C++17

In Detail

From language fixes, constexpr_if, fold expressions, std::any, filesystem... to parallel algorithms and more.

See what are the exciting features of C++17!

BARTLOMIEJ FILIPEK

BFILIPEK.COM

The new C++ Standard - C++17 - was recently approved!

> C++17 is formally approved
> 2017-09-06 by Herb Sutter
> herbsutter.com/2017/09/06/c17-is-formally-approved/

It’s a good occasion to learn and understand what the new features are.

Table of Contents:

Intro

The book is compiled from my recent blog post series about C++17 features:

1. Fixes and deprecation
2. Language clarification
3. Templates
4. Attributes
5. Simplification
6. Library changes - Filesystem
7. Library changes - Parallel STL
8. Library changes - Utilities
9. Wrap up/Summary

Documents & Links

First of all, if you want to dig into the standard on your own, you can read the latest draft here:

N4687, 2017-07-30, **Working Draft, Standard for Programming Language C++** - from isocpp.org. It's the last draft before it got accepted, so it's almost as the final spec. Also have a look here for more information about the papers and the status: isocpp : Standard C++.

Also, you can grab my list of concise descriptions of all of the C++17 - It's a one-page reference card, pdf language features: **grab it here**.

Links:

- Compiler support: C++ compiler support
- The official paper with changes: P0636r0: Changes between C++14 and C++17 DIS
- There's also a talk from Bryce Lelbach: C++Now 2017: C++17 Features
- My master C++17 features post: C++17 Features
- Jason Turner: C++ Weekly channel, where he covered most (or even all!) of C++17 features.

And the books:

- **C++17 STL Cookbook** by Jacek Galowicz
  - my review of this book: link here
- **Modern C++ Programming Cookbook** by Marius Bancila
  - my review of this book: link here
- **C++ Templates: The Complete Guide** (2nd Edition) by David Vandevoorde, Nicolai M. Josuttis, Douglas Gregor

Fixes and Deprecation

The draft for the language contains now over 1586 pages! Due to compatibility requirements, the new features are added, but not much is removed. Fortunately, there are some things that could go away.

Removed things

Removing trigraphs

Trigraphs are special character sequences that could be used when a system doesn't support 7-bit ASCII - like in ISO 646 character set . For example `??=` generated `#`, `??-` produces `~`. BTW: All of C++'s basic source character set fits in 7-bit ASCII. The sequences are rarely used and by removing them the translation phase of the code might be simpler.

If you want to know more: c++03 - Purpose of Trigraph sequences in C++? - Stack Overflow, or Digraphs and trigraphs Wikipedia.

More details in: N4086. If you really need trigraphs with Visual Studio, take a look at /Zc:trigraphs switch. Also, other compilers might leave the support in some way or the other. Other compiler status: done in GCC: 5.1 and Clang: 3.5.

Removing register keyword

The `register` keyword was deprecated in the 2011 C++ standard as it has no meaning. Now it's being removed. This keyword is reserved and might be repurposed in the future revisions (for example `auto` keyword was reused and now is something powerful).

More details: P0001R1, MSVC 2017: 15.3, GCC: 7.0 and Clang: 3.8.

Remove Deprecated operator++(bool)

This operator is deprecated for a very long time! In C++98 is was decided that it's better not to use it. But only in C++17, the committee agreed to remove it from the language.

More details: P0002R1, MSVC 2017: 15.3, GCC: 7.0 and Clang: 3.8.

Removing Deprecated Exception Specifications from C++17

In C++17 exception specification will be part of the type system (see P0012R1). Still the standard contains old and deprecated exception specification that appeared to be not practical and not used.

For example:

```
void fooThrowsInt(int a) throw(int) {
   printf_s("can throw ints\n");
```

```
    if (a == 0)
       throw 1;
}
```

The above code is deprecated since C++11. The only practical exception declaration is `throw()` that mean - this code won't throw anything. But since C++11 it's advised to use `noexcept`.

For example in clang 4.0 you'll get the following error:

```
error: ISO C++1z does not allow dynamic exception specifications [-Wdynamic-exception-
spec]
note: use 'noexcept(false)' instead
```

More details: P0003R5, MSVC 2017: **not yet**. Done in GCC: 7.0 and Clang: 4.0.

Removing auto_ptr

This is one of my favorite update to the language!

In C++11 we got smart pointers: `unique_ptr`, `shared_ptr` and `weak_ptr`. Thanks to the move semantics the language could finally support proper unique resource transfers. `auto_ptr` was old and buggy thing in the language see the full reasons here - why is auto_ptr deprecated. It should be almost automatically converted to `unique_ptr`. For some time `auto_ptr` was deprecated (since C++11). Many compilers would report this like:

```
warning: 'template<class> class std::auto_ptr' is deprecated
```

Now it goes into a zombie state, and basically, your code won't compile.

Here's the error from: MSVC 2017 when using `/std:c++latest`:

```
error C2039: 'auto_ptr': is not a member of 'std'
```

If you need help with the conversion from `auto_ptr` to `unique_ptr` you can check Clang Tidy, as it provides auto conversion: Clang Tidy: modernize-replace-auto-ptr.

More details: N4190

In the linked paper N4190: there are also other library items that were removed: `unary_function` / `binary_function`, `ptr_fun()`, and `mem_fun()` / `mem_fun_ref()`, `bind1st()` / `bind2nd()` and `random_shuffle`.

Fixes

We can argue what is a fix in a language standard and what is not. Below I've picked three things that sound to me like a fix for something that was missed in the previous standards.

New auto rules for direct-list-initialization

Since C++11 we got a strange problem where:

```
auto x { 1 };
```

Is deduced as `initializer_list`. With the new standard, we can fix this, so it will deduce `int` (as most people would initially guess).

To make this happen, we need to understand two ways of initialization: copy and direct.

```
auto x = foo(); // copy-initialization
auto x{foo}; // direct-initialization, initializes an
             // initializer_list (until C++17)
int x = foo(); // copy-initialization
int x{foo}; // direct-initialization
```

For the direct initialization, C++17 introduces new rules:

```
For a braced-init-list with only a single element, auto
deduction will deduce from that entry;
For a braced-init-list with more than one element, auto
deduction will be ill-formed.
```

For example:

```
auto x1 = { 1, 2 }; // decltype(x1) is std::initializer_list<int>
auto x2 = { 1, 2.0 }; // error: cannot deduce element type
auto x3{ 1, 2 }; // error: not a single element
auto x4 = { 3 }; // decltype(x4) is std::initializer_list<int>
auto x5{ 3 }; // decltype(x5) is int
```

More details in N3922 and also in Auto and braced-init-lists, by Ville Voutilainen.

Already working since MSVC 14.0, GCC: 5.0, Clang: 3.8.

static_assert with no message

Self-explanatory. It allows just to have the condition without passing the message, the version with the message will also be available. It will be compatible with other asserts like BOOST_STATIC_ASSERT (that didn't take any message from the start).

```
static_assert(std::is_arithmetic_v<T>, "T must be arithmetic");
static_assert(std::is_arithmetic_v<T>); // no message needed since C++17
```

More details: N3928, supported in MSVC 2017, GCC: 6.0 and Clang: 2.5.

Different begin and end types in range-based for

Since C++11 range-based for loop was defined internally as:

```
{
   auto && __range = for-range-initializer;
   for ( auto __begin = begin-expr,
```

```
                __end = end-expr;
                __begin != __end;
                ++__begin ) {
          for-range-declaration = *__begin;
          statement
    }
}
```

As you can see, `__begin` and `__end` have the same type. That might cause some troubles - for example when you have something like a sentinel that is of a different type.

In C++17 it's changed into:

```
{
  auto && __range = for-range-initializer;
  auto __begin = begin-expr;
  auto __end = end-expr;
  for ( ; __begin != __end; ++__begin ) {
    for-range-declaration = *__begin;
    statement
  }
}
```

Types of `__begin` and `__end` might be different; only the comparison operator is required. This little change allows Range TS users a better experience.

More details in P0184R0, supported in MSVC 2017, GCC: 6.0 and Clang: 3.6.

Section Summary

The language standard grows, but there's some movement in the committee to remove and clean some of the features. For compatibility reasons, we cannot delete all of the problems, but one by one we can get some improvements.

Next time we'll address language clarifications: like guaranteed copy elision or expression evaluation order. So stay tuned!

Language Clarification

You all know this… C++ is a very complex language, and some (or most? :)) parts are quite confusing. One of the reasons for the lack of clarity might be a free choice for the implementation/compiler - for example to allow for more aggressive optimizations or be backward (or C) compatible. Sometimes, it's simply a lack of time/effort/cooperation. C++17 reviews some of most popular 'holes' and addressed them. In the end, we get a bit clearer way of how things might work.

Features described in this section:

- Evaluation order
- Copy elision (optional optimization that seems to be implemented across all of the popular compilers)
- Exceptions
- Memory allocations for (over)aligned data

Stricter expression evaluation order

This one is tough, so please correct me if I am wrong here, and let me know if you have more examples and better explanations. I've tried to confirm some details on slack/Twitter, and hopefully I am not writing nonsenses here :)

Let's try:

C++ doesn't specify any evaluation order for function parameters. **Dot.**

For example, that's why `make_unique` is not just a syntactic sugar, but actually it guarantees memory safety:

With `make_unique`:

```
foo(make_unique<T>(), otherFunction());
```

And with explicit `new`.

```
foo(unique_ptr<T>(new T), otherFunction());
```

In the above code we know that `new T` is quaranteed to happen before `unique_ptr` construction, but that all. For example, `new T` might happen first, then `otherFunction()`, and then `unique_ptr` constructor.
When `otherFunction` throws, then `new T` generates a leak (as the unique pointer is not yet created). When you use `make_unique`, then it's not possible to leak, even when the order of execution is random. More of such problems in GotW #56: Exception-Safe Function Calls

With the accepted proposal the order of evaluation should be 'practical.'

