C++20/17/14/11

Overview

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

C++20 includes the following new language features: - concepts - designated initializers - template syntax for lambdas - range-based for loop with initializer - likely and unlikely attributes - deprecate implicit capture of this - class types in non-type template parameters - constexpr virtual functions - explicit(bool) - char8_t - immediate functions - using enum

C++20 includes the following new library features: - concepts library

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++14 includes the following new language features: - binary literals - generic lambda expressions - lambda capture initializers - return type deduction - decltype(auto) - relaxing constraints on constexpr functions - variable templates - [[deprecated]] attribute

C++14 includes the following new library features: - user-defined literals for standard library types - compile-time integer sequences - std::make_unique

C++11 includes the following new language features: - move semantics - variadic templates - rvalue references - forwarding references - initializer lists - static assertions - auto - lambda expressions - decltype - type aliases - nullptr - strongly-typed enums - attributes - constexpr - delegating constructors - user-defined literals - explicit virtual overrides - final specifier - default functions - deleted functions - range-based for loops - special member functions for move semantics - converting constructors - explicit conversion functions - inline-namespaces - non-static data member initializers - right angle brackets - ref-qualified member functions - trailing return types - noexcept specifier

C++11 includes the following new library features: - std::move - std::forward - std::thread - std::to_string - type traits - smart pointers - std::chrono - tuples - std::tie - std::array - unordered containers - std::make_shared - memory model - std::async - std::begin/end

C++20 Language Features

Concepts

Concepts are named compile-time predicates which constrain types. They take the following form:

```
template < template-parameter-list >
concept concept-name = constraint-expression;
```

where `constraint-expression` evaluates to a constexpr Boolean. *Constraints* should model semantic requirements, such as whether a type is a numeric or hashable. A compiler error results if a given type does not satisfy the concept it's bound by (i.e. `constraint-expression` returns `false`). Because constraints are evaluated at compile-time, they can provide more meaningful error messages and runtime safety.

```
// `T` is not limited by any constraints.
template <typename T>
concept AlwaysSatisfied = true;
// Limit `T` to integrals.
template <typename T>
concept Integral = std::is_integral_v<T>;
// Limit `T` to both the `Integral` constraint and signedness.
template <typename T>
concept SignedIntegral = Integral<T> && std::is_signed_v<T>;
// Limit `T` to both the `Integral` constraint and the negation of the `SignedIntegral` con
template <typename T>
concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
```

There are a variety of syntactic forms for enforcing concepts:

```
// Forms for function parameters:
// `T` is a constrained type template parameter.
template <MyConcept T>
void f(T v);

// `T` is a constrained type template parameter.
template <typename T>
  requires MyConcept<T>
void f(T v);

// `T` is a constrained type template parameter.
template <typename T>
void f(T v) requires MyConcept<T>;

// `v` is a constrained deduced parameter.
void f(MyConcept auto v);
```

```
// `v` is a constrained non-type template parameter.
template <MyConcept auto v>
void g();

// Forms for auto-deduced variables:
// `foo` is a constrained auto-deduced value.
MyConcept auto foo = ...;

// Forms for lambdas:
// `T` is a constrained type template parameter.
auto f = []<MyConcept T> (T v) {
  // ...
};
// `T` is a constrained type template parameter.
auto f = []<typename T> requires MyConcept<T> (T v) {
  // ...
};
// `T` is a constrained type template parameter.
auto f = []<typename T> (T v) requires MyConcept<T> {
  // ...
};
// `v` is a constrained deduced parameter.
auto f = [](MyConcept auto v) {
  // ...
};
// `v` is a constrained non-type template parameter.
auto g = []<MyConcept auto v> () {
  // ...
};
```

The `requires` keyword is used either to start a requires clause or a requires expression:

```
template <typename T>
  requires MyConcept<T> // `requires` clause.
void f(T);

template <typename T>
concept Callable = requires (T f) { f(); }; // `requires` expression.

template <typename T>
  requires requires (T x) { x + x; } // `requires` clause and expression on same line.
T add(T a, T b) {
  return a + b;
}
```

Note that the parameter list in a requires expression is optional. Each requirement in a requires expression are one of the following:

- **Simple requirements** - asserts that the given expression is valid.

```
template <typename T>
concept Callable = requires (T f) { f(); };
```

- **Type requirements** - denoted by the `typename` keyword followed by a type name, asserts that the given type name is valid.

```
struct Foo {
  int foo;
};

struct Bar {
  using value = int;
  value data;
};

struct Baz {
  using value = int;
  value data;
};

// Using SFINAE, enable if `T` is a `Baz`.
template <typename T, typename = std::enable_if_t<std::is_same_v<T, Baz>>>
struct S {};

template <typename T>
using Ref = T&;

template <typename T>
concept C = requires {
                     // Requirements on type `T`:
  typename T::value; // A) has an inner member named `value`
  typename S<T>;     // B) must have a valid class template specialization for `S`
  typename Ref<T>;   // C) must be a valid alias template substitution
};

template <C T>
void g(T a);

g(Foo{}); // ERROR: Fails requirement A.
g(Bar{}); // ERROR: Fails requirement B.
g(Baz{}); // PASS.
```

- **Compound requirements** - an expression in braces followed by a trailing return type or type constraint.

```
template <typename T>
concept C = requires(T x) {
  {*x} -> typename T::inner; // the type of the expression `*x` is convertible to `T::inner
  {x + 1} -> std::Same<int>; // the expression `x + 1` satisfies `std::Same<decltype((x + 1
  {x * 1} -> T; // the type of the expression `x * 1` is convertible to `T`
};
```

- **Nested requirements** - denoted by the `requires` keyword, specify additional constraints (such as those on local parameter arguments).

```
template <typename T>
concept C = requires(T x) {
  requires std::Same<sizeof(x), size_t>;
};
```

See also: concepts library.

