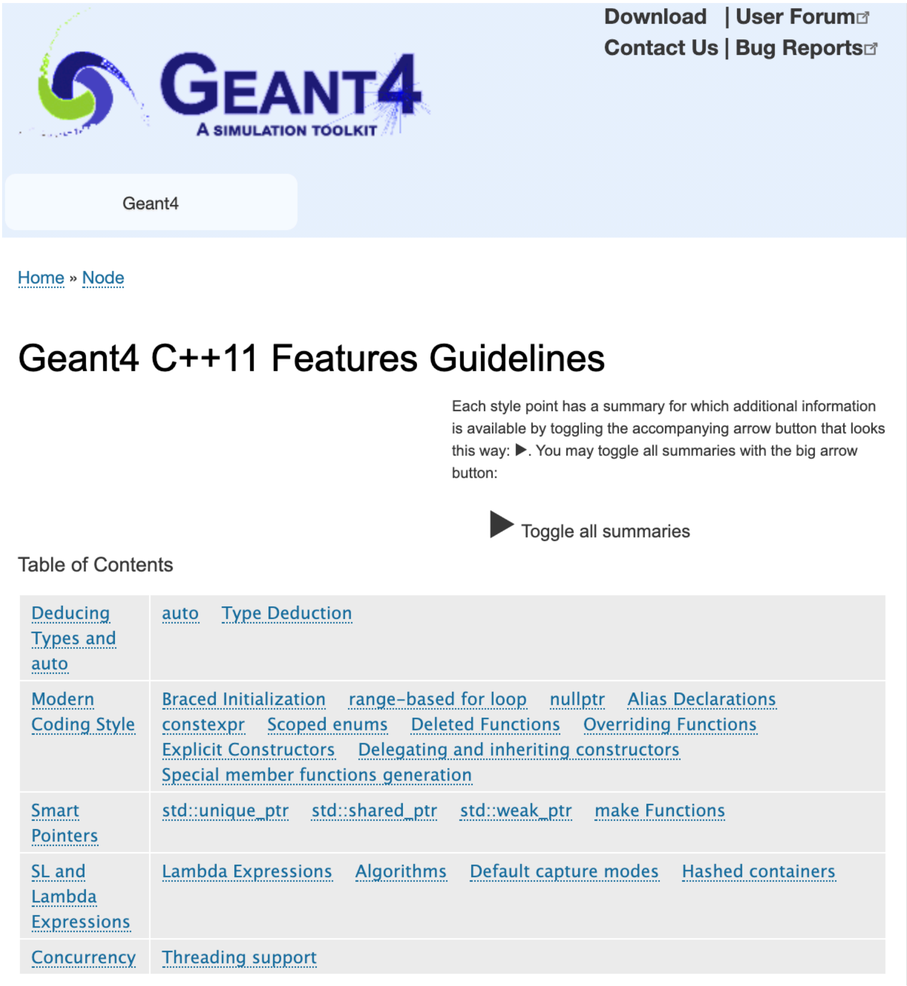
C++17 Mini Tutorial

I. Hrivnacova, IJCLab Orsay (CNRS/IN2P3)

Virtual Geant4 Seminar,
22 June 2021

Introduction

- The guidelines for use of the C++11 features relevant to Geant4 code development and tutorial slides
  - presented at the Geant4 Collaboration Meeting in Fermilab, 2015
  - see C++11 Guidelines, tutorial slides
- This mini course
  - Tutorial slides for C++17, plus some C++14
  - The update of the guidelines will follow



Download | User Forum
Contact Us | Bug Reports

GEANT4 A SIMULATION TOOLKIT

Geant4

Home » Node

Geant4 C++11 Features Guidelines

Each style point has a summary for which additional information is available by toggling the accompanying arrow button that looks this way: ▶. You may toggle all summaries with the big arrow button:

▶ Toggle all summaries

Table of Contents

Deducing Types and auto	auto Type Deduction
Modern Coding Style	Braced Initialization range-based for loop nullptr Alias Declarations constexpr Scoped enums Deleted Functions Overriding Functions Explicit Constructors Delegating and inheriting constructors Special member functions generation
Smart Pointers	std::unique_ptr std::shared_ptr std::weak_ptr make Functions
SL and Lambda Expressions	Lambda Expressions Algorithms Default capture modes Hashed containers
Concurrency	Threading support

C++ Standards

So far, six revisions of the C++ standard:

- **C++98:** in 1998, the ISO working group standardized C++ for the first time as ISO/IEC 14882.
  - In 2003, it published a new version of the C++ standard called ISO/IEC 14882:2003, which fixed problems identified in C++98.
- **C++11:** in 2011, a **major revision** of the standard 14882:2011 (was informally referred to as “C++0x”) included many additions to both the core language and the standard library
- **C++14**: in 2014, released as **a small extension to C++11**, featuring mainly bug fixes and small improvements (also known as C++1y)
- **C++17**: in 2017, a **major revision** 14882:2017 (informally known as C++1z)
- **C++20**: in 2020, a **new major revision** 14882:2020 adds more new major features

and currently working on the next revision, C++23.

About This Mini-Course

- Compiled from the following sources:
  - C++17 - The Complete Guide, First Edition by Nicolai M. Josuttis. Leanpub book, version published on 2020-09-26.
  - stackoverflow.com
  - isocpp.org/wiki
  - en.cppreference.com
  - www.codingame.com/playgrounds
  - www.learncpp.com/cpp-tutorial
  - arne-mertz.de

- Not all important features could be covered in this tutorial, some of features are presented as bare introduction

Basic Language Features

Return type deduction (C++14)
Structured Bindings
if and switch with Initialization
Inline Variables
Lambda Extensions
New Attributes and Attribute Features
Preprocessor Condition __has_include
Relaxed Enum Initialization from Integral Values (Backup Slide)

Return Type Deduction (C++14)

- Trailing return type (C++11)

```
int f1(double, int);

// C++11
auto f2(double, int) -> int;

// makes possible
template < typename T1, typename T2 >
auto Foo(T1 t1, T2 t2) -> decltype(T1*T2)
{
  ...
}
```

- Return type deduction (C++14)

```
int foo();
int bar();
double dbar();

auto f3() { return foo() * 42; }
  // ok, deduces int

// Multiple return supported
auto g(bool b) {
  // with arbitrary control flow
  if( b ) {
    return foo() * 42;
   }
   return bar();      // OK
   // return dbar(); // ERROR: types do not match
}

template < typename T1, typename T2 >
auto Foo(T1 t1, T2 t2)
{ … }
```

Return Type Deduction - 2

- Compiler can deduce return types for any function, no matter how complex
  - The only condition is that **each return statement must have the exact same type**.
- Restricted to inline functions, function templates and helper functions that are used only **inside a single translation unit.**
  - Not applicable for a “normal” function that is declared in a header but implemented elsewhere
- Using return type deduction is **encouraged where technically possible**.
  - Can help to make the types you use more consistent

- Example of use in a class

```
class HasAContainer {
  typedef std::vector<int> container_t;

 public:
  auto begin() const {
    return std::begin(values);
  }
  auto itemAt(container_t::size_type index) const {
    return values[index];
  }
  //...
 private:
  container_t values;
};
```

Structured Bindings

- Allow you to initialize multiple entities with the elements or members of an object

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

MyStruct ms;

// C++17
auto [u, v] = ms;
// Each syntax for
initializations is supported
auto [u2, v2] {ms};
auto [u3, v3] (ms);
```

```
MyStruct getStruct() {
  return MyStruct{42, "hello"};
}

// You can assign the result to two entities
// directly giving local names to the returned data
members
auto[id, val] = getStruct();

// Here, id and val are names for the members i and s
of the returned structure. They have the corresponding
types, int and std::string, and can be used as two
different objects:
if (id > 30) {
  std::cout << val;
}
```

Structured Bindings - 2

- Iterate over the elements of a std::map<>

```
// BEFORE C++17
for (const auto& elem : mymap) {
   std::cout << elem.first << ": " << elem.second << '\n';
}
```

```
// C++17
for (const auto& [key, val] : mymap) {
   std::cout << key << ": " << val << '\n';
}
```

Structured Bindings - 3

Can be used:

- For **structures and classes** where all non-static data members are public, you can bind each non-static data member to exactly one name.
- For **raw arrays,** you can bind a name to each element of the array.
- For any type, with a suitable **tuple-like API.**
  - **`std::pair<>`**, **`std::tuple<>`**, and **`std::array<>`** are examples of types that provide such API.

`if` and `switch` with Initialization

- An initialization clause can be now specified before the usual condition or selection clause in the `if` and `switch` control structures

```
// BEFORE C++17
auto val = GetValue();
if (condition(val))
  // on success
else
  // on false...
```