Examples:

- in `f(a, b, c)` - the order of evaluation of a, b, c is still unspecified, but any parameter is fully evaluated before the next one is started. Especially important for complex expressions.
  - if I'm correct that fixes a problem with `make_unique` vs `unique_ptr<T>(new T())`. As function argument must be fully evaluated before other arguments are.
- chaining of functions already work from left to right, but the order of evaluation of inner expressions might be different. look here: c++11 - Does this code from "The C++ Programming Language" 4th edition section 36.3.6 have well-defined behavior? - Stack Overflow. To be correct "The expressions are indeterminately sequenced with respect

to each other", see Sequence Point ambiguity, undefined behavior?.

- now, with C++17, chaining of functions will work as expected when they contain such inner expressions, i.e., they are evaluated from left to right: `a(expA).b(expB).c(expC)` is evaluated from left to right and `expA` is evaluated before calling b…
- when using operator overloading order of evaluation is determined by the order associated with the corresponding built-in operator:
  - so `std::cout << a() << b() << c()` is evaluated as a, b, c.

And from the paper:

> the following expressions are evaluated in the order a, then b, then c:
> 1. a.b
> 2. a->b
> 3. a->*b
> 4. a(b1, b2, b3)
> 5. b @= a
> 6. a[b]
> 7. a << b
> 8. a >> b

And the most important part of the spec is probably:

> The initialization of a parameter, including every associated value computation and side effect, is indeterminately sequenced with respect to that of any other parameter.

StackOverflow: What are the evaluation order guarantees introduced. by C++17?

More details in: P0145R3 and P0400R0. Not yet supported in Visual Studio 2017, GCC 7.0, Clang 4.0

Guaranteed copy elision

Currently, the standard allows eliding in the cases like:

- when a temporary object is used to initialize another object (including the object returned by a function, or the exception object created by a throw-expression)
- when a variable that is about to go out of scope is returned or thrown
- when an exception is caught by value

But it's up to the compiler/implementation to elide or not. In practice, all the constructors' definitions are required. Sometimes elision might happen only in release builds (optimized), while Debug builds (without any optimization) won't elide anything.

With C++17 we'll get clear rules when elision happens, and thus constructors might be entirely omitted.

Why might it be useful?

- allow returning objects that are not movable/copyable - because we could now skip copy/move constructors. Useful in factories.
- improve code portability, support 'return by value' pattern rather than use 'output params.'

Example:

```
// based on P0135R0
struct NonMoveable
{
  NonMoveable(int);
  // no copy or move constructor:
  NonMoveable(const NonMoveable&) = delete;
  NonMoveable(NonMoveable&&) = delete;

  std::array<int, 1024> arr;
};

NonMoveable make()
{
  return NonMoveable(42);
}

// construct the object:
auto largeNonMovableObj = make();
```

The above code wouldn't compile under C++14 as it lacks copy and move constructors. But with C++17 the constructors are not required - because the object `largeNonMovableObj` will be constructed in place.

Defining rules for copy elision is not easy, but the authors of the proposal suggested new, simplified types of value categories:

- `glvalue` - 'A `glvalue` is an expression whose evaluation computes the location of an object, bit-field, or function. '
- `prvalue` - A `prvalue` is an expression whose evaluation initializes an object, bit-field, or operand of an operator, as specified by the context in which it appears

In short: `prvalues` perform initialization, `glvalues` produce locations.

Unfortunately, in C++17 we'll get copy elision only for temporary objects, not for Named RVO (so it covers only the first point, not for Named Return Value Optimization). Maybe C++20 will follow and add more rules here?

More details: P0135R0, MSVC 2017: **not yet**. GCC: 7.0, Clang: 4.0.

Exception specifications part of the type system

Previously exception specifications for a function didn't belong to the type of the function, but now it will be part of it.

We'll get an error in the case:

```
void (*p)();
void (**pp)() noexcept = &p; // error: cannot convert to
                             // pointer to noexcept function

struct S { typedef void (*p)(); operator p(); };
void (*q)() noexcept = S(); // error: cannot convert to
                            // pointer to noexcept
```

One of the reasons for adding the feature is a possibility to allow for better optimization. That can happen when you have a guarantee that a function is for example `noexcept`.

Also in C++17 Exception specification is cleaned up: Removing Deprecated Exception Specifications from C++17 - it's so-called 'dynamic exception specifications'. Effectively, you can only use `noexcept` specifier for declaring that a function might throw something or not.

More details: P0012R1, MSVC 2017: **not yet**, GCC 7.0, Clang 4.0.

Dynamic memory allocation for over-aligned data

When doing SIMD or when you have some other memory layout requirements, you might need to align objects specifically. For example, in SSE you need a 16-byte alignment (for AVX 256 you need a 32-byte alignment). So you would define a vector4 like:

```
class alignas(16) vec4
{
    float x, y, z, w;
};
auto pVectors = new vec4[1000];
```

Note: **alignas** specifier is available sice C++11.

In C++11/14 you have no guarantee how the memory will be aligned. So often you have to use some special routines like `_aligned_malloc` / `_aligned_free` to be sure the alignment is preserved. That's not nice as it's not working with C++ smart pointers and also make memory allocations/deletions visible in the code (we should stop using raw new and delete, according to Core Guidelines).

C++17 fixes that hole by introducing additional memory allocation functions that use align parameter:

```
void* operator new(size_t, align_val_t);
void* operator new[](size_t, align_val_t);
void operator delete(void*, align_val_t);
void operator delete[](void*, align_val_t);
void operator delete(void*, size_t, align_val_t);
void operator delete[](void*, size_t, align_val_t);
```

now, you can allocate that `vec4` array as:

```
auto pVectors = new vec4[1000];
```

No code changes, but it will magically call:

```
operator new[](sizeof(vec4), align_val_t(alignof(vec4)))
```

In other words, `new` is now aware of the alignment of the object.

More details in P0035R4. MSVC 2017: **not yet**, GCC: 7.0, Clang: 4.0.

Templates

Do you work a lot with templates and meta-programming?

With C++17 we get a few nice improvements: some are small, but also there are notable features as well. I'm aware that for C++17 everyone wanted to have concepts, and as you know, we didn't get them. But does it mean C++17 doesn't improve templates/template meta-programming? Far from that! In my opinion, we get excellent features.

Let's look at:

- Template argument deduction for class templates
- `template<auto>`
- Fold expressions
- `constexpr if`
- Plus some smaller, detailed improvements/fixes

BTW: if you're really brave you can still use concepts! They are merged into GCC so you can play with them even before they are finally published.

Template argument deduction for class templates

I have good and bad news for you :)

Do you often use `make<T>` functions to construct a templated object (like `std::make_pair`)?
With C++17 you can forget about (most of) them and just use regular a constructor :)
That also means that a lot of your code - those `make<T>` functions can now be removed.

The reason?

C++17 filled a gap in the deduction rules for templates. Now the template deduction can happen for standard class templates and not just for functions.

For instance, the following code is (and was) legal:

```
void f(std::pair<int, char>);

// call:
f(std::make_pair(42, 'z'));
```

Because `std::make_pair` is a template function (so we can perform template deduction).

But the following wasn't (before C++17)

```
void f(std::pair<int, char>);

// call:
f(std::pair(42, 'z'));
```

Looks the same, right? This was not OK because `std::pair` is a template class, and template classes could not apply type deduction in their initialization.

But now we can do that so that the above code will compile under C++17 conformant compiler.

What about creating local variables like tuples or pairs?

```
std::pair<int, double> p(10, 0.0);
// same as
std::pair p(10, 0.0); // deduced automatically!
```

Try in Compiler Explorer: example, GCC 7.1.

This can substantially reduce complex constructions like

```
std::lock_guard<std::shared_timed_mutex,
        std::shared_lock<std::shared_timed_mutex>> lck(mut_, r1);
```

Can now become:

```
std::lock_guard lck(mut_, r1);
```

Note, that partial deduction cannot happen, you have to specify all the template parameters or none:

```
std::tuple t(1, 2, 3);              // OK: deduction
std::tuple<int,int,int> t(1, 2, 3); // OK: all arguments are provided
std::tuple<int> t(1, 2, 3);         // Error: partial deduction
```

Also if you're adventurous you can create your custom class template deduction guides: see here for more information: recent post: Arne Mertz: Modern C++ Features - Class Template Argument Deduction.

BTW: why not all `make` functions can be removed? For example, consider `make_unique` or `make_shared` are they only for 'syntactic sugar' ? I'll leave this as an open question for now.

More details in

- P0091R3
- Simon Brand: Template argument deduction for class template constructors
- Class template deduction(since C++17) - cppreference.

MSVC **not yet**, GCC: 7.0, Clang: **not yet**.

