# Highly Opinionated Taste of Modern C++

Brett Viren

September 21, 2021

# Topics

- C++ standards
- Tour of select C++ features
- References

# Disclaimers

I am not a deep C++ expert.

- I reserve the right to say, "I dunno"!

Intentionally neglect some C++ features.

- C++ is a big language, getting bigger, not all is relevant.
- I stick to the context of porting WCP algorithms to WCT.
  - WCT algorithm code should be relatively "linear".
  - Use of elaborate, advanced C++ forms is likely a sign that the algorithm should be factored into smaller WCT components.
  - But, a lot of "smaller" parts of modern C++ is very useful here.
- If I miss some C++ feature you love, we can discuss.
  - Such discussions can go on a long time....

# C++ standards

- WCP @14, WCT @17, *art* @17.
  - Should be able to change WCP to 17, no problem.
- Mostly backwards compatible.
  - I never used what has been dropped.
  - Issues are rare, so don't worry.
  - Rare but happens: FNAL's nominally C++17 compiler doesn't support C++17's `std::filesystem`.

**C++ standards**

| Year | C++ Standard | Informal name |
|---|---|---|
| **1998** | ISO/IEC 14882:1998[29] | C++98 |
| **2003** | ISO/IEC 14882:2003[30] | C++03 |
| **2011** | ISO/IEC 14882:2011[31] | C++11, C++0x |
| **2014** | ISO/IEC 14882:2014[32] | C++14, C++1y |
| **2017** | ISO/IEC 14882:2017[33] | C++17, C++1z |
| **2020** | ISO/IEC 14882:2020[12] | C++20, C++2a |

Wikipedia

# C++ Version Support in Popular Compilers

| | C++14 | C++17 |
|---|---|---|
| GCC | 5+ (e8,e12+) | 8+ (e12+) |
| Clang | 3.4+ (cN+) | 5+ (cN+) |

- FNAL/UPS qualifier in parenthesis, all recent FNAL/UPS builds are C++17.
  - e20 = GCC 9.9, e21 = GCC 10.1, c7 = Clang 7, c8 = Clang 10
- But: C++14 tends to be default, requiring use of `-std=c++17`.

```
$ g++ --version
g++ (Debian 10.2.1-6) 10.2.1 20210110

$ g++ -dM -E -x c++ /dev/null | grep -F __cplusplus
#define __cplusplus 201402L

$ g++ -std=c++17 -dM -E -x c++ /dev/null | grep -F __cplusplus
#define __cplusplus 201703L
```

# Whence C++20, C++23

C++20

- C++20 published Dec 2020.
- Compiler support not 100% yet.
- `modules` and `ranges` likely become important.
- See Kyle Knoepfel's nice overviews: part 1, part 2, part 3
- My guess 1-3 years from now HEP may start to adopt.

C++23

- The proto feature list has some enticing things.
- Which is first: DUNE data or HEP adopting C++23?

# Tour Overview

- Roughly in order of C++ version.
- Describe feature, why it is useful.
- Give an example of usage in WCT.

# C++11

- C++11 is old but I think some may still not be so commonly used.
- The wikipedia page on C++11 is very comprehensive.

# nullptr (C++11)

- A "zero pointer" value that is not of type `int`.
  - Always use `nullptr`, never NULL nor `(void*)0`.
- `nullptr` is convertible to any type of pointer, including "smart" pointers.

```
bool MyFrameSource::operator()(std::shared_ptr<IFrame>& out)
{
    if (end_of_file()) {
        out = nullptr; // WCT uses nullptr to mark EOS
        return false;
    }
    // ...
}
```

# using (C++11)

## Define a *type alias*

- Old style used typedef and was a bit clumsy/confusing.
- New using is more clear (can also be be templated).

```
// from WireCellSio::FrameFileSource
using array_xxfrw = Eigen::Array<float, Eigen::Dynamic,
                                 Eigen::Dynamic, Eigen::RowMajor>;
array_xxfrw arr;
```

## Strip off a namespace

```
using WireCell::units;
double tick = 0.5*units::us;

using WireCell::units::us;
double tick = 0.5*us;
```

- **Never** use "using" in the default namespace in a header file.
  - At your digression in implementation files, just be wary of name collisions.