```
// C++17
if (auto val = GetValue(); condition(val)) {
    // on success
} else {
    // on false...
}
```

- *The `val` value is valid for the whole `if` statement (including the optional else part)*
- *The scope of `val` is limited to the `if` statement only*

`if` and `switch` with Initialization - 2

- An example where used with structured binding

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

- *`map::insert` returns a pair consisting of an iterator to the inserted element (or to the element that prevented the insertion) and a `bool` denoting whether the insertion took place*

Inline Variables - Motivation

- For portability and ease of integration
- Limitations up to C++17:
  - no global variables/objects in a library entirely contained in header files
  - non-const static member inside the class structure is not allowed
  - a variable defined outside the class structure is also an error if part of a header file included by multiple CPP files

```
class MyClass {
  static std::string msg{"OK"};
    // compile-time ERROR
  …
};
```

```
class MyClass {
  static std::string msg;
...
};

std::string MyClass::msg{"OK"};
   // Link ERROR if included by multiple CPP files
```

Inline Variables

- Since C++17, you can define a variable/object in a header file as `inline`:

```
class MyClass {
   inline static std::string msg{"OK"}; // OK since C++17
   ...
};

inline MyClass myGlobalObj;
    // OK even if included/defined by multiple CPP files
```

- The same semantics as a function declared inline:
  - It can be defined in multiple translation units, provided all definitions are identical.
  - It must be defined in every translation unit in which it is used.

Inline Variables - 2

- For static data members, `constexpr` implies `inline` now
- Since C++17, the following declaration defines the static data member n:

```
struct D {
   static constexpr int n = 5;
     // C++11/C++14: declaration
     // since C++17: definition
};

// That is, it is the same as
struct D {
   inline static constexpr int n = 5;
};
```

- By using `thread_local` you can also make an inline variable unique for each thread:

```
struct ThreadData {
   inline static thread_local std::string name;
      // unique name per thread
   ...
};

inline thread_local std::vector<std::string> cache;
      // one cache per thread
```

Lambda Extensions

- Lambdas were introduced with C++11
- Generic lambdas were introduced with C++14
  - Lambda function parameters can be `auto` to let the compiler deduce the type.
- C++17 allows the use of lambdas in even more places:
  - In constant expressions (i.e., at compile time)
  - In places where you need a copy of the current object (e.g. when calling lambdas in threads)
    - Not explained in this tutorial

Generic Lambdas (C++14)

- C++11: you have to state the parameter type

```
// C++11
auto size = [](const unordered_map<wstring, vector<string>>& m) { return m.size(); };
```

- C++14: lambda function parameters can be `auto` to let the compiler deduce the type
  - generic lambda that will work with any suitable type and just do the right thing

```
// C++14: new expressive power
auto size = [](const auto& m) { return m.size(); };
```

- More examples in a backup slide

constexpr Lambdas (C++17)

- Example: using the result of calling a lambda as a compile-time argument for the declaration of the size of a `std::array<>`

```
// implicitly constexpr since C++17
auto squared = [](auto val) {
   return val*val;
};

std::array<int,squared(5)> a;
   // OK since C++17 => std::array<int,25>
```

- Using features that are not allowed in `constexpr` contexts disables this ability
- To find out at compile time whether a lambda is valid for a compile-time context, you can declare it as `constexpr`:

```
// OK since C++17
auto squared3 = [](auto val) constexpr {
   return val*val;
};
```

- More examples in a backup slide

New Attributes and Attribute Features

- Attributes = formal annotations that enable or disable warnings
  - Since C++11
  - For example: `[[noreturn]]`(C++11), `[[deprecated]]`(C++14)
- New attributes with C++17:
  - `[[nodiscard]]`, `[[maybe_unused]]`, `[[fallthrough]]` (see a backup slide for the last one)
- Attributes can now be also used in a few more places and with some additional convenience
  - They can be used to mark namespaces, enumerators (values of enumeration types)

Attribute `[[nodiscard]]`

- To encourage warnings by the compiler if a return value of a function is not used
  - However, the compiler is not required to issue a warning
- Can be used to signal misbehaviour when return values are not used:
  - **Memory leaks**, such as not using returned allocated memory
  - **Unexpected or non-intuitive behavior** such as getting different/unexpected behavior when not using the return value
  - …
- An example is the member function `empty()`, which checks whether an object (container/string) has no elements. Programmers pretty often call this function to "empty" the container (remove all elements):

```
cont.empty();
```

- This can often be detected using the new attribute:

```
class MyContainer {
  ...
  public:
    [[nodiscard]] bool empty() const;
};
```

Attribute `[[maybe_unused]]`

- Can be used to avoid warnings by the compiler for not using a name or entity.
- The attribute may be applied to:
  - the declaration of a class
  - a type definition with typedef or using
  - a variable
  - a non-static data member
  - a function
  - an enumeration type
  - an enumerator (enumeration value).

- One application is to name a parameter without (necessarily) using it:

```
void foo(int val,
         [[maybe_unused]] std::string msg)
{
#ifdef DEBUG
   log(msg);
#endif
   ...
}
```

Preprocessor Condition `__has_include`

- C++17 extends the preprocessor to allow you to check whether a specific header file could be included
  - The conditions inside `__has_include(...)` evaluate to 1 `(true)` if a corresponding `#include` command would be valid
  - Nothing else matters (e.g., the answer does not depend on whether the file was already included).

```
#if __has_include(<filesystem>)
#  include <filesystem>
#  define HAS_FILESYSTEM 1
#elif __has_include(<experimental/filesystem>)
#  include <experimental/filesystem>
#  define HAS_FILESYSTEM 1
#  define FILESYSTEM_IS_EXPERIMENTAL 1
#elif __has_include("filesystem.hpp")
#  include "filesystem.hpp"
#  define HAS_FILESYSTEM 1
#  define FILESYSTEM_IS_EXPERIMENTAL 1
#else
# define HAS_FILESYSTEM 0
#endif
```

Template Features

Class Template Argument Deduction
Fold Expressions
Extended Using Declarations

Class Template Argument Deduction (CTAD)

- Before C++17, you always had to explicitly specify all template arguments for class templates.
- For example:

```
std::complex<double> c{5.1,3.3};
std::mutex mx;
std::lock_guard<std::mutex> lg(mx);
```

  - the `double` or `std::mutex` cannot be omitted

- Since C++17, by using class template argument deduction (CTAD), you can omit explicit definition of the template arguments if the constructor is able to deduce all template parameters:

```
std::complex c{5.1,3.3};
std::mutex mx;
std::lock_guard lg{mx};
```

- You can even let containers deduce element types:

```
std::vector v1 {1, 2, 3};
std::vector v2 {“hello”, “world”};
```

Using CTAD

- Whenever the arguments passed to a constructor can be used to deduce the class template parameters

```
// Before C++17
std::complex<double> c1{1.1, 2.2};

std::tuple<int, char, std::nullptr_t>
  t{42, 'x', nullptr};
```

```
// C++17
std::complex c1{1.1, 2.2};
std::complex c5{5,3.3};
  // ERROR: attempts to int
  // and double as T
std::tuple t{42, 'x', nullptr};
```

- We can get rid of several convenience function templates:

```
// Before C++17
std::vector<int> v;
std::pair<typename std::vector<int>::iterator,
          typename std::vector<int>::iterator>
  p(v.begin(), v.end());
// Could be simplified with
auto p = std::make_pair(v.begin(), v.end());
```

```
// C++17
std::vector<int> v;
std::pair p(v.begin(),
            v.end());
```

No Partial CTAD

```
template<typename T1, typename T2, typename T3 = T2>
class C {
 public:
   C (T1 x = {}, T2 y = {}, T3 z = {}) {...}
   ...
};
```

Note that the third template parameter
has a default value.

```
// All deduced
C c1(22, 44.3, "hi"); // OK
C c2(22, 44.3);       // OK
C c3("hi", "guy");    // OK
```

```
// All specified
C<string, string, int> c7;            // OK
C<int, string> c8(52, "my");          // OK
C<string, string> c9("a", "b", "c"); // OK
```

```
// Only some deduced:
C<string> c4("hi", "my"); // ERROR: only T1 explicitly defined
C<> c5(22, 44.3);         // ERROR: neither T1 not T2 explicitly defined
C<> c6(22, 44.3, 42);     // ERROR: neither T1 nor T2 explicitly defined
```

More on CTAD

- Specific **deduction guides** can be defined to provide additional class template argument deductions or fix existing deductions defined by constructors
- Deduction guides can be used to force “decay”
  - In general, even outside templates, arguments passed by value decay, while arguments passed by reference do not decay. Decay means that raw arrays convert to pointers and that top-level qualifiers, such as const, and references are ignored.
- Deduction guides do not have to be templates and do not have to apply to constructors.
- The C++ standard library provides **deduction guides for pairs and tuples, arrays, for containers deduction from iterators**
- No Deduction Guides for Smart Pointers
  - Ambiguity when the type of a pointer to one object and an array of objects have or decay to the same type

Fold Expressions

- A feature to compute the result of using a binary operator over all the arguments of a parameter pack (variadic template)

```
template<typename... T>
auto foldSum (T... args) {
  return (... + args);
    // ((arg1 + arg2) + arg3) ...
}
```