Declaring non-type template parameters with auto

This is another part of the strategy to use `auto` everywhere. With C++11 and C++14 you can use it to automatically deduce variables or even return types, plus there are also generic lambdas. Now you can also use it for deducing non-type template parameters.

For example:

```
template <auto value> void f() { }

f<10>();              // deduces int
```

This is useful, as you don't have to have a separate parameter for the type of non-type parameter. Like:

```
template <typename Type, Type value> constexpr Type TConstant = value;
                // ^^^^                          ^^^^
constexpr auto const MySuperConst = TConstant<int, 100>;
```

with C++17 it's a bit simpler:

```
template <auto value> constexpr auto TConstant = value;
                            // ^^^^
constexpr auto const MySuperConst = TConstant <100>;
```

So no need to write `Type` explicitly.

As one of the advanced uses a lot of papers/blogs/talks point to an example of Heterogeneous compile time list:

```
template <auto ... vs> struct HeterogenousValueList {};
using MyList = HeterogenousValueList<'a', 100, 'b'>;
```

Before C++17 it was not possible to declare such list directly, some wrapper class would have to be provided first.

More details in

- P0127R2 - Declaring non-type template parameters with auto
- P0127R1 - Declaring non-type template arguments with auto - motivation, examples, discussion.
- c++ - Advantages of auto in template parameters in C++17 - Stack Overflow
- Trip report: Summer ISO C++ standards meeting (Oulu) | Sutter's Mill

MSVC **not yet**, GCC: 7.0, Clang: 4.0.

Fold expressions

With C++11 we got variadic templates which is a great feature, especially if you want to work with a variable number of input parameters to a function. For example, previously (pre C++11) you had to write several different versions of a function (like one for one parameter, another for two parameters, another for three params…).

Still, variadic templates required some additional code when you wanted to implement 'recursive' functions like `sum`, `all`. You had to specify rules for the recursion:

For example:

```
auto SumCpp11(){
    return 0;
}

template<typename T1, typename... T>
auto SumCpp11(T1 s, T... ts){
    return s + SumCpp11(ts...);
}
```

And with C++17 we can write much simpler code:

```
template<typename ...Args> auto sum(Args ...args)
```

```
{
    return (args + ... + 0);
}

// or even:

template<typename ...Args> auto sum2(Args ...args)
{
    return (args + ...);
}
```

Fold expressions over a parameter pack.

Expression	Expansion
(… op pack)	((pack1 op pack2) op …) op packN
(init op … op pack)	(((init op pack1) op pack2) op …) op packN
(pack op …)	pack1 op (… op (packN-1 op packN))
(pack op … op init)	pack1 op (… op (packN-1 op (packN op init)))

Also by default we get the following values for empty parameter packs:

Operator	default value
&&	true
||	false
,	void()

Here's quite nice implementation of a `printf` using folds P0036R0:

```
template<typename ...Args>
void FoldPrint(Args&&... args) {
    (cout << ... << forward<Args>(args)) << '\n';
}
```

Or a fold over a comma operator:

```
template<typename T, typename... Args>
void push_back_vec(std::vector<T>& v, Args&&... args)
{
    (v.push_back(args), ...);
}
```

In general, fold expression allows writing cleaner, shorter and probably easier to read code.

More details in:

- N4295 and P0036R0
- "Using fold expressions to simplify variadic function templates" in Modern C++ Programming Cookbook.
- Simon Brand: Exploding tuples with fold expressions
- Baptiste Wicht: C++17 Fold Expressions
- Fold Expressions - ModernesCpp.com

MSVC **not yet**, GCC: 6.0, Clang: 3.6 (N4295)/3.9(P0036R0).

`constexpr if`

This is a big one!

The static-if for C++!

The feature allows you to discard branches of an if statement at compile-time based on a constant expression condition.

```
if constexpr(cond)
     statement1; // Discarded if cond is false
else
     statement2; // Discarded if cond is true
```

For example:

```
template <typename T>
auto get_value(T t) {
    if constexpr (std::is_pointer_v<T>)
        return *t;
    else
        return t;
}
```

This removes a lot of the necessity for tag dispatching and SFINAE and even for `#ifdefs`.

I'd like to return to this feature when we are discussing features of C++17 that simplify the language. I hope to come back with more examples of `constexpr if`.

More details in:

- P0292R2
- Simon Brand: Simplifying templates and #ifdefs with if constexpr

MSVC 2017, GCC: 7.0, Clang: 3.9.

Other

In C++17 there are also other language features related to templates. In this section, I wanted to focus on biggest enhancements, so I'll just mention the other briefly:

- Allow `typename` in a template template parameters: N4051.
    - Allows you to use `typename` instead of `class` when declaring a template template parameter. Normal type parameters can use them interchangeably, but template template parameters were restricted to `class`.
- DR: Matching of template template-arguments excludes compatible templates: P0522R0.

    - Improves matching of template template arguments. Resolves Core issue CWG 150.

- Allow constant evaluation for all non-type template arguments: N4268
    - Remove syntactic restrictions for pointers, references, and pointers to members that appear as non-type template parameters:
- `constexpr` lambdas: P0170R1
    - Lambda expressions may now be constant expressions.

Section summary

Is C++17 improving templates and meta-programming? Definitely!

We have really solid features like template deduction for class templates, `template<auto>` plus some detailed features that fix some of the problems.

Still, for me, the most powerful features, that might have a significant impact on the code is `constexpr if` and folds. They greatly clean up the code and make it more readable.

Attributes

“C++ Attributes… what?”

There were almost 40% votes like that in my recent Twitter survey. Maybe It would be good to introduce that little-known feature?

There’s even a good occasion, as in C++17 we’ll get even more useful stuff connected with attributes.

Have you ever used `__declspec`, `__attribute` or `#pragma` directives in your code?

For example:

```
struct S { short f[3]; } __attribute__ ((aligned (8)));

void fatal () __attribute__ ((noreturn));
```

Or for DLL import/export in MSVC:

```
#if COMPILING_DLL
    #define DLLEXPORT __declspec(dllexport)
#else
    #define DLLEXPORT __declspec(dllimport)
#endif
```

Those are existing forms of compiler specific attributes/annotations.

So what’s an attribute?

An attribute is additional information that can be used by the compiler to produce code. It might be utilized for optimization or some specific code generation (like DLL stuff, OpenMP, etc.).

Contrary to other languages as C#, in C++ that meta information is fixed by the compiler, you cannot add user-defined attributes. In C# you can just ‘derive’ from `System.Attribute`.

Here’s the deal about C++11 attributes:

With the modern C++, we get more and more standardized attributes that will work with other compilers. So we’re moving a bit from compiler specific annotation to standard forms.

Before C++11

In short: it was (and still is) a mess :)

`#pragma`, `_declspec`, `__attribute` … a lot of variations and compiler specific keywords.

GCC specific attributes

- Attribute Syntax - Using the GNU Compiler Collection (GCC)

- Using the GNU Compiler Collection (GCC): Common Function Attributes

MSVC specific attributes

- __declspec | Microsoft Docs

Clang specific attributes

- Attributes in Clang — Clang 5 documentation

The document lists also what syntax is supported, so a lot of those attributes can be already used in modern C++11 form.

Attributes in C++11 and C++14

C++11 did one step to minimize the need to use vendor specific syntax. As I see, the target is to move as much as compiler specific into standardized forms.

The First thing:

With C++11 we got a nicer form of specifying annotations over our code.

The basic syntax is just `[[attr]]` or `[[namespace::attr]]`.

You can use `[[att]]` over almost anything: types, functions, enums, etc., etc.

For example:

```
[[abc]] void foo()
{

}
```

In C++11 we have the following attributes:

- `[[noreturn]]`
  - for example `[[noreturn]] void terminate() noexcept;`
- `[[carries_dependency]]`
  - mostly to help optimizing multi-threaded code and when using different memory models.
  - good answer: What does the `carries_dependency` attribute mean? - Stack Overflow

C++14 added:

- `[[deprecated]]` and `[[deprecated("reason")]]`
  - Marking as deprecated in C++14 – Joseph Mansfield

Note: there no need to use attributes for alignment as there's `alignas` separate keyword for that. Before C++11 in GCC you would use `__attribute__ ((aligned (N)))`.

Have a look at also this article Modern C++ Features - Attributes - at Simplify C++.

You know a bit about the old approach, C++11/14… so what the deal about C++17?

C++17 additions

With C++17 we get three more standard attributes

- `[[fallthrough]]`
- `[[nodiscard]]`
- `[[maybe_unused]]`

Plus three supporting features.

