Recent Changes in C++

Some Highlights

Toby Allsopp
toby@mi6.gen.nz

Auckland C++ Meetup
14 June 2016

This talk

- Can only scratch the surface
- Focuses on what *I* think is important
  - stuff you should use in your own code
  - stuff you need to know to read other code

Overview

- Introduction
- Type deduction
- Lambda expressions
- Move semantics
- Range-based for loop
- Smart pointers
- Others

History of the standard

- C++98 was the first ISO standard
- C++03 had only minor tweaks
- C++11 was massive, 13 years since the last major update
- C++14 was much more modest
- C++17 also looks to be pretty modest

auto

Instead of

```
map<string, string>::iterator it = m.find("foo");
```

you can write

```
auto it = m.find("foo");
```

- No more **typedef** `std::map<string, string> FooBarMap`
- Doesn't work for member variables, function parameters (but wait for generic lambdas)

New function syntax

You can put the return type *after* the parameter list.

```
int f(double x) { return 7; }
double g(string s) { return 4.2; }

auto f(double x) -> int { return 7; }
auto g(string s) -> double { return 4.2; }
```

This is super useful when the return type of a function template depends on its parameters, e.g.

```
template<typename L, typename R>
auto add(L l, R r) -> decltype(l + r) {
  return l + r;
}
```

Function return type deduction (C++14)

In C++14, you can leave out the return type entirely, e.g.

```
template<typename L, typename R>
auto add(L l, R r) {
  return l + r;
}
```

You can use this for all your functions if you want as long as all callers can see the definition.

Lambdas

Given

```
vector<int> v = { 1, 1, 2, 3 };
int limit = 1;
```

then

```
auto it = find_if(v.begin(), v.end(),
  [limit](int x) -> bool { return x > limit; });
```

is equivalent to

```
struct pred {
  int limit;
  pred(int limit) : limit(limit) {}
  bool operator()(int x) const { return x > limit; }
};
auto it = find_if(v.begin(), v.end(), pred(limit));
```

Lambda capturing spec

C++11	
Nothing	`[]`
Everything by reference	`[&]`
Everything by value	`[=]`
Something by reference	`[&something]`
Something by value	`[something]`
One of each	`[&byref,byval]`
C++14	
Expression by value	`[p=std::move(up)]`
C++17	
`this` by value	`[*this]`

Lambda return type deduction

C++11

- Return type can be omitted if the body is a single return statement (or has no return statement).
- `[]() { return 3; }` is deduced to return `int` (like `auto` does)

C++14

- Return type can be omitted even if multiple statements.

std::function

- Lambda expressions have anonymous types
- Say you want to store some in a vector
  - They all need to be the same type!
- std::function is a wrapper for anything that can be called
  - lambdas
  - function pointers
  - anything with operator()

```
vector<function<string(int)>> v;
v.push_back(&to_string<int>);
v.push_back([](int i) { return string(i); });
```

Generic lambdas (C++14)

You can use **auto** for your lambda's parameters:

```
auto mul = [](auto x, auto y) { return x * y; };
int a = mul(2, 3);
double b = mul(2, 3.2);
```

and get a templated function call operator like this:

```
struct mul_lambda {
  template<typename X, typename Y>
  auto operator()(X x, Y y) { return x * y; }
};
auto mul = mul_lambda();
```

This is really useful in certain circumstances (visitor pattern) but beware overuse.

Move semantics

- A combination of features
  - rvalue references
  - move constructors and assignment
- Can speed up code by avoiding copying
- Can allow non-copyable objects to be transferred (see `unique_ptr`)

Rvalue references

- An rvalue is kind of a temporary value
- In `v.push_back(string("123"))`, the string is an rvalue
- In contrast to an lvalue, an rvalue has no named storage location
- rvalues can bind to const lvalue references
- lvalues cannot bind to rvalue references

```
void l(int &i);  // lvalue reference
void r(int &&i); // rvalue reference

int v = 3;
l(v); // OK - lvalue to lvalue reference
r(v); // NOT OK - lvalue to rvalue reference
r(std::move(v)); // OK - rvalue ref to rvalue ref
l(3); // NOT OK - rvalue to non-const lvalue ref
r(3); // OK - rvalue to rvalue reference
```

Move construction and assignment

- A move constructor is like a copy constructor except the source is passed by rvalue reference
- The idea is that it gets called when the source is “going away” in some sense — it can be destructive
- The other idea is that this makes the operation more efficient, e.g. by just transferring a pointer
- The move assignment operator works on the same principle

```
struct foo {
  foo(); // default constructor
  foo(const foo&); // copy constructor
  foo(foo &&); // move constructor
  foo &operator=(const foo&); // copy assignment
  foo &operator=(foo &&); // move assignment
};
```

Range-based for loop

Instead of

```
for (auto it = v.begin(); it != v.end(); ++it) {
  const string &x = *it;
}
```

you can now write

```
for (const string &x : v) {
}
```

- Saves typing
- More readable
- More efficient (slightly, maybe)

Smart pointers

- `auto_ptr` deprecated (removed in C++17)
  - copy semantics are BROKEN
  - can't put it in a `vector`
- `unique_ptr` is its replacement
  - move-only, made possible by rvalue references
- `shared_ptr` and `weak_ptr`
  - `make_shared<T>(x, y, ...)` is useful

nullptr

```
int x = NULL; // sure, why not?
int y = nullptr; // no way, pointers only
```

```
int foo(double x) { return 7; }
int foo(const char *s) { return s ? *s : 42; }

foo(NULL); // 7? huh?
foo(nullptr); // ahh, 42 :)
```

And the rest

- Initializer lists
- Uniform initialization
- `override` and `final`
- `enum class`
- Angle brackets >>
- Variadic templates
- Variable templates (C++14)
- Threading
  - `std::thread`
  - `std::mutex`
  - `std::future`
  - `std::async`
  - `thread_local`
- `= default` and `= delete`
- `static_assert`
- `constexpr`
- `long long int`
- `alignof` and `alignas`
- Tuples
- Hash tables
- Regular expressions
- Literals
  - User-defined literals
  - Binary literals (C++14)
  - More literals (C++14)

Questions?

decltype

Use to say that something has the same type as something else.

```
vector<int> v = {1,2,3};
auto a = v[1]; // int
decltype(v[1]) b = v[1]; // int&
decltype(auto) c = v[1]; // int& (C++14)
```

Not that useful most of the time.

using type aliases

- More readable alternative to typedef

typedef	using
`typedef map<K,V> m`	`using m = map<K,V>`
`typedef void (*f)(int)`	`using f = void (*)(int)`

- Can be templated

```
template<typename T>
using myvector = vector<T>;
```

References

- https://en.wikipedia.org/wiki/C%2B%2B11
- https://en.wikipedia.org/wiki/C%2B%2B14
- https://en.wikipedia.org/wiki/C%2B%2B17
- http://cppreference.com