Remedial C++ 17 Language features

Pete Williamson
petewil00@hotmail.com

C++ Landscape

Original C++

C++ 98

C++ 11 (previously called C++ 0X)

C++ 14

C++ 17

C++ 20

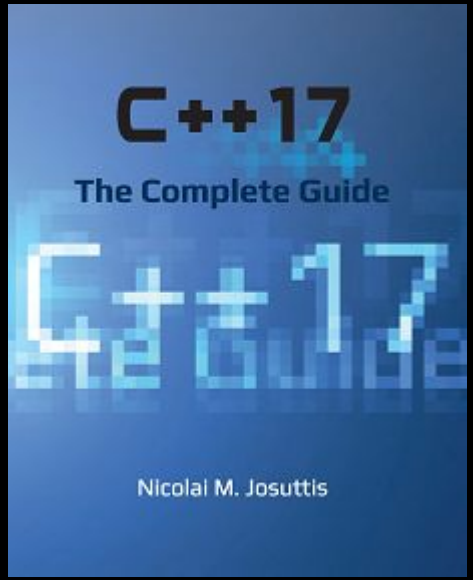
C++ 23 (in planning)

C++17 The Complete Guide

By Nicolai M. Josuttis

Was the source for many of the examples and the organization for this talk.

https://www.cppstd17.com/



C++ 17 language changes (part 1)

- Making the text message for static_assert optional
- Allow typename (as an alternative to class) in a template template parameter
- New rules for auto deduction from braced-init-list
- Nested namespace definitions, e.g., namespace X::Y { … } instead of namespace X { namespace Y { … } }
- Allowing attributes for namespaces and enumerators
- New standard attributes [[fallthrough]], [[maybe_unused]] and [[nodiscard]]
- UTF-8 (u8) character literals[16][19] (UTF-8 string literals have existed since C++11; C++17 adds the corresponding character literals for consistency, though as they are restricted to a single byte they can only store ASCII)
- Hexadecimal floating-point literals
- Constant evaluation for all non-type template arguments
- Fold expressions, for variadic templates
- A compile-time static if with the form if constexpr(expression)
- Structured binding declarations, allowing auto [a, b] = getTwoReturnValues();

C++ 17 language changes (part 2)

- Initializers in **if** and **switch** statements
- copy-initialization and direct-initialization of objects of type T from prvalue expressions of type T (ignoring top-level cv-qualifiers) shall result in no copy or move constructors from the prvalue expression. See copy elision for more information.
- Some extensions on over-aligned memory allocation
- Class template argument deduction (CTAD), introducing constructor deduction guides, eg. allowing std::pair(5.0, false) instead of explicitly specifying constructor arguments types std::pair<double, bool>(5.0, false) or without an additional helper template function std::make_pair(5.0, false).
- Inline variables, which allows the definition of variables in header files without violating the one definition rule. The rules are effectively the same as inline functions
- __has_include, allowing the availability of a header to be checked by preprocessor directives
- The keyword register is now reserved and unused (formerly deprecated)
- Value of __cplusplus changed to 201703L
- Exception specifications were made part of the function type

C++ 17 library changes

- Most of Library Fundamentals TS I, including:
  - std::string_view, a read-only non-owning reference to a character sequence or string-slice
  - std::optional, for representing optional objects, a data type that may not always be returned by a given algorithm with support for non-return
  - std::any, for holding single values of any type
- std::uncaught_exceptions, as a replacement of std::uncaught_exception in exception handling
- New insertion functions try_emplace and insert_or_assign for std::map and std::unordered_map key-value associative data structures
- Uniform container access: std::size, std::empty and std::data
- Definition of "contiguous iterators"
- A file system library based on boost::filesystem
- Parallel versions of STL algorithms
- Additional mathematical special functions, including elliptic integrals and Bessel functions
- std::variant, a tagged union container
- std::byte, allowing char to be replaced for data types intending to model a byte of data as a byte rather than a character
- Logical operator traits: std::conjunction, std::disjunction and std::negation

Structured Bindings

Structured Bindings

What it does:

Give local names to structure members

Example: (from Josuttis)

```
struct MyStruct { int i = 0; std::string s }
MyStruct ms;

auto [u, v] = ms;
```