```
foldSum(47, 11, val, -1);
→ return 47 + 11 + val + -1;
```

```
foldSum(std::string("hello"), "world", "!");
→ return std::string("hello") + "world" + "!";
```

- `(... + args)` – “post-adds”:
  `=> (arg1 + arg2) + arg3) …`
- `(args + ...)` - “pre-adds”:
  `=> (arg1 + (arg2 + arg3))…`

Extended Using Declarations

- Using declarations were extended to allow a comma-separated list of names

```
class Base {
  public:
    void a();
    void b();
    void c();
};
```

```
class Derived : private Base {
  public:
    // Before C++17
    using Base::a;
    using Base::b;
    using Base::c;
};
```

- Using declarations can also apply to parameter packs
- Can be used in generic meta programming to derive all operations of the same kind from a variadic list of base classes

```
class Derived : private Base {
  public:
    // C++17
    using Base::a, Base::b, Base::c;
};
```

New Library Components

std::make_unique (C++14)
std::optional<>
std::variant<>
std::any
String Views
The Filesystem Library

std::make_unique (C++14)

- Smart pointers since C++11
- Presented in Geant4 C++11 Guidelines tutorial
  - With a work-around for `std::make_unique`, only part of C++14.

make Functions

- *Prefer* `std::make_unique`*(*) and* `std::make_shared` *to direct use of new.*
  - Compared to direct use of new, make functions eliminate source code duplication, improve exception safety and (some) make code faster

```
{
  std::unique_ptr<Widget> upw(new Widget>);
  std::shared_ptr<Widget> spw(new Widget>);
}
```

  - Without make function (previously was in green, now in red)

```
{
  auto upw1(std::make_unique<Widget>());
  auto upw1(std::make_shared<Widget>());
}
```

  - With make function (Widget type is not duplicated)

20th Geant4 Collaboration Meeting, 30 September 2015, Fermilab 39

std::optional<>

- Defines an object of a certain type with an additional Boolean member/flag that signals whether a value exists in a type-safe way.
- Simulate semantics similar to pointers, where we can express having no value by using nullptr.

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

std::optional<std::string> create(bool b) {
  if (b) {
    return "Godzilla";
  }
  return std::nullopt;
}

int main() {
  for (auto v : {true, false}) {
    std::cout
      << "create(" << std::boolalpha << v << ") returned "
      << create(v).value_or("empty") << '\n';
   }
}
```

```
Output:
create(true) returned Godzilla
create(false) returned empty
```

`std::optional<>`

Types

- Header file: **`<optional>`**
- Defined classes, objects:
  - `namespace std {`

    `template<typename T> class optional;`

    `}`
  - `nullopt` of type `std::nullopt_t` as a “value” for optional objects that have no value
  - Exception class `std::bad_optional_access`, derived from `std::exception`, for value access without a value.
- Operations overview in backup slides

Optional Data Members

```
#include <optional>
#include <iostream>
#include <vector>

struct Name
{
   std::string first;
   std::optional<std::string> middle;
   std::string last;
};

std::ostream& operator << (
  std::ostream& strm, const Name& n) {
  strm << n.first << ' ';
  if (n.middle) {
    strm << *n.middle << ' ';
  }
  return strm << n.last;
}
```

```
int main()
{
   Name n{"Jim", std::nullopt, "Knopf"};
   std::cout << n << '\n';

   Name m{"Donald", "Ervin", "Knuth"};
   std::cout << m << '\n';
}
```

```
Output:
Jim Knopf
Donald Ervin Knuth
```

- See also examples of construction, accessing value and comparison in backup slides

`std::variant<>`

- A `std::variant<>` holds a value of various *alternatives*, which usually have different types
- Adopted from C unions, without the drawbacks of the C language
- A closed discriminated *union* with a specified list of possible types
  - Where the type of the current value is always known
  - That can hold values of any specified type
  - That you can derive from
- Has internal memory for the maximum size of the underlying types plus a fixed overhead to manage which alternative is used.
  - No heap memory is allocated
- Enables a new form of polymorphism and dealing with heterogeneous collections - not covered here.

std::variant<> - 2

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

int main()
{
  std::variant<int, std::string> var{"hi"};
  std::cout << var.index() << '\n'; // prints 1
  var = 42;
  std::cout << var.index() << '\n'; // prints 0
  try {
    int i = std::get<0>(var);
    std::string s = std::get<std::string>(var);
 }
  catch (const std::bad_variant_access& e) {
    std::cerr << "EXCEPTION: " << e.what() << '\n';
  }
}
```

- Represents a type-safe union.

Initialized with string alternative

Now holds int alternative

Access by index

Access by type (throws exception here)

In case a wrong type/index is used

```
Output:
1
0
EXCEPTION: bad_variant_access
```

Types

- Header file: **`<variant>`**
- Defined classes, objects:
  - `namespace std {`

    `template<typename Types...> class variant;;`

    `}`
    - A *variadic* class template (a feature introduced with C++11 that allows you to deal with an arbitrary number of types)
  - Other class templates: `std::variant_size, std::variant_alternative,` value `std::variant_npos`
  - Type `std::monostate`
  - Exception class `std::bad_variant_access`, derived from `std::exception`.
- Operations overview and more examples (accessing and changing value, comparison) in backup slides

Construction

```
std::variant<int, int, std::string> v1;
```

The default constructor calls the default constructor of the first alternative. The alternative is value initialized: it is `0`, `false`, or `nullptr` for fundamental types.
`v1` sets first `int` to `0`, `index()==0`

```
std::variant<long, int> v2{42};
std::cout << v2.index() << '\n';
```

If a value is passed for initialization, the best matching type is used:
v2.index() *prints 1*

```
std::variant<long, long> v3{42};          // ERROR
std::variant<int, float> v4{42.3};        // ERROR
std::variant<int, long double> v6{42.3}; // ERROR
```

The call is ambiguous if two types match equally well

- Class template argument deduction cannot be used.
- There is **no** `make_variant<>()` convenience function (unlike for `std::optional<>` and `std::any`).
  - Neither make sense, because the whole goal of a variant is to deal with multiple alternatives.

std::monostate

- Can serve as a first alternative type to make the variant type default constructible, where the first type has no default constructor.

```
struct NoDefConstr {
   NoDefConstr(int i) {
     std::cout << "NoDefConstr::NoDefConstr(int) called\n";
   }
};
std::variant<NoDefConstr, int> v1; // ERROR: cannot default construct first type

std::variant<std::monostate, NoDefConstr, int> v2; // OK
std::cout << "index: " << v2.index() << '\n';      // prints 0
```

- Objects of type `std::monostate` always have the same state. Thus, they always compare to be equal.
- Their own purpose is to represent an alternative type so that the variant has *no value of any other type.*

`std::any`

- In general, value objects in C++ are declared to have a specific type that cannot be changed
- `std::any` is a value type that can change its type while still having type safety
  - Objects can hold values of any arbitrary type but they know which type the value has that they currently hold
  - No need to specify the possible types when declaring an object of this type
- Objects contain both the contained value and the type of the contained value
- To use the current value with its type, an `any_cast<>` is necessary

std::any - 2

```
std::any a;
std::any b = 4.3;
a = 42;
b = std::string{"hi"};

if (a.type() == typeid(std::string)) {
  std::string s = std::any_cast<std::string>(a);
  useString(s);
}
else if (a.type() == typeid(int)) {
  useInt(std::any_cast<int>(a));
}
```

- *a is empty*
- *b has value 4.3 of type `double`*
- *a has value 42 of type `int`*
- *b has value "hi" of type `std::string`*

Types

- Header file: **`<any>`**
- Defined classes
  - `namespace std {`

    `class any;`

    `}`
    - Not a class template at all
  - Exception class `std::bad_any_cast`, which is derived from `std::bad_cast`, which is derived from `std::exception` if type conversions fail
- Operations overview and more examples (construction, accessing and changing value, comparison) in backup slides

String Views Motivation

- Consider the following example:

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

int main()
{
  char text[]{ "hello" };
  std::string str{ text };
  std::string more{ str };

  std::cout << text << ' '
            << str << ' '
            << more << '\n';
}
```

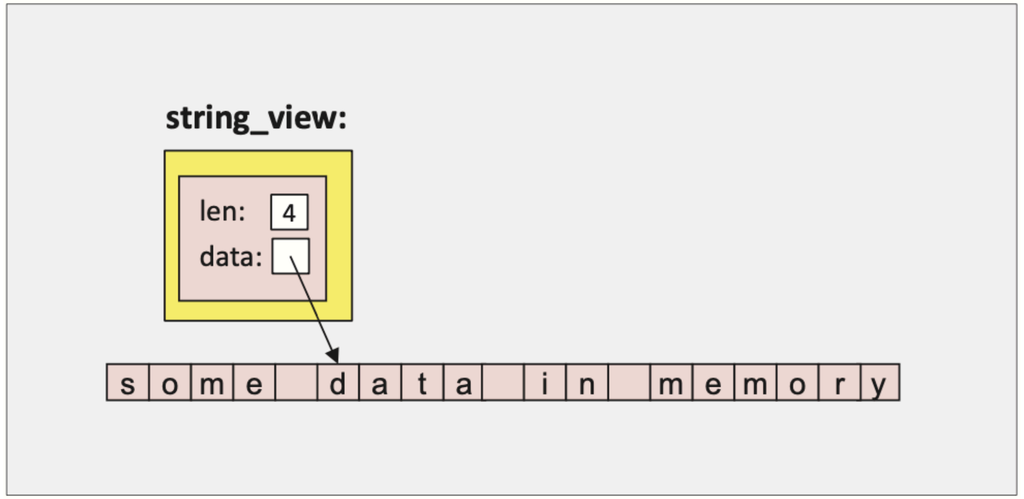
```
Output:
hello hello hello
```

Internally, `main` copies the string `"hello"` 3 times, resulting in **4 copies**:

- The string literal `"hello"`, which is known at compile-time and stored in the binary
- One copy is created when we create the `char[]`
- Two `std::string` objects create one copy of the string each.

