

Sticky Bits – Powered by Feabhas

A blog looking at developing software
for real-time and embedded systems

Demystifying C++ lambdas

Posted on [March 7, 2014](#) by [Glennan Carnie](#)

A new (but not so welcome?) addition

Lambdas are one of the new features added to C++ and seem to cause considerable consternation amongst many programmers. In this article we'll have a look at the syntax and underlying implementation of lambdas to try and put them into some sort of context.

Functors and the Standard Library

With STL algorithms the processing on each element is performed by a user-supplied unary or binary functor object. For common operations, the STL-supplied functors can be used (for example `std::divides`), but for bespoke manipulations a bespoke function or functor must be created.

```
class X
{
public:
    void op() { cout << "X::op()" << endl; }
};

class Functor
{
public:
    void operator() (X& elem) { elem.op(); }
};

int main()
{
    vector<X> v;
    v.push_back(X());
    v.push_back(X());
    Functor f;

    for_each(v.begin(), v.end(), f);
}
```

Use functor object

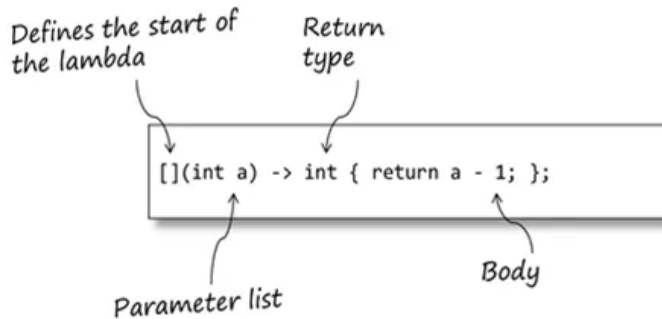
A functor is a class that provides an implementation of `operator()`.

In the case of functors used with the STL algorithms the `operator()` function must take either one parameter (for a unary procedure) or two parameters (for binary procedures) of appropriate types.

Creating bespoke functors can be a lot of effort; especially if the functor is only used in one specific place. These bespoke functors also unnecessarily 'clutter up' the code.

Introducing lambdas

A lambda is an ad-hoc, locally-scoped function (well, more strictly, a functor). Basically, lambdas are syntactic sugar, designed to reduce a lot of the work required in creating ad-hoc functor classes.



The brackets (`[]`) mark the declaration of the lambda; it can have parameters, and it should be followed by its body (the same as any other function).

When the lambda is executed the parameters are passed using the standard ABI mechanisms. One difference between lambdas and functions: lambda parameters can't have defaults.

The body of the lambda is just a normal function body, and can be arbitrarily complex (although, as we'll see, it's generally good practice to keep lambda bodies relatively simple).

Note the lambda uses a trailing return type declaration. This is (no doubt) to simplify parsing (since types are not valid function parameters).

The return type may be omitted if:

- The return type is void
- The compiler can deduce the return type (lambda body is `return <type>`)

A lambda is an object (hence why we're referring to it as a functor, rather than a function) so has a type and can be stored. However, the type of the lambda is only known by the compiler (since it is compiler-generated), so you must use `auto` for declaration instances of the lambda.

```

void func()
{
    auto lambda = [](int a) -> int { return a - 1; }; // MUST use auto since the lambda
                                                    // is compiler-generated and
                                                    // therefore has an unknown type.

    int i = lambda(100); // Now you can call the lambda
                        // just like a normal function.
}

int main()
{
    lambda(100); // ERROR! lambda is not in scope
}

```

block-scoped 'function'!

Lambdas allow ad-hoc functions to be declared at block scope (something that was illegal before in C++). The lambda function (functor) is only available within the scope of `func()` in this example; unlike a function, which

would have global scope (or file scope if declared as static).

This means I can replace my manually-created functor in the STL algorithm with a lambda:

```
int main()
{
    vector<X> v;
    v.push_back(X());
    v.push_back(X());

    auto lambda = [] (X& elem) -> void { elem.op(); };

    for_each(v.begin(), v.end(), lambda);
}
```

Use lambda in algorithm

In the code above the `for_each` algorithm calls the lambda with each element in the container it turns.

So, that's neat. But what has it really gained me?

Inline lambdas

Since a lambda is a scoped functor we can define it just in the scope where we actually need it – in our case within the `for_each` algorithm.

```
class X
{
public:
    void op();
    int getVal();
};

int main()
{
    vector<X> v;
    v.push_back(X());
    v.push_back(X());

    for_each(v.begin(), v.end(), [](X& elem) { elem.op(); });

    auto i =
        find_if(v.begin(), v.end(), [](X& elem) -> bool { return (elem.getVal() != 0); });
}
```

Lambda is scoped to the for_each algorithm

Lambda is scoped to the find_if algorithm

Now, the lambda is defined within the body of the algorithm and effectively only exists for the lifetime of the algorithm. We have reduced the scope of the code to just where it is needed (one of the principles of good modular design).

