C++17

Overview

Many of these descriptions and examples are taken from various resources (see Acknowledgements section) and summarized in my own words.

C++17 includes the following new language features:

- template argument deduction for class templates
- declaring non-type template parameters with auto
- folding expressions
- new rules for auto deduction from braced-init-list
- constexpr lambda
- lambda capture this by value
- inline variables
- nested namespaces
- structured bindings
- selection statements with initializer
- constexpr if
- utf-8 character literals
- direct-list-initialization of enums
- fallthrough, nodiscard, maybe_unused attributes

C++17 includes the following new library features:

- std::variant
- std::optional
- std::any
- std::string_view
- std::invoke
- std::apply
- std::filesystem
- std::byte
- splicing for maps and sets
- parallel algorithms

C++17 Language Features

Template argument deduction for class templates

Automatic template argument deduction much like how it's done for functions, but now including class constructors.

```
template <typename T = float>
struct MyContainer {
  T val;
```

```
  MyContainer() : val{} {}
  MyContainer(T val) : val{val} {}
  // ...
};
MyContainer c1 {1}; // OK MyContainer<int>
MyContainer c2; // OK MyContainer<float>
```

Declaring non-type template parameters with auto

Following the deduction rules of `auto`, while respecting the non-type template parameter list of allowable types[*], template arguments can be deduced from the types of its arguments:

```
template <auto... seq>
struct my_integer_sequence {
  // Implementation here ...
};

// Explicitly pass type `int` as template argument.
auto seq = std::integer_sequence<int, 0, 1, 2>();
// Type is deduced to be `int`.
auto seq2 = my_integer_sequence<0, 1, 2>();
```

* - For example, you cannot use a `double` as a template parameter type, which also makes this an invalid deduction using `auto`.

Folding expressions

A fold expression performs a fold of a template parameter pack over a binary operator.

- An expression of the form `(... op e)` or `(e op ...)`, where `op` is a fold-operator and `e` is an unexpanded parameter pack, are called *unary folds*.
- An expression of the form `(e1 op ... op e2)`, where `op` are fold-operators, is called a *binary fold*. Either `e1` or `e2` is an unexpanded parameter pack, but not both.

```
template <typename... Args>
bool logicalAnd(Args... args) {
    // Binary folding.
    return (true && ... && args);
}
bool b = true;
bool& b2 = b;
logicalAnd(b, b2, true); // == true
```

```
template <typename... Args>
auto sum(Args... args) {
    // Unary folding.
```

```
    return (... + args);
}
sum(1.0, 2.0f, 3); // == 6.0
```

New rules for auto deduction from braced-init-list

Changes to `auto` deduction when used with the uniform initialization syntax. Previously, `auto x {3};` deduces a `std::initializer_list<int>`, which now deduces to `int`.

```
auto x1 {1, 2, 3}; // error: not a single element
auto x2 = {1, 2, 3}; // x2 is std::initializer_list<int>
auto x3 {3}; // x3 is int
auto x4 {3.0}; // x4 is double
```

constexpr lambda

Compile-time lambdas using `constexpr`.

```
auto identity = [](int n) constexpr { return n; };
static_assert(identity(123) == 123);
```

```
constexpr auto add = [](int x, int y) {
  auto L = [=] { return x; };
  auto R = [=] { return y; };
  return [=] { return L() + R(); };
};

static_assert(add(1, 2)() == 3);
```

```
constexpr int addOne(int n) {
  return [n] { return n + 1; }();
}

static_assert(addOne(1) == 2);
```

Lambda capture `this` by value

Capturing `this` in a lambda's environment was previously reference-only. An example of where this is problematic is asynchronous code using callbacks that require an object to be available, potentially past its lifetime. `*this` (C++17) will now make a copy of the current object, while `this` (C++11) continues to capture by reference.

```
struct MyObj {
  int value {123};
  auto getValueCopy() {
    return [*this] { return value; };
  }
  auto getValueRef() {
    return [this] { return value; };
  }
};
MyObj mo;
auto valueCopy = mo.getValueCopy();
auto valueRef = mo.getValueRef();
mo.value = 321;
valueCopy(); // 123
valueRef(); // 321
```

Inline variables

The inline specifier can be applied to variables as well as to functions. A variable declared inline has the same semantics as a function declared inline.

```
// Disassembly example using compiler explorer.
struct S { int x; };
inline S x1 = S{321}; // mov esi, dword ptr [x1]
                      // x1: .long 321

S x2 = S{123};        // mov eax, dword ptr [.L_ZZ4mainE2x2]
                      // mov dword ptr [rbp - 8], eax
                      // .L_ZZ4mainE2x2: .long 123
```

