Fermilab | U.S. DEPARTMENT OF ENERGY | Office of Science

C++20 - Part 1

art stakeholder meeting
7 July 2020

C++ standards over the years (1)

- **C++98** – First ISO standard
- **C++03** – Primarily addressed defects of C++98
- **C++11** – **Major shift in coding**
  - Move semantics
  - Automatic type deduction
  - Variadic templates
  - Compile-time computation with `constexpr`
- **C++14** – Minor improvements over C++11
  - Automatic type deduction of function return values
  - Generic lambda expressions

C++ standards over the years (2)

- **C++17** – Moderate changes to language
  - Structured bindings
  - Compile-time conditionals – `if constexpr (...) {}`
  - Guaranteed copy elision (RVO)
  - Class template argument deduction – `std::vector v{1, 2, 3}`
  - Library enhancements such as
    - `std::variant`
    - `std::optional`
    - `std::filesystem`

C++ standards over the years (2)

- **C++17** – Moderate changes to language
  - Structured bindings
  - Compile-time conditionals – `if constexpr (...) {}`
  - Guaranteed copy elision (RVO)
  - Class template argument deduction – `std::vector v{1, 2, 3}`
  - Library enhancements such as
    - `std::variant`
    - `std::optional`
    - `std::filesystem`
- **C++20** – **Major shift in coding**

C++20 – the big four

There are four features that receive the most attention:

- **Modules**
  - Separating declarations and definitions no longer necessary
  - Removes need for `#include` statements
- **Coroutines**
  - Interruptible functions that retain state (akin to Python generators)
- **Concepts**
  - Means of expressing constraints a given type must model
- **Ranges**
  - Library extension that shifts emphasis from iterators to ranges

I will not focus on any of these today–I will discuss smaller features. I plan to go through ranges and concepts in the next few meetings.

Topics covered today

- Mathematical constants
- Container improvements
- Range-based `for` loops with initializers
- Designated initializers

Mathematical constants (1)

How do you calculate the area of a circle?

```
#include <cmath>  // Common, but non-conformant solution

double area_of_circle(double const radius) {
  return M_PI * radius * radius;
}
```

Mathematical constants (1)

How do you calculate the area of a circle?

```cpp
#include <cmath>  // Common, but non-conformant solution

double area_of_circle(double const radius) {
  return M_PI * radius * radius;
}
```

```cpp
#include "TMath.h"  // ROOT-provided solution

double area_of_circle(double const radius) {
  return TMath::Pi() * radius * radius;
}
```

Mathematical constants (1)

How do you calculate the area of a circle?

```cpp
#include <cmath>  // Common, but non-conformant solution

double area_of_circle(double const radius) {
  return M_PI * radius * radius;
}
```

```cpp
#include "TMath.h"  // ROOT-provided solution

double area_of_circle(double const radius) {
  return TMath::Pi() * radius * radius;
}
```

```cpp
#include <numbers>  // As of C++20

double area_of_circle(double const radius) {
  return std::numbers::pi * radius * radius;
}
```

Mathematical constants (2)

Be wary of `using` *directives*:

```
#include <numbers>  // As of C++20

using namespace std::numbers;
void MyModule::produce(art::Event& e) //  Uh oh!
{
  ...
}
```

Mathematical constants (2)

Be wary of `using` *directives*:

```
#include <numbers>  // As of C++20

using namespace std::numbers;
void MyModule::produce(art::Event& e) //  Uh oh!
{
  ...
}
```

The variable `e` is a name corresponding to `std::numbers::e`, or the mathematical constant of `2.718...`

Mathematical constants (2)

Use `using` *declarations* instead:

```
#include <numbers>  // As of C++20

using std::numbers::pi;
void MyModule::produce(art::Event& e) //  Safe
{
  ...
}
```

`std::string` enhancements

Current method of checking for substring at beginning or end of string (e.g.):

```
// Before C++20
std::string s{"It was the best of times"};
bool starts_with_it = s.find("It") == 0;
auto const pos = s.rfind("times");
bool ends_with_times =
  pos != std::string::npos &&
  s.compare(pos, size(s) - strlen("times"), "times") == 0;
```