[[fallthrough]] attribute

Indicates that a fall-through in a switch statement is intentional and a warning should not be issued for it.

```
switch (c) {
case 'a':
    f(); // Warning! fallthrough is perhaps a programmer error
case 'b':
    g();
[[fallthrough]]; // Warning suppressed, fallthrough is ok
case 'c':
    h();
}
```

More details in: P0188R1 and P0068R0 - reasoning.
GCC: 7.0, Clang: 3.9, MSVC: 15.0

[[nodiscard]] attribute

`[[nodiscard]]` is used to stress that the return value of a function is not to be discarded, on pain of a compiler warning.

```
[[nodiscard]] int foo();
void bar() {
    foo(); // Warning! return value of a
           // nodiscard function is discarded
}
```

This attribute can also be applied to types in order to mark all functions which return that type as `[[nodiscard]]`:

```
[[nodiscard]] struct DoNotThrowMeAway{};
DoNotThrowMeAway i_promise();
void oops() {
    i_promise(); // Warning emitted, return value of a
                 // nodiscard function is discarded
}
```

More details:

- P0189R1 (Wording),
- P0068R0 - reasoning.
- `nodiscard` in Jason Turner's C++ Weekly

GCC: 7.0, Clang: 3.9, MSVC: not yet

[[maybe_unused]] attribute

Suppresses compiler warnings about unused entities when they are declared with `[[maybe_unused]]`.

```
static void impl1() { ... } // Compilers may warn about this
[[maybe_unused]] static void impl2() { ... } // Warning suppressed


void foo() {
   int x = 42; // Compilers may warn about this
   [[maybe_unused]] int y = 42; // Warning suppressed
}
```

More details:

- P0212R1,
- P0068R0 - reasoning.
- `maybe_unused` in Jason Turner's C++ Weekly

GCC: 7.0, Clang: 3.9, MSVC: not yet

For more examples of those C++17 attributes you can see Simon Brand's recent article: C++17 attributes - maybe_unused, fallthrough and nodiscard.

Attributes for namespaces and enumerators

Permits attributes on enumerators and namespaces.

```
enum E {
  foobar = 0,
  foobat [[deprecated]] = foobar
};

E e = foobat; // Emits warning

namespace [[deprecated]] old_stuff{
    void legacy();
}

old_stuff::legacy(); // Emits warning
```

More details in:

- N4266,
- N4196 (reasoning)

GCC: 4.9 (namespaces)/ 6 (enums), Clang: 3.4, MSVC: 14.0

Ignore unknown attributes

That's mostly for clarification.

Before C++17 if you tried to use some compiler specific attribute, you might even get an error when compiling in another compiler that doesn't support it. Now, the compiler simply omits the attribute specification and won't report anything (or just a warning). This wasn't mentioned in the standard, so needed a clarification.

```
// compilers which don't
// support MyCompilerSpecificNamespace will ignore this attribute
[[MyCompilerSpecificNamespace::do_special_thing]]
void foo();
```

For example in GCC 7.1 there's a warnings:

```
warning: 'MyCompilerSpecificNamespace::do_special_thing'
scoped attribute directive ignored [-Wattributes]
void foo();
```

More details in:

- P0283R2 - Standard and non-standard attributes - wording
- P0283R1 - Standard and non-standard attributes - more description

MSVC **not yet**, GCC: Yes, Clang: 3.9.

Using attribute namespaces without repetition

Other name for this feature was "Using non-standard attributes" in P0028R3 and PDF: P0028R2 (rationale, examples).

Simplifies the case where you want to use multiple attributes, like:

```
void f() {
    [[rpr::kernel, rpr::target(cpu,gpu)]] // repetition
    do-task();
}
```

Proposed change:

```
void f() {
    [[using rpr: kernel, target(cpu,gpu)]]
    do-task();
}
```

That simplification might help when building tools that automatically translate annotated such code into different programming models.

More details in: P0028R4
GCC: 7.0, Clang: 3.9, MSVC: **not yet**

Section Summary

> @cppreference.com
> Attributes available in C++17
>
> - `[[noreturn]]`
> - `[[carries_dependency]]`
> - `[[deprecated]]`
> - `[[deprecated("reason")]]`
> - `[[fallthrough]]`
> - `[[nodiscard]]`
> - `[[maybe_unused]]`

I hope that after reading you understood the need of attributes: what are they and when are they useful. Previously each compiler could specify each own syntax and list of available attributes, but in modern C++ the committee tried to standardize this: there are some extracted, common parts. Plus each compiler is not blocked to add it's own extensions. Maybe at some point, we'll move away from `__attribute` or `_declspec` or `#pragma'?

There's also quite important quote from Bjarne Stroustrup's C++11 FAQ/Attributes:

> There is a reasonable fear that attributes will be used to create language dialects. The recommendation is to use attributes to only control things that do not affect the meaning of a program but might help detect errors (e.g. `[[noreturn]]`) or help optimizers (e.g. `[[carries_dependency]]`).

Code Simplification

With each C++ standard, we aim for simpler, cleaner and more expressive code. C++17 offers several “big” language features that should make our code nicer. Let’s have a look.

You might say that most of the new language features (not to mention The Standard Library improvements) are there to write simpler and cleaner code. The “C++17 in details” series reviews most of the bigger things, still, I tried to pick a few features that make your code more compact right off the bat.

- Structured bindings/Decomposition declarations
- Init-statement for if/switch
- Inline variables
- constexpr if (again!)
- a few other mentions

Structured Binding Declarations

Do you often work with tuples?

If not, then you should probably start looking into it. Not only are tuples suggested for returning multiple values from a function, but they’ve also got special language support - so that the code is even easier and cleaner.

For example (got it from `std::tie` at cppreference):

```
std::set<S> mySet;

S value{42, "Test", 3.14};
std::set<S>::iterator iter;
bool inserted;

// unpacks the return val of insert into iter and inserted
std::tie(iter, inserted) = mySet.insert(value);

if (inserted)
    std::cout << "Value was inserted\n";
```

Notice that you need to declare `iter` and `inserted` first. Then you can use `std::tie` to make the magic… Still, it’s a bit of code.

With C++17:

```
std::set<S> mySet;

S value{42, "Test", 3.14};

auto [iter, inserted] = mySet.insert(value);
```

One line instead of three! It’s also easier to read and safer, isn’t it?

Also, you can now use `const` and write `const auto [iter, inserted]` and be const correct.

Structured Binding is not only limited to tuples, we have three cases:

1. If the initializer is an array:

```
// works with arrays:
double myArray[3] = { 1.0, 2.0, 3.0 };
auto [a, b, c] = myArray;
```

2. if the initializer supports `std::tuple_size<>` and provides `get<N>()` function (the most common case I think):

```
auto [a, b] = myPair; // binds myPair.first/second
```

In other words, you can provide support for your classes, assuming you add `get<N>` interface implementation.

3. if the initializer's type contains only non static, public members:

```
struct S { int x1 : 2; volatile double y1; };
S f();
const auto [ x, y ] = f();
```

Now it's also quite easy to get a reference to a tuple member:

```
auto& [ refA, refB, refC, refD ] = myTuple;
```

And one of the **coolest usage** (support to for loops!):

```
std::map myMap;
for (const auto & [k,v] : myMap)
{
    // k - key
    // v - value
}
```

BTW: Structured Bindings or Decomposition Declaration?

For this feature, you might have seen another name "decomposition declaration" in use. As I see this, those two names were considered, but now the standard (the draft) sticks with "Structured Bindings."

Hasn't C++ Become More Pythonic? - as was written in one blog post from Jeff Preshing? :)

More Details in:

- Section: 11.5 Structured binding declarations [dcl.struct.bind]
- P0217R3
- P0144R0
- P0615R0: Renaming for structured bindings
- c++ today: Structured Binding (C++17 inside)
- C++17 Structured Bindings – Steve Lorimer

Working in GCC: 7.0, Clang: 4.0, MSVC: in VS 2017.3.

Init-statement for if/switch

New versions of the if and switch statements for C++:

`if (init; condition)` and `switch (init; condition)`.

Previously you had to write:

```
{
    auto val = GetValue();
    if (condition(val))
        // on success
    else
        // on false...
}
```

Look, that `val` has a separate scope, without that it 'leaks' to enclosing scope.

Now you can write:

```
if (auto val = GetValue(); condition(val))
    // on success
else
    // on false...
```

`val` is visible only inside the `if` and `else` statements, so it doesn't 'leak.'
`condition` might be any condition, not only if `val` is true/false.

Why is this useful?

Let's say you want to search for a few things in a string:

```
const std::string myString = "My Hello World Wow";

const auto it = myString.find("Hello");
if (it != std::string::npos)
    std::cout << it << " Hello\n"

const auto it2 = myString.find("World");
if (it2 != std::string::npos)
    std::cout << it2 << " World\n"
```

We have to use different names for `it` or enclose it with a separate scope:

```
{
    const auto it = myString.find("Hello");
    if (it != std::string::npos)
        std::cout << it << " Hello\n"
}

{
    const auto it = myString.find("World");
    if (it != std::string::npos)
        std::cout << it << " World\n"
}
```

The new if statement will make that additional scope in one line:

```
if (const auto it = myString.find("Hello"); it != std::string::npos)
    std::cout << it << " Hello\n";

if (const auto it = myString.find("World"); it != std::string::npos)
    std::cout << it << " World\n";
```

As mentioned before, the variable defined in the if statement is also visible in the `else` block. So you can write:

```
if (const auto it = myString.find("World"); it != std::string::npos)
    std::cout << it << " World\n";
else
    std::cout << it << " not found!!\n";
```

Plus, you can use it with structured bindings (following Herb Sutter code):

```
// better together: structured bindings + if initializer
if (auto [iter, succeeded] = mymap.insert(value); succeeded) {
    use(iter);  // ok
    // ...
} // iter and succeeded are destroyed here
```

More details in

- P0305R1
- C++ Weekly - Ep 21 C++17's `if` and `switch` Init Statements

GCC: 7.0, Clang: 3.9, MSVC: in VS 2017.3.

Inline variables

With Non-Static Data Member Initialization (see my post about it here), we can now declare and initialize member variables in one place. Still, with static variables (or `const static`) you usually need to define it in some `cpp` file.

