Lambda Functions in C++11 - the Definitive Guide

By Alex Allain

One of the most exciting features of C++11 is ability to create lambda functions (sometimes referred to as closures). What does this mean? A lambda function is a function that you can write inline in your source code (usually to pass in to another function, similar to the idea of a functor or function pointer). With lambda, creating quick functions has become much easier, and this means that not only can you start using lambda when you'd previously have needed to write a separate named function, but you can start writing more code that relies on the ability to create quick-and-easy functions. In this article, I'll first explain why lambda is great--with some examples--and then I'll walk through all of the details of what you can do with lambda.

Why Lambdas Rock

Imagine that you had an address book class, and you want to be able to provide a search function. You might provide a simple search function, taking a string and returning all addresses that match the string. Sometimes that's what users of the class will want. But what if they want to search only in the domain name or, more likely, only in the username and ignore results in the domain name? Or maybe they want to search for all email addresses that also show up in another list. There are a lot of potentially interesting things to search for. Instead of building all of these options into the class, wouldn't it be nice to provide a generic "find" method that takes a procedure for deciding if an email address is interesting? Let's call the method findMatchingAddresses, and have it take a "function" or "function-like" object.

```
#include <string>
#include <vector>

class AddressBook
{
    public:
    // using a template allows us to ignore the differences between functors, function pointers
    // and lambda
    template<typename Func>
    std::vector<std::string> findMatchingAddresses (Func func)
    {
        std::vector<std::string> results;
        for ( auto itr = _addresses.begin(), end = _addresses.end(); itr != end; ++itr )
        {
            // call the function passed into findMatchingAddresses and see if it matches
            if ( func( *itr ) )
            {
                results.push_back( *itr );
            }
        }
        return results;
    }

    private:
    std::vector<std::string> _addresses;
};
```

Anyone can pass a function into the findMatchingAddresses that contains logic for finding a particular function. If the function returns true, when given a particular address, the address will be returned. This kind of approach was OK in earlier version of C++, but it suffered from one fatal flaw: it wasn't quite convenient enough to create functions. You had to go define it somewhere else, just to be able to pass it in for one simple use. That's where lambdas come in.

Basic Lambda Syntax

Before we write some code to solve this problem, let's see the really basic syntax for lambda.

```
#include <iostream>

using namespace std;

int main()
{
    auto func = [] () { cout << "Hello world"; };
    func(); // now call the function
}
```

Okay, did you spot the lambda, starting with []? That identifier, called the capture specification, tells the compiler we're creating a lambda function. You'll see this (or a variant) at the start of every lambda function.

Next up, like any other function, we need an argument list: (). Where is the return value? Turns out that we don't need to give one. In C++11, if the

compiler can deduce the return value of the lambda function, it will do it rather than force you to include it. In this case, the compiler knows the function returns nothing. Next we have the body, printing out "Hello World". This line of code doesn't actually cause anything to print out though--we're just creating the function here. It's almost like defining a normal function--it just happens to be inline with the rest of the code.

It's only on the next line that we call the lambda function: func() -- it looks just like calling any other function. By the way, notice how easy this is to do with auto! You don't need to sweat the ugly syntax of a function pointer.

Applying Lambda in our Example

Let's look at how we can apply this to our address book example, first creating a simple function that finds email addresses that contain ".org".

```
AddressBook global_address_book;

vector<string> findAddressesFromOrgs ()
{
    return global_address_book.findMatchingAddresses(
        // we're declaring a lambda here; the [] signals the start
        [] (const string& addr) { return addr.find( ".org" ) != string::npos; }
    );
}
```

Once again we start off with the capture specifier, [], but this time we have an argument--the address, and we check if it contains ".org". Once again, nothing inside the body of this lambda function is executed here yet; it's only inside findMatchingAddresses, when the variable func is used, that the code inside the lambda function executes.

In other words, each loop through findMatchingAddresses, it calls the lambda function and gives it the address as an argument, and the function checks if it contains ".org".