`std::string` enhancements

Current method of checking for substring at beginning or end of string (e.g.):

```
// Before C++20
std::string s{"It was the best of times"};
bool starts_with_it = s.find("It") == 0;
auto const pos = s.rfind("times");
bool ends_with_times =
  pos != std::string::npos &&
  s.compare(pos, size(s) - strlen("times"), "times") == 0;
```

```
// With C++20
std::string s{"It was the best of times"};
s.starts_with("It");
s.ends_with("times");
```

Checking for inclusion in an unordered container

Current method of checking if a supplied parameter value is allowed (e.g.):

```
namespace {
  std::set allowed_values{"fast", "slow"};
}
```

```
// Before C++20
void validate(std::string const& speed) {
  if (allowed_values.find(speed) != cend(allowed_values)) return;
  throw art::Exception{...};
}
```

Checking for inclusion in an unordered container

Current method of checking if a supplied parameter value is allowed (e.g.):

```
namespace {
  std::set allowed_values{"fast", "slow"};
}
```

```
// Before C++20
void validate(std::string const& speed) {
  if (allowed_values.find(speed) != cend(allowed_values)) return;
  throw art::Exception{...};
}
```

```
// With C++20
void validate(std::string const& speed) {
  if (allowed_values.contains(speed)) return;
  throw art::Exception{...};
}
```

Range-based `for` loops with initializers (1)

Allows for localizing the scope of auxiliary variables:

```
void f() {
  for (size_t i = 0; auto const& hit : make_hits()) {
    g(i, hit);
    ++i;
  }
}
```

Range-based `for` loops with initializers (2)

Avoids dangling references:

```
class Tracks {
  vector<Track> tracks_;
public:
  auto const& data() const { return tracks_; }
}

Tracks make_tracks() { ... }

void f() {
  for (auto const& track : make_tracks().data()) { // Uh oh!
    ...
  }
}
```

Range-based `for` loops with initializers (2)

Avoids dangling references:

```
class Tracks {
  vector<Track> tracks_;
public:
  auto const& data() const { return tracks_; }
}

Tracks make_tracks() { ... }

void f() {
  for (auto const& track : make_tracks().data()) { // Uh oh!
    ...
  }
}
```

The `make_tracks` function returns a temporary object, whose lifetime is *not* extended by the range-based for. This means that `track` **points to invalid memory**.

Range-based `for` loops with initializers (2)

The solution to this problem looks different in C++20:

```
Tracks make_tracks() { ... }

void f() {
  auto const tracks = make_tracks();
  for (auto const& track : tracks.data()) { // Okay
    ...
  }
}

void f_cpp20() {
  for (auto const tracks = make_tracks();
          auto const& track : tracks.data()) { // C++20
    ...
  }
}
```

Designated initializers

C++20 supports designated initializers, which have been supported by the C language since the C99 standard.

```
struct Coordinate { double x{}, y{}, z{0.}; };

// Before C++20
Coordinate c1;                  // (0,0,0)
Coordinate c2{};                // (0,0,0)
Coordinate c3{1};               // (1,0,0)
Coordinate c4{0, 2};            // (0,2,0)

// With C++20
Coordinate c5{.y=2};            // (0,2,0)
Coordinate c6{.y=2, .x=0}; // Ordering error - not allowed
```

Other features not covered

- Calendar and timezone support (via `<chrono>` library)
  - Goodbye `getTimeOfDay`, hello `std::chrono::time_of_day`
- `std::span`
  - Non-owning view into a container
- `std::source_location`
  - Goodbye *many* macros
- etc.

art support for C++20

- There is no C++20-supported build of the *art* stack.
  - ROOT does not plan to support C++20 until the end of 2020.
- The `e21` qualifier has been introduced.
  - GCC 10 build with C++20 enabled
  - No SciSoft products are yet available with this qualifier
- No Clang qualifier yet available with C++20 enabled.

art support for C++20

- There is no C++20-supported build of the *art* stack.
  - ROOT does not plan to support C++20 until the end of 2020.
- The `e21` qualifier has been introduced.
  - GCC 10 build with C++20 enabled
  - No SciSoft products are yet available with this qualifier
- No Clang qualifier yet available with C++20 enabled.

Next time: more `const*` facilities , the `format` library, and ranges.

References

- C++20 features
  - https://en.cppreference.com/w/cpp/20
- C++20 compiler status
  - https://en.cppreference.com/w/Template:cpp/compiler_support/20