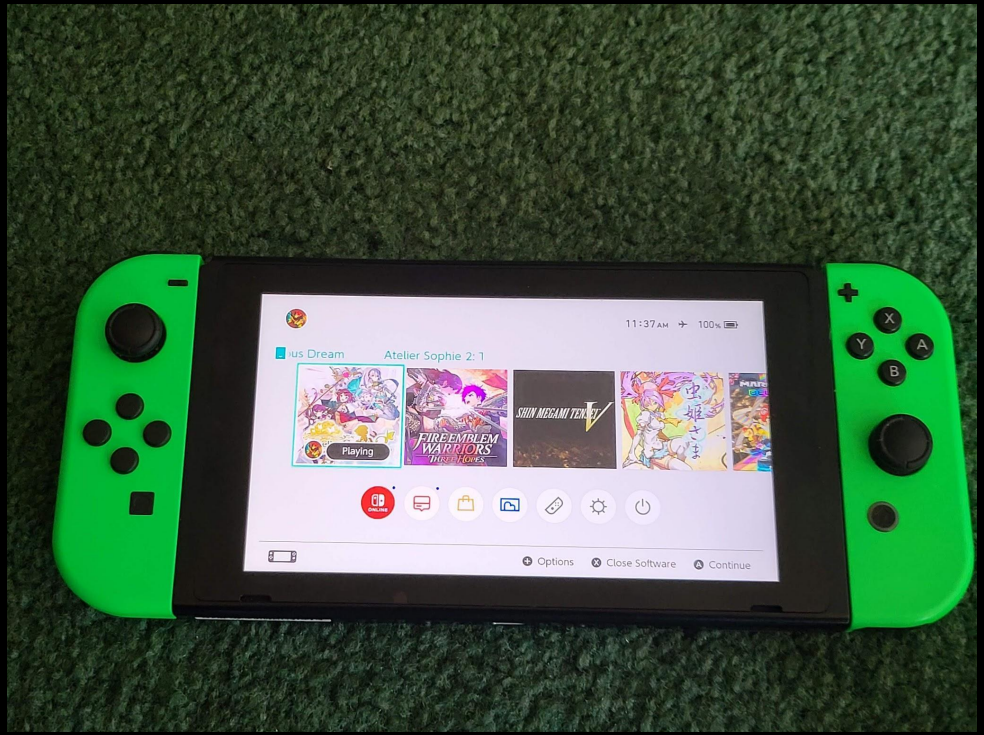
Structured Bindings

When to use it:

Give local names to variables when a function returns a struct - like multiple return values.

```
// keeps struct alive in scope.
auto[id, val] = getStruct();
```

Initialization inside if and switch



Initialization inside if and switch

What it does:

Lets you init something before checking it

Example:

```
if (rc = file_open(foo); rc == SUCCESS) { … }
```

When to use it:

Not sure that I would, I think that the code clearer with init outside. Just the same, we should be able to read it in case somebody else writes it.

Inline variables

Inline variables

What it does:

Just use “inline”, and you can define variables in header files

Example:

In foo.h:

```
inline Foo globalFooObject;
```

Inline variables

When to use it:

You're making a class entirely contained in a header file.

You want a single global object in your program, and the header might be included in many cpp files. Before you had to ensure that only one cpp file contained the object, now you don't need to put it in any of them.

Inline variables

Notes:

1. `constexpr` implies inline, so you can use `constexpr` to get the same effect.
2. You can use `inline` with `thread_local` to get one copy of the variable per thread.

Aggregate extensions

Aggregate extensions

Uh, what was an aggregate again?