Variable Capture with Lambdas

Although these kinds of simple uses of lambd are nice, variable capture is the real secret sauce that makes a lambda function great. Let's imagine that you want to create a small function that finds addresses that contain a specific name. Wouldn't it be nice if you could write code something like this?

```
// read in the name from a user, which we want to search
string name;
cin>> name;
return global_address_book.findMatchingAddresses(
    // notice that the lambda function uses the the variable 'name'
    [&] (const string& addr) { return addr.find( name ) != string::npos; }
);
```

It turns out that this example is completely legal--and it shows the real value of lambda. We're able to take a variable declared outside of the lambda (name), and use it inside of the lambda. When findMatchingAddresses calls our lambda function, all the code inside of it executes--and when addr.find is called, it has access to the name that the user passed in. The only thing we needed to do to make it work is tell the compiler we wanted to have variable capture. I did that by putting [&] for the capture specification, rather than []. The empty [] tells the compiler not to capture any variables, whereas the [&] specification tells the compiler to perform variable capture.

Isn't that marvelous? We can create a simple function to pass into the find method, capturing the variable name, and write it all in only a few lines of code. To get a similar behavior without C++11, we'd either need to create an entire functor class or we'd need a specialized method on AddressBook. In C++11, we can have a single simple interface to AddressBook that can support any kind of filtering really easily.

Just for fun, let's say that we want to find only email addresses that are longer than a certain number of characters. Again, we can do this easily:

```
int min_len = 0;
cin >> min_len;
return global_address_book.findMatchingAddresses( [&] (const string& addr) { return addr.length() >= min_len; } );
```

By the way, to steal a line from Herb Sutter, you should get used to seeing "});" This is the standard end-of-function-taking-lambda syntax, and the more you start seeing and using lambda in your own code, the more you'll see little piece of syntax.

Lambda and the STL

One of the biggest beneficiaries of lambda functions are, no doubt, power users of the standard template library algorithms package. Previously, using algorithms like for_each was an exercise in contortions. Now, though, you can use for_each and other STL algorithms almost as if you were writing a normal loop. Compare:

```
vector<int> v;
v.push_back( 1 );
v.push_back( 2 );
//...
for ( auto itr = v.begin(), end = v.end(); itr != end; itr++ )
{
    cout << *itr;
}
```

with

```
vector<int> v;
v.push_back( 1 );
v.push_back( 2 );
//...
for_each( v.begin(), v.end(), [] (int val)
{
    cout << val;
} );
```

That's pretty good looking code if you ask me--it reads, and is structured, like a normal loop, but you're suddenly able to take advantage of the goodness that for_each provides over a regular for loop--for example, guarantees that you have the right end condition. Now, you might wonder, won't this kill performance? Well, here's the kicker: it turns out that for_each has about the same performance, and is sometimes even **faster** than a regular for loop. (The reason: it can take advantage of loop unrolling.)

If you're interested in more on C++11 lambda and the benefits to the STL, you'll enjoy this video of Herb Sutter talking about C++11 lambdas.

I hope this STL example shows you that lambda functions are more than just a slightly more convenient way of creating functions--they allow you to create entirely new ways of writing programs, where you have code that takes other functions as data and allows you to abstract away the processing of a particular data structure. for_each works on a list, but wouldn't it be great to have similar functions for working with trees, where all you had to do was right some code that would process each node, and not have to worry about the traversal algorithm? This kind of decomposition where one function worries about the structure of data, while delegating the data processing to another function can be quite powerful. With lambda, C++11 enables this new kind of programming. Not that you couldn't have done it before--for_each isn't new--it's just that you wouldn't have wanted to do it before.

More on the new Lambda Syntax

By the way, the parameter list, like the return value is also optional if you want a function that takes zero arguments. Perhaps the shortest possible lambda expression is:

```
[] {}
```

Which is a function that takes no arguments and does nothing. An only slightly more compelling example:

```
using namespace std;
#include <iostream>

int main()
{
    [] { cout << "Hello, my Greek friends"; }();
}
```

Personally, I'm not yet sold on omitting the argument list; I think the [] () structure tends to help lambda functions stand out a little more in the code, but time will tell what standards people come up with.

Return Values

By default, if your lambda does not have a return statement, it defaults to void. If you have a simple return expression, the compiler will deduce the type of the return value:

```
[] () { return 1; } // compiler knows this returns an integer
```

If you write a more complicated lambda function, with more than one return value, you should specify the return type. (Some compilers, like GCC, will let you get away without doing this, even if you have more than one return statement, but the standard doesn't guarantee it.)

