* pa;

A getA()  // returns an A instance copied from contents of pa
{
    A a;
    a.i = pa - > i;
    a.j = pa - > j;
    return a;
```

```
}

struct B;  // as above

B b1{ GetA() };
B b2(b1);  // error C2280
```

- Static member functions do not support cv-qualifiers.

  Previous versions of Visual C++ 2015 allowed static member functions to have cv-qualifiers. This behavior is due to a regression in Visual C++ 2015 and Visual C++ 2015 Update 1; Visual C++ 2013 and previous versions of Visual C++ reject code written in this way. The behavior of Visual C++ 2015 and Visual C++ 2015 Update 1 is incorrect and does not conform to the C++ standard. Visual Studio 2015 Update 2 rejects code written in this way and issues compiler error C2511 instead.

  ```
  error C2511: 'void A::func(void) const': overloaded member function not found
  in 'A'
  ```

  Example (before)

  C++

  ```
  struct A
  {
      static void func();
  };

  void A::func() const {}  // C2511
  ```

  Example(after)

  C++

  ```
  struct A
  {
      static void func();
  };

  void A::func() {}  // removed const
  ```

- Forward declaration of enum is not allowed in WinRT code (affects /ZW only)

  Code compiled for the Windows Runtime (WinRT) doesn't allow **enum** types to be forward declared, similarly to when managed C++ code is compiled for the .Net Framework using the /clr compiler switch. This behavior is ensures that the size of an enumeration is always known and can be correctly projected to the WinRT type system. The compiler rejects code written in this way and issues compiler error C2599 together with compiler error C3197.

  ```
  error C2599: 'CustomEnum': the forward declaration of a WinRT enum is not
  allowed
  ```

```
error C3197: 'public': can only be used in definitions
```

Example (before)

C++

```
namespace A {
    public enum class CustomEnum : int32;  // forward declaration; error C2599,
error C3197
}

namespace A {
    public enum class CustomEnum : int32
    {
        Value1
    };
}

public ref class Component sealed
{
public:
    CustomEnum f()
    {
        return CustomEnum::Value1;
    }
};
```

Example (after)

C++

```
// forward declaration of CustomEnum removed

namespace A {
    public enum class CustomEnum : int32
    {
        Value1
    };
}

public ref class Component sealed
{
public:
    CustomEnum f()
    {
        return CustomEnum::Value1;
    }
};
```

- Overloaded non-member operator new and operator delete may not be declared inline (Level 1 (/W1) on-by-default)

  Previous versions of the compiler do not issue a warning when non-member operator new and operator delete functions are declared inline. Code written in this way is ill-formed (no diagnostic required) and can cause memory issues resulting from mismatched new and delete operators (especially when used together with sized deallocation) that can be difficult to diagnose. The compiler now issues compiler warning C4595 to help identify code written in this way.

  ```
  warning C4595: 'operator new': non-member operator new or delete functions may
  not be declared inline
  ```

  Example (before)

  C++

  ```
  inline void* operator new(size_t sz)  // warning C4595
  {
      ...
  }
  ```

  Example (after)

  C++

  ```
  void* operator new(size_t sz)  // removed inline
  {
      ...
  }
  ```

  Fixing code that's written in this way might require that the operator definitions be moved out of a header file and into a corresponding source file.

Conformance Improvements in Update 3

- std::is_convertable now detects self-assignment (standard library)

  Previous versions of the std::is_convertable type-trait did not correctly detect self-assignment of a class type when its copy constructor is deleted or private. Now, std::is_convertable<>::value is correctly set to **false** when applied to a class type with a deleted or private copy constructor.

  There is no compiler diagnostic associated with this change.

  Example

  C++

```
#include <type_traits>

            class X1
{
            public:
            X1(const X1&) = delete;
            };

            class X2
{
            private:
            X2(const X2&);
            };

static_assert(std::is_convertible<X1&, X1>::value, "BOOM");
static_assert(std::is_convertible<X2&, X2>::value, "BOOM");
```

  In previous versions of Visual C++, the static assertions at the bottom of this example pass because std::is_convertable<>::value was incorrectly set to **true**. Now, std::is_convertable<>::value is correctly set to **false**, causing the static assertions to fail.

- Defaulted or deleted trivial copy and move constructors respect access specifiers

  Previous versions of the compiler did not check the access specifier of defaulted or deleted trivial copy and move constructors before allowing them to be called. This old behavior was incorrect and does not conform to the C++ standard. In some cases, this old behavior created a risk of silent bad code generation, resulting in unpredictable runtime behavior. The compiler now checks the access specifier of defaulted or deleted trivial copy and move constructors to determine whether it can be called, and if not, issues compiler warning C2248 as a result.

  ```
  error C2248: 'S::S' cannot access private member declared in class 'S'
  ```

  Example (before)

  C++

  ```
  class S {
  public:
      S() = default;
  private:
      S(const S&) = default;
  };

  void f(S);  // pass S by value

  int main()
  {
      S s;
      f(s);  // error C2248, can't invoke private copy constructor
  }
  ```

  Example (after)

  C++

  ```
  class S {
  public:
      S() = default;
  private:
      S(const S&) = default;
  };

  void f(const S&);  // pass S by reference

  int main()
  {
      S s;
      f(s);
  }
  ```

- Deprecation of attributed ATL code support (Level 1 (/W1) on-by-default)

Previous versions of the compiler supported attributed ATL code. As the next phase of removing support for attributed ATL code that began in Visual C++ 2008, attributed ATL code has been deprecated. The compiler now issues compiler warning C4467 to help identify this kind of deprecated code.