Remember how we can initialize a class using curly braces? The part with the curlys is the aggregate.

```
MyData foo{“Meaning of Universe”, 42}
```

Aggregate Extensions

What it does:

Aggregates can have base classes now

You can skip inner curly braces too

Aggregate Extensions

Example:

```
struct Data {
  std::string name;
  double value;
}

Data x {“Foo”, 1.4};
```

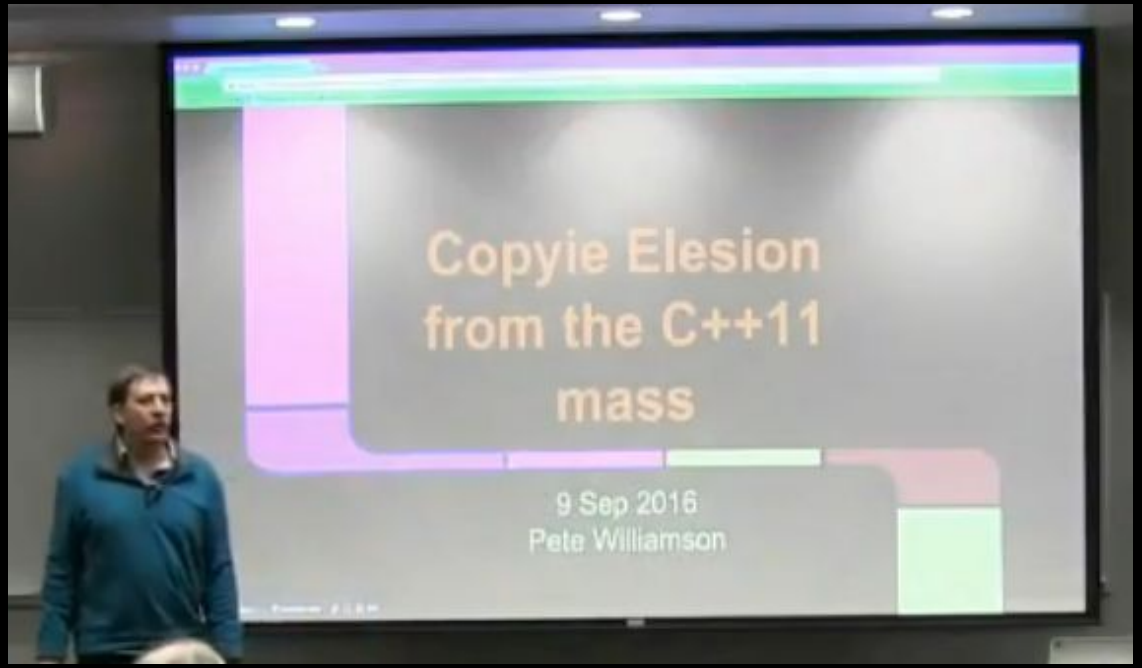
```
struct MoreData: Data {
  bool checked;
}

MoreData y{{“test1”, 1.8}, false};
MoreData z{“test2”, 5.4, true};
```

Aggregate Extensions

When to use it:

You want to initialize a derived class with an aggregate.

Mandatory Copy Elision



Mandatory Copy Elision

What it does:

In some cases, the compiler now has to construct an object in its final location instead of making a temporary and copying it. This used to be optional for the compiler.

Since it was optional, we had to write a copy constructor and move operator for the class. Now we can skip that, but only for unmaterialized objects.

Mandatory Copy Elision

Example (Josuttis):

```
class MyClass { … }
void foo(MyClass params) { … } // params is init by arg
MyClass bar() { return MyClass(); }

int main() {
  foo(MyClass{});    // pass temp to init param
  MyClass x = bar(); // use temp to init x
  foo(bar());        // use returned temp to init param
}
```

Mandatory Copy Elision

When to use it:

1. This guarantees better perf by avoiding copies
2. This lets us get away without writing copy ctor, move operator - sometimes.

Mandatory Copy Elision

When we can’t use it:

We can’t use it for materialized objects:

```
MyClass foo() {
  MyClass obj;
  …
  // This has already been materialized
  return obj;
}
```

Lambda extensions

I rode through the desert on a function with no name.

Lambda extensions

What it does:

A number of small improvements to Lambdas:

- They are constexpr if they can safely be made constexpr => You can use them at compile time.
- You can now capture the this pointer [*this] to get a copy of the current object in case the lambda lives longer than the calling scope.
- You can now capture objects by const reference

Lambda extensions examples

```
auto squared = [ ](auto val) {
  Return val * val;
};
// This can now be called at compile time.
std::array<int, squared(5)> a;
```

Lambda extensions examples

```
class C {
  private:
    Std::string name;
  public:
    void foo() {
      auto l1 = [*this] {
        std::cout << name << ‘\n’; };
    }
```

Lambda extensions

When to use it:

Helpful in moving more code to compile time to speed up your app.