Lambdas take advantage of the optional new C++11 return value syntax of putting the return value after the function. In fact, you must do this if you want to specify the return type of a lambda. Here's a more explicit version of the really simple example from above:

```
[] () -> int { return 1; } // now we're telling the compiler what we want
```

Throw Specifications

Although the C++ standards committee decided to deprecate throw specifications (except for a few cases I'll cover in a later article), they didn't remove them from the language, and there are tools that do static code analysis to check exception specifications, such as PC Lint. If you are using one of these tools to do compile time exception checking, you really want to be able to say which exceptions your lambda function throws. The main reason I can see for doing this is when you're passing a lambda function into another function as an argument, and that function expects the lambda function to throw only a specific set of exceptions. By providing an exception spec for your lambda function, you could allow a tool like PC Lint to check that for you. If you want to do that, it turns out you can. Here's a lambda that specifies that it takes no arguments and does not throw an exception:

```
[] () throw () { /* code that you don't expect to throw an exception*/ }
```

How are Lambda Closures Implemented?

So how does the magic of variable capture really work? It turns out that the way lambdas are implemented is by creating a small class; this class overloads the operator(), so that it acts just like a function. A lambda function is an instance of this class; when the class is constructed, any variables in the surrounding enviroment are passed into the constructor of the lambda function class and saved as member variables. This is, in fact, quite a bit like the idea of a functor that is already possible. The benefit of C++11 is that doing this becomes almost trivially easy--so you can use it all the time, rather than only in very rare circumstances where writing a whole new class makes sense.

C++, being very performance sensitive, actually gives you a ton of flexibility about what variables are captured, and how--all controlled via the capture specification, []. You've already seen two cases--with nothing in it, no variables are captured, and with &, variables are captured by reference. If you make a lambda with an empty capture group, [], rather than creating the class, C++ will create a regular function. Here's the full list of options:

[]	Capture nothing (or, a scorched earth strategy?)
[&]	Capture any referenced variable by reference
[=]	Capture any referenced variable by making a copy
[=, &foo]	Capture any referenced variable by making a copy, but capture variable foo by reference
[bar]	Capture bar by making a copy; don't copy anything else
[this]	Capture the this pointer of the enclosing class

Notice the last capture option--you don't need to include it if you're already specifying a default capture (= or &), but the fact that you can capture the this pointer of a function is super-important because it means that you don't need to make a distinction between local variables and fields of a class when writing lambda functions. You can get access to both. The cool thing is that you don't need to explicitly use the this pointer; it's really like you are writing a function inline.

```
class Foo
{
public:
    Foo () : _x( 3 ) {}
    void func ()
    {
        // a very silly, but illustrative way of printing out the value of _x
        [this] () { cout << _x; } ();
    }

private:
        int _x;
};

int main()
{
    Foo f;
    f.func();
}
```

Dangers and Benefits of Capture by Reference

When you capture by reference, the lambda function is capable of modifying the local variable outside the lambda function--it is, after all, a reference. But this also means that if you return a lamba function from a function, you shouldn't use capture-by-reference because the reference will not be valid after the function returns.

What type is a Lambda?

The main reason that you'd want to create a lambda function is that someone has created a function that expects to receive a lambda function. We've already seen that we can use templates to take a lambda function as an argument, and auto to hold onto a lambda function as a local variable. But how do you name a specific lambda? Because each lambda function is implemented by creating a separate class, as you saw earlier, even single lambda function is really a different type--even if the two functions have the same arguments and the same return value! But C++11 does include a convenient wrapper for storing any kind of function--lambda function, functor, or function pointer: std::function.

std::function

The new std::function is a great way of passing around lambda functions both as parameters and as return values. It allows you to specify the exact types for the argument list and the return value in the template. Here's out AddressBook example, this time using std::function instead of templates. Notice that we do need to use the 'functional' header file.

```
#include <functional>
#include <vector>

class AddressBook
{
    public:
    std::vector<string> findMatchingAddresses (std::function<bool (const string&)> func)
    {
        std::vector<string> results;
```

```
        for ( auto itr = _addresses.begin(), end = _addresses.end(); itr != end; ++itr )
        {
            // call the function passed into findMatchingAddresses and see if it matches
            if ( func( *itr ) )
            {
                results.push_back( *itr );
            }
        }
        return results;
    }

    private:
    std::vector<string> _addresses;
};
```

One big advantage of std::function over templates is that if you write a template, you need to put the whole function in the header file, whereas std::function does not. This can really help if you're working on code that will change a lot and is included by many source files.