C++11 and `constexpr` keyword allow you to declare and define static variables in one place, but it's limited to constexpr'essions only. I've even asked the question: c++ - What's the difference between static constexpr and static inline variables in C++17? - Stack Overflow - to make it a bit clear.

Ok, but what's the deal with this feature?

Previously only methods/functions could be specified as `inline`, but now you can do the same with variables, inside a header file.

> A variable declared inline has the same semantics as a function declared inline: it can be defined, identically, in multiple translation units, must be defined in every translation unit in which it is used, and the behavior of the program is as if there was exactly one variable.

```
struct MyClass
{
    static const int sValue;
};
```

```
inline int const MyClass::sValue = 777;
```

Or even:

```
struct MyClass
{
    inline static const int sValue = 777;
};
```

Also, note that `constexpr` variables are `inline` implicitly, so there's no need to use `constexpr inline myVar = 10;`.

Why can it simplify the code?

For example, a lot of header only libraries can limit the number of hacks (like using inline functions or templates) and just use inline variables.

The advantage over `constexpr` is that your initialization expression doesn't have to be `constexpr`.

More info in:

- P0386R2
- SO: What is an inline variable and what is it useful for?

GCC: 7.0, Clang: 3.9, MSVC: in VS 2017.3.

constexpr if

I've already introduced this feature in my previous post about templates: templates/constexpr-if. It was only a brief description, so now we can think about examples that shed a bit more light on the feature.

Regarding code samples? Hmm... As you might recall `constexpr if` can be used to replace several tricks that were already done:

- SFINAE technique to remove not matching function overrides from the overload set
  - you might want to look at places with C++14's std::enable_if - that should be easily replaced by `constexpr if`.
- Tag dispatch

So, in most of the cases, we can now just write a `constexpr if` statement and that will yield much cleaner code. This is especially important for metaprogramming/template code that is, I think, complex by its nature.

A simple example: Fibonacci:

```
template<int  N>
constexpr int fibonacci() {return fibonacci<N-1>() + fibonacci<N-2>(); }
template<>
constexpr int fibonacci<1>() { return 1; }
template<>
constexpr int fibonacci<0>() { return 0; }
```

Now, it can be written almost in a 'normal' (no compile time version):

```
template<int N>
constexpr int fibonacci()
{
    if constexpr (N>=2)
        return fibonacci<N-1>() + fibonacci<N-2>();
    else
        return N;
}
```

In C++ Weekly episode 18 Jason Turner makes an example that shows that `constexpr if` won't do any short circuit logic, so the whole expression must compile:

```
if constexpr (std::is_integral<T>::value &&
              std::numeric_limits<T>::min() < 10)
{

}
```

For `T` that is `std::string` you'll get a compile error because `numeric_limits` are not defined for strings.

In the C++Now 2017: Bryce Lelbach "C++17 Features"/16th minute there's a nice example, where `constexpr if` can be used to define `get<N>` function - that could work for structured bindings.

```
struct S
{
    int n;
    std::string s;
    float d;
};

template <std::size_t I>
auto& get(S& s)
{
    if constexpr (I == 0)
        return s.n;
    else if constexpr (I == 1)
        return s.s;
    else if constexpr (I == 2)
        return s.d;
}
```

Versus previously you would have needed to write:

```
template <> auto& get<0>(S &s) { return s.n; }
template <> auto& get<1>(S &s) { return s.s; }
template <> auto& get<2>(S &s) { return s.d; }
```

As you can see it's questionable which is the simpler code here. Although in this case, we've used only a simple `struct`, with some real world examples the final code would be much more complex and thus `constexpr if` would be cleaner.

More details:

- C++ Weekly Special Edition - Using C++17's constexpr if - YouTube - real examples from Jason and his projects.
- C++17: let's have a look at the constexpr if – FJ - I've taken the idea of fibonacci example from there.
- C++ 17 vs. C++ 14 — if-constexpr – LoopPerfect – Medium - a lot of interesting examples

MSVC 2017.3, GCC: 7.0, Clang: 3.9.

Other features

We can argue that most of the new features of C++ simplify the language in one way or another. In this post, I focused on the bigger parts, also without doing much repetition.

Still, just for recall you might want to consider the following features, which also make the code simpler:

- `template <auto>` - see here.
- Fold Expressions - already mentioned in my previous post in the series.
- Template argument deduction for class templates - mentioned here.

Not to mention a lot of library features! But we'll cover them later :)

Section Summary

In my opinion, C++17 makes real progress towards compact, expressive and easy to read code.

One of the best things is `constexpr if` that allows you to write template/metaprogramming code in a similar way to standard code. For me, it's a huge benefit (as I am always frightened of those scary template tricks).

The second feature: structured bindings (that works even in for loops) feels like code from dynamic languages (like Python).

As you can see all of the mentioned features are already implemented in GCC and Clang. If you work with the recent versions of those compilers you can immediately experiment with C++17. Soon, a lot of those features will be available in VS: VS 2017.3

Parallel Algorithms

Writing multithreaded code is hard. You'd like to utilize all of the machine's processing power, keeping code simple and avoid data races at the same time.

With C++11/14 we've finally got threading into the standard library. You can now create `std::thread` and not just depend on third party libraries or a system API. What's more, there's also async processing with futures.

For example, in 2014 I wrote about using async tasks in this article: Tasks with std::future and std::async .

Multithreading is a significant aspect of modern C++. In the committee, there's a separate "SG1, Concurrency" group that works on bringing more features to the standard.

What's on the way?

- Coroutines,
- Atomic Smart pointers,
- Transactional Memory,
- Barriers,
- Tasks blocks.
- Parallelism
- Compute
- Executors
- Heterogeneous programming models support
- maybe something more?

And why do we want to bring all of those features?

There's a famous talk from Sean Parent about better concurrency. It was a keynote at CppNow 2012, here's a recent version from 2016 from code::dive 2016.

Do you know how much processing power of a typical desktop machine we can utilize using **only** the core version of C++/Standard Library?

50%,
100%?
10%?

Sean in his talk explained that we can usually access only around 0,25% with single-threaded C++ code and maybe a few percent when you add threading from C++11/14.

So where's the rest of the power?

GPU and Vectorization (SIMD) from CPU.

GPU power	CPU vectorization	CPU threading	Single Thread
75%	20%	4%	0,25%

Of course, some third party APIs allow you to access GPU/vectorization: for example, we have CUDA, OpenCL, OpenGL, vectorized libraries, etc. There's even a chance that your compiler will try to auto-vectorize some of the code. Still, we'd like to have such support directly from the Standard Library. That way common code can be used on many platforms.

With C++11/14 we got a lot of low-level features. But it's still tough to use them effectively. What we need is an abstraction. Ideally, code should be auto-threaded/parallelized, of course with some guidance from a programmer.

C++17 moves us a bit and allows to use more computing power: it unlocks the auto vectorization/auto parallelization feature for algorithms in the Standard Library.

Plus of course, not everything can be made parallel/multi threaded as there's Amdahl's law. So always using 100% (110% with CPU boost :)) of the machine power is only a theoretical case. Still, it's better to strive for it rather than write everything single-threaded.

Overview

I've already mentioned the reasoning why we want to have so many 'tools' for multithreading/computing in the Standard.

The TS paper describing what was merged into the Standard: P0024R2

The new feature looks surprisingly simple from a user point of view. You just have a new parameter that can be passed to most of the std algorithms: this new parameter is the **execution policy**.

```
std::algorithm_name(policy, /* normal args... */);
```

I'll go into the detail later, but the general idea is that you call an algorithm and then you specify **how** it can be executed. Can it be parallel, maybe vectorized, or just serial.

That hint is necessary because the compiler cannot deduce everything from the code (at least not yet :)). We, as authors of the code, only know if there are any side effects, possible race conditions, dead locks, or if there's no sense in running it parallel (like if you have a small collection of items).

Current implementation

I hope this article will be soon updated, but for now, I have bad news.

Unfortunately, as of today, any of the major compilers support the feature.

Hovewer you can play with the following implementations/API's:

- Codeplay: http://github.com/KhronosGroup/SyclParallelSTL
- HPX: http://stellar-group.github.io/hpx/docs/html/hpx/manual/parallel.html
    - You can have a look at Rainer's article: C++17: New Parallel Algorithms where he used HPX for code samples.
- Parallel STL - https://parallelstl.codeplex.com/
- Intel - https://software.intel.com/en-us/get-started-with-pstl
- n3554 - proposal implementation (inited by Nvidia) https://github.com/n3554/n3554
- Thibaut Lutz: http://github.com/t-lutz/ParallelSTL

Execution policies

The execution policy parameter will tell the algorithm how it should be executed. We have the following options:

- `sequenced_policy` - is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and require that a parallel algorithm's execution may not be parallelized.

    - the corresponding global object is `std::execution::seq`

- **`parallel_policy`** - is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized.
    - the corresponding global object is `std::execution::par`

- **`parallel_unsequenced_policy`** - is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized and vectorized.
    - the corresponding global object is `std::execution::par_unseq`

Note that those are unique types, with their corresponding global objects. It's not just an enum.

Sequential execution seems obvious, but what's the difference between `par` and `par_unseq`?