Designated initializers

C-style designated initializer syntax. Any member fields that are not explicitly listed in the designated initializer list are default-initialized.

```
struct A {
  int x;
  int y;
  int z = 123;
};

A a {.x = 1, .z = 2}; // a.x == 1, a.y == 0, a.z == 2
```

Template syntax for lambdas

Use familiar template syntax in lambda expressions.

```
auto f = []<typename T>(std::vector<T> v) {
  // ...
};
```

Range-based for loop with initializer

This feature simplifies common code patterns, helps keep scopes tight, and offers an elegant solution to a common lifetime problem.

```
for (std::vector v{1, 2, 3}; auto& e : v) {
  std::cout << e;
}
// prints "123"
```

likely and unlikely attributes

Provides a hint to the optimizer that the labelled statement is likely/unlikely to have its body executed.

```
int random = get_random_number_between_x_and_y(0, 3);
[[likely]] if (random > 0) {
  // body of if statement
  // ...
}

[[unlikely]] while (unlikely_truthy_condition) {
  // body of while statement
  // ...
}
```

Deprecate implicit capture of this

Implicitly capturing `this` in a lamdba capture using `[=]` is now deprecated; prefer capturing explicitly using `[=, this]` or `[=, *this]`.

```
struct int_value {
  int n = 0;
  auto getter_fn() {
    // BAD:
    // return [=]() { return n; };

    // GOOD:
    return [=, *this]() { return n; };
  }
};
```

Class types in non-type template parameters

Classes can now be used in non-type template parameters. Objects passed in as template arguments have the type `const T`, where `T` is the type of the object, and has static storage duration.

```
struct foo {
  foo() = default;
  constexpr foo(int) {}
};
```

```
template <foo f>
auto get_foo() {
  return f;
}

get_foo(); // uses implicit constructor
get_foo<foo{123}>();
```

constexpr virtual functions

Virtual functions can now be `constexpr` and evaluated at compile-time. `constexpr` virtual functions can override non-`constexpr` virtual functions and vice-versa.

```
struct X1 {
  virtual int f() const = 0;
};

struct X2: public X1 {
  constexpr virtual int f() const { return 2; }
};

struct X3: public X2 {
  virtual int f() const { return 3; }
};

struct X4: public X3 {
  constexpr virtual int f() const { return 4; }
};

constexpr X4 x4;
x4.f(); // == 4
```

explicit(bool)

Conditionally select at compile-time whether a constructor is made explicit or not. `explicit(true)` is the same as specifying `explicit`.

```
struct foo {
  // Specify non-integral types (strings, floats, etc.) require explicit construction.
  template <typename T>
  explicit(!std::is_integral_v<T>) foo(T) {}
};
```

```
foo a = 123; // OK
foo b = "123"; // ERROR: explicit constructor is not a candidate (explicit specifier evalua
foo c {"123"}; // OK
```

char8_t

Provides a standard type for representing UTF-8 strings.

```
char8_t utf8_str[] = u8"\u0123";
```

Immediate functions

Similar to `constexpr` functions, but functions with a `consteval` specifier must produce a constant. These are called `immediate functions`.

```
consteval int sqr(int n) {
  return n * n;
}

constexpr int r = sqr(100); // OK
int x = 100;
int r2 = sqr(x); // ERROR: the value of 'x' is not usable in a constant expression
                 // OK if `sqr` were a `constexpr` function
```

using enum

Bring an enum's members into scope to improve readability. Before:

```
enum class rgba_color_channel { red, green, blue, alpha };

std::string_view to_string(rgba_color_channel channel) {
  switch (channel) {
    case rgba_color_channel::red:   return "red";
    case rgba_color_channel::green: return "green";
    case rgba_color_channel::blue:  return "blue";
    case rgba_color_channel::alpha: return "alpha";
  }
}
```

After:

```
enum class rgba_color_channel { red, green, blue, alpha };

std::string_view to_string(rgba_color_channel channel) {
  switch (my_channel) {
    using enum rgba_color_channel;
    case red:   return "red";
    case green: return "green";
    case blue:  return "blue";
    case alpha: return "alpha";
  }
}
```

C++20 Library Features

Concepts library

Concepts are also provided by the standard library for building more complicated concepts. Some of these include:

Core language concepts: - `Same` - specifies two types are the same. - `DerivedFrom` - specifies that a type is derived from another type. - `ConvertibleTo` - specifies that a type is implicitly convertible to another type. - `Common` - specifies that two types share a common type. - `Integral` - specifies that a type is an integral type. - `DefaultConstructible` - specifies that an object of a type can be default-constructed.

Comparison concepts: - `Boolean` - specifies that a type can be used in Boolean contexts. - `EqualityComparable` - specifies that `operator==` is an equivalence relation.

Object concepts: - `Movable` - specifies that an object of a type can be moved and swapped. - `Copyable` - specifies that an object of a type can be copied, moved, and swapped. - `Semiregular` - specifies that an object of a type can be copied, moved, swapped, and default constructed. - `Regular` - specifies that a type is *regular*, that is, it is both `Semiregular` and `EqualityComparable`.

Callable concepts: - `Invocable` - specifies that a callable type can be invoked with a given set of argument types. - `Predicate` - specifies that a callable type is a Boolean predicate.

See also: concepts.

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;
    }
  }
}
// vs.
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. "'c++ [[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' c++ // 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"};
```

```cpp
// 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));
```

```cpp
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.

```cpp
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.

```cpp
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`
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),
// Sort elements using sequential execution policy
auto result2 = std::sort(std::execution::seq, std::begin(longVector), std::end(longVector));
```

C++14 Language Features

Binary literals

Binary literals provide a convenient way to represent a base-2 number. It is possible to separate digits with `'`.