String Views

- `string_view` provides a view of a string that is defined elsewhere
- It allows us to deal with character sequences like strings without allocating memory for them
  - Refer to external character sequences without owning them
- Using such a string view is **cheap and fast**
- It has many of the functions that we know from `std::string`



String Views - 2

- We can re-write the previous code to use `std::string_view`:

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

int main()
{
  std::string_view text{ "hello" };
  std::string_view str{ text };
  std::string_view more{ str };

  std::cout << text << ' '
            << str << ' '
            << more << '\n';
}
```

```
Output:
hello hello hello
```

- Unlike `std::string`, which keeps its own copy of the string, **`std::string_view` provides a view of a string that is defined elsewhere**.
- The output is the same, but **no more copies of the string "hello" are created**.
  - `text` is only a view onto the string `"hello"`, not allocated at run-time, so no copy has to be created.
  - A copy of a `std::string_view` observes the same string as the copied-from `std::string_view` is observing.
- This means that neither `str` nor `more` create any copies.

String Views - 3

- `std::string_view` is not only fast, but has many of the functions that we know from `std::string`

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

int main()
{
  std::string_view str{ "Trains are fast!" };

  std::cout << str.length() << '\n';                    // 16
  std::cout << str.substr(0, str.find(' ')) << '\n'; // Trains
  std::cout << (str == "Trains are fast!") << '\n';  // 1
}
```

String Views - 4

- Because `std::string_view` doesn't create a copy of the string, if we change the viewed string, the changes are reflected in the `std::string_view`:

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

int main()
{
  char arr[]{ "Gold" };
  std::string_view str{ arr };

  std::cout << str << '\n'; // Gold

  // Change 'd' to 'f' in arr
  arr[3] = 'f';

  std::cout << str << '\n'; // Golf
}
```

- When you use a string view, it's best to avoid modifications to the underlying string for the remainder of the string view's life to prevent confusion and errors.
- `string_view` is **potentially dangerous**
  - It is up to the programmer to ensure that the referred character sequence is still valid when using a string_view

String View Types

- Header file: **`<string_view>`**
- A couple of specializations of class `basic_string_view<>`:
  - Class `std::string_view` = specialization for characters of type `char`:

```
namespace std {
  using string_view = basic_string_view<char>;
}
```

  - For strings that use wider character sets, such as Unicode or some Asian character sets:

```
namespace std {
  using u16string_view = basic_string_view<char16_t>;
  using u32string_view = basic_string_view<char32_t>;
  using wstring_view = basic_string_view<wchar_t>;
}
```

- Operations overview in backup slides

Differences Compared to std::string

- The underlying character sequence is **read-only**.
  - There is no operation that enables modification of the characters. You can only assign a new value, swap values, and shrink the view to a subset of the character sequence.
- The character sequence is **not guaranteed to be null terminated**.
- The value returned by `data()` **can be `nullptr`**. For example, `nullptr` is returned after initializing a string view with the default constructor.
  - Due to these you should always use `size()` before accessing characters via `operator[]` or `data()` (unless you know better).
- There is **no allocator support**.

Safe Use of String Views

- Use string views **instead of C-style strings**
- Prefer string views over std::string **for read-only strings, unless you already have a std::string**
- **Use `std::string_view` with care**
  - Make sure that the underlying string viewed with a `std::string_view` does not go out of scope and isn't modified while using the `std::string_view`
  - Using `std::string_view` of a non-null-terminated string can cause undefined behavior (eg. call to `data()`)

Safe Use of String Views - 2

- **Do not:**
  - Do not use string views in APIs that pass the argument to a `string`.
  - Do not initialize `string` members from string view parameters
  - Do not return a string view [unless it is just a forwarded input argument or you signal the danger by, for example, naming the function accordingly]
  - Never use a returned value to initialize a string view

The Filesystem Library

- With C++17, the Boost Filesystem library adopted as a C++ standard library
  - For this it was adjusted to new language features,, cleaned up, and extended to provide some missing pieces
- First example: how to use a passed string for file type checking

```
#include <iostream>
#include <filesystem>

int main(int argc, char* argv[])
{ // ...
  std::filesystem::path p{argv[1]};
    // p represents a filesystem path (might not exist)
  if (is_regular_file(p)) {   // is path p a regular file?
    std::cout << p << " exists with "
              << file_size(p) << " bytes\n";
  }
  else if (is_directory(p)) { // is path p a directory?
    std::cout << p << " is a directory\n";
  }
  else if (exists(p)) {       // does path p actually exist?
    std::cout << p << " is a special file\n";
  }
  else {
    std::cout << "path " << p << " does not exist\n";
  }
}
```

Namespace

- The filesystem library has its own sub-namespace `filesystem` inside `std`.
- Common convention to introduce the shortcut fs:

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

- This allows us to use `fs::current_path()` instead of `std::filesystem::current_path()`
- It is recommended to always qualify the namespace of filesystem calls explicitly
  - although using argument dependent lookup (ADL) usually works.
  - Not qualifying filesystem calls might sometimes result in unintended behavior.

The Filesystem Library

G4FileSystem.hh

- Some older compilers, we support, supply `<filesystem>` in the `std::experimental` namespace, and the implementation may be in a separate library from the main standard library.
- Geant4 supplies work arounds for these in the `G4Filesystem.hh` header of the `G4globman` module.
  - It simply detects the appropriate header and namespace to use, creating a `G4fs` namespace alias to `std::filesystem` or `std::experimental::filesystem` as required
- Geant4 code requiring use of filesystem can then be written as, e.g.

Instead of:

```
#include <filesystem>

namespace fs = std::filesystem;
fs::path p;
```

Use:

```
#include "G4Filesystem.hh"

G4fs::path p;
```

Paths

- `path` represents the (potential) location of a file within a filesystem
- The path can be **relative** (so that the file location depends on the current working directory) or **absolute**.
- Different formats are possible:
  - A **generic** format, which is portable
  - A **native** format, which is specific to the underlying file system
    - On POSIX-based operating systems there is no difference
    - On Windows, the generic format /tmp/test.txt is a valid native format in addition to \tmp\test.txt, which is also supported
- Special file names `“.”` and `“..”` exist
- See more details on path format in a backup slide

Path Normalization

The filesystem library provides functions for both **lexical normalization** (not taking the filesystem into account) and **filesystem-dependent normalization**.

Path	POSIX normalized	Windows normalized
foo/.///bar/../	foo/	foo\
//host/../foo.txt	//host/foo.txt	\\host\foo.txt
./f/../.f/	.f/	.f\
C:bar/../	.	C:
C:/bar/..	C:/	C:\
C:\bar\..	C:\bar\..	C:\
/./../data.txt	/data.txt	\data.txt
././	.	.

Normalized path:

- File names are separated only by a single preferred directory separator.
- `"."` is not used unless the whole path is nothing but "." (representing the current directory).
- `".."` file names are not present unless they are at the beginning of a relative path.
- The path only ends with a directory separator if the trailing file name is a directory with a name other than "." or “..”

Member Function versus Free-Standing Functions

- Several functions are provided as both member and free-standing functions
- **Member functions** are **cheap**.
  - Pure lexical operations, do not take the actual filesystem into account (no operating systems calls are necessary)

```
mypath.is_absolute(); // check whether path is absolute or relative
```

- **Free-standing functions** are **expensive**
  - Usually take the actual filesystem into account (operating systems calls are necessary)

```
equivalent(path1, path2); // true if both paths refer to the same file
```

- The same functionality operating both lexically and by taking the actual filesystem into account is sometimes provided:

```
std::filesystem::path fromP, toP;
...
toP.lexically_relative(fromP);
relative(toP, fromP);
```

yield lexical path from fromP to toP
yield actual path from fromP to toP

Argument Dependent Lookup (ADL)

- You do not have to specify the full namespace `std::filesystem` when calling free-standing filesystem functions and an argument has a filesystem specific type
- You only have to qualify the call when implicit conversions from other types are used

