C++14

Overview

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

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++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 object
// 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 (C++11)`.

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 (C++11) for more information on `std::unique_ptr` and `std::shared_ptr`.