```
0b110 // == 6
0b1111'1111 // == 255
```

Generic lambda expressions

C++14 now allows the `auto` type-specifier in the parameter list, enabling polymorphic lambdas.

```
auto identity = [](auto x) { return x; };
int three = identity(3); // == 3
std::string foo = identity("foo"); // == "foo"
```

Lambda capture initializers

This allows creating lambda captures initialized with arbitrary expressions. The name given to the captured value does not need to be related to any variables in the enclosing scopes and introduces a new name inside the lambda body. The initializing expression is evaluated when the lambda is *created* (not when it is *invoked*).

```
int factory(int i) { return i * 10; }
auto f = [x = factory(2)] { return x; }; // returns 20

auto generator = [x = 0] () mutable {
  // this would not compile without 'mutable' as we are modifying x on each call
  return x++;
};
auto a = generator(); // == 0
auto b = generator(); // == 1
auto c = generator(); // == 2
```

Because it is now possible to *move* (or *forward*) values into a lambda that could previously be only captured by copy or reference we can now capture move-only types in a lambda by value. Note that in the below example the `p` in the capture-list of `task2` on the left-hand-side of `=` is a new variable private to the lambda body and does not refer to the original `p`.

```
auto p = std::make_unique<int>(1);

auto task1 = [=] { *p = 5; }; // ERROR: std::unique_ptr cannot be copied
// vs.
auto task2 = [p = std::move(p)] { *p = 5; }; // OK: p is move-constructed into the closure
// the original p is empty after task2 is created
```

Using this reference-captures can have different names than the referenced variable.

```
auto x = 1;
auto f = [&r = x, x = x * 10] {
  ++r;
  return r + x;
};
f(); // sets x to 2 and returns 12
```

Return type deduction

Using an `auto` return type in C++14, the compiler will attempt to deduce the type for you. With lambdas, you can now deduce its return type using `auto`, which makes returning a deduced reference or rvalue reference possible.

```
// Deduce return type as `int`.
auto f(int i) {
 return i;
}

template <typename T>
auto& f(T& t) {
  return t;
}

// Returns a reference to a deduced type.
auto g = [](auto& x) -> auto& { return f(x); };
int y = 123;
int& z = g(y); // reference to `y`
```

decltype(auto)

The `decltype(auto)` type-specifier also deduces a type like `auto` does. However, it deduces return types while keeping their references and cv-qualifiers, while `auto` will not.

```
const int x = 0;
auto x1 = x; // int
decltype(auto) x2 = x; // const int
int y = 0;
int& y1 = y;
auto y2 = y1; // int
decltype(auto) y3 = y1; // int&
int&& z = 0;
auto z1 = std::move(z); // int
decltype(auto) z2 = std::move(z); // int&&

// Note: Especially useful for generic code!

// Return type is `int`.
auto f(const int& i) {
 return i;
}

// Return type is `const int&`.
decltype(auto) g(const int& i) {
 return i;
}

int x = 123;
static_assert(std::is_same<const int&, decltype(f(x))>::value == 0);
static_assert(std::is_same<int, decltype(f(x))>::value == 1);
static_assert(std::is_same<const int&, decltype(g(x))>::value == 1);
```

See also: `decltype`.

Relaxing constraints on constexpr functions

In C++11, `constexpr` function bodies could only contain a very limited set of syntaxes, including (but not limited to): `typedef`s, `using`s, and a single `return` statement. In C++14, the set of allowable syntaxes expands greatly to include the most common syntax such as `if` statements, multiple `return`s, loops, etc.

```
constexpr int factorial(int n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
factorial(5); // == 120
```

Variable templates

C++14 allows variables to be templated:

```
template<class T>
constexpr T pi = T(3.1415926535897932385);
template<class T>
constexpr T e  = T(2.7182818284590452353);
```

[[deprecated]] attribute

C++14 introduces the `[[deprecated]]` attribute to indicate that a unit (function, class, etc) is discouraged and likely yield compilation warnings. If a reason is provided, it will be included in the warnings.

```
[[deprecated]]
void old_method();

[[deprecated("Use new_method instead")]]
void legacy_method();
```

C++14 Library Features

User-defined literals for standard library types

New user-defined literals for standard library types, including new built-in literals for `chrono` and `basic_string`. These can be `constexpr` meaning they can be used at compile-time. Some uses for these literals include compile-time integer parsing, binary literals, and imaginary number literals.

```
using namespace std::chrono_literals;
auto day = 24h;
day.count(); // == 24
std::chrono::duration_cast<std::chrono::minutes>(day).count(); // == 1440
```

Compile-time integer sequences

The class template `std::integer_sequence` represents a compile-time sequence of integers. There are a few helpers built on top: * `std::make_integer_sequence<T, N...>` - creates a sequence of `0, ..., N - 1` with type `T`. * `std::index_sequence_for<T...>` - converts a template parameter pack into an integer sequence.

Convert an array into a tuple:

```
template<typename Array, std::size_t... I>
decltype(auto) a2t_impl(const Array& a, std::integer_sequence<std::size_t, I...>) {
  return std::make_tuple(a[I]...);
}

template<typename T, std::size_t N, typename Indices = std::make_index_sequence<N>>
decltype(auto) a2t(const std::array<T, N>& a) {
  return a2t_impl(a, Indices());
}
```

std::make_unique

`std::make_unique` is the recommended way to create instances of `std::unique_ptr`s due to the following reasons: * Avoid having to use the `new` operator. * Prevents code repetition when specifying the underlying type the pointer shall hold. * Most importantly, it provides exception-safety. Suppose we were calling a function `foo` like so:

```
foo(std::unique_ptr<T>{new T{}}, function_that_throws(), std::unique_ptr<T>{new T{}});
```

The compiler is free to call `new T{}`, then `function_that_throws()`, and so on… Since we have allocated data on the heap in the first construction of a `T`, we have introduced a leak here. With `std::make_unique`, we are given exception-safety:

```
foo(std::make_unique<T>(), function_that_throws(), std::make_unique<T>());
```

See the section on smart pointers for more information on `std::unique_ptr` and `std::shared_ptr`.

C++11 Language Features

Move semantics

Move semantics is mostly about performance optimization: the ability to move an object without the expensive overhead of copying. The difference between a copy and a move is that a copy leaves the source unchanged, and a move will leave the source either unchanged or radically different – depending on what the source is. For plain old data, a move is the same as a copy.

To move an object means to transfer ownership of some resource it manages to another object. You could think of this as changing pointers held by the source object to be moved, or now held, by the destination object; the resource remains in its location in memory. Such an inexpensive transfer of resources is extremely useful when the source is an `rvalue`, where the potentially dangerous side-effect of changing the source after the move is redundant since the source is a temporary object that won't be accessible later.

Moves also make it possible to transfer objects such as `std::unique_ptr`s, smart pointers that are designed to hold a pointer to a unique object, from one scope to another.

See the sections on: rvalue references, defining move special member functions, `std::move`, `std::forward`, `forwarding references`.

Rvalue references

C++11 introduces a new reference termed the *rvalue reference*. An rvalue reference to `A`, which is a non-template type parameter (such as `int`, or a user-defined type), is created with the syntax `A&&`. Rvalue references only bind to rvalues.

Type deduction with lvalues and rvalues:

```cpp
int x = 0; // `x` is an lvalue of type `int`
int& xl = x; // `xl` is an lvalue of type `int&`
int&& xr = x; // compiler error -- `x` is an lvalue
int&& xr2 = 0; // `xr2` is an lvalue of type `int&&` -- binds to the rvalue temporary, `0`
```

See also: `std::move`, `std::forward`, `forwarding references`.

Forwarding references

Also known (unofficially) as *universal references*. A forwarding reference is created with the syntax `T&&` where `T` is a template type parameter, or using `auto&&`. This enables two major features: move semantics; and *perfect forwarding*, the ability to pass arguments that are either lvalues or rvalues.

Forwarding references allow a reference to bind to either an lvalue or rvalue depending on the type. Forwarding references follow the rules of *reference collapsing*: * `T& &` becomes `T&` * `T& &&` becomes `T&` * `T&& &` becomes `T&` * `T&& &&` becomes `T&&`

`auto` type deduction with lvalues and rvalues:

```cpp
int x = 0; // `x` is an lvalue of type `int`
auto&& al = x; // `al` is an lvalue of type `int&` -- binds to the lvalue, `x`
auto&& ar = 0; // `ar` is an lvalue of type `int&&` -- binds to the rvalue temporary, `0`
```

Template type parameter deduction with lvalues and rvalues:

```cpp
// Since C++14 or later:
void f(auto&& t) {
  // ...
}

// Since C++11 or later:
template <typename T>
void f(T&& t) {
  // ...
}

int x = 0;
f(0); // deduces as f(int&&)
f(x); // deduces as f(int&)

int& y = x;
f(y); // deduces as f(int& &&) => f(int&)

int&& z = 0; // NOTE: `z` is an lvalue with type `int&&`.
f(z); // deduces as f(int&& &) => f(int&)
f(std::move(z)); // deduces as f(int&& &&) => f(int&&)
```

See also: `std::move`, `std::forward`, `rvalue references`.

Variadic templates

The `...` syntax creates a *parameter pack* or expands one. A template *parameter pack* is a template parameter that accepts zero or more template arguments (non-types, types, or templates). A template with at least one parameter pack is called a *variadic template*.

```
template <typename... T>
struct arity {
  constexpr static int value = sizeof...(T);
};
static_assert(arity<>::value == 0);
static_assert(arity<char, short, int>::value == 3);
```

An interesting use for this is creating an *initializer list* from a *parameter pack* in order to iterate over variadic function arguments.

```
template <typename First, typename... Args>
auto sum(const First first, const Args... args) -> decltype(first) {
  const auto values = {first, args...};
  return std::accumulate(values.begin(), values.end(), First{0});
}

sum(1, 2, 3, 4, 5); // 15
sum(1, 2, 3);       // 6
sum(1.5, 2.0, 3.7); // 7.2
```

Initializer lists

A lightweight array-like container of elements created using a “braced list” syntax. For example, `{ 1, 2, 3 }` creates a sequences of integers, that has type `std::initializer_list<int>`. Useful as a replacement to passing a vector of objects to a function.

```
int sum(const std::initializer_list<int>& list) {
  int total = 0;
  for (auto& e : list) {
    total += e;
  }

  return total;
}

auto list = {1, 2, 3};
```

```
sum(list); // == 6
sum({1, 2, 3}); // == 6
sum({}); // == 0
```

Static assertions

Assertions that are evaluated at compile-time.

```
constexpr int x = 0;
constexpr int y = 1;
static_assert(x == y, "x != y");
```

auto

`auto`-typed variables are deduced by the compiler according to the type of their initializer.

