Background

❖ C++ evolution defined by ISO committee
❖ Your CERN representative - but really ours: HENP’s!
❖ Just back from feature-freezing C++ meeting
❖ Remember C++98 → C++11?
C++ 20
C++ 2.0
Overview

❖ Process
❖ Big features and where they matter
❖ Little features and where they matter
❖ The Future
❖ Conclusion
Process
Areas of Work

- Evolution in several areas, in parallel. Several just missed C++17. Many major features ended up in C++20.
- Features are proposals or Technical Specifications (TS)
C++ Technical Specifications

- Major work items are sometimes progressing outside the standard:
- TS allows to test-drive
- gain implementation + usage experience before entering the standard
C++ 20 and TSes

❖ Merging
❖ Modules
❖ Co-routines
❖ Concepts
❖ Ranges
❖ Many almost ready for C++17 - and C++20 became the TS treasure chest!
❖ … and there is still more!
Small Features
Selection of small features in C++17, 20

- span
- format
- optional + variant (C++17)
- structured binding (C++17)
- if constexpr (C++17)
- if with variable declaration (C++17)
- (string) literals as template parameters
- more constexpr, consteval
- class template argument deduction
Too much!

- Selected those most relevant for us in HENP
- For the rest: check CppCon + MeetingCpp + CppNow
Selection of small features in C++17, 20

✓ span
✓ format
✓ optional + variant (C++17)
✓ structured binding (C++17)
✓ if constexpr (C++17)
✓ if with variable declaration (C++17)

✓ = covered in this talk!

❖ (string) literals as template parameters
❖ more constexpr, consteval
❖ class template argument deduction
if with variable declaration (C++17)

- Fixes scope of if-condition variables, compared to earlier:

```cpp
if (auto v = f(); !v.get())
  auto v = f();
if (!v.get()) {
  // but I need v only in here...
}```
if with variable declaration (C++17)

Fixes scope of if-condition variables, compared to earlier:

```cpp
if (auto v = f(); !v.get())
```

```cpp
auto v = f();
if (!v.get()) {
    // but I need v only in here...
}
```
if constexpr

```cpp
template <class T>
void *begin_if(T& v) {
    if constexpr (is_container<T>)
        return &*v.begin();
    return nullptr;
}
```

- Does not compile the branch if false
- Valid code for T being bool despite v.begin()
Structured Binding

```cpp
for (auto && [k, v] : myMap) {
    cout << "key: " << k
    << "val: " << v << '\n';
}
```

- Great way of “receiving” multiple (2, 3, 4,...) struct members or tuple<> elements
- Handle multiple values being passed in for or if statements
optional, variant

- `std::optional`: holds one or none
- `std::variant`: holds one of a set (union)
- C++20 will possibly also have `std::expected`: value or error
- Extremely powerful for writing safe, compact code

```
variant<double, string> v{17.};
assert(get<double>(v) > 16);
v.emplace<string>("ABC");
cout << get<1>(v); // good!
get<0>(v); // throws!
```
**span**

- Whether std::vector, std::array or C-style array
- Refers to a contiguous array of given size
- Wonderful as function parameter
- Fixed-size or runtime-size

```cpp
void func(span<double,4> lv);
array<double, 4> jetLV{...};
func(jetLV);
```
string message = format("The answer is {}.", 42);

- Not yet guaranteed for C++20 - but expected!
- An efficient and nice way to format strings in C++, finally
- printf-format plus so much more
- Incl. user-extensible: format your classes
Big Features
BIG FEATURES
Big C++ features since C++14

✓ = covered in this talk!

✓ Contracts
✓ Concepts
✓ Ranges
✓ Modules
✓ <=>

❖ Coroutines
❖ std::filesystem (C++17)
Contracts (1/3): Intro

- Specify what your function expects
- Can be checked by compiler
- Check can be turned off
- Also enables optimizations
Contracts (2/3): Setting Expectations

- expects (or “pre”?): condition on arguments
- ensures (or “post”): post-condition, a guarantee by the function
- assert: a check to be performed within the function

```cpp
int f(int i) [[expects: i > 0]];  
int g(string& s) [[ensures: !s.empty()]] {  
    [[assert: s.empty()]]  
    s = "ABC";
}
```
Contracts (3/3): Validating Expectations

- Contract levels default, audit, axiom; compiler flag selects what to check:
  - --default: checks default
  - --off: nothing
  - --audit: default and audit
- axiom is thus never checked: good for optimizer hints; expensive checks can be audit

```c
int f(int i) [[default pre: i > 0]];  
int g(int i) [[audit pre: i > 0]];    
int h(int i) [[axiom pre: i > 0]];   
```
Concepts

- Document and restrict template parameters
- Better error messages
- User-Oriented feature for library authors, i.e. us!

```cpp
template <ConvertibleTo<string> T>
class DoesSomethingWithAString;
```
Ranges

- Generalized iterators working on everything that has a begin
- Much, much nicer syntax
- More than just syntax: a way of writing algorithms without mentioning data!

```c++
if (auto evens = vec | view::filter(is_even)) {
  // Do something with an even number.
}
```

- Note use of | to pipe into filter; use of range as boolean expression
Modules

- Dramatic build time reduction
- Hides implementation details, similar to header / source
- See “Migrating large codebases to C++ Modules" by ROOT team’s Yuka Takahashi on Wednesday, 19:00, Track 1!
Spaceship $\leftrightarrow$

- finally a default comparison!
- reduces code clutter and bugs

```
class A { int i; }
bool operator==(A, A) {...}
bool operator!=(A, A) {...}
bool operator>(A, A) {...}
bool operator<(A, A) {...}
bool operator>=(A, A) {...}
bool operator<=(A, A) {...}
```
Spaceship $<=>$

- finally a default comparison!
- reduces code clutter and bugs

```cpp
class A { int i; };
bool operator==(A, A) {...}
bool operator!=(A, A) {...}
bool operator>(A, A) {...}
bool operator<(A, A) {...}
bool operator>=(A, A) {...}
bool operator<=(A, A) {...}

class A {
    auto operator<=>(A) const = default;
    int i;
};
```
C++20 will change how we write code

Goals are simplicity, 0-cost, faster programs, common features in the library

Implementations are on their way, most of C++17 already available
C++23 will be much smaller
Implementing language features for library where needed: stdlib modules, more concepts
Several major features did not make it: networking, executors
More features in the works, e.g. reflection
Your C++ Representative

- (disclaimer: not my main job)
- HENP has > 100 million lines of C++ code
  - Move away from C++?
  - But replacing 100MLOC might be more work than improving C++!
- I will be at ACAT all week: share your ideas, pet peeves, thoughts about C++'s future role in HENP codes
Conclusion (1/3)

❖ C++ learned a lesson: evolve or be dead
❖ Relevant to us: better, more maintainable code
❖ We won't need all features, but we have a palette to select from
Many features are targeted at us: math special functions (C++17), ranges, concurrency, compile times of large-scale code.

We should benefit from what the language and its tools provide.

We need to evolve tooling and code to benefit.

Need to upgrade our coding guidelines, selecting “allowed” features: contracts? concepts? coroutines?
Are we investing enough in our code and education, given C++’s pace?