```
namespace fs = std::filesystem;
create_directory(fs::path{"tmpdir"}); // OK
remove(fs::path{"tmpdir"});           // OK
fs::create_directory("tmpdir");       // OK
fs::remove("tmpdir");                 // OK
```

```
create_directory("tmpdir"); // ERROR                    (*)
remove("tmpdir"); // OOPS: calls C function remove() (**)
```

- The call (*) will fail to compile because we do not pass an argument of the filesystem namespace meaning, the symbol `create_directory` is not looked up in that namespace.
- Depending on the header file you include (indirectly), the call (**) might compile finding the C function `remove()`, which behaves slightly differently: it might also remove the specified file but might not remove empty directories.
- **Recommendation: to explicitly qualify all free-standing filesystem calls**

Error Handling

- Filesystems are a source of errors:
  - Files might not exist, file operations might not be allowed, or operations might violate resource limits
  - While a program is running other processes might create, modify or remove files meaning that advance checks are no guarantee for no errors
- A mixed approach:
  - By default, filesystem errors are handled as **exceptions**.
  - However, you can handle **specific errors** locally if you have or want to.
- Filesystem operations usually have two overloads:
  1. By default the operations throw a filesystem_error exceptions on errors.
  2. By passing an additional out parameter, you can get an error code on error instead.
     - Note that in the latter case, you might still have special return values, signalling a specific error that is not handled as an exception.

Error Handling - Exception

```
try {
  ...
  if (!fs::create_directory(p)) {
    std::cout << p << " already exists\n";
  }
  ...
}
catch (const fs::::filesystem_error& e) {
  std::cout << "EXCEPTION: " << e.what() << '\n';
  std::cout << " path: " << e.path1() << '\n';
}
```

exception on error (unless path exists)

path exists

Error Handling - Error Codes

```
std::error_code ec;
fs::create_directory(p, ec);
```

Set error code on error

```
if (ec) {
  std::cout << "ERROR: " << ec.message() << "\n";
}
```

Check if error code set (due to error)

```
if (ec == std::errc::read_only_file_system) {
  // if specific error code set
  std::cout << "ERROR: " << p << " is read-only\n";
}
```

We can also check against specific error codes

```
std::error_code ec;
if (!fs::create_directory(p, ec)) {
  std::cout << "can't create directory " << p << "\n";
  std::cout << "error: " << ec.message() << "\n";
}
```

We still can check the return value of create_directory()

File Types

- `file_type` = an enumeration type

```
namespace std::filesystem {
   enum class file_type {
      regular, directory,
      symlink, block, character,
      fifo, socket,
      ...
      none, not_found, unknown,
   };
}
```

- Platforms might provide additional file type values. However, using them is not portable.

Value	Meaning
`regular`	Regular file
`directory`	Directory file
`symlink`	Symbolic link file
`block`	Block-special file
`character`	Character-special file
`fifo`	FIFO or pipe file
`socket`	Socket file
...	Additional implementation-defined file type
`none`	The type of the file is not known (yet)
`unknown`	The file exists but the type could not be determined
`not_found`	Pseudo-type indicating the file was not found

Iterating Over Directories

- We can iterate over directories or all files of a filesystem (sub)tree
- **A range-based for loop** (the most convenient way) :

```
// Iterate over all files in a directory
for (const auto& e : fs::directory_iterator(dir)) {
   std::cout << e.path() << '\n';
}

// Iterate recursively over all files in a filesystem (sub)tree
for (const auto& e : fs::recursive_directory_iterator(dir)) {
   std::cout << e.path() << '\n';
}
```

The passed argument dir can be a path or anything implicitly convertible to a path (especially all forms of strings).

- See also iterating with use of `begin()` and `end()` in a backup slide

Library Extensions and Expert Utilities

Container Extensions
Parallel STL Algorithms
Polymorphic Memory Resources (PMR)

Container Extensions

Node Handles

- The associative or unordered containers are node-based data structures, C++17 adds the **ability to splice a node out** of a container and
  - Modify keys of (unordered) maps or values of (unordered) sets
  - Move elements between (unordered) sets and maps
  - Merge elements from one (unordered) set or map into another
- New functions:
  - `extract()` - extracts nodes from the container
  - `merge()` - splices nodes from another container

`extract` is the only way to change a key of a map element without reallocation

```
map<int, string> m{{1, "mango"},
                   {2, "papaya"},
                   {3, "guava"}};

auto nh = m.extract(2);
nh.key() = 4;

m.insert(move(nh));
// m == {{1, "mango"},
//       {3, "guava"},
//       {4, "papaya"}}
```

Emplace Improvements

- `emplace` (C++11) - takes the arguments necessary to construct an object in place, whereas insert copies objects into the vector
- C++17: the emplace functions now **return a reference to the inserted objects** also for the sequential containers (std::vector<>, ...) and the container adapters (srd::stack, ..)
  - As was already the case for associative containers in C++11
- C++17: **new std::map functions**:
  - `try_emplace()` emplaces a new value with defined move semantics
  - `insert_or_assign()` is a slightly improved way to insert/update elements

```
struct Foo
{
  Foo(int n, double x);
};

std::vector<Foo> v;
v.insert(someIterator, Foo(42, 3.1416));
v.emplace(someIterator, 42, 3.1416);

// Before C++17
myVector.emplace_back( ... );
foo(myVector.back());

// C++17
foo(myVector.emplace_back( ... ));
```

Container Support for Incomplete Types

- Since C++17, `std::vector`, `std::list`, and `std::forward_list` are required to support incomplete types.
  - The main motivation: you can now have a type, which recursively has a member of a container of its type

```
struct Node {
  std::string value;
  std::vector<Node> children;
  // Node is an incomplete type here
  // OK since C++17
};
```

Parallel STL Algorithms

- Many algorithms were extended by a new first argument (execution policy) to specify whether and how to run the algorithm in parallel threads
  - The old way without this argument is, of course, still supported
- **Execution policies;**
  - `std::execution::seq` - sequential execution
  - `std::execution::par` - parallel sequenced execution
  - `std::execution::par_unseq` - parallel unsequenced (vectorized) execution
- In addition, some supplementary algorithms were introduced that specifically support parallel processing

```
#include <algorithm>
#include <execution>


// old, still valid API
for_each(coll.begin(), coll.end(),
         [](auto& val) {
            // ... do some computation
         });

// C++17
for_each(std::execution::par,
         coll.begin(), coll.end(),
         [](auto& val) {
            // ... do some computation
         });
```

Polymorphic Memory Resources (PMR)

- Until C++17, using allocators (correctly) was in many ways both tricky and clumsy
- C++17 now provides a fairly easy-to-use approach for predefined and user-defined ways of memory allocation
  - which can be used for standard types and user-defined types
- Standard memory resources are provided in `<memory_resource>`
- Custom memory resources can be defined by deriving from `std::pmr::memory_resource`
- Here, we give just one example on the next slide without presenting all the features available

PMR - Example

```
std::array<std::byte, 200000> buf;
```

Allocate some memory on the stack using the new type `std::byte` (but you could also just use `char`)

```
std::pmr::monotonic_buffer_resource pool{
  buf.data(), buf.size()};
```

Initialize a `monotonic_buffer_resource` with this memory, passing its address and its size

```
std::pmr::vector<std::string> coll{&pool};
```

Finally, we use a `std::pmr::vector`, which takes the memory resource for all its allocations

- The class `monotonic_buffer_resource` is derived from the class `memory_resource` and can therefore be used as a memory resource for a polymorphic allocator.
- By passing the address of our memory resource, we ensure that the vector uses our memory resource as a polymorphic allocator.

```
// The declaration above is just a shortcut for:
std::vector<std::string, std::pmr::polymorphic_allocator<std::string>> coll{&pool};
```

BACKUP SLIDES

Generic Lambdas (C++14)

- C++11: you have to state the parameter type

```
// C++11
for_each( begin(v), end(v), [](decltype(*cbegin(v)) x) { cout << x; } );
sort( begin(w), end(w), [](const shared_ptr<some_type>& a,
                           const shared_ptr<some_type>& b) { return (*a)<(*b); } );
auto size = [](const unordered_map<wstring, vector<string>>& m) { return m.size(); };
```

- C++14: lambda function parameters can be `auto` to let the compiler deduce the type

```
// C++14: just deduce the type
for_each( begin(v), end(v), [](const auto& x) { cout << x; } );
sort( begin(w), end(w), [](const auto& a, const auto& b) { return (*a)<(*b); } );
```

- C++14: generic lambda that will work with any suitable type and just do the right thing

```
// C++14: new expressive power
auto size = [](const auto& m) { return m.size(); };
```

constexpr Lambdas - 2

- Using features that are not allowed in `constexpr` contexts disables this ability but you can still use the lambda in runtime contexts:

```
auto squared2 = [](auto val) {
   // implicitly constexpr since C++17
   static int calls = 0;
      // OK, but disables lambda for constexpr
      //contexts
   ...
   return val*val;
};

std::array<int,squared2(5)> a;
   // ERROR: static variable in compile-time
context
std::cout << squared2(5) << '\n';
   // OK
```

- To find out at compile time whether a lambda is valid for a compile-time context, you can declare it as `constexpr`:

```
// OK since C++17
auto squared3 = [](auto val) constexpr {
   return val*val;
};
```

Attribute `[[fallthrough]]`

- Can be used to avoid warnings by the compiler for not having a break statement after a sequence of one or more case labels inside a switch statement.

- Note that the attribute has to be used in an empty statement. It must therefore end with a semicolon.
- Using the attribute as the last statement in a switch statement is not allowed.