```
auto a = 3.14; // double
auto b = 1; // int
auto& c = b; // int&
auto d = { 0 }; // std::initializer_list<int>
auto&& e = 1; // int&&
auto&& f = b; // int&
auto g = new auto(123); // int*
const auto h = 1; // const int
auto i = 1, j = 2, k = 3; // int, int, int
auto l = 1, m = true, n = 1.61; // error -- `l` deduced to be int, `m` is bool
auto o; // error -- `o` requires initializer
```

Extremely useful for readability, especially for complicated types:

```
std::vector<int> v = ...;
std::vector<int>::const_iterator cit = v.cbegin();
// vs.
auto cit = v.cbegin();
```

Functions can also deduce the return type using `auto`. In C++11, a return type must be specified either explicitly, or using `decltype` like so:

```
template <typename X, typename Y>
auto add(X x, Y y) -> decltype(x + y) {
  return x + y;
}
add(1, 2); // == 3
add(1, 2.0); // == 3.0
add(1.5, 1.5); // == 3.0
```

The trailing return type in the above example is the *declared type* (see section on decltype) of the expression x + y. For example, if x is an integer and y is a double, decltype(x + y) is a double. Therefore, the above function will deduce the type depending on what type the expression x + y yields. Notice that the trailing return type has access to its parameters, and this when appropriate.

Lambda expressions

A lambda is an unnamed function object capable of capturing variables in scope. It features: a *capture list*; an optional set of parameters with an optional trailing return type; and a body. Examples of capture lists: * [] - captures nothing. * [=] - capture local objects (local variables, parameters) in scope by value. * [&] - capture local objects (local variables, parameters) in scope by reference. * [this] - capture this pointer by value. * [a, &b] - capture objects a by value, b by reference.

```
int x = 1;

auto getX = [=] { return x; };
getX(); // == 1

auto addX = [=](int y) { return x + y; };
addX(1); // == 2

auto getXRef = [&]() -> int& { return x; };
getXRef(); // int& to `x`
```

By default, value-captures cannot be modified inside the lambda because the compiler-generated method is marked as const. The mutable keyword allows modifying captured variables. The keyword is placed after the parameter-list (which must be present even if it is empty).

```
int x = 1;

auto f1 = [&x] { x = 2; }; // OK: x is a reference and modifies the original

auto f2 = [x] { x = 2; }; // ERROR: the lambda can only perform const-operations on the cap
// vs.
auto f3 = [x]() mutable { x = 2; }; // OK: the lambda can perform any operations on the cap
```

decltype

decltype is an operator which returns the *declared type* of an expression passed to it. cv-qualifiers and references are maintained if they are part of the expression. Examples of decltype:

```
int a = 1; // `a` is declared as type `int`
decltype(a) b = a; // `decltype(a)` is `int`
const int& c = a; // `c` is declared as type `const int&`
decltype(c) d = a; // `decltype(c)` is `const int&`
decltype(123) e = 123; // `decltype(123)` is `int`
int&& f = 1; // `f` is declared as type `int&&`
decltype(f) g = 1; // `decltype(f) is `int&&`
decltype((a)) h = g; // `decltype((a))` is int&
```

```
template <typename X, typename Y>
auto add(X x, Y y) -> decltype(x + y) {
  return x + y;
}
add(1, 2.0); // `decltype(x + y)` => `decltype(3.0)` => `double`
```

See also: `decltype(auto)`.

Type aliases

Semantically similar to using a `typedef` however, type aliases with `using` are easier to read and are compatible with templates.

```
template <typename T>
using Vec = std::vector<T>;
Vec<int> v; // std::vector<int>

using String = std::string;
String s {"foo"};
```

nullptr

C++11 introduces a new null pointer type designed to replace C's `NULL` macro. `nullptr` itself is of type `std::nullptr_t` and can be implicitly converted into pointer types, and unlike `NULL`, not convertible to integral types except `bool`.

```
void foo(int);
void foo(char*);
foo(NULL); // error -- ambiguous
foo(nullptr); // calls foo(char*)
```

Strongly-typed enums

Type-safe enums that solve a variety of problems with C-style enums including: implicit conversions, inability to specify the underlying type, scope pollution.

```
// Specifying underlying type as `unsigned int`
enum class Color : unsigned int { Red = 0xff0000, Green = 0xff00, Blue = 0xff };
// `Red`/`Green` in `Alert` don't conflict with `Color`
enum class Alert : bool { Red, Green };
Color c = Color::Red;
```

Attributes

Attributes provide a universal syntax over `__attribute__(...)`, `__declspec`, etc.

```
// `noreturn` attribute indicates `f` doesn't return.
[[ noreturn ]] void f() {
  throw "error";
}
```

constexpr

Constant expressions are expressions evaluated by the compiler at compile-time. Only non-complex computations can be carried out in a constant expression. Use the `constexpr` specifier to indicate the variable, function, etc. is a constant expression.

```
constexpr int square(int x) {
  return x * x;
}

int square2(int x) {
  return x * x;
}

int a = square(2);  // mov DWORD PTR [rbp-4], 4

int b = square2(2); // mov edi, 2
                    // call square2(int)
                    // mov DWORD PTR [rbp-8], eax
```

`constexpr` values are those that the compiler can evaluate at compile-time:

```
const int x = 123;
constexpr const int& y = x; // error -- constexpr variable `y` must be initialized by a cons
```

Constant expressions with classes:

```
struct Complex {
  constexpr Complex(double r, double i) : re{r}, im{i} { }
  constexpr double real() { return re; }
  constexpr double imag() { return im; }

private:
  double re;
  double im;
};

constexpr Complex I(0, 1);
```

Delegating constructors

Constructors can now call other constructors in the same class using an initializer list.

```
struct Foo {
  int foo;
  Foo(int foo) : foo{foo} {}
  Foo() : Foo(0) {}
};

Foo foo;
foo.foo; // == 0
```

User-defined literals

User-defined literals allow you to extend the language and add your own syntax. To create a literal, define a `T operator "" X(...) { ... }` function that returns a type `T`, with a name `X`. Note that the name of this function defines the name of the literal. Any literal names not starting with an underscore are reserved and won't be invoked. There are rules on what parameters a user-defined literal function should accept, according to what type the literal is called on.

Converting Celsius to Fahrenheit:

```
// `unsigned long long` parameter required for integer literal.
long long operator "" _celsius(unsigned long long tempCelsius) {
  return std::llround(tempCelsius * 1.8 + 32);
}
24_celsius; // == 75
```

String to integer conversion:

```
// `const char*` and `std::size_t` required as parameters.
int operator "" _int(const char* str, std::size_t) {
  return std::stoi(str);
}
```

```
"123"_int; // == 123, with type `int`
```

Explicit virtual overrides

Specifies that a virtual function overrides another virtual function. If the virtual function does not override a parent's virtual function, throws a compiler error.