I like the example from Bryce Adelstein's talk:

If we have a code like

```
double mul(double x,double y) {
    return x * y;
}

std::transform(
    // "Left" input sequence.
    x.begin(), x.end(),
    y.begin(), // "Right" input sequence.
    z.begin(),// Output sequence.
    mul);
```

The sequential operations that will be executed with the following instructions:

```
load x[i]
load y[i]
mul
store into z[i]
```

With the `par` policy the whole `mul()` for the i-th element will be executed on one thread, the operations won't be interleaved. But different `i` can be on a different thread.

With `par_unseq` `mul()` each operation can be on a different thread, interleaved. In practice it can be vectorized like:

```
load x[i... i+3]
load y[i...i+3]
mul // four elements at once
store into z[i...i+3]
```

Plus, each of such vectorized invocation might happen on a different thread.

With `par_unseq` function invocations might be interleaved, so it's not allowed to use vectorized unsafe code: like using mutexes, memory allocation… More on that here: @cppreference.

Also, the current approach allows to provide nonstandard policies, so compiler/library vendors might be able to provide their extensions.

Let's now see what algorithms were updated to handle the new policy parameter.

Algorithm update

Most of the algorithms (that operates on containers/ranges) from the Standard Library can handle execution policy.

What have we here?

- adjacent difference, adjacent find.
- all_of, any_of, none_of
- copy
- count
- equal
- fill
- find
- generate
- includes
- inner product
- in place merge, merge
- is heap, is partitioned, is sorted
- lexicographical_compare
- min element, minmax element
- mismatch
- move
- n-th element
- partial sort, sort copy
- partition
- remove + variations
- replace + variations
- reverse / rotate
- search
- set difference / intersection / union /symmetric difference
- sort
- stable partition
- swap ranges
- transform
- unique

The full list can be found here: @cppreference.

A simple example:

```
std::vector<int> v = genLargeVector();

// standard sequential sort
std::sort(v.begin(), v.end());

// explicitly sequential sort
std::sort(std::seq, v.begin(), v.end());

// permitting parallel execution
std::sort(std::par, v.begin(), v.end());

// permitting vectorization as well
std::sort(std::par_unseq, v.begin(), v.end());
```

New algorithms

A few existing algorithms weren't 'prepared' for parallelism, but instead we have new, similar versions:

- `for_each` - similar to `std::for_each` except returns `void` .
- `for_each_n` - applies a function object to the first n elements of a sequence.
- `reduce` - similar to `std::accumulate` , except out of order execution.
- `exclusive_scan` - similar to `std::partial_sum` , excludes the i-th input element from the i-th sum.
- `inclusive_scan` - similar to `std::partial_sum` , includes the i-th input element in the i-th sum
- `transform_reduce` - applies a functor, then reduces out of order
- `transform_exclusive_scan` - applies a functor, then calculates exclusive scan
- `transform_inclusive_scan` - applies a functor, then calculates inclusive scan

For example, we can use `for_each` (or new `for_each_n`) with an execution policy, but assuming we don't want to use the return type of the original `for_each` .

Also, there's an interesting case with **reduce**. This new algorithm provides a parallel version of accumulate. But it's important to know the difference.

Accumulate returns the sum of all the elements in a range (or a result of a binary operation that can be different than just a sum).

```
std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int sum = std::accumulate(v.begin(), v.end(), /*init*/0);
```

The algorithm is sequential only; a parallel version will try to compute the final sum using a tree approach (sum sub-ranges, then merge the results, divide and conquer). Such method can invoke the binary operation/sum in a nondeterministic order. Thus if `binary_op` is not associative or not commutative, the behavior is also non-deterministic.

For example, we'll get the same results for accumulate and reduce for a vector of integers (when doing a sum), but we might get a slight difference for a vector of floats or doubles. That's because floating point operations are not associative.

Section Summary

Is that the end for today?

Multithreading/Concurrency/Parallelism are huge topics to discover and understand. I'll hope to return with some more examples (possibly with some working implementation in common compilers!). So for now, I've described only the tip of an iceberg :)

From this post, I'd like you to remember that concurrency/parallelism is one of the key areas in the C++ standard and a lot of work is being done to bring more features.

With C++17 we get a lot of algorithms that can be executed in a parallel/vectorized way. That's amazing, as it's a solid abstraction layer. With this making, apps is much easier. A similar thing could be achieved possibly with C++11/14 or third-party APIs, but now it's all in the standard.

Below I've also gathered a few valuable resources/articles/talks so that you can learn more.

Additional Resources

The original paper for the spec: P0024R2

The initial TS paper: PDF: A Parallel Algorithms Library | N3554

ModernesCpp articles about parallel STL:

- C++17: New Parallel Algorithms of the Standard Template Library
- Parallel Algorithm of the Standard Template Library - ModernesCpp.com

Bryce Adelstein's talk about parallel algorithms. Contains a lot of examples for map reduce (transform reduce) algorithm: CppCon 2016: Bryce Adelstein Lelbach "The C++17 Parallel Algorithms Library and Beyond" YouTube

And the Sean Parent talk about better concurrency in C++: code::dive 2016 conference – Sean Parent – Better Code: Concurrency - YouTube

Filesystem

Although C++ is an old programming language, its Standard Library misses a few basic things. Features that Java or .NET had for years were/are not available in STL. With C++17 there's a nice improvement: for example, we now have the standard filesystem!

Traversing a path, even recursively is so simple now!

Although the Standard Library lacks some important features, you could always use Boost with its thousands of sub-libraries and do the work. The C++ Committee and the Community decided that the Boost libraries are so important that some of the systems were merged into the Standard. For example smart pointers (although improved with the move semantics in C++11), regular expressions, and much more.

The similar story happened with the filesystem. Let's try to understand what's inside.

Filesystem Overview

I think the Committee made a right choice with this feature. The filesystem library is nothing new, as it's modeled directly over Boost filesystem, which is available since 2003 (with the version 1.30). There are only a little differences, plus some wording changes. Not to mention, all of this is also based on POSIX.

Thanks to this approach it's easy to port the code. Moreover, there's a good chance a lot of developers are already familiar with the library. (Hmmm… so why I am not that dev? :))

The library is located in the `<filesystem>` header. It uses namespace `std::filesystem`.

The final paper is P0218R0: Adopt the File System TS for C++17 but there are also others like P0317R1: Directory Entry Caching, PDF: P0430R2–File system library on non-POSIX-like operating systems, P0492R2… All in all, you can find the final spec in the C++17 draft: the "filesystem" section, 30.10.

We have three/four core parts:

- The `path` object
- `directory_entry`
- Directory iterators
- Plus many supportive functions
  - getting information about the path
  - files manipulation: copy, move, create, symlinks
  - last write time
  - permissions
  - space/filesize
  - …

Compiler/Library support

Depending on the version of your compiler you might need to use `std::experimental::filesystem` namespace.

- GCC: You have to specify `-lstdc++fs` when you want filesystem. Implemented in `<experimental/filesystem>`.

- Clang should be ready with Clang 5.0
  - https://libcxx.llvm.org/cxx1z_status.html
- Visual Studio: In VS 2017 (2017.2) you still have to use `std::experimental` namespace, it uses TS implementation.
  - See the reference link and also C++17 Features In Visual Studio 2017 Version 15.3 Preview.
  - Hopefully by the end of the year VS 2017 will fully implement C++17 (and STL)

Examples

All the examples can be found on my Github: **github.com/fenbf/articles/cpp17**.

I've used Visual Studio 2017 Update 2.

Working with the Path object

The core part of the library is the `path` object. Just pass it a string of the path, and then you have access to lots of useful functions.

For example, let's examine a path:

```
namespace fs = std::experimental::filesystem;

fs::path pathToShow(/* ... */);
cout << "exists() = " << fs::exists(pathToShow) << "\n"
     << "root_name() = " << pathToShow.root_name() << "\n"
     << "root_path() = " << pathToShow.root_path() << "\n"
     << "relative_path() = " << pathToShow.relative_path() << "\n"
     << "parent_path() = " << pathToShow.parent_path() << "\n"
     << "filename() = " << pathToShow.filename() << "\n"
     << "stem() = " << pathToShow.stem() << "\n"
     << "extension() = " << pathToShow.extension() << "\n";
```

Here's an output for a file path like `"C:\Windows\system.ini"`:

```
exists() = 1
root_name() = C:
root_path() = C:\
relative_path() = Windows\system.ini
parent_path() = C:\Windows
filename() = system.ini
stem() = system
extension() = .ini
```

What's great about the above code?

It's so simple to use! But there's more cool stuff:

For example, if you want to iterate over all the elements of the path just write:

```
int i = 0;
```

```
for (const auto& part : pathToShow)
    cout << "path part: " << i++ << " = " << part << "\n";
```

The output:

```
path part: 0 = C:
path part: 1 = \
path part: 2 = Windows
path part: 3 = system.ini
```

We have several things here:

- the path object is implicitly convertible to `std::wstring` or `std::string`. So you can just pass a path object into any of the file stream functions.
- you can initialize it from a string, const char*, etc. Also, there's support for `string_view`, so if you have that object around there's no need to convert it to `string` before passing to `path`. PDF: WG21 P0392
- `path` has `begin()` and `end()` (so it's a kind of a collection!) that allows iterating over every part.

What about composing a path?

We have two options: using append or operator `/=`, or operator `+=`.

- append - adds a path with a directory separator.
- concat - only adds the 'string' without any separator.