```
void commentPlace(int place)
{
    switch (place) {
        case 1:
            std::cout << "very ";
            [[fallthrough]];
        case 2:
            std::cout << "well\n";
            break;
        default:
            std::cout << "OK\n";
            break;
    }
}
```

Passing the place 1 will print:

```
very well
```

Relaxed Enum Initialization from Integral Values

- For enumerations with a fixed underlying type you can use an integral value of that type for direct list initialization.
  - Applies to unscoped enumerations with a specified type and all scoped enumerations
  - Cannot be used for unscoped enumerations (enum without class) with no specified underlying type

```
// Unscoped enum with underlying type
enum Color : int { red, green, blue};
Color c1{0};     // OK since C++17 (ERROR before C++17)
Color c2 = 0;    // still ERROR
Color c3(0);     // still ERROR
Color c4 = {0}; // still ERROR

// Scoped enum (class) with default underlying type
enum class Weekday { mon, tue, wed, thu, fri, sat, sun };
Weekday d1{1};  // OK since C++17 (ERROR before C++17)

// Scoped enum (class) with specified underlying type:
enum class Weekday2 : char { mon, tue, wed, thu, fri, sat, sun };
Weekday2 d2{1};  // OK since C++17 (ERROR before C++17)

// Unscoped enum without underlying type
enum Color { red, green, blue};
Color c{0};     // ERROR
```

`std::optional<>` Operations

Operation	Effect
constructors	Creates an optional object (might or might not call constructor for contained type)
`make_optional<>()`	Creates an optional object initialized by the passed value(s)
destructor	Destroys an optional object
`=`	Assigns a new value
`emplace()`	Assigns a new value to the contained type
`reset()`	Destroys any value (makes the object empty)
`has_value()`	Returns whether the object has a value
conversion to `bool`	Returns whether the object has a value
`*`	Value access (undefined behavior if no value)
`->`	Access to member of the value (undefined behavior if no value)
`value()`	Value access (exception if no value)
`value_or()`	Value access (fallback argument if no value)
`swap()`	Swaps values between two objects
`==, !=, <, <=, >, >=`	Compares optional objects
`hash<>`	Function object type to compute hash values

std::optional<>

Construction

```
std::optional<int> o1;
std::optional<int> o2(std::nullopt);
```

An optional object that does not have a value

```
std::optional o3{42};
std::optional o4{"hello"};
```

Pass a value to initialize the contained type:
`o3` deduces `optional<int>`
`o4` deduces `optional<const char*>`

```
std::optional<std::string> o10{o4};
```

Copy optional objects (including type conversions) provided their underlying type supports copying

```
auto o13 = std::make_optional(3.0);
auto o14 = std::make_optional("hello");
auto o15 = std::make_optional<std::complex<double>>(3.0, 4.0);
```

Using function `make_optional<>()`, which allows an initialization with single or multiple arguments

Accessing the Value

```
std::optional o{42};
if (o) ...    // true
if (!o) ...   // false
if (o.has_value()) ...   // true
```

Check whether an optional object has a value

```
std::cout << o; // ERROR
```

No I/O operators are defined

```
std::optional o{std::pair{42, "hello"}};
auto p = *o;
std::cout << o->first;
```

Pointer syntax is provided to access the value
`*o` initializes `p` as `pair<int,string>`
prints `42`

```
o = std::nullopt;
std::cout << *o; // undefined behaviour
```

!!! Without a value is undefined behavior:

```
if (o) std::cout << *o;
  // OK (might output nothing)
std::cout << o.value();
  // OK (but throws an exception if no value)
```

Note that both `operator*` and `value()` return the contained object by reference. You have to be careful when calling these operations directly for temporary return values.

Accessing the Value - 2

```
// You can ask for the value and pass a fallback value
// which is used if the optional object has no value.
std::cout << o.value_or("NO VALUE”);
   // OK (writes NO VALUE if no value)

// More efficient
std::cout << o ? o->c_str() : "fallback";
```

There is a subtle difference between `value()` and `value_or()`: `value_or()` always returns by value, while `value()` returns by reference.

This means that calls to `value_or()` allocate memory, while a call to `value()` never does.

`std::optional<>`

Comparisons

- Usual comparison operators
- Operands can be an optional object, an object of the contained type, and `std::nullopt`.
  - If **both** operands are **objects with a value**, the corresponding operator of the contained type is used.
  - If **both** operands are **objects without a value**, they are considered to be **equal** (==, <=, and >= yield true and all other comparisons yield false).
  - If **only one operand is an object with a value**, the operand without a value is considered to be less than the other operand.

```
std::optional<int> o0;
std::optional<int> o1{42};

o0 == std::nullopt // yields true
o0 == 42           // yields false
o0 < 42            // yields true
o0 > 42            // yields false
o1 == 42           // yields true
o0 < o1            // yields true

// For optional objects of unsigned int
// there is a value less than 0:
std::optional<unsigned> uo;
uo < 0             // yields true
uo < -42           // yields true

// For optional objects of bool,
// there is a value less than false:
std::optional<bool> bo;
bo < false         // yields true

if (!uo.has_value()) // RECOMMENDED
if (uo < 0)          // DO NOT USE
```

std::variant<> Operations

Operation	Effect
constructors	Creates a variant object (might call constructor for underlying type)
destructor	Destroys a variant object
=	Assigns a new value
emplace<T>()	Destroys the old value and assigns a new value to the alternative with type T
emplace<Idx>()	Destroys the old value and assigns a new value to the alternative with index Idx
valueless_by_exception()	Returns whether the variant has no value due to an exception
index()	Returns the index of the current alternative
swap()	Swaps values between two objects
==, !=, <, <=, >, >=	Compares variant objects
hash<>	Function object type for computing hash values
holds_alternative<T>()	Returns whether there is a value for type T
get<T>()	Returns the value for the alternative with type T
get<Idx>()	Returns the value for the alternative with index Idx
get_if<T>()	Returns a pointer to the value for the alternative with type T or nullptr
get_if<Idx>()	Returns a pointer to the value for the alternative with index Idx or nullptr
visit()	Performs an operation for the current alternative

Accessing the Value with get<>()

```
// Create variant
std::variant<int, int, std::string> var;
```

Sets first int to 0, index()==0

```
// Call get<>()
auto a = std::get<double>(var);// ERROR
auto b = std::get<4>(var);     // ERROR
auto c = std::get<int>(var);   // ERROR
```

Compile-time ERROR:
no `double`
no 4th alternative
int twice

- You can pass the index or, provided a type is not used more than once, its type.
- Using an invalid index or invalid/ambiguous type results in a compile-time error.

```
try {
  auto s = std::get<std::string>(var);// throws
  auto i = std::get<0>(var); // OK
  auto j = std::get<1>(var); // throws
}
catch (const std::bad_variant_access& e) {
  std::cout << "Exception: "
            << e.what() << '\n';
}
```

Exception caught in case of an invalid access:
`s` throws exception (first `int` currently set)
`i` OK, `i==0`
`j` throws exception (other `int` currently set)

- If the passed type or index requests the value of an alternative currently not set, a variant object throws an exception.

Accessing the Value with get_if<>()

```
if (auto ip = std::get_if<1>(&var); ip != nullptr) {
   std::cout << *ip << '\n'; }
else {
   std::cout << "alternative with index 1 not set\n";
}
```

Note that if with initialization is used

```
if (auto ip = std::get_if<1>(&var)) { ... }
```

The initialization can be used as a condition directly

- API to access the value with the option to check whether it exists
- You have to pass a pointer to a variant to `get_if<>()`, which then returns either a pointer to the current value or `nullptr`

Changing the Value

```
// Create variant
std::variant<int, int, std::string> var;
```

Sets first int to 0, index()==0

```
var = "hello";
```

Assignment: directly assigns the new value if the variant currently holds the matching alternative
Sets `string, index()==2`

```
var.emplace<1>(42);
```

`emplace()` operation: always destroys the old value and assigns the new value
Sets second `int, index()==1`

```
std::variant<int, int, std::string> var;
std::get<0>(var) = 77; // OK
std::get<1>(var) = 99; // throws
if (auto p = std::get_if<1>(&var); p) {
  *p = 42;
}
```

`get<>()` or `get_if<>()` can be used to assign a new value to the current alternative:
`get<0>` OK, because first int already set
`get<1>` throws exception (other int currently set)
`get_if<1>` modifies second int if it is set

- *`std::variant<>` can be also modified using variant **visitors** (not presented here)*

Comparisons

```
std::variant<std::monostate, int, std::string> v1, v2{"hello"}, v3{42};
std::variant<std::monostate, std::string, int> v4;
```

- Usual comparison operators
- Operands must be **variants of the same type** (i.e., they have the same alternatives in the same order)
  - A variant with *a value of an earlier alternative* is less than a variant with a value with a later alternative.
  - If two variants have *the same alternative*, the corresponding operators for the type of the alternatives are evaluated.
    - Note that all objects of type `std::monostate` are always equal.