```
warning C4467: Usage of ATL attributes is deprecated
```

If you want to continue using attributed ATL code until support is removed from the compiler, you can disable this warning by passing the /Wv:18 or /wd:4467 command line arguments to the compiler, or by adding #pragma warning(disable:4467) in your source code.

Example 1 (before)

C++

```
[uuid("594382D9-44B0-461A-8DE3-E06A3E73C5EB")]
class A {};
```

Example 1 (after)

C++

```
__declspec(uuid("594382D9-44B0-461A-8DE3-E06A3E73C5EB")) A {};
```

Sometimes you might need or want to create an IDL file to avoid the use deprecated ATL attributes, as in the example code below

Example 2 (before)

C++

```
[emitidl];
[module(name = "Foo")];

[object, local, uuid("9e66a290-4365-11d2-a997-00c04fa37ddb")]
__interface ICustom {
    HRESULT Custom([in] long l, [out, retval] long *pLong);
    [local] HRESULT CustomLocal([in] long l, [out, retval] long *pLong);
};

[coclass, appobject, uuid("9e66a294-4365-11d2-a997-00c04fa37ddb")]
class CFoo : public ICustom
{
    // ...
};
```

First, create the *.idl file; the vc140.idl generated file can be used to obtain an *.idl file containing the interfaces and annotations.

Next, add a MIDL step to your build to make sure that the C++ interface definitions are generated.

Example 2 IDL (after)

C++

```
import "docobj.idl";

[
    object,
    local,
    uuid(9e66a290 - 4365 - 11d2 - a997 - 00c04fa37ddb)
]

interface ICustom : IUnknown {
    HRESULT  Custom([in] long l, [out, retval] long *pLong);
    [local] HRESULT  CustomLocal([in] long l, [out, retval] long *pLong);
};

[version(1.0), uuid(29079a2c - 5f3f - 3325 - 99a1 - 3ec9c40988bb)]
library Foo
{
    importlib("stdole2.tlb");
importlib("olepro32.dll");
[
    version(1.0),
    appobject,
    uuid(9e66a294 - 4365 - 11d2 - a997 - 00c04fa37ddb)
]

coclass CFoo {
    interface ICustom;
};
}
```

Then, use ATL directly in the implementation file, as in the example code below.

Example 2 Implementation (after)

C++

```
#include <idl.header.h>
#include <atlbase.h>

class ATL_NO_VTABLE CFooImpl :
    public ICustom,
    public ATL::CComObjectRootEx< CComMultiThreadModel>
{
public:
    BEGIN_COM_MAP(CFooImpl)
        COM_INTERFACE_ENTRY(ICustom)
    END_COM_MAP()
};
```

- Precompiled header (PCH) files and mismatched #include directives (only affects /Wall /WX)

  Previous versions of the compiler accepted mismatched **#include** directives in source files between -Yc and -Yu compilations when using precompiled header (PCH) files. Code written in this way is no longer accepted by the compiler. The compiler now issues compiler warning CC4598 to help identify mismatched **#include** directives when using PCH files.

  ```
  warning C4598: 'b.h': included header file specified for Ycc.h at position 2
  does not match Yuc.h at that position
  ```

  Example (before):

  X.cpp (-Ycc.h)

  C++

  ```
  #include "a.h"
  #include "b.h"
  #include "c.h"
  ```

  Z.cpp (-Yuc.h)

  C++

  ```
  #include "b.h"
  #include "a.h"  // mismatched order relative to X.cpp
  #include "c.h"
  ```

  Example (after)

  X.cpp (-Ycc.h)

  C++

  ```
  #include "a.h"
  #include "b.h"
  #include "c.h"
  ```

  Z.cpp (-Yuc.h)

  C++

  ```
  #include "a.h"
  #include "b.h" // matched order relative to X.cpp
  #include "c.h"
  ```

- Precompiled header (PCH) files and mismatched include directories (only affects /Wall /WX)

  Previous versions of the compiler accepted mismatched include directory (-I) command line arguments to the compiler between -Yc and -Yu compilations when using precompiled header (PCH) files. Code written in this way is no longer accepted by the compiler. The compiler now issues compiler warning CC4599 to help identify mismatched include directory (-I) command line arguments when using PCH files.

  ```
  warning C4599: '-I..' : specified for Ycc.h at position 1 does not match Yuc.h
  at that position
  ```

  Example (before)

  msdos

  ```
  cl /c /Wall /Ycc.h -I.. X.cpp
  cl /c /Wall /Yuc.h Z.cpp
  ```

  Example (after)

  msdos

  ```
  cl /c /Wall /Ycc.h -I.. X.cpp
  cl /c /Wall /Yuc.h -I.. Z.cpp
  ```