```
struct A {
  virtual void foo();
  void bar();
};

struct B : A {
  void foo() override; // correct -- B::foo overrides A::foo
  void bar() override; // error -- A::bar is not virtual
  void baz() override; // error -- B::baz does not override A::baz
};
```

Final specifier

Specifies that a virtual function cannot be overridden in a derived class or that a class cannot be inherited from.

```
struct A {
  virtual void foo();
};

struct B : A {
  virtual void foo() final;
};

struct C : B {
  virtual void foo(); // error -- declaration of 'foo' overrides a 'final' function
};
```

Class cannot be inherited from.

```
struct A final {};
struct B : A {}; // error -- base 'A' is marked 'final'
```

Default functions

A more elegant, efficient way to provide a default implementation of a function, such as a constructor.

```
struct A {
  A() = default;
  A(int x) : x{x} {}
  int x {1};
};
A a; // a.x == 1
A a2 {123}; // a.x == 123
```

With inheritance:

```
struct B {
  B() : x{1} {}
  int x;
};

struct C : B {
  // Calls B::B
  C() = default;
};

C c; // c.x == 1
```

Deleted functions

A more elegant, efficient way to provide a deleted implementation of a function. Useful for preventing copies on objects.

```
class A {
  int x;

public:
  A(int x) : x{x} {};
  A(const A&) = delete;
  A& operator=(const A&) = delete;
};

A x {123};
A y = x; // error -- call to deleted copy constructor
y = x; // error -- operator= deleted
```

Range-based for loops

Syntactic sugar for iterating over a container's elements.

```
std::array<int, 5> a {1, 2, 3, 4, 5};
for (int& x : a) x *= 2;
```

```
// a == { 2, 4, 6, 8, 10 }
```

Note the difference when using `int` as opposed to `int&`:

```
std::array<int, 5> a {1, 2, 3, 4, 5};
for (int x : a) x *= 2;
// a == { 1, 2, 3, 4, 5 }
```

Special member functions for move semantics

The copy constructor and copy assignment operator are called when copies are made, and with C++11's introduction of move semantics, there is now a move constructor and move assignment operator for moves.

```
struct A {
  std::string s;
  A() : s{"test"} {}
  A(const A& o) : s{o.s} {}
  A(A&& o) : s{std::move(o.s)} {}
  A& operator=(A&& o) {
   s = std::move(o.s);
   return *this;
  }
};

A f(A a) {
  return a;
}

A a1 = f(A{}); // move-constructed from rvalue temporary
A a2 = std::move(a1); // move-constructed using std::move
A a3 = A{};
a2 = std::move(a3); // move-assignment using std::move
a1 = f(A{}); // move-assignment from rvalue temporary
```

Converting constructors

Converting constructors will convert values of braced list syntax into constructor arguments.

```
struct A {
  A(int) {}
  A(int, int) {}
  A(int, int, int) {}
};
```

```
A a {0, 0}; // calls A::A(int, int)
A b(0, 0); // calls A::A(int, int)
A c = {0, 0}; // calls A::A(int, int)
A d {0, 0, 0}; // calls A::A(int, int, int)
```

Note that the braced list syntax does not allow narrowing:

```
struct A {
  A(int) {}
};

A a(1.1); // OK
A b {1.1}; // Error narrowing conversion from double to int
```

Note that if a constructor accepts a `std::initializer_list`, it will be called instead:

```
struct A {
  A(int) {}
  A(int, int) {}
  A(int, int, int) {}
  A(std::initializer_list<int>) {}
};

A a {0, 0}; // calls A::A(std::initializer_list<int>)
A b(0, 0); // calls A::A(int, int)
A c = {0, 0}; // calls A::A(std::initializer_list<int>)
A d {0, 0, 0}; // calls A::A(std::initializer_list<int>)
```

Explicit conversion functions

Conversion functions can now be made explicit using the `explicit` specifier.

```
struct A {
  operator bool() const { return true; }
};

struct B {
  explicit operator bool() const { return true; }
};

A a;
if (a); // OK calls A::operator bool()
bool ba = a; // OK copy-initialization selects A::operator bool()

B b;
```

```
if (b); // OK calls B::operator bool()
bool bb = b; // error copy-initialization does not consider B::operator bool()
```

Inline namespaces

All members of an inline namespace are treated as if they were part of its parent namespace, allowing specialization of functions and easing the process of versioning. This is a transitive property, if A contains B, which in turn contains C and both B and C are inline namespaces, C's members can be used as if they were on A.

```
namespace Program {
  namespace Version1 {
    int getVersion() { return 1; }
    bool isFirstVersion() { return true; }
  }
  inline namespace Version2 {
    int getVersion() { return 2; }
  }
}

int version {Program::getVersion()};              // Uses getVersion() from Version2
int oldVersion {Program::Version1::getVersion()}; // Uses getVersion() from Version1
bool firstVersion {Program::isFirstVersion()};    // Does not compile when Version2 is adde
```

Non-static data member initializers

Allows non-static data members to be initialized where they are declared, potentially cleaning up constructors of default initializations.

```
// Default initialization prior to C++11
class Human {
    Human() : age{0} {}
  private:
    unsigned age;
};
// Default initialization on C++11
class Human {
  private:
    unsigned age {0};
};
```

Right angle brackets

C++11 is now able to infer when a series of right angle brackets is used as an operator or as a closing statement of typedef, without having to add whitespace.