For example:

```
fs::path p1("C:\\temp");
p1 /= "user";
p1 /= "data";
cout << p1 << "\n";

fs::path p2("C:\\temp\\");
p2 += "user";
p2 += "data";
cout << p2 << "\n";
```

output:

```
C:\temp\user\data
C:\temp\userdata
```

What can we do more?

Let's find a file size (using `file_size`):

```
uintmax_t ComputeFileSize(const fs::path& pathToCheck)
{
    if (fs::exists(pathToCheck) &&
        fs::is_regular_file(pathToCheck))
    {
        auto err = std::error_code{};
        auto filesize = fs::file_size(pathToCheck, err);
        if (filesize != static_cast<uintmax_t>(-1))
            return filesize;
    }
```

```
    return static_cast<uintmax_t>(-1);
}
```

Or, how to find the last modified time for a file:

```
auto timeEntry = fs::last_write_time(entry);
time_t cftime = chrono::system_clock::to_time_t(timeEntry);
cout << std::asctime(std::localtime(&cftime));
```

Isn't that nice? :)

As an additional information, most of the functions that work on a `path` have two versions:

- One that throws: `filesystem_error`
- Another with `error_code` (system specific)

Let's now take a bit more advanced example: how to traverse the directory tree and show its contents?

Traversing a path

We can traverse a path using two available iterators:

- `directory_iterator`
- `recursive_directory_iterator` - iterates recursively, but the order of the visited files/dirs is unspecified, each directory entry is visited only once.

In both iterators the directories `.` and `..` are skipped.

Ok... show me the code:

```
void DisplayDirTree(const fs::path& pathToShow, int level)
{
    if (fs::exists(pathToShow) && fs::is_directory(pathToShow))
    {
        auto lead = std::string(level * 3, ' ');
        for (const auto& entry : fs::directory_iterator(pathToShow))
        {
            auto filename = entry.path().filename();
            if (fs::is_directory(entry.status()))
            {
                cout << lead << "[+] " << filename << "\n";
                DisplayDirTree(entry, level + 1);
                cout << "\n";
            }
            else if (fs::is_regular_file(entry.status()))
                DisplayFileInfo(entry, lead, filename);
            else
                cout << lead << " [?]" << filename << "\n";
        }
    }
}
```

The above example uses not a recursive iterator but does the recursion on its own. This is because I'd like to present the files in a nice, tree style order.

We can also start with the root call:

```
void DisplayDirectoryTree(const fs::path& pathToShow)
{
    DisplayDirectoryTree(pathToShow, 0);
}
```

The core part is:

```
for (auto const & entry : fs::directory_iterator(pathToShow))
```

The code iterates over `entries`, each entry contains a path object plus some additional data used during the iteration.

Not bad, right?

Of course there's more stuff you can do with the library:

- Create files, move, copy, etc.
- Work on symbolic links, hard links
- Check and set file flags
- Count disk space usage, stats

Today I wanted to give you a general glimpse over the library. As you can see there are more potential topics for the future.

More resources

You might want to read:

- Chapter 7, "Working with Files and Streams" - of **Modern C++ Programming Cookbook**.
    - examples like: Working with filesystem paths, Creating, copying, and deleting files and directories, Removing content from a file, Checking the properties of an existing file or directory, searching.
- The whole Chapter 10 ""Filesystem" from "**C++17 STL Cookbook**"
    - examples: path normalizer, Implementing a grep-like text search tool, Implementing an automatic file renamer, Implementing a disk usage counter, statistics about file types,Implementing a tool that reduces folder size by substituting duplicates with symlinks
- C++17- std::byte and std::filesystem - ModernesCpp.com
- How similar are Boost filesystem and the standard C++ filesystem libraries? - Stack Overflow

Section Summary

I think the filesystem library is an excellent part of the C++ Standard Library. A lot of time I had to use various API to do the same tasks on different platforms. Now, I'll be able to just use one API that will work for probably 99.9% cases.

The feature is based on Boost, so not only a lot of developers are familiar with the code/concepts, but also it's proven to work in many existing projects.

And look at my samples: isn't traversing a directory and working with paths so simple now? I am happy to see that everting can be achieved using `std::` prefix and not some strange API :)

Standard Library Utilities

What I like about C++17 is that it finally brings a lot of features and patterns that are well known but come from other libraries. For example, for years programmers have been using boost libraries. Now, many of boost sub -libraries are merged into the standard. That merging process makes the transition to the modern C++ much easier, as most of the time the code will just compile and work as expected. Not to mention is the fact that soon you won't need any third party libraries.

Let's have a look at the following features:

- `std::any` - adapted from boost any
- `std::variant` - and the corresponding boost variant
- `std::optional` - boost optional library
- `std::string_view`
- Searchers for `std::search`
- Plus a few other mentions

Library Fundamentals V1 TS and more

Most of the utilities described today (`std::optional` , `std::any` , `std::string_view` , searchers) comes from so called "Library Fundamentals V1". It was in Technical Specification for some time, and with the paper "P0220R1 - Adopt Library Fundamentals V1 TS Components for C++17 (R1")" it got merged into the standard.

Support:

- Libc++ C++1Z Status
- Visual Studio Support
- GCC/libstdc++, a lot of features are in `<experimental/>` namespace/headers.

When I describe the features, I write "compiler" support, but when discussing library features, I should mention the library implementation. For the sake of simplification, I'll just stick to compiler name as each common compiler (GCC, Clang, MSVC) have its separate libs.

And now the features:

`std::any`

A better way to handle any type and replace `void*` .

Node from n4562:

> The discriminated type may contain values of different types but does not attempt conversion between them, i.e. 5 is held strictly as an int and is not implicitly convertible either to "5" or to 5.0. This indifference to interpretation but awareness of type effectively allows safe, generic containers of single values, with no scope for surprises from ambiguous conversions.

In short, you can assign any value to existing `any` object:

```
auto a = std::any(12);
a = std::string("hello world");
a = 10.0f;
```

When you want to read a value you have to perform a proper cast:

```
auto a = std::any(12);
std::cout << std::any_cast<int>(a) << '\n';

try
{
    std::cout << std::any_cast<std::string>(a) << '\n';
}
catch(const std::bad_any_cast& e)
{
    std::cout << e.what() << '\n';
}
```

Here's a bigger sample:

```
#include <string>
#include <iostream>
#include <any>
#include <map>

int main()
{
    auto a = std::any(12);

    // set any value:
    a = std::string("Hello!");
    a = 16;

    // we can read it as int
    std::cout << std::any_cast<int>(a) << '\n';

    // but not as string:
    try
    {
        std::cout << std::any_cast<std::string>(a) << '\n';
    }
    catch(const std::bad_any_cast& e)
    {
        std::cout << e.what() << '\n';
    }

    // reset and check if it contains any value:
    a.reset();
    if (!a.has_value())
    {
        std::cout << "a is empty!" << "\n";
    }

    // you can use it in a container:
    std::map<std::string, std::any> m;
```

```
    m["integer"] = 10;
    m["string"] = std::string("Hello World");
    m["float"] = 1.0f;

    for (auto &[key, val] : m)
    {
        if (val.type() == typeid(int))
            std::cout << "int: " << std::any_cast<int>(val) << "\n";
        else if (val.type() == typeid(std::string))
            std::cout << "string: " << std::any_cast<std::string>(val) << "\n";
        else if (val.type() == typeid(float))
            std::cout << "float: " << std::any_cast<float>(val) << "\n";
    }
}
```

Notes

- `any` object might be empty.
- `any` shouldn't use any dynamically allocated memory, but it's not guaranteed by the spec.

More info in:

- n4562: any
- std::any - cppreference.com
- Boost.Any - 1.61.0
  - c++ - What is the typical usage of boost any library? - Stack Overflow
- Conversations: I'd Hold Anything for You [1] | Dr Dobb's

MSVC VS 2017, GCC: 7.0, Clang: 4.0

`std::variant`

Type safe unions!

With a regular `union` you can only use POD types, and it's not safe - for instance, it won't tell you which variant is currently used. With `std::variant` it's only possible to access types that are declared.

For example:

```
std::variant<int, float, std::string> abc;
```

`abc` can only be initialized with `int`, `float` or `string` and nothing else. You'll get a compile time error when you try to assign something else.

To access the data, you can use:

- `std::get` with index or type of the alternative. It throws `std::bad_variant_access` on errors.
- `std::get_if` - returns a pointer to the element or `nullptr`;
- or use `std::visit` method that has usage especially for containers with variants.

A bigger playground:

```
#include <string>
```

```
#include <iostream>
#include <variant>

struct SampleVisitor
{
    void operator()(const int &i) const { std::cout << "int: " << i << "\n"; }
    void operator()(const float& f) const { std::cout << "float: " << f << "\n"; }
};

int main()
{
    std::variant<int, float> intOrFloat;
    static_assert(std::variant_size_v<decltype(intOrFloat)> == 2);

    // default initialized to the first alternative, should be 0
    std::visit(SampleVisitor(), intOrFloat);

    // won't compile:
    // error: no match for 'operator=' (operand types are 'std::variant<int, float>' and
'std::__cxx11::basic_string<char>')
    // intOrFloat = std::string("hello");

    // index will show the currently used 'type'
    std::cout << "index = " << intOrFloat.index() << std::endl;
    intOrFloat = 100.0f;
    std::cout << "index = " << intOrFloat.index() << std::endl;

    // try with get_if:
    if (const auto intPtr (std::get_if<int>(&intOrFloat)); intPtr)
        std::cout << "int!" << *intPtr << "\n";
    else if (const auto floatPtr (std::get_if<float>(&intOrFloat)); floatPtr)
        std::cout << "float!" << *floatPtr << "\n";

    // visit:
    std::visit(SampleVisitor(), intOrFloat);
    intOrFloat = 10;
    std::visit(SampleVisitor(), intOrFloat);
}
```

Notes:

- Variant is not allowed to allocate additional (dynamic) memory.
- A variant is not permitted to hold references, arrays, or the type void.
- The first alternative must always be default constructible
- A variant is default initialized with the value of its first alternative.
- If the first alternative type is not default constructible, then the variant must use `std::monostate` as the first alternative

More info:

- P0088R3: Variant: a type-safe union for C++17 (v8). - note that Variant wasn't in the Library Fundamentals, it was a separate proposal.