If you want to check if a variable of type std::function is currently holding a valid function, you can always treat it like a boolean:

```
std::function<int ()> func;
// check if we have a function (we don't since we didn't provide one)
if ( func )
{
    // if we did have a function, call it
    func();
}
```

A Note About Function Pointers

Under the final C++11 spec, if you have a lambda with an empty capture specification, then it can be treated like a regular function and assigned to a function pointer. Here's an example of using a function pointer with a capture-less lambda:

```
typedef int (*func)();
func f = [] () -> int { return 2; };
f();
```

This works because a lambda that doesn't have a capture group doesn't need its own class--it can be compiled to a regular old function, allowing it to be passed around just like a normal function. Unfortunately, support for this feature is not included in MSVC 10, as it was added to the standard too late.

Making Delegates with Lambdas

Let's look at one more example of a lambda function--this time to create a delegate. What's a delgate, you ask? When you call a normal function, all you need is the function itself. When you call a method on an object, you need two things: the function and the object itself. It's the difference between func() and obj.method(). To call a method, you need both. Just passing in the address of the method into a function isn't enough; you need to have an object to call the method on.

Let's look at an example, starting with some code that again expects a function as an argument, into which we'll pass a delegate.

```
#include <functional>
#include <string>

class EmailProcessor
{
public:
    void receiveMessage (const std::string& message)
    {
        if ( _handler_func )
        {
            _handler_func( message );
        }
        // other processing
    }
    void setHandlerFunc (std::function<void (const std::string&)> handler_func)
    {
        _handler_func = handler_func;
    }

private:
        std::function<void (const std::string&)> _handler_func;
};
```

This is a pretty standard pattern of allowing a callback function to be registered with a class when something interesting happens.

But now let's say we want another class that is responsible for keeping track of the longest message received so far (why do you want to do this? Maybe you are a bored sysadmin). Anyway, we might create a little class for this:

```
#include <string>

class MessageSizeStore
{
    public:
    MessageSizeStore () : _max_size( 0 ) {}
    void checkMessage (const std::string& message )
    {
        const int size = message.length();
        if ( size > _max_size )
        {
            _max_size = size;
        }
    }
    int getSize ()
    {
        return _max_size;
    }

private:
    int _max_size;
};
```

What if we want to have the method checkMessage called whenever a message arrives? We can't just pass in checkMessage itself--it's a method, so it needs an object.

```
EmailProcessor processor;
MessageSizeStore size_store;
processor.setHandlerFunc( checkMessage ); // this won't work
```

We need some way of binding the variable size_store into the function passed to setHandlerFunc. Hmm, sounds like a job for lambda!

```
EmailProcessor processor;
MessageSizeStore size_store;
processor.setHandlerFunc(
        [&] (const std::string& message) { size_store.checkMessage( message ); }
);
```

Isn't that cool? We are using the lambda function here as glue code, allowing us to pass a regular function into setHandlerFunc, while still making a call onto a method--creating a simple delegate in C++.

In Summary

So are lambda functions really going to start showing up all over the place in C++ code when the language has survived for decades without them? I think so--I've started using lambda functions in production code, and they are starting to show up all over the place--in some cases shortening code, in some cases improving unit tests, and in some cases replacing what could previously have only been accomplished with macros. So yeah, I think lambdas rock way more than any other Greek letter.

Lambda functions are available in GCC 4.5 and later, as well as MSVC 10 and version 11 of the Intel compiler.

Next: Range-Based For Loops Range-based for loops make iterating over vectors and other containers very easy
Previous: How auto, decltype, and the new function syntax work together to create better code Learn about some of the new type inference features of C++11