It can also be used to declare and define a static member variable, such that it does not need to be initialized in the source file.

```
struct S {
  S() : id{count++} {}
  ~S() { count--; }
  int id;
  static inline int count{0}; // declare and initialize count to 0 within
the class
};
```

Nested namespaces

Using the namespace resolution operator to create nested namespace definitions.

```
namespace A {
  namespace B {
    namespace C {
      int i;
    }
  }
}
```

The code above can be written like this:

```
namespace A::B::C {
  int i;
}
```

Structured bindings

A proposal for de-structuring initialization, that would allow writing `auto [ x, y, z ] = expr;` where the type of `expr` was a tuple-like object, whose elements would be bound to the variables `x`, `y`, and `z` (which this construct declares). *Tuple-like objects* include `std::tuple`, `std::pair`, `std::array`, and aggregate structures.

```
using Coordinate = std::pair<int, int>;
Coordinate origin() {
  return Coordinate{0, 0};
}

const auto [ x, y ] = origin();
x; // == 0
y; // == 0
```

```
std::unordered_map<std::string, int> mapping {
  {"a", 1},
  {"b", 2},
  {"c", 3}
};

// Destructure by reference.
for (const auto& [key, value] : mapping) {
  // Do something with key and value
}
```

Selection statements with initializer

New versions of the `if` and `switch` statements which simplify common code patterns and help users keep scopes tight.

```
{
  std::lock_guard<std::mutex> lk(mx);
  if (v.empty()) v.push_back(val);
}
// vs.
if (std::lock_guard<std::mutex> lk(mx); v.empty()) {
  v.push_back(val);
}
```

```
Foo gadget(args);
switch (auto s = gadget.status()) {
  case OK: gadget.zip(); break;
  case Bad: throw BadFoo(s.message());
}
// vs.
switch (Foo gadget(args); auto s = gadget.status()) {
  case OK: gadget.zip(); break;
  case Bad: throw BadFoo(s.message());
}
```

constexpr if

Write code that is instantiated depending on a compile-time condition.

```
template <typename T>
constexpr bool isIntegral() {
  if constexpr (std::is_integral<T>::value) {
    return true;
  } else {
    return false;
  }
}
static_assert(isIntegral<int>() == true);
static_assert(isIntegral<char>() == true);
static_assert(isIntegral<double>() == false);
struct S {};
static_assert(isIntegral<S>() == false);
```

UTF-8 character literals

A character literal that begins with `u8` is a character literal of type `char`. The value of a UTF-8 character literal is equal to its ISO 10646 code point value.

```
char x = u8'x';
```

Direct list initialization of enums

Enums can now be initialized using braced syntax.

```
enum byte : unsigned char {};
byte b {0}; // OK
byte c {-1}; // ERROR
byte d = byte{1}; // OK
byte e = byte{256}; // ERROR
```

fallthrough, nodiscard, maybe_unused attributes

C++17 introduces three new attributes: `[[fallthrough]]`, `[[nodiscard]]` and `[[maybe_unused]]`.

- `[[fallthrough]]` indicates to the compiler that falling through in a switch statement is intended behavior.

```
switch (n) {
  case 1: [[fallthrough]]
    // ...
  case 2:
    // ...
    break;
}
```

- `[[nodiscard]]` issues a warning when either a function or class has this attribute and its return value is discarded.

```
[[nodiscard]] bool do_something() {
  return is_success; // true for success, false for failure
}

do_something(); // warning: ignoring return value of 'bool do_something()',
                // declared with attribute 'nodiscard'
```

```
// Only issues a warning when `error_info` is returned by value.
struct [[nodiscard]] error_info {
  // ...
};

error_info do_something() {
  error_info ei;
```

```
  // ...
  return ei;
}

do_something(); // warning: ignoring returned value of type 'error_info',
                // declared with attribute 'nodiscard'
```

- `[[maybe_unused]]` indicates to the compiler that a variable or parameter might be unused and is intended.

```
void my_callback(std::string msg, [[maybe_unused]] bool error) {
  // Don't care if `msg` is an error message, just log it.
  log(msg);
}
```

C++17 Library Features

std::variant

The class template `std::variant` represents a type-safe `union`. An instance of `std::variant` at any given time holds a value of one of its alternative types (it's also possible for it to be valueless).

```
std::variant<int, double> v{ 12 };
std::get<int>(v); // == 12
std::get<0>(v); // == 12
v = 12.0;
std::get<double>(v); // == 12.0
std::get<1>(v); // == 12.0
```

std::optional

The class template `std::optional` manages an optional contained value, i.e. a value that may or may not be present. A common use case for optional is the return value of a function that may fail.

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

create(false).value_or("empty"); // == "empty"
create(true).value(); // == "Godzilla"
// optional-returning factory functions are usable as conditions of while
and if
```

```
if (auto str = create(true)) {
  // ...
}
```

std::any

A type-safe container for single values of any type.