```
   v1 == v4   // COMPILE-TIME ERROR
   v1 == v2   // yields false
   v1 < v2    // yields true
   v1 < v3    // yields true
   v2 < v3    // yields false

v1 = "hello";
   v1 == v2   // yields true

v2 = 41;
   v2 < v3    // yields true
```

`std::any` Operations

Operation	Effect
constructors	Creates an any object (might call constructor for underlying type)
`make_any()`	Creates an any object (passing value(s) to initialize it)
destructor	Destroys an any object
`=`	Assigns a new value
`emplace<T>()`	Assigns a new value with the type `T`
`reset()`	Destroys any value (makes the object empty)
`has_value()`	Returns whether the object has a value
`type()`	Returns the current type as `std::type_info` object
`any_cast<T>()`	Uses the current value as value of type `T` (exception/`nullptr` if other type)
`swap()`	Swaps values between two objects

Construction

```
std::any a1;
```

By default initialized by being empty

```
std::any a2 = 42;
std::any a3 = "hello";
```

If a value is passed for initialization, its decayed type is used as the type of the contained value:
a2 contains value of type `int`
a3 contains value of type `const char*`

```
std::any a6{std::complex{3.0, 4.0}};
```

To initialize with multiple arguments, you have to create the object (or use `std::in_place_type` *- not demonstrated here)*

```
auto a10 = std::make_any<float>(3.0);
auto a11 = std::make_any<std::string>("hello");
auto a13 = std::make_any<std::complex<double>>(3.0, 4.0);
```

Function `make_any<>()` *can be used for single or multiple arguments*
You always have to explicitly specify the initialized type

- Not covered: use of the `in_place_type` tags to hold a type different to the type of the initial value; passing an initializer list

Changing the Value

```
std::any a;
a = 42;
a = “hello";
```

Assignment:
a contains value of type `int`
a contains value of type `const char*`

```
a.emplace<std::string>("hello");
a.emplace<std::complex<double>>(4.4, 5.5);
```

`emplace()` *operation*
a contains value of type `std::string`
a contains value of type `std::complex<double>`

Accessing the Value

- To access the contained value, you have to cast it to its type with a `std::any_cast<>`

```
std::any_cast<std::string>(a);
std::any_cast<std::string&>(a);
std::any_cast<const std::string&>(a);
```

yield copy of the value
write access by reference
read access by reference

- If the cast fails, a `std::bad_any_cast exception` is thrown. To avoid exception handling, the address of an any object can be passed. It returns `nullptr` if the cast fails:

```
if (auto sp{std::any_cast<std::string>(&a)}; sp != nullptr) {
  ... // use *sp for write access to the value of a
}
if (auto sp{std:any_cast<const std::string>(&a)}; sp != nullptr) {
   ... // use *sp for read access to the value of a
}
```

```
std::any_cast<std::string&>(&a); // RUNTIME ERROR
```

Casting to a reference results in a runtime error

String View Operations

Operation	Effect
constructors	Creates or copies a string view
destructor	Destroys a string view
`=`	Assigns a new value
`swap()`	Swaps values between two strings views
`==`, `!=`, `<`, `<=`, `>`, `>=`, `compare()`	Compares string views
`empty()`	Returns whether the string view is empty
`size()`, `length()`	Returns the number of characters
`max_size()`	Returns the maximum possible number of characters
`[]`, `at()`	Accesses a character (read-only)
`front()`, `back()`	Accesses the first or last character (read-only)
`<<`	Writes the value to a stream
`copy()`	Copies or writes the contents to a character array
`data()`	Returns the value as `nullptr` or constant character array (note: no terminating null character)
find functions	Searches for a certain substring or character
`begin()`, `end()`	Provides iterator support (`const_iterator` only)
`cbegin()`, `cend()`	Provides constant iterator support
`rbegin()`, `rend()`	Provides constant reverse iterator support
`crbegin()`, `crend()`	Provides constant reverse iterator support
`substr()`	Returns a certain substring
`remove_prefix()`	Removes leading characters
`remove_suffix()`	Removes trailing characters
`hash<>`	Function object type to compute hash values

Construction

```
std::string_view sv1;
auto p = sv1.data();
std::cout << sv1[0]; // ERROR
```

Create with the default constructor
p yields `nullptr`
ERROR: no valid character

```
std::string_view sv2{"hello"};
std::cout << sv2;          // OK
std::cout << sv2.size();   // 5
std::cout << sv2.at(5);    // throws
std::cout << sv2[5];       // undefined behavior
std::cout << sv2.data();   // OOPS !
```

*Initialize with a **`string` literal***
OK
`size()` give 5
throws `std::out_of_range` exception
undefined behavior
OOPS: only works because `'\0'` is behind `sv2`

```
std::string_view sv3{"hello", 6};
std::cout << sv3.size(); //6
std::cout << sv3.at(5);  // OK
std::cout << sv3[5];     // OK
std::cout << sv3.data(); // OK
```

*Initialize with **a string literal and the number of characters** including the null terminator: **NOTE: 6 to include '\0'***
`size()` give 6
OK, prints the value of '\0'
OK, prints the value of '\0'
OK

- Note that the value of the string view can be null terminated, although the null terminator is not part of the value (sv2)
- or you can ensure that '\0' is part of the string view (sv3).

String View

Construction - 2

```
std::string s = "hello";
std::cout << s.size(); // 5
std::cout << s.at(5);  // throws
std::cout << s[5];     // OK
std::cout << s.data(); // OK
```

Create and inspect ***a string***

`size()` *give 5*

throws `std::out_of_range` *exception*

`OK, prints the value of '\0'`

`OK`

```
std::string_view sv{s};
std::cout << sv.size(); // 5
std::cout << sv.at(5);  // throws
std::cout << sv[5];     // undefined behavior
std::cout << sv.data(); // OOPS !!!
```

Create ***from a string***

`size()` *give 5*

throws `std::out_of_range` *exception*

`undefined behavior`

`OOPS: only works because '\0' is behind sv`

Modifying String View

```
std::string_view sv1 = "hey";
std::string_view sv2 = "world";
sv1.swap(sv2);
sv2 = sv1;
```

Assign a new value or swap the values of two string views

```
std::string_view sv = "I like my kindergarten";
sv.remove_prefix(2);
sv.remove_suffix(8);
std::cout << sv;
```

Skip leading or trailing characters
Prints: `like my kind`

```
std::string_view sv1 = "hello";
std::string_view sv2 = "world";
auto s1 = sv1 + sv2;  // ERROR
```

No support for operator+

```
auto s2 = std::string(sv1) + std::string(sv2); // OK
```

Paths - 2

- The generic path format is as follows:

`[rootname] [rootdir] [relativepath]`

  - **root name** : optional , implementation-specific (
    - e.g. //host on POSIX systems and C: on Windows systems)
  - **root directory** : optional, a directory separator
  - **relative path:** a sequence of file names separated by directory separators

- A directory separator consists of one or multiple `'/'` or implementation-specific preferred directory separators (eg. the backslash on Windows)
- Examples of portable generic paths are:

  `//host1/bin/hello.txt .`

  `tmp/`

  `/a/b//../c`

  - Note that the last path refers to the same location as `/a/c`

Iterating Over Directories - 2

- **Directory iterators** are ranges - global overloads of `begin()` and `end()` are provided:
  - `begin()` yields the iterator itself
  - `end()` yields the end iterator, which you can also create with the default constructor

- Therefore, you can also iterate as follows:

```
fs::directory_iterator di{p};
for (auto pos = begin(di); pos != end(di); ++pos) {
   std::cout << pos->path() << '\n';
}

for (fs::directory_iterator pos{p};
     pos != fs::directory_iterator{};
     ++pos) {
   std::cout << pos->path() << '\n';
}
```

Path Creation

Call	Effect
`path`{*charseq*}	Initializes a path from a character sequence
`path`{*beg*,*end*}	Initializes a path from a range
`u8path`(*u8string*)	Yields a path from a UTF-8 string
`current_path()`	Yields the path of the current working directory
`temp_directory_path()`	Yields the path for temporary files

The Filesystem Library Operations

Path Inspection

Call	Effect
`p.empty()`	Yields whether a path is empty
`p.is_absolute()`	Yields whether a path is absolute
`p.is_relative()`	Yields whether a path is relative
`p.has_filename()`	Yields whether a path is neither a directory nor a root name
`p.has_stem()`	Same as `has_filename()` (as any file name has a stem)
`p.has_extension()`	Yields whether a path has an extension
`p.has_root_name()`	Yields whether a path has a root name
`p.has_root_directory()`	Yields whether a path has a root directory
`p.has_root_path()`	Yields whether a path has a root name or a root directory
`p.has_parent_path()`	Yields whether a path has a parent path
`p.has_relative_path()`	Yields whether a path consists of more than just root elements
`p.filename()`	Yields the file name (or the empty path)
`p.stem()`	Yields the file name without extension (or the empty path)
`p.extension()`	Yields the extension (or the empty path)
`p.root_name()`	Yields the root name (or the empty path)
`p.root_directory()`	Yields the root directory (or the empty path)
`p.root_path()`	Yields the root elements (or the empty path)
`p.parent_path()`	Yields the parent path (or the empty path)
`p.relative_path()`	Yields the path without root elements (or the empty path)
`p.begin()`	Yields the begin of the path elements
`p.end()`	Yields the end of the path elements

The Filesystem Library Operations

Path I/O and Conversion

Call	Effect
`strm << p`	Writes the value of a path as quoted string
`strm >> p`	Reads the value of a path as quoted string
`p.string()`	Yields the path as a `std::string`
`p.wstring()`	Yields the path as a `std::wstring`
`p.u8string()`	Yields the path as a UTF-8 string of type `std::u8string`
`p.u16string()`	Yields the path as a UTF-16 string of type `std::u16string`
`p.u32string()`	Yields the path as a UTF-32 string of type `std::u32string`
`p.string<...>()`	Yields the path as a `std::basic_string<...>`
`p.lexically_normal()`	Yields `p` as normalized path
`p.lexically_relative(p2)`	Yields the path from `p2` to `p` (empty path if none)
`p.lexically_proximate(p2)`	Yields the path from `p2` to `p` (`p` if none)

The Filesystem Library Operations

Path Modifications

Call	Effect
`p = p2`	Assigns a new path
`p = sv`	Assigns a string (view) as a new path
`p.assign(p2)`	Assigns a new path
`p.assign(sv)`	Assigns a string (view) as a new path
`p.assign(beg, end)`	Assigns elements of the range from `beg` to `end` to the path
`p1 / p2`	Yields the path that appends `p2` as sub-path of path `p1`
`p /= sub`	Appends `sub` as sub-path to path `p`
`p.append(sub)`	Appends `sub` as sub-path to path `p`
`p.append(beg, end)`	Appends elements of the range from `beg` to `end` as sub-paths to path `p`
`p += str`	Appends the characters of `str` to path `p`
`p.concat(str)`	Appends the characters of `str` to path `p`
`p.concat(beg, end)`	Appends elements of the range from `beg` to `end` to path `p`
`p.remove_filename()`	Removes a trailing file name from the path
`p.replace_filename(repl)`	Replaces the trailing file name (if any)
`p.replace_extension()`	Removes any trailing file name extension
`p.replace_extension(repl)`	Replaces the trailing file name extension (if any)
`p.clear()`	Makes the path empty
`p.swap(p2)`	Swaps the values of two paths
`swap(p1, p2)`	Swaps the values of two paths
`p.make_preferred()`	Replaces directory separators in `p` with native format and yields the modified `p`

Path Comparisons and Other

Call	Effect
`p1 == p2`	Yields whether two paths are equal
`p1 != p2`	Yields whether two paths are not equal
`p1 < p2`	Yields whether a path is less than another
`p1 <= p2`	Yields whether a path is less than or equal to another
`p1 >= p2`	Yields whether a path is greater than or equal to another
`p1 > p2`	Yields whether a path is greater than another
`p.compare(p2)`	Yields whether `p2` is less than, equal to, or greater than `p`
`p.compare(sv)`	Yields whether `p2` is less than, equal to, or greater than the string (view) `sv` converted to a path
`equivalent(p1, p2)`	Expensive path comparison taking the filesystem into account

File Types

Call	Effect
`exists(p)`	Yields whether there is a file that can be accessed
`is_symlink(p)`	Yields whether the file `p` exists and is a symbolic link
`is_regular_file(p)`	Yields whether the file `p` exists and is a regular file
`is_directory(p)`	Yields whether the file `p` exists and is a directory
`is_other(p)`	Yields whether the file `p` exists and is neither regular nor a directory nor a symbolic link
`is_block_file(p)`	Yields whether the file `p` exists and is a block-special file
`is_character_file(p)`	Yields whether the file `p` exists and is a character-special file
`is_fifo(p)`	Yields whether the file `p` exists and is a FIFO or pipe file
`is_socket(p)`	Yields whether the file `p` exists and is a socket

The Filesystem Library Operations

File Status

Call	Effect
`status(p)`	Yields the `file_status` of the file `p` (following symbolic links)
`symlink_status(p)`	Yields the `file_status` of `p` (not following symbolic links)

The possible calls for a `file_status` object `fs`:

Call	Effect
`exists(fs)`	Yields whether a file exists
`is_regular_file(fs)`	Yields whether the file exists and is a regular file
`is_directory(fs)`	Yields whether the file exists and is a directory
`is_symlink(fs)`	Yields whether the file exists and is a symbolic link
`is_other(fs)`	Yields whether the file exists and is neither regular nor a directory nor a symbolic link
`is_block_file(fs)`	Yields whether the file exists and is a block-special file
`is_character_file(fs)`	Yields whether the file exists and is a character-special file
`is_fifo(fs)`	Yields whether the file exists and is a FIFO or pipe file
`is_socket(fs)`	Yields whether the file exists and is a socket
`fs.type()`	Yields the `file_type` of the file
`fs.permissions()`	Yields the permissions of the file

The Filesystem Library Operations

Create and Delete Files

Call	Effect
`create_directory(p)`	Creates a directory
`create_directory(p, attrPath)`	Creates a directory with attributes of `attrPath`
`create_directories(p)`	Creates a directory and all directories above that do not exist yet
`create_hard_link(to, new)`	Creates another filesystem entry `new` for the existing file `to`
`create_symlink(to, new)`	Creates a symbolic link from `new` to `to`
`create_directory_symlink(to, new)`	Creates a symbolic link from `new` to the directory `to`
`copy(from, to)`	Copies a file of any type
`copy(from, to, options)`	Copies a file of any type with `options`
`copy_file(from, to)`	Copies a file (but not directory or symbolic link)
`copy_file(from, to, options)`	Copies a file with `options`
`copy_symlink(from, to)`	Copies a symbolic link (`to` refers to where `from` refers)
`remove(p)`	Removes a file or empty directory
`remove_all(p)`	Removes `p` and recursively all files in its subtree (if any)

The Filesystem Library Operations

Modify Existing Files

Call	Effect
`rename(old, new)`	Renames and/or moves a file
`last_write_time(p, newtime)`	Changes the timepoint of the last write access
`permissions(p, prms)`	Replaces the permissions of a file with `prms`
`permissions(p, prms, mode)`	Modifies the permissions of a file according to `mode`
`resize_file(p, newSize)`	Changes the size of a regular file

Directory Entry Operations

Call	Effect
`e.path()`	Yields the filesystem `path` for the current entry
`e.exists()`	Yields whether the file exists
`e.is_regular_file()`	Yields whether the file exists and is a regular file
`e.is_directory()`	Yields whether the file exists and is a directory
`e.is_symlink()`	Yields whether the file exists and is a symbolic link
`e.is_other()`	Yields whether the file exists and is neither regular nor a directory nor a symbolic link
`e.is_block_file()`	Yields whether the file exists and is a block-special file
`e.is_character_file()`	Yields whether the file exists and is a character-special file
`e.is_fifo()`	Yields whether the file exists and is a FIFO or pipe file
`e.is_socket()`	Yields whether the file exists and is a socket
`e.file_size()`	Yields the size of a file
`e.hard_link_count()`	Yields the number of hard links
`e.last_write_time()`	Yields the timepoint of the last write to a file
`e.status()`	Yields the status of the file `p`
`e.symlink_status()`	Yields the file status (following symbolic links) `p`
`e1 == e2`	Yields whether the two entry paths are equal
`e1 != e2`	Yields whether the two entry paths are not equal
`e1 < e2`	Yields whether an entry path is less than another
`e1 <= e2`	Yields whether an entry path is less than or equal to another
`e1 >= e2`	Yields whether an entry path is greater than or equal to another
`e1 > e2`	Yields whether an entry path is greater than another
`e.assign(p)`	Replaces the path of `e` with `p` and updates all entry attributes
`e.replace_filename(p)`	Replaces the file name of the current path of `e` with `p` and updates all entry attributes
`e.refresh()`	Updates all cached attributes for this entry

The Filesystem Library Operations

Other

Call	Effect
`equivalent(p1, p2)`	Yields whether `p1` and `p2` refer to the same file
`space(p)`	Yields information about the disk space available at path `p`
`current_path(p)`	Sets the path of the current working directory to `p`