```
typedef std::map<int, std::map <int, std::map <int, int> > > cpp98LongTypedef;
typedef std::map<int, std::map <int, std::map <int, int>>>   cpp11LongTypedef;
```

Ref-qualified member functions

Member functions can now be qualified depending on whether `*this` is an lvalue or rvalue reference.

```
struct Bar {
  // ...
};

struct Foo {
  Bar getBar() & { return bar; }
  Bar getBar() const& { return bar; }
  Bar getBar() && { return std::move(bar); }
  Bar getBar() const&& { return std::move(bar); }
private:
  Bar bar;
};

Foo foo{};
Bar bar = foo.getBar(); // calls `Bar getBar() &`

const Foo foo2{};
Bar bar2 = foo2.getBar(); // calls `Bar Foo::getBar() const&`

Foo{}.getBar(); // calls `Bar Foo::getBar() &&`
std::move(foo).getBar(); // calls `Bar Foo::getBar() &&`

std::move(foo2).getBar(); // calls `Bar Foo::getBar() const&&`
```

Trailing return types

C++11 allows functions and lambdas an alternative syntax for specifying their return types.

```
int f() {
  return 123;
}
// vs.
```

```
auto f() -> int {
  return 123;
}
```

```
auto g = []() -> int {
  return 123;
};
```

This feature is especially useful when certain return types cannot be resolved:

```
// NOTE: This does not compile!
template <typename T, typename U>
decltype(a + b) add(T a, U b) {
    return a + b;
}

// Trailing return types allows this:
template <typename T, typename U>
auto add(T a, U b) -> decltype(a + b) {
    return a + b;
}
```

In C++14, decltype(auto) can be used instead.

Noexcept specifier

The `noexcept` specifier specifies whether a function could throw exceptions. It is an improved version of `throw()`.

```
void func1() noexcept;        // does not throw
void func2() noexcept(true);  // does not throw
void func3() throw();         // does not throw

void func4() noexcept(false); // may throw
```

Non-throwing functions are permitted to call potentially-throwing functions. Whenever an exception is thrown and the search for a handler encounters the outermost block of a non-throwing function, the function std::terminate is called.

```
extern void f();  // potentially-throwing
void g() noexcept {
    f();          // valid, even if f throws
    throw 42;     // valid, effectively a call to std::terminate
}
```

C++11 Library Features

std::move

`std::move` indicates that the object passed to it may have its resources transferred. Moves can often be more efficient than copies. Using objects that have been moved from should be used with care, as they can be left in an unspecified state (see: What can I do with a moved-from object?).

A definition of `std::move` (performing a move is nothing more than casting to an rvalue):

```
template <typename T>
typename remove_reference<T>::type&& move(T&& arg) {
  return static_cast<typename remove_reference<T>::type&&>(arg);
}
```

Transferring `std::unique_ptr`s:

```
std::unique_ptr<int> p1 {new int{0}}; // in practice, use std::make_unique
std::unique_ptr<int> p2 = p1; // error -- cannot copy unique pointers
std::unique_ptr<int> p3 = std::move(p1); // move `p1` into `p3`
                                         // now unsafe to dereference object held by `p1`
```

std::forward

Returns the arguments passed to it as-is, either as an lvalue or rvalue references, and includes cv-qualification. Useful for generic code that need a reference (either lvalue or rvalue) when appropriate, e.g factories. Used in conjunction with `forwarding references`.

A definition of `std::forward`:

```
template <typename T>
T&& forward(typename remove_reference<T>::type& arg) {
  return static_cast<T&&>(arg);
}
```

An example of a function `wrapper` which just forwards other `A` objects to a new `A` object's copy or move constructor:

```
struct A {
  A() = default;
  A(const A& o) { std::cout << "copied" << std::endl; }
  A(A&& o) { std::cout << "moved" << std::endl; }
};

template <typename T>
A wrapper(T&& arg) {
  return A{std::forward<T>(arg)};
}

wrapper(A{}); // moved
A a;
wrapper(a); // copied
wrapper(std::move(a)); // moved
```

See also: `forwarding references`, `rvalue references`.

std::thread

The `std::thread` library provides a standard way to control threads, such as spawning and killing them. In the example below, multiple threads are spawned to do different calculations and then the program waits for all of them to finish.

```
void foo(bool clause) { /* do something... */ }

std::vector<std::thread> threadsVector;
threadsVector.emplace_back([]() {
  // Lambda function that will be invoked
});
threadsVector.emplace_back(foo, true);  // thread will run foo(true)
for (auto& thread : threadsVector) {
  thread.join(); // Wait for threads to finish
}
```

std::to_string

Converts a numeric argument to a `std::string`.

```
std::to_string(1.2); // == "1.2"
std::to_string(123); // == "123"
```

Type traits

Type traits defines a compile-time template-based interface to query or modify the properties of types.

```
static_assert(std::is_integral<int>::value);
static_assert(std::is_same<int, int>::value);
static_assert(std::is_same<std::conditional<true, int, double>::type, int>::value);
```

Smart pointers

C++11 introduces new smart(er) pointers: `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`. `std::auto_ptr` now becomes deprecated and then eventually removed in C++17.

`std::unique_ptr` is a non-copyable, movable smart pointer that properly manages arrays and STL containers. **Note: Prefer using the `std::make_X` helper functions as opposed to using constructors. See the sections for std::make_unique and std::make_shared.**

```
std::unique_ptr<Foo> p1 {new Foo{}};  // `p1` owns `Foo`
if (p1) {
  p1->bar();
}

{
  std::unique_ptr<Foo> p2 {std::move(p1)};  // Now `p2` owns `Foo`
  f(*p2);

  p1 = std::move(p2);  // Ownership returns to `p1` -- `p2` gets destroyed
}

if (p1) {
  p1->bar();
}
// `Foo` instance is destroyed when `p1` goes out of scope
```

A `std::shared_ptr` is a smart pointer that manages a resource that is shared across multiple owners. A shared pointer holds a *control block* which has a few components such as the managed object and a reference counter. All control block access is thread-safe, however, manipulating the managed object itself is *not* thread-safe.