```
std::any x {5};
x.has_value() // == true
std::any_cast<int>(x) // == 5
std::any_cast<int&>(x) = 10;
std::any_cast<int>(x) // == 10
```

std::string_view

A non-owning reference to a string. Useful for providing an abstraction on top of strings (e.g. for parsing).

```
// Regular strings.
std::string_view cppstr {"foo"};
// Wide strings.
std::wstring_view wcstr_v {L"baz"};
// Character arrays.
char array[3] = {'b', 'a', 'r'};
std::string_view array_v(array, std::size(array));
```

```
std::string str {"   trim me"};
std::string_view v {str};
v.remove_prefix(std::min(v.find_first_not_of(" "), v.size()));
str; //  == "   trim me"
v; // == "trim me"
```

std::invoke

Invoke a `Callable` object with parameters. Examples of `Callable` objects are `std::function` or `std::bind` where an object can be called similarly to a regular function.

```
template <typename Callable>
class Proxy {
  Callable c;
public:
  Proxy(Callable c): c(c) {}
  template <class... Args>
  decltype(auto) operator()(Args&&... args) {
```

```
    // ...
    return std::invoke(c, std::forward<Args>(args)...);
  }
};
auto add = [](int x, int y) {
  return x + y;
};
Proxy<decltype(add)> p {add};
p(1, 2); // == 3
```

std::apply

Invoke a `Callable` object with a tuple of arguments.

```
auto add = [](int x, int y) {
  return x + y;
};
std::apply(add, std::make_tuple(1, 2)); // == 3
```

std::filesystem

The new `std::filesystem` library provides a standard way to manipulate files, directories, and paths in a filesystem.

Here, a big file is copied to a temporary path if there is available space:

```
const auto bigFilePath {"bigFileToCopy"};
if (std::filesystem::exists(bigFilePath)) {
  const auto bigFileSize {std::filesystem::file_size(bigFilePath)};
  std::filesystem::path tmpPath {"/tmp"};
  if (std::filesystem::space(tmpPath).available > bigFileSize) {
    std::filesystem::create_directory(tmpPath.append("example"));
    std::filesystem::copy_file(bigFilePath, tmpPath.append("newFile"));
  }
}
```

std::byte

The new `std::byte` type provides a standard way of representing data as a byte. Benefits of using `std::byte` over `char` or `unsigned char` is that it is not a character type, and is also not an arithmetic type; while the only operator overloads available are bitwise operations.

```
std::byte a {0};
std::byte b {0xFF};
int i = std::to_integer<int>(b); // 0xFF
```

```
std::byte c = a & b;
int j = std::to_integer<int>(c); // 0
```

Note that `std::byte` is simply an enum, and braced initialization of enums become possible thanks to direct-list-initialization of enums.

Splicing for maps and sets

Moving nodes and merging containers without the overhead of expensive copies, moves, or heap allocations/deallocations.

Moving elements from one map to another:

```
std::map<int, string> src {{1, "one"}, {2, "two"}, {3, "buckle my shoe"}};
std::map<int, string> dst {{3, "three"}};
dst.insert(src.extract(src.find(1))); // Cheap remove and insert of { 1,
"one" } from `src` to `dst`.
dst.insert(src.extract(2)); // Cheap remove and insert of { 2, "two" } from
`src` to `dst`.
// dst == { { 1, "one" }, { 2, "two" }, { 3, "three" } };
```

Inserting an entire set:

```
std::set<int> src {1, 3, 5};
std::set<int> dst {2, 4, 5};
dst.merge(src);
// src == { 5 }
// dst == { 1, 2, 3, 4, 5 }
```

Inserting elements which outlive the container:

```
auto elementFactory() {
  std::set<...> s;
  s.emplace(...);
  return s.extract(s.begin());
}
s2.insert(elementFactory());
```

Changing the key of a map element:

```
std::map<int, string> m {{1, "one"}, {2, "two"}, {3, "three"}};
auto e = m.extract(2);
e.key() = 4;
m.insert(std::move(e));
// m == { { 1, "one" }, { 3, "three" }, { 4, "two" } }
```

Parallel algorithms

Many of the STL algorithms, such as the `copy`, `find` and `sort` methods, started to support the *parallel execution policies*: `seq`, `par` and `par_unseq` which translate to "sequentially", "parallel" and "parallel unsequenced".

```
std::vector<int> longVector;
// Find element using parallel execution policy
auto result1 = std::find(std::execution::par, std::begin(longVector),
std::end(longVector), 2);
// Sort elements using sequential execution policy
auto result2 = std::sort(std::execution::seq, std::begin(longVector),
std::end(longVector));
```