From a readability perspective we have put the functionality right where it is being used; unlike a functor, which may well be defined in another module (that the programmer has to go and find, and/or understand out of context to where it's being used).

Under the hood

When you define a lambda the compiler uses that to create an ad-hoc functor class. The functor name is compiler-generated (and probably won't be anything human readable)

User code

```
[](X& elem) { elem.op(); }
```

Compiler generated (conceptual)

```
class _SomeCompilerGeneratedName_  
{  
public:  
    void operator() (X& elem) const  
    {  
        elem.op();  
    }  
};
```

The lambda body is used to generate the operator() method on the functor class. The client code is modified to use the new lambda-functor.

```
int main()  
{  
    vector<X> v;  
    v.push_back(X());  
    v.push_back(X());  
  
    for_each(v.begin(), v.end(), [](X& elem) { elem.op(); });  
}
```

Compiler generated (conceptual)

```
class _SomeCompilerGeneratedName_  
{  
    // As previous...  
};  
  
int main()  
{  
    vector<X> v;  
    v.push_back(X());  
    v.push_back(X());  
  
    for_each(v.begin(), v.end(), _SomeCompilerGeneratedName_{});  
}
```

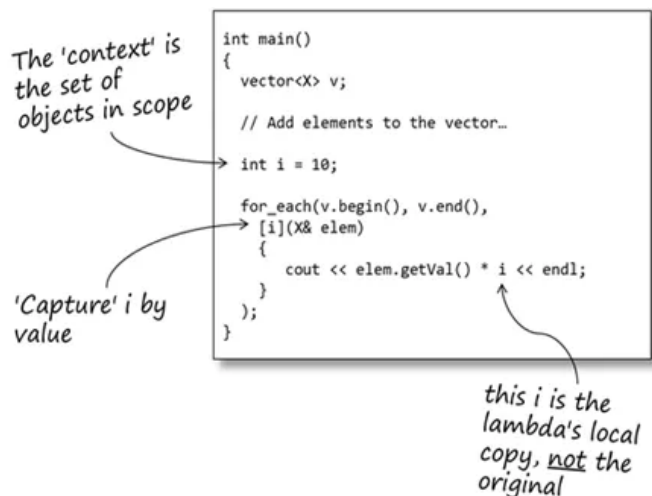
Capturing the context

Sometimes it's useful and convenient to be able to access objects from the lambda's containing scope – that is, the scope within which the lambda was defined. We could pass them in to the lambda as parameters (just like a normal function); however, this doesn't work with algorithms since the algorithm has no mechanism for passing extra parameters from your code (how could it?)

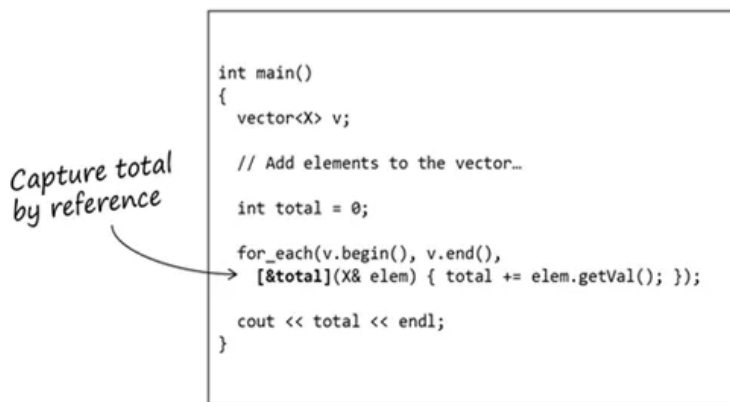
If you were writing your own functor you could do this by passing in the appropriate parameters to the constructor of the functor. C++ provides a convenient mechanism for achieving this with lambdas called ‘capturing the context’.

The context of a lambda is the set of objects that are in scope when the lambda is called. The context objects may be captured then used as part of the lambda’s processing.

Capturing an object by name makes a lambda-local copy of the object.



Capturing an object by reference allows the lambda to manipulate its context. That is, the lambda can change the values of the objects it has captured by reference.



A word of warning here: A lambda is, as we’ve seen, just an object and, like other objects it may be copied, passed as a parameter, stored in a container, etc. The lambda object has its own scope and lifetime which may, in some circumstances, be different to those objects it has ‘captured’. Be very careful when capturing local objects by reference because a lambda’s lifetime may exceed the lifetime of its capture list. In other words, the lambda may have a reference to an object no longer in scope!

All variables in scope can be captured using a default-capture. This makes available all automatic variables currently in scope.

```

int i;
double d;
X theX;
std::vector<double> v(1000);

auto lam1 = [&]() { /* code... */ };    // Capture everything by reference
auto lam2 = [=]() { /* code... */ };    // Capture everything by value

```

*Be careful of
the overheads*

Note, the compiler is only required to make copies of captured objects if they are actually used within the body of the lambda.

