C++ 11

6 years ago now, the C++ standards body met and ratified a new version of c++, with new features. These features are now in wide use, and offer some real advantages over c++98. Lets talk about some of the big ones:

auto keyword

You can ask the compiler to infer the type of a variable using the `auto` keyword.

decltype keyword

The new operator decltype takes an expression and "returns" its type:

```
const vector<int> vi;
typedef decltype (vi.begin()) CIT;
CIT another_const_iterator;
```

uniform initialization syntax

You can now use trailing {} to initalize everything.

delete and default methods

You can now intruct the compiler and your peers (and perhaps you after a couple of months) what your intention is regarding canonical methods (constructor, destructor, copy, move, assignment, and move assignment operators).

Just follow the method declaration with an `=delete` or `=default`.

nullptr

It used to be that we reserved the 0 address as an indicator that a pointer was uninitialized. We used the NULL define as well. However, there are issues that arise with this approach, so c++11 introduces `nullptr` which has the same meaning but doesn't equal zero. Its better.

delegating constructor

You can now call a another constructor of the same class. (you have always been able to do this in python)

```
class M //C++11 delegating constructors
{
 int x, y;
 char *p;
public:
 M(int v) : x(v), y(0), p(new char [MAX]) {} //#1 target
 M(): M(0) {cout<<"delegating ctor"<<endl;} //#2 delegating
};
```

for range form

for(auto& f: iterable) {
}

```
std::vector<std::string> foo = {"one","two","three"};
for(auto& c : foo) {
    cout << c << endl;
}
```

lambdas

anonymous functions.

```
std::string fred("fred");
auto func = [](std::string& s){ cout << s << endl; }
func(fred);
```

capture regions

Capture regions let you specify how the lambda closes over variables in the current scope. It is a comma-separated list of zero or more captures, optionally beginning with a capture-default.

- [=] copy
- [&] reference
- [this] - this pointer

You can name vars as well.

```
std::string me{"fred"};

auto fx = [=]() {cout << me << endl;};
auto fx2  = [&](){cout << me << endl;};
me = "you";

fx();
```

```
fx2();
```

RValue References

C++11 has introduced the notion of Rvalues and LValues. Lvalues are named values. They are called LValues because they can appear on the left hand side of an assignment operation. RValues, on the other hand, are anonymous variables which cannot appear on the left hand side of an assignment.

```
std::string foo = "fred";
// foo is an LValue
// "fred" is an RValue
```

References to anonymous, unnamed data. Syntax for this is a double ampersand &&.

```
std::string foo&& = "me";
```

Useful for function arguments, allowing you to call functions with anonymous data.
Also useful for transferring ownership, which we will look at later.

```
void foo(std::string&& f) {
    cout << f << endl;
};
foo("bar");
```

std::move

convert an LValue to an RValue.

```
std::string mystring{"bar"};
foo(std::move(mystring));
```

Move Semantics

Often times, one would like to avoid creating unnecessary copies of data. This can happen for a variety of reasons, including returning from functions, aggregation (this+that), etc. Move semantics allow you to move a value from one scope to another cheaply. It is only applicable to types which manage memory internally.

move constructor

The move constructor is similar to the copy constructor, except that it takes a non-const rvalue reference. For it to work with stl containers, it also has to be declared as `noexcept`. (see example). `noexcept` means that if an exception gets thrown in the method, your whole program will shut down immediately. Why does the STL demand this of you? The authors are Jerks. Also, the move constructor makes a strong guarantee that the transaction will result in a complete transfer of state from one object to another; this guarantee cannot be satisfied should an exception be raised during the execution. So, we instruct the compiler, via `noexcept`, to shut down should one occur.

```
Example {
    string* m_name;
public:
    Example() : m_name{new string() } {}
    Example(Example&& rhs) noexcept {
        m_name = rhs.m_name;
        rhs.m_name = nullptr;
    }
    ...
}
```

Its job is to move data from the instance passed in as an argument to `this` instance, valid state.

move assignment operator

The move assignment operator is similar to the assignment operator, except that it takes a non-const rvalue reference as an argument, and it also must move data from the argument to `this` instance, leaving the passed in instance in a valid state.

```
Example {
    string* m_name;
public:
    Example() : m_name{new string() } {}
    ...

    Example& operator=(Example&& rhs) noexcept {
        if (this != &rhs) {
            if(m_name) delete m_name;
            m_name = rhs.m_name;
            rhs.m_name = nullptr;
        }
        return *this;
    }
}
```

STL

The stl has gotten a face lift. There are a number of newish containers that are now part of the stl

- unordered_map
- unordered_multimap
- unordered_set
- unordered_multiset

Threading library

Now we have a threading library in the stl, providing a uniform approach to multi-threading across oses. Yay.

Smart Pointers

Smart pointers take care of RAII and express ownership. There are two varieties - shared, which allows for multiple owners, and unique, which only allows for a single owner.

shared_ptr

Shared ptr is a reference counted implementation. You can share underlying data between multiple instances, and only delete memory when the last instance goes out of scope. Shared pointers have a companion function `std::make_shared` which can be used to initialize a shared_ptr without ever calling new.

```
std::shared_ptr<std::string> mep;
{
    auto foop = std::make_shared<std::string>("foo");
    cout << *foop << endl;
    mep = foop;
}
cout << mep << endl;
```

unique_ptr

Unique pointer is a bit more restrictive. It expresses exclusive ownership, and, as such, does not provide a copy or assignment operator. This means that you cannot copy it around at all. You have to use std::move if you want to transfer ownership.

```
auto fredp = std::unique_ptr<std::string>(new string("fred"));
cout << *fredp << endl;
```

retrieving the raw pointer

If you want to retrieve a raw pointer from a smart pointer, simply call `.get()` on it.

New algorithms

The C++11 Standard Library defines new algorithms that mimic the set theory operations all_of(), any_of() and none_of(). The following listing applies the predicate ispositive() to the range [first, first+n) and uses all_of(), any_of() and none_of() to examine the range's properties:

```
#include <algorithm>
//C++11 code
//are all of the elements positive?
all_of(first, first+n, ispositive()); //false
//is there at least one positive element?
any_of(first, first+n, ispositive());//true
// are none of the elements positive?
none_of(first, first+n, ispositive()); //false
```

A new category of copy_n algorithms is also available. Using copy_n(), copying an array of 5 elements to another array is a cinch:

```
#include
int source[5]={0,12,34,50,80};
int target[5];
//copy 5 elements from source to target
copy_n(source,5,target);
```

The algorithm iota() creates a range of sequentially increasing values, as if by assigning an initial value to *first, then incrementing that value using prefix ++. In the following listing, iota() assigns the consecutive values {10,11,12,13,14} to the array arr, and {'a', 'b', 'c'} to the char array c.

```
#include <numeric>
int a[5]={0};
char c[3]={0};
iota(a, a+5, 10); //changes a to {10,11,12,13,14}
iota(c, c+3, 'a'); //{'a','b','c'}
```