C++11

Overview

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

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
- char32_t and char16_t
- raw string literals

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
- std::ref
- memory model
- std::async
- std::begin/end

C++11 Language Features

Move semantics

Moving an object means to transfer ownership of some resource it manages to another object.

The first benefit of move semantics is performance optimization. When an object is about to reach the end of its lifetime, either because it's a temporary or by explicitly calling `std::move`, a move is often a cheaper way to transfer resources. For example, moving a `std::vector` is just copying some pointers and internal state over to the new vector -- copying would involve having to copy every single contained element in the vector, which is expensive and unnecessary if the old vector will soon be destroyed.

Moves also make it possible for non-copyable types such as `std::unique_ptr`s (smart pointers) to guarantee at the language level that there is only ever one instance of a resource being managed at a time, while being able to transfer an instance between scopes.

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

Rvalue references

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

Type deduction with lvalues and rvalues:

```
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`

void f(int& x) {}
```

```
void f(int&& x) {}

f(x);  // calls f(int&)
f(xl); // calls f(int&)
f(3);  // calls f(int&&)
f(std::move(x)) // calls f(int&&)

f(xr2);           // calls f(int&)
f(std::move(xr2)) // calls f(int&& x)
```

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 *perfect forwarding*: the ability to pass arguments while maintaining their value category (e.g. lvalues stay as lvalues, temporaries are forwarded as 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:

```
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:

```
// 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); // T is int, deduces as f(int &&) => f(int&&)
f(x); // T is int&, deduces as f(int& &&) => f(int&)

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

int&& z = 0; // NOTE: `z` is an lvalue with type `int&&`.
f(z); // T is int&, deduces as f(int& &&) => f(int&)
f(std::move(z)); // T is int, 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` by reference.
- `[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 captured value
// vs.
```

```
auto f3 = [x]() mutable { x = 2; }; // OK: the lambda can perform any
operations on the captured value
```

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) (C++14)`.

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 constant expression
```

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 added
```

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); }
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) (C++14)` 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
}
```

char32_t and char16_t

Provides standard types for representing UTF-8 strings.

```
char32_t utf8_str[] = U"\u0123";
char16_t utf8_str[] = u"\u0123";
```

Raw string literals

C++11 introduces a new way to declare string literals as "raw string literals". Characters issued from an escape sequence (tabs, line feeds, single backslashes, etc.) can be inputted raw while preserving formatting. This is useful, for example, to write literary text, which might contain a lot of quotes or special formatting. This can make your string literals easier to read and maintain.

A raw string literal is declared using the following syntax:

```
R"delimiter(raw_characters)delimiter"
```

where:

- `delimiter` is an optional sequence of characters made of any source character except parentheses, backslashes and spaces.
- `raw_characters` is any raw character sequence; must not contain the closing sequence `")delimiter"`.

Example:

```
// msg1 and msg2 are equivalent.
const char* msg1 = "\nHello,\n\tworld!\n";
const char* msg2 = R"(
Hello,
    world!
)";
```

C++11 Library Features

std::move

`std::move` indicates that the object passed to it may have its resources transferred. 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 reference):

```
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 while maintaining their value category and cv-qualifiers. Useful for generic code and 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 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 pointer that manages its own heap-allocated memory. **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_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.
- 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`.

std::ref

`std::ref(val)` is used to create object of type `std::reference_wrapper` that holds reference of val. Used in cases when usual reference passing using `&` does not compile or `&` is dropped due to type deduction. `std::cref` is similar but created reference wrapper holds a const reference to val.

```
// create a container to store reference of objects.
auto val = 99;
auto _ref = std::ref(val);
_ref++;
auto _cref = std::cref(val);
//_cref++; does not compile
std::vector<std::reference_wrapper<int>>vec; // vector<int&>vec does not
```

```
compile
vec.push_back(_ref); // vec.push_back(&i) does not compile
cout << val << endl; // prints 100
cout << vec[0] << endl; // prints 100
cout << _cref; // prints 100
```

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.
2. `std::launch::async` Run the callable object on a new thread.
3. `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
```