I would avoid functions where you capture a copy of the object, but there could be edge cases where it helps. If the lambda is complicated enough to need a copy of the object, a normal function may be a better fit.

New Attributes

New attributes

What it does:

C++ has had attributes (which act like Java annotations) since C++ 11. C++ 17 adds several new ones.

- `[[nodiscard]]` - encourages warnings if result is unused.
- `[[maybe_unused]]` - avoid warnings if not used
- `[[fallthrough]]` - avoid warnings for intentional case statement fallthrough

New attributes examples

```
[[nodiscard]] int* foo();

int bar(int val,
        [[maybe_unused]] int debug_code)
{...}
```

[[fallthrough]] example

```
switch (error) {
  case (404):
    printf(“Page not found”);
    [[fallthrough]];
  case ():
    printf”(“bad request”);
    return REQUEST_ERROR;
}
```

When to use new attributes

Sparingly. Every one of these, in my opinion, marks a suboptimal design choice, consider refactoring the code first. Might be useful for adding to old code as you increase warning level.

Nested namespaces

Nested namespaces

What it does:

Allows an easier way to specify a nested namespace

Example:

```
namespace A::B::C {
    …
}
```

When to use it:

This is good, use away.

Defined expression eval order

Defined expression eval order

What it does:

Some things we assumed in the past to work as expected were really undefined behavior, this defines it.

Now for many operators (not all), it is guaranteed they work in an order we would expect: expr 1, then expr2

```
expr1[expr2]      expr1.expr2       expr1.*expr2
expr1->*expr2     expr1 << expr2    expr1 >> expr2
expr2 = expr1     expr2 += expr1    expr2 *= expr1
```

Defined expression eval order

Example: (Jousittis)

```
std::string str =
    “I heard it even works if you don’t believe”;

std::replace(0,8,””).
    replace(s.find(“even”), 4, “sometimes”).
    replace(s.find(“you don’t”, 9, “I”);
```

You might expect: “it sometimes works if I believe”

Defined expression eval order

Example (continued):

This could produce different results depending on which relative order the find and replace calls operate in

Before C++ 17, you might get:

It sometimes workIdon’t believe

It even worksometiIdon’t believe

It even worsometimesf youllieve

Defined Expression Eval order

When to use it:

We **will** be using it, like it or not (but it should be good)

// still undefined, but we shouldn't do this.

```
i = i++ + i;
```

Relaxed Enum init from int

Relaxed Enum init from int

What it does:

You can now init an enum using the underlying int

Example:

```
enum class Weekday { mon, tue, wed, … }
Weekday day1{0};  // OK with C++ 17
Weekday day2 = 0; // Still an error
```

Relaxed enum from int

When to use it:

Allows you to define a new enum type based on the underlying integral type without specifying all values

```
enum MyInt : char { };
MyInt int1{42};
```

Not sure I would use it this way.

Fixed direct list init with auto

Fixed direct list init with auto

What it does:

Before:

```
int x{42}; // initializes an int
auto a{42};  // error
auto b{1,2,3}; initializes a std::initializer
list
```

After:

```
auto a{42}; // initializes an int
auto b{1, 2, 3}; // now an error
```

Fixed direct list init with auto

When to use it:

Note that this is a breaking change as one old case is now an error.

I wouldn’t go out of my way to use this. While auto is good, I’d generally prefer to use the actual type if I didn’t really need the auto. Might be helpful in a template.

Hexadecimal floating point literals

Hexadecimal floating point literals

What it does:

You can specify float literals in hex to get an exact representation.

Example:

```
std::initializer_list<double> values {
    0x1p4,          // 16
    0xC.68p+2,      // 49.625
};
```

Hexadecimal floating point literals

The mantissa is written in hex.

The exponent is base 2, and written in decimal

0xCp2 is the decimal value 48 = 12 * 2^2

When to use:

When you really want an exact value.

UTF-8 character literals

UTF-8 character literals

What it does:

Before you could use the u8 prefix for strings, but not for characters. Now you can use it for characters too.

Example:

```
auto ch = u8'8';
```

When to use:

You need a single UTF-8 literal character.

Smaller features

Exception specifications as type

Exception specifications as type

What it does:

The exception specification is now officially part of the type of the function

Example:

```
void funcGuaranteedNotToThrow() noexcept;
void funcThatMightThrow();
```

Exception specification as type

When to use it:

You might want a function pointer of a type guaranteed not to throw, and want an error if you try to assign a throwing function to it:

```
void(*func_pointer) () noexcept;
func_pointer = funcGuaranteedNotToThrow;  // OK
Func_pointer = funcThatMightThrow;        // Error
```

static_assert: message optional

What it does:

Now in static_assert() you don’t need the “message”

Example:

```
static_assert(sizeof(int)>=4, “Needs big ints”);
static_assert(sizeof(int)>=4);  // OK in C++ 17
```

When to use:

Sparingly. I think asserts are better with a message.

Preprocessor __has__include

What it does:

In the preprocessor, check to see if a header could be included.

Example:

```
#if __has_include(<filesystem>)
#  include <filesystem>
#endif
```

Preprocessor __has__include

When to use it:

Might switch on different but similar includes depending on what is available in the current compilation environment (which platform you are compiling for).

Note it doesn’t guarantee the file has the expected contents, only that the file exists.

Note preprocessor only, can’t use in the code itself.

Template features

Class template argument deduction

Class Template Argument Deduction

What it does:

Deduces template arguments whenever there is enough information (most of the time).

Examples:

```
From: std::complex<double> c{3.14, 2.72}
To:   std::complex c{3.14. 2.72}
```

Class Template Argument Deduction

When to use it:

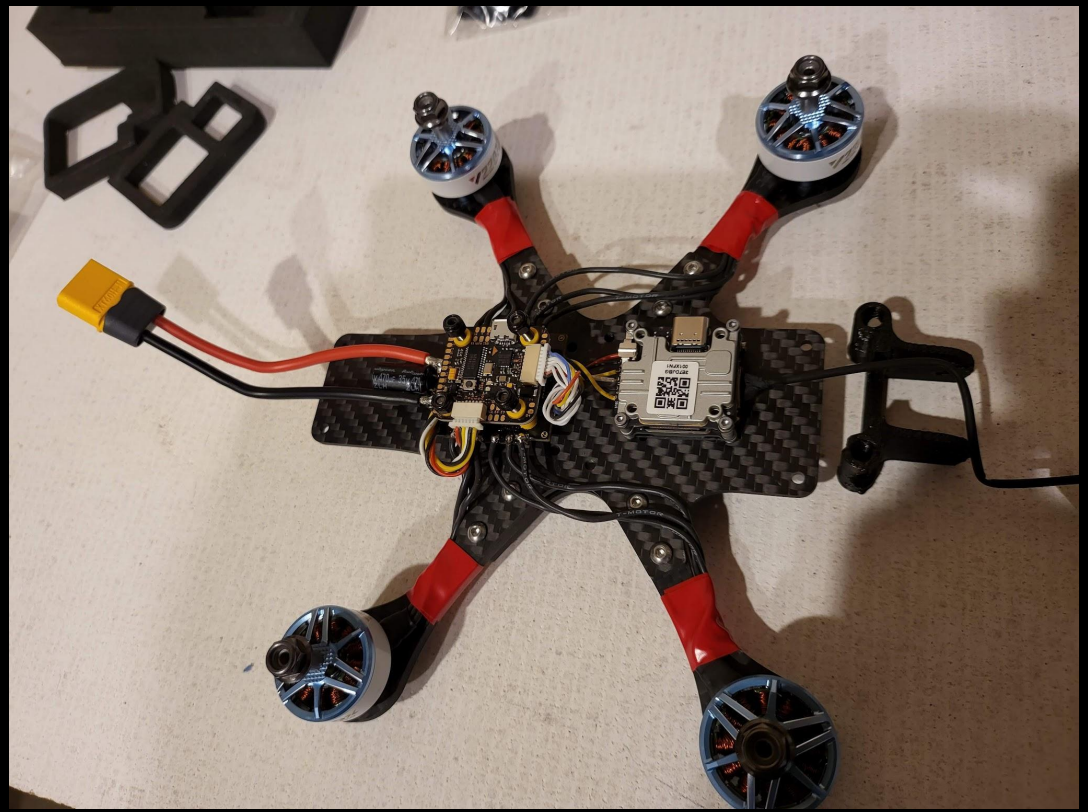
When the class type is obvious to any maintenance programmers.

Class Template Argument Deduction

More details:

- It works for copying
- You can make a wrapper class
  that deduces the type of a callback lambda
- It won't work for partial argument deduction
- We can replace make_pair<> with ctor{}
- You can provide deduction guides if the compiler chooses wrong

Compile time if



Compile time if (thanks, Walter!)

What it does:

As it says, calculates the if/then/else at compile time.

Compile time if: example - Josuttis

```
template <typename T> std::string asString(T x) {
  if constexpr(std::is_same_v<T, std::string>) {
    return x;
  } else if constexpr(std::is_arithmetic_v<T>) {
    return std::to_string(x);
  } else {
    return std::string(x);
  }
}
```

Compile time if - when to use

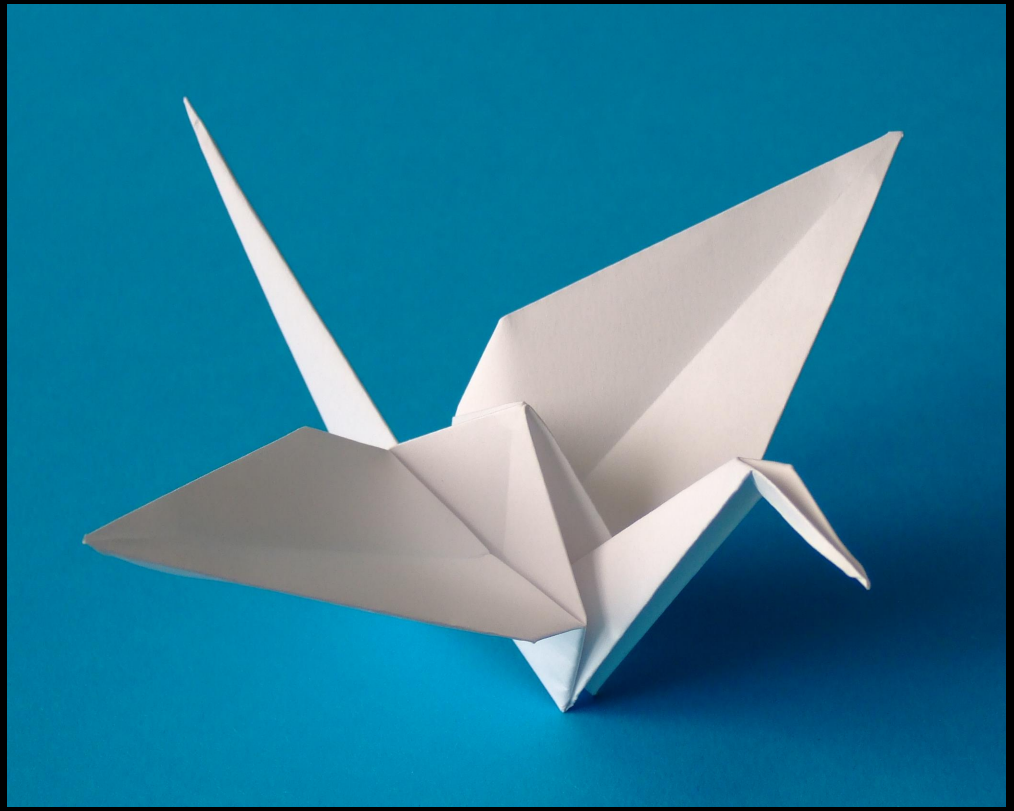
- Allows templates that would otherwise never compile like the example.
- Could be used for perfect forwarding (& -> &, && -> &&) of return values
- Helpful for tag dispatching
- Can be used outside templates to reduce code size

Compile time if gotchas

Gotchas:

- The unused branches must still be syntactically correct. It could affect the return type.
- Return type could be affected by “else” even if then returns.
- Does not short circuit the condition like regular if.

Fold Expressions



Fold expressions

What it does:

Apply a binary operator to all the arguments.

Example:

```
template<typename… T>
auto addAll(T… args) {
    return (... + args);
}
```

Fold Expression types

Unary left fold: `( … op args )`
Unary right fold: `( args op … )`
Binary left fold: `( value op … op args )`
Binary right fold: `( args op … op value )`

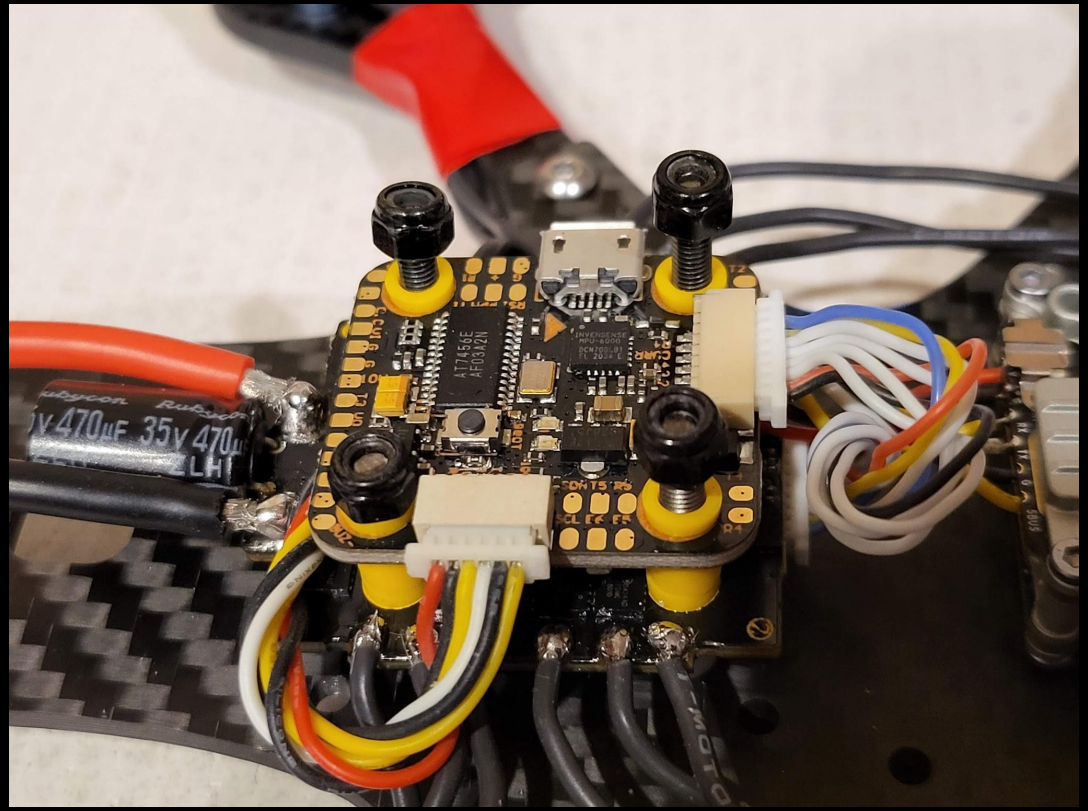
We usually want a left fold, which starts from the left:
((arg1 op arg2} op arg3) op …
Especially important for strings instead of just “+”

Fold Expressions

When to use them:

Before this feature, you had to write a recursive template to do this, stressing out both the compiler and the programmer

String Literals for Template Param



String Literals for Template Param

What it does:

You used to need to have “external linkage” or “internal linkage” to pass a string as a template parameter. Now you can do it with “no linkage”

Before we explain this more, a quick digression:

String linkage types

String linkage types

```
extern const char hello[] = “Hello”; // external
const char hello11[] = “Howdy”; // internal
void somefunc() {
    // This declaration has no linkage.
    static const char hello17[] = “Greetings”;
    ...
}
```

String Literals for Template Param

Example:

```
template<const char* string>
class Greeting { ... }
void somefunc() {
    ...
    Greeting<Hello17> message17; // OK in C++17
    Greeting<”Hi there”> badMessage;  // Error
```

String Literals for Template Param

When to use it:

Before this, you could have a template that takes a pointer for an argument, but you couldn’t have a compile time function that returned the address of something. This feature fixes that hole.

Placeholder types as Templ. param

Placeholder types as Templ. Param

What it does:

You can use auto to declare a non-type template parameter.

Placeholder types as Templ. Param

Example:

```
template<auto T> class Foo {  ... }

Foo<42> foo1;  // type of T is int
Foo<'p'> foo2;  // type of T is char
```

Placeholder types as Templ. Param

When to use it:

1. A template that could take either a string or a char.
2. Define compile time constants
3. Using auto as a variable template param

Extended Using Declarations

Extended Using Declarations

What it does:

You can now use a comma separated list of names in a using declaration. That lets you use it with parameter packs.

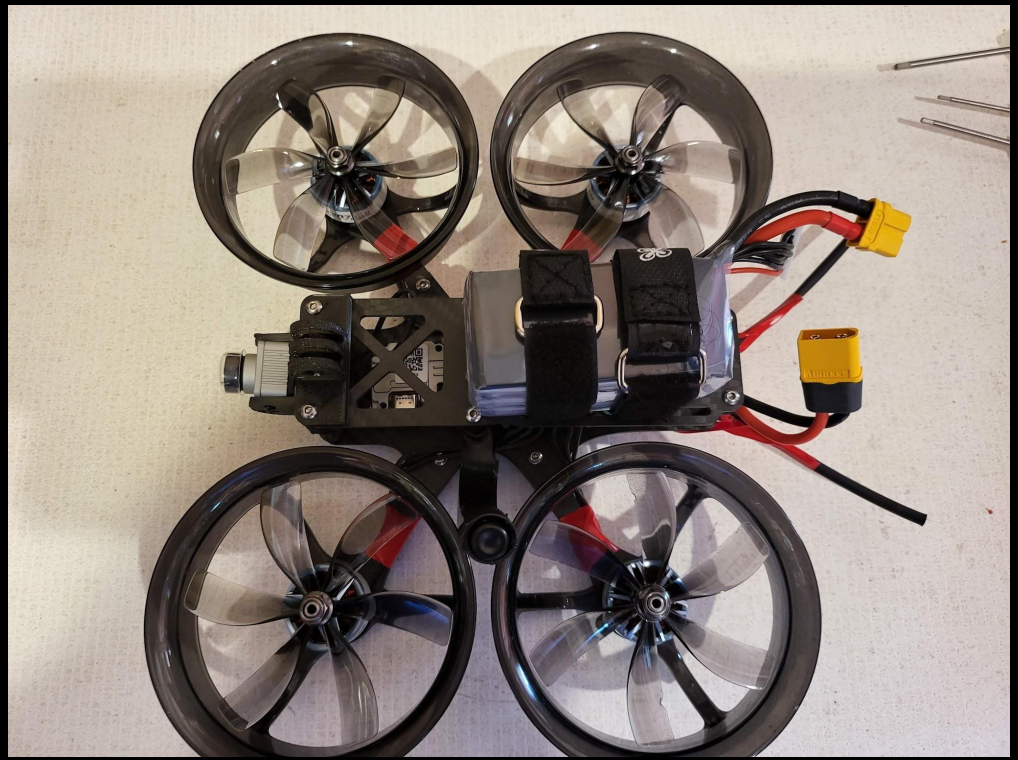
Example:

```
class Derived : private Base {
    using Base::a, Base::b, Base::c;
};
```

Extended using declarations

When to use it:

Maybe with varadic parameter packs or inheriting constructors, but I don’t see this as useful for most code. All the examples I saw had added complexity for only a small benefit.



New Library Components

New Library Componets

Coming “soon”, talk planned for next year in April

Links:

Josuttis: C++ 17 The Complete Guide

https://www.cppstd17.com/

Wikipedia: https://en.wikipedia.org/wiki/C%2B%2B17

Anthony Caldera:
https://github.com/AnthonyCalandra/modern-cpp-features/blob/master/CPP17.md

CPP Reference: https://en.cppreference.com

For Googlers:
https://g3doc.corp.google.com/experimental/users/tkoeppe/g3doc/c++17_changes.html?cl=head

Thanks for listening!

Any questions?