```
void foo(std::shared_ptr<T> t) {
  // Do something with `t`...
}

void bar(std::shared_ptr<T> t) {
  // Do something with `t`...
}

void baz(std::shared_ptr<T> t) {
  // Do something with `t`...
}

std::shared_ptr<T> p1 {new T{}};
```

```
// Perhaps these take place in another threads?
foo(p1);
bar(p1);
baz(p1);
```

std::chrono

The chrono library contains a set of utility functions and types that deal with *durations*, *clocks*, and *time points*. One use case of this library is benchmarking code:

```
std::chrono::time_point<std::chrono::steady_clock> start, end;
start = std::chrono::steady_clock::now();
// Some computations...
end = std::chrono::steady_clock::now();

std::chrono::duration<double> elapsed_seconds = end - start;
double t = elapsed_seconds.count(); // t number of seconds, represented as a `double`
```

Tuples

Tuples are a fixed-size collection of heterogeneous values. Access the elements of a `std::tuple` by unpacking using `std::tie`, or using `std::get`.

```
// `playerProfile` has type `std::tuple<int, const char*, const char*>`.
auto playerProfile = std::make_tuple(51, "Frans Nielsen", "NYI");
std::get<0>(playerProfile); // 51
std::get<1>(playerProfile); // "Frans Nielsen"
std::get<2>(playerProfile); // "NYI"
```

std::tie

Creates a tuple of lvalue references. Useful for unpacking `std::pair` and `std::tuple` objects. Use `std::ignore` as a placeholder for ignored values. In C++17, structured bindings should be used instead.

```
// With tuples...
std::string playerName;
std::tie(std::ignore, playerName, std::ignore) = std::make_tuple(91, "John Tavares", "NYI")

// With pairs...
std::string yes, no;
std::tie(yes, no) = std::make_pair("yes", "no");
```

std::array

`std::array` is a container built on top of a C-style array. Supports common container operations such as sorting.

```
std::array<int, 3> a = {2, 1, 3};
std::sort(a.begin(), a.end()); // a == { 1, 2, 3 }
for (int& x : a) x *= 2; // a == { 2, 4, 6 }
```

Unordered containers

These containers maintain average constant-time complexity for search, insert, and remove operations. In order to achieve constant-time complexity, sacrifices order for speed by hashing elements into buckets. There are four unordered containers: * `unordered_set` * `unordered_multiset` * `unordered_map` * `unordered_multimap`

std::make_shared

`std::make_shared` is the recommended way to create instances of `std::shared_ptrs` due to the following reasons: * Avoid having to use the `new` operator. * Prevents code repetition when specifying the underlying type the pointer shall hold. * It provides exception-safety. Suppose we were calling a function `foo` like so:

```
foo(std::shared_ptr<T>{new T{}}, function_that_throws(), std::shared_ptr<T>{new T{}});
```

The compiler is free to call `new T{}`, then `function_that_throws()`, and so on... Since we have allocated data on the heap in the first construction of a `T`, we have introduced a leak here. With `std::make_shared`, we are given exception-safety:

```
foo(std::make_shared<T>(), function_that_throws(), std::make_shared<T>());
```

- Prevents having to do two allocations. When calling `std::shared_ptr{ new T{} }`, we have to allocate memory for `T`, then in the shared pointer we have to allocate memory for the control block within the pointer.

See the section on smart pointers for more information on `std::unique_ptr` and `std::shared_ptr`.

Memory model

C++11 introduces a memory model for C++, which means library support for threading and atomic operations. Some of these operations include (but aren't limited to) atomic loads/stores, compare-and-swap, atomic flags, promises, futures, locks, and condition variables.

See the sections on: std::thread

std::async

`std::async` runs the given function either asynchronously or lazily-evaluated, then returns a `std::future` which holds the result of that function call.

The first parameter is the policy which can be: 1. `std::launch::async | std::launch::deferred` It is up to the implementation whether to perform asynchronous execution or lazy evaluation. 1. `std::launch::async` Run the callable object on a new thread. 1. `std::launch::deferred` Perform lazy evaluation on the current thread.

```
int foo() {
  /* Do something here, then return the result. */
  return 1000;
}

auto handle = std::async(std::launch::async, foo);  // create an async task
auto result = handle.get();  // wait for the result
```

std::begin/end

`std::begin` and `std::end` free functions were added to return begin and end iterators of a container generically. These functions also work with raw arrays which do not have begin and end member functions.

```
template <typename T>
int CountTwos(const T& container) {
  return std::count_if(std::begin(container), std::end(container), [](int item) {
    return item == 2;
  });
}

std::vector<int> vec = {2, 2, 43, 435, 4543, 534};
int arr[8] = {2, 43, 45, 435, 32, 32, 32, 32};
auto a = CountTwos(vec); // 2
auto b = CountTwos(arr);  // 1
```

Acknowledgements

- cppreference - especially useful for finding examples and documentation of

new library features.
- C++ Rvalue References Explained - a great introduction I used to understand rvalue references, perfect forwarding, and move semantics.
- clang and gcc's standards support pages. Also included here are the proposals for language/library features that I used to help find a description of, what it's meant to fix, and some examples.
- Compiler explorer
- Scott Meyers' Effective Modern C++ - highly recommended book!
- Jason Turner's C++ Weekly - nice collection of C++-related videos.
- What can I do with a moved-from object?
- What are some uses of decltype(auto)?
- And many more SO posts I'm forgetting...

Author

Anthony Calandra

Content Contributors

See: https://github.com/AnthonyCalandra/modern-cpp-features/graphs/contributors

License