Under the hood (again)

When you add a capture list to a lambda the compiler adds appropriate member variables to the lambda-functor class and a constructor for initialising these variables.

User code

```
[&total, offset](X& elem) { total += elem.get1() + offset; }
```

Compiler generated (conceptual)

```

class _SomeCompilerGeneratedName_
{
public:
    _SomeCompilerGeneratedName_(int& t, int o) : total_{t}, offset_{o} {}
    void operator() (X& elem) const {total_ += elem.getVal() + offset_;}

private:
    int& total_;    // Context captured by reference
    int  offset_;   // Context captured by value
};

```

It is relatively easy to see now why capturing the context has potential overheads: for every object captured by value a copy of the original is made; for every object captured by reference a reference is stored.

User code

```
int main()
{
    vector<X> v;

    // Add elements to the vector...

    int total = 0;
    int offset = 1;

    for_each(v.begin(), v.end(),
        [&total, &offset](X& elem) { total += elem.getVal() + offset; });

    cout << total << endl;
}
```

Compiler generated (conceptual)

```
class _SomeCompilerGeneratedName_
{
    // As previous...
};

int main()
{
    vector<X> v;

    // Add elements to the vector...

    int total = 0;
    int offset = 1;

    for_each(v.begin(), v.end(), _SomeCompilerGeneratedName_{total, offset});

    cout << total << endl;
}
```

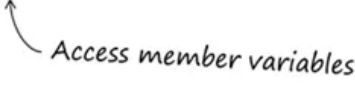
Lambdas within member functions

It is perfectly possible (and quite likely) that we may want to use lambdas inside class member functions. Remember, a lambda is a unique (and separate) class of its own so when it executes it has its own context. Therefore, it does not have direct access to any of the class' member variables.

To capture the class' member variables we must capture the `this` pointer of the class. We now have full access to all the class' data (including private data, as we are inside a member function).

```
class Filter
{
public:
    Filter(vector<int> src) : v(src) {}

    void filter()
    {
        remove_if(v.begin(), v.end(),
            [this](int i) -> bool { return (i < level); });
    }
private:
    vector<int> v;
    int level;
};
```



Access member variables

Callable objects

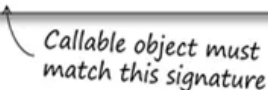
Callable object is a generic name for any object that can be called like a function:

- A member function (pointer)
- A free function (pointer)
- A functor
- A lambda

In C we have the concept of a pointer-to-function, which allows the address of any function to be stored (providing its signature matches that of the pointer). However, a pointer-function has a different signature to a pointer-to-member-function; which as a different signature to a lambda. What would be nice is a generalised 'pointer-to-callable-object' that could store the address of any callable object (providing its signature matched, of course).

`std::function` is a template class that can hold any callable object that matches its signature. `std::function` provides a consistent mechanism for storing, passing and accessing these objects.

```
std::function <<Return Type> (<Parameter List>>)
```



Callable object must match this signature

`std::function` can be thought of as a generic pointer-to-function that can point at any callable object, provided the callable object matches the signature of the `std::function`. And, unlike C's pointer-to-function, the C++ compiler provides strong type-checking on the parameters of the callable object (including the return type).


```
#include <functional>

class SimpleCallback
{
public:
    SimpleCallback (std::function<void(void)> f) : callback(f) {}
    void execute();

private:
    std::function<void(void)> callback; // void (*callback)(void)
};

void SimpleCallback::execute()
{
    if (callback != nullptr)          // Is the function valid?
    {
        callback();                  // Call like a normal function
    }
}
```

std::function provides an overload for operator!= to allow it to be compared to nullptr (so it can act like a function-pointer).

Our SimpleCallback class can be used with any callable type – functors, free functions or lambdas, without any change, since they all match the signature required by callback.

With functors...

```
class Functor
{
public:
    void operator()() { cout << "Functor" << endl; }
};

int main()
{
    Functor functor;
    SimpleCallback callback(functor);
    callback.execute();
}
```

With functions...

```
void func()
{
    cout << "Free function" << endl;
}

int main()
{
    SimpleCallback callback(func);
    callback.execute();
}
```

...or with lambdas

```
int main()
{
    SimpleCallback callback([]() { cout << "Lambda" << endl; });
    callback.execute();
}
```

In summary

Despite the awkward syntax lambdas are not a mechanism to be feared or despised. They merely provides a useful way of simplify code and reducing programmer effort. In essence they are no more than syntactic sugar. In this respect they are no more detrimental than operator overloading.

For more information on other aspects of C++11, have a look at our training course [AC++11-401 – Transitioning to C++11](#).

For more on C++ programming – particularly for embedded and real-time applications – visit the Feabhas website. You may find the following of interest:

[C++-501 – C++ for Embedded Developers](#)

[C++-502 – C++ for Real-Time Developers](#)

[AC++-501 – Advanced C++](#)



Glennan Carnie

Technical Consultant at [Feabhas Ltd](#)

Glennan is an embedded systems and software engineer with over 20 years