MSVC VS 2017, GCC: 7.0, Clang: 4.0?

`std::optional`

Another and elegant way to return objects from functions that are allowed to be empty.

For example:

```
std::optional<std::string> ostr = GetUserResponse();

if (ostr)
    ProcessResponse(*ostr);
else
    Report("please enter a valid value");
```

In the simple sample above `GetUserResponse` returns optional with a possible string inside. If a user doesn't enter a valid value `ostr` will be empty. It's much nicer and expressive than using exceptions, nulls, output params or other ways of handling empty values.

A better example:

```
#include <optional>
#include <iostream>
#include <string>

std::optional<int> GetInt(int r)
{
    if (r % 2 == 0)
        return r/2;

    return { };
}

void ShowOptionalInt(const std::optional<int>& oi)
{
    if (oi)
        std::cout << "int ok: " << *oi << "\n";
    else
        std::cout << "bad int...\n";
}

int main()
{
    std::cout << sizeof(int) << ", " << sizeof(std::optional<int>) << "\n";
    std::cout << sizeof(double) << ", " << sizeof(std::optional<double>) << "\n";
    std::cout << sizeof(std::string) << ", " << sizeof(std::optional<std::string>) <<
"\n";
    auto oi = GetInt(10);
    ShowOptionalInt(oi);
    auto oi2 = GetInt(11);
    ShowOptionalInt(oi2);
}
```

Notes:

- Implementations are not permitted to use additional storage, such as dynamic memory, to allocate its contained value. The contained value shall be allocated in a region of the optional storage suitably aligned for the type T.

More info:

- n4562: optional

- Boost Optional
- Efficient optional values | Andrzej's C++ blog
- Recipe "Safely signalizing failure with std::optional" from C++17 STL Cookbook.

MSVC VS 2017, GCC: 7.0, Clang: 4.0?

`string_view`

Although passing strings got much faster with move semantics from C++11, there's still a lot of possibilities to end up with many temporary copies.

A much better pattern to solve the problem is to use a string view. As the name suggests instead of using the original string, you'll only get a non-owning view of it. Most of the time it will be a pointer to the internal buffer and the length. You can pass it around and use most of the common string functions to manipulate.

Views work well with string operations like sub string. In a typical case, each substring operation creates another, smaller copy of some part of the string. With string view, `substr` will only map a different portion of the original buffer, without additional memory usage, or dynamic allocation.

Another important reason for using views is the consistency: what if you use other implementations for strings? Not all devs have the luxury to work only with the standard strings. With views, you can just write (or use) existing conversion code, and then string view should handle other strings in the same way.

In theory `string_view` is a natural replacement for most of `const std::string&`.

Still, it's important to remember that it's only a non-owning view, so if the original object is gone, the view becomes rubbish.

If you need a real string, there's a separate constructor for `std::string` that accepts a `string_view`. For instance, the filesystem library was adapted to handle string view (as input when creating a path object).

Ok, but let's play with the code:

```
#include <string>
#include <iostream>
#include <string_view>

// we need to overload 'new' to see what's
// hapenning under the hood...
void* operator new(std::size_t n)
{
    std::cout << "new " << n << " bytes\n";
    return malloc(n);
}

int main()
{
    // the original string, one allocation:
    std::string str {"Hello Amazing Programming World" };

    // another allocation for the substring, separate 'copy'
    auto subStr = str.substr(str.find("Programming"));
    std::cout << subStr << "\n";

    // no allocation for the sub range:
```

```
    std::string_view strView { str };
    auto subView = strView.substr(str.find("Programming"));
    std::cout << subView << "\n";
}
```

More info:

- n4562: `string_view` and also N3921, string_view: a non-owning reference to a string, revision 7
- What is string_view? - Stack Overflow
- C++17 string_view – Steve Lorimer
- Modernescpp - string_view
- foonathan::blog() - std::string_view accepting temporaries: good idea or horrible pitfall?

MSVC VS 2017, GCC: 7.0, Clang: 4.0?

Searchers

When you want to find one object in a `string`, you can just use find or some other alternative. But the task complicates when there's a need to search for a pattern (or a sub range) in a string.

The naive approach might be `O(n*m)` (where `n` is the length of the whole string, `m` is the length of the pattern).

But there are much better alternatives. For example Bayer-Moore with the complexity of `O(n+m)`.

C++17 updated `std::search` algorithm in two ways:

- you can now use execution policy to run the default version of the algorithm but in a parallel way.
- you can provide a Searcher object that handles the search.

For now we have three searchers:

- `default_searcher`
- `boyer_moore_searcher`
- `boyer_moore_horspool_searcher`

You can play with the example here:

```
#include <iostream>
#include <string>
#include <algorithm>
#include <functional>
// in gcc 7.1 still have to use experimental namespace/headers
#include <experimental/functional>
#include <experimental/algorithm>
#include <chrono>

template <typename TFunc> void RunAndMeasure(TFunc func)
{
    const auto start = std::chrono::steady_clock::now();
    func();
    const auto end = std::chrono::steady_clock::now();
    std::cout << std::chrono::duration <double, std::milli> (end - start).count() << "
ms\n";
}
```

```
int main()
{
    std::string in;
    std::string needle = "Hello Amazing Programming World";
    RunAndMeasure([&]() {
        const int maxIters = 20000000u;
        for (int i = 0; i < maxIters; ++i)
        {
            in += "abcd ";
            if (i % 7)
                in += "xyz, uvw ";
            if (i == maxIters - 100)
                in += " $" + needle + "$ ";
        }
    });

    RunAndMeasure([&]() {
    auto it = std::experimental::search(in.begin(), in.end(),
                  std::experimental::default_searcher(
                      needle.begin(), needle.end()));
    if(it == in.end())
        std::cout << "The string " << needle << " not found\n";
    });

    RunAndMeasure([&]() {
    auto it = std::experimental::search(in.begin(), in.end(),
                  std::experimental::boyer_moore_searcher(
                      needle.begin(), needle.end()));
    if(it == in.end())
        std::cout << "The string " << needle << " not found\n";
    });

    RunAndMeasure([&]() {
    auto it = std::experimental::search(in.begin(), in.end(),
                  std::experimental::boyer_moore_horspool_searcher(
                      needle.begin(), needle.end()));
    if(it == in.end())
        std::cout << "The string " << needle << " not found\n";
    });
}
```

- Which version is the fastest?
- Is this better than just `std::string::find`?

More info:

- N3905 Extending std::search to use Additional Searching Algorithms (Version 4)

MSVC VS 2017.3, GCC: 7.0, Clang: 3.9?

Other Changes in STL

- `shared_ptr` with array - P0414R2: Merging shared_ptr changes from Library Fundamentals to C++17. So far `unique_ptr` was able to handle arrays. Now it's also possible to use `shared_ptr`.

- Splicing Maps and Sets - PDF P0083R2 - we can now move nodes from one tree based container (maps/sets) into other ones, without additional memory overhead/allocation.
- Mathematical special functions - PDF: P0226R1
- Improving `std::pair` and `std::tuple` - N4387) - pair/tuple obey the same initialization rules as their underlying element types.
- Sampling - n4562: Sampling - new algorithm that selects `n` elements from the sequence
- Elementary string conversions - P0067R5, new function `to_chars` that handles basic conversions, no need to use stringstream, sscanf, itoa or other stuff.

Summary

Did I miss something in the whole feature descriptions? Yes!

There are many other changes in STL and the language that would fill many more pages. But let's stop for now.

If you want to dig deeper try to read the spec/draft or look at the official paper with changes: P0636r0: Changes between C++14 and C++17 DIS.

What do I like the most?

From the language:

- `constexpr if` - very powerful tool, allows you to write template/metaprogramming code in a similar way to standard code.
- Structured bindings - moves C++ closer to dynamic languages
- Template argument deduction for class templates
    - And other template features

From STL:

- Filesystem - a significant portion of the library, that will make code much easier and common across many platforms.
- type safe helpers: `std::any`, `std::optional`, `std::variant` - we can now replace `void*` or C style unions. The code should be safer.
- string features: like `string_view`, string conversions, searchers.
- parallelism - very powerful abstraction for threading.

Still, there's a lot of stuff to learn/teach! I've just described the features, but the another part of the equation is to use them effectively. And that needs experience.

I hope that my package about C++17 gave you enough information to start your experiments. Play with the features, learn how they can be added to your projects. And also, don't forget to have fun.

Thanks for reading!
Bartek
bfilipek.com