# `default` / `delete` class methods (C++11)

- Can use keywords `default` or `delete` to define body of special class methods.
- Use where appropriate, no need to always use.
- Use for constructors, destructor

```
struct A {
    A(int x);                       // implicitly, no default constructor
};
struct B {
    B(int x);                       // alone, no default constructor
    B() = default;                  // restore default constructor
    B(const B&) = default;          // default copy constructor
};
struct C {
    C(const C&) = delete;           // no copy construction allowed
    void* operator new(std::size_t) = delete;   // no heap
    void* operator new[](std::size_t) = delete; // allowed
};
```

See more examples: https://en.cppreference.com/w/cpp/language/default_constructor

# Initialization (C++11)

```
// Member initialization
struct MyData {
    int x{0}; float y{0}; std::string z{""}
};
```

- Prefer *member initialization* to initializing in a custom constructor.

```
// Initialization list
int zero{};
MyData md{42, 6.9, "hello world!"}, empty{};
std::vector<MyData> mds{
    {42, 6.9, "hello world!"},
    {0, 0.0, "bye bye"}
};
```

- **Always** initialize in some way!

https://en.cppreference.com/w/cpp/language/list_initialization

https://en.cppreference.com/w/cpp/language/aggregate_initialization

# std::shared_ptr<T> (C++11)

- Safe sharing of data constructed on the heap.
- Holds a reference count on an underlying, shared pointer.
- Object deleted when all shared_ptr<T> destroyed, reference is zero.
- Can hold a nullptr, (re)setting ptr does not change copies.
- WCT uses shared_ptr<IData> in passing all IData to INode calls and shared_ptr<ISomeComponent> when getting from NamedFactory

```
// okay, but not best, we have no "delete" so bad to see a "new":
std::shared_ptr<IFrame> f1(new SimpleFrame(...));

// Niether "new" nor "delete" have we.
std::shared_ptr<IFrame> f2 = std::make_shared<SimpleFrame>(...);
```

See executable example: https://en.cppreference.com/w/cpp/memory/shared_ptr

# Casting std::shared_ptr<T> (C++11,17)

- std::dynamic_pointer_cast() shared equivalent to dynamic_cast()
- Also, static_pointer_cast(), const_pointer_cast() (C++11) and reinterpret_pointer_cast() (C++17)

```
struct A { virtual ~A() = default };
struct B : A { virtual ~B() = default };
// implicit down cast
std::shared_ptr<A> a = std::make_shared<B>();
// explicit up cast
auto b = std::dynamic_pointer_cast<B>(a);
```

See executable example: https://en.cppreference.com/w/cpp/memory/shared_ptr/pointer_cast

# std::weak_ptr<T> (C++11)

- Provide non-owning reference to T held in some std::shared_ptr<T>.
- Creation does not increase the shared reference count.
- Becomes invalid when last std::shared_ptr<T> destructs.
- Must convert to std::shared_ptr<T> to use the pointer.

```
auto sp = std::make_shared<Type>(...);

// perhaps, some other context:
std::weak_ptr<Type> wp = sp;
if (auto sp2 = wp.lock()) {
    // use as shared_ptr
}
else {
    // handle fact of nullptr
}
```

See executable example: https://en.cppreference.com/w/cpp/memory/weak_ptr

# std::unique_ptr<T> (C++11, 14)

- Assured deletion of data in a unique context.
  - Underlying pointer deleted when unique_ptr<T> destroyed.
  - Good to use as class data member.
- Copying unique_ptr<T> transfers underlying pointer.
- unique_ptr<T> comes with C++11, make_unique<T>() with C++14.

```
template<typename T>
std::unique_ptr<T> pass_through(std::unique_ptr<T> p)
{   return p;  }

void func()
{   // Avoid use of explicit "new"
    std::unique_ptr<Type> p = std::make_unique<Type>(...);
    auto p2 = pass_through(std::move(p));
    // p is nullptr after std::move()
}
// underlying pointer deleted as p2 leaves scope
```

See executable example: https://en.cppreference.com/w/cpp/memory/unique_ptr

# auto and variable type deduction (C++11, 14)

- Type deduced through assignment.
- Still take care about const, and value vs reference.

```
std::vector<int> v{1,2,3};
const std::vector<int> cv{1,2,3};
auto a = v[0];                    // copy
auto& b = v[1]; b=22;             // reference
auto& c = cv[2];                  // error

auto val = some_function();       // what is val?
```

Using auto can save programmers time but can also obscure meaning.

- Human must check some_function() definition to learn val type.

# auto and return type deduction (C++11)

- Can replace explicit return type with auto for functions if type can be deduced.
- All returns must be same type, no implicit casting.
- Argument types must still be explicit, or templated
- C++20 will allow auto for argument types.

```
int x=1;                                    // avoid globals...
auto f() { return x; }
auto g() { return 42; }
auto h(int y) { return y; }
template<class T> auto tf(T t) { return t; }
// In C++20 above can be simplified
// auto tf(auto t) { return t; }
```

See more examples https://en.cppreference.com/w/cpp/language/function

# Range based `for` (C++11)

- A simpler syntax for looping over containers.
- Iterators elided and direct types used.
- Use of `auto` makes syntax even simpler.

```
std::vector v{1,2,3};

// mutable reference
for (auto& x : v) { x *= 2; }

// copy
for (auto  x : v) { cout << x << endl; }

// const reference
const auto& cv = v;
for (const auto& x: cv) { ... };
```

See more: https://en.cppreference.com/w/cpp/language/range-for

# `tie` (C++11)

- We can unpack a `pair` to a pair of variables.

```
std::pair<int,int> pi;
int a,b;
std::tie(a,b) = pi;
```

Maybe not super useful compared to what comes next.

# Structured bindings (C++17)

- With `auto` the tie-like unpacking is simpler and more general.

```
// mt becomes a tuple<int,char,float>
auto mt = std::make_tuple(1, 'a', 2.3);
get<0>(mt);                          // ->1
get<1>(mt);                          // ->'a'

// Fully unpack via "structured bindings".
auto  [ i, c, d ] = mt;              // copies
auto& [ i, c, d ] = mt;              // references

// Can unpack structs
struct A { int x{0}, float y{0}; };
A func() { return A{1,2.0}; }

// elsewhere:
auto [a, b] = func();
```

More examples: https://en.cppreference.com/w/cpp/language/structured_binding

# Range based `for` with `map`

- Simpler loop syntax works with other contiainers

```
std::map<int,float> mif;
// as pair<int,float>
for (const auto& p : mif) {
    cout << p.first << " " << p.second << endl;
}
// as unpacked int,float
for (const auto& [f,s] : mif) {
    cout << f << " " << s << endl;
}
```

# Modify container during iteration (don't)

- Like with iterators, a loop body should avoid to modify the container.
- In general, do not add elements to a container while iterating.
- May be safe to add to ordered container before current iteration.
- Best to iterate read-only, fill a second container.

```
std::vector<int> v{1,2,3};
for (auto& x : v) {
    v.push_back(x*10);              // definitely bad, infinite loop!
}
std::map<int,int> m{ {1,1}, {2,2}, {3,3} };
for (auto& [a,b] : m) {
    m[-a] = -b;                     // maybe safe, -a < a
    m[2*a] = 2*b;                   // likely bad, infinite loop!
}
std::map<int,int> m2;
for (const auto& [a,b] : m) {       // safe loop
    m2[a] = b;
    m2[-a] = -b;
    m2[2*a] = 2*b;
}
m = m2;                             // replace original, if needed
```

# std::function (C++11)

Create single functional object type from variety of function-like types.

```
int func(float x) { return x; }
struct CallMe { int operator()(float x) { return x; } };
struct WithMeth { int foo(float x) { return x; } };

using MyFunc = std::function<int (float)>;

MyFunc f1 = [](float) { return 42; };
MyFunc f2 = func;
MyFunc f3 = CallMe{};
WithMeth wm;
MyFunc f4 = std::bind(&WithMeth::foo, wm, _1);
```

- In some contexts, may simply use `auto` instead of explicit `std::function`.
- Use `std::function` in function/method parameter or return type.
  - Example of *type erasure*

See examples: https://en.cppreference.com/w/cpp/utility/functional/function

# Lambda functions (C++11, 14)

An *anonymous functional object* which can *capture* context.

- Direct analog of lambda functions from LISP, Python, etc.

```
[capture](parameters) -> return_type { function_body }
```

```
std::vector<int> v{3,1,2};
std::sort(v.begin(), v.end(),
          [](const int& a, const int& b) -> bool {
              return a > b; // reverse sort
          });
```

Capture:

- `[ & ]` captures all variables by reference
- `[ = ]` captures all variables by value
- `[ x, &y ]` captures just variables `x` by value and `y` by reference

Take care of capture lifetime if lambda leaves local context!

See executable example: https://en.cppreference.com/w/cpp/language/lambda

# More with lambda functions (C++14)

## Generic lambdas

```
auto lambda = [](auto x, auto y) {return x + y;};
```

## Capture expressions (closures)

```
int x = 1;
auto adder = [x](int y){ return x + y; };
adder(4);                                  // -> 5
```

Closures are essentially a class instances. Don't go wild. If you find yourself making complex sets of closures on same data better make an honest `class`!

# Nested namespace (C++17)

A small thing, but write this:

```
namespace WireCell {
    namespace SigProc {
        // ...
    }
}
```

When it is so much cooler to write this:

```
namespace WireCell::SigProc {
    // ...
}
```

Think of all the indenting we now save!

# any (C++17)

A type which can hold an instance of any type and try to get it back again.

```
std::any a = 1;                          // holds an int
int b = std::any_cast<int>(a);           // okay, gets it back
float f = std::any_cast<float>(b); // throws bad_any_cast
```

In WCT we use `boost::any` instead of `std::any`.

- The two are essentially identical.
- There was no `std::any` back when WCT first needed it.
- Used for *type erasure* in passing `IData` in *data flow graph*
- Node/algorithm code sees data after a *type reinstatement*.

Unlikely to need to use `any` in algorithm code, but you'll now know it when you see it.

See https://en.cppreference.com/w/cpp/utility/any and

https://www.boost.org/doc/libs/1_77_0/doc/html/any.html

# Boost - core dependency for WCT (and WCP)

Breeding ground for some C++ standard stuff, WCT uses:

- `boost::any` already mentioned
- `boost::string`, see `WireCellUtil/String.h`, `split()`, `format()`
- `boost::exception`, see `WireCellUtil/Exception.h`, Python-esque.
- Graph lib, `ICluster`, `IndexedGraph`, 3D blob tiling/solving/clustering, DFP graph,
- `boost::iostreams`, bzip/bzip2/tar input/output filters for JSON/Numpy config and data.
- Multidimensional array lib, `SimpleTensor`, `RayGrid`
- `boost::program_options`, `WireCell::Main` class.

Many more useful things in Boost.

- Before reinventing the wheel, browse the list of Boost libraries: `https://www.boost.org/doc/libs/`
- But, not everything there should be preferred over alternatives.
- Be wary of thread-related as there are bigger issues here.

# Some things not covered

- template metaprogramming
- variadic templates
- "move" semantics (showed some for `std::unique_ptr<T>`)
- threading support
- `decltype()`
- `std::atomic`
- `constexpr`
- `std::regex`

# References

There are of course mountains of C++ reference material.

- Wikipedia pages are pretty good for feature overviews C++11, C++14, C++17 (page for C++17 is a bit sparse).
- The C++ Programming Language, 4th ed, Bjarne Stroustrup. The definitive tome but only goes up to C++11. Probably nice to have especially as one begins (I have a very old edition from 20 years ago and it was very good).
- Effective Modern C++, Scott Meyers. Goes up to C++14, includes a lot on "move semantics" and new initialization forms. Well written and rather helpful but may covers things less important for algorithm developers and more for toolkit/framework internals or other large scale systems.
- C++ Reference wiki very well organized, comprehensive and is essentially definitive and includes into C++23. Includes examples that can be directly executed. I go to this every day I work on C++.
- cplusplus.com an older, less comprehensive reference and tutorials. But still useful. A good augmentation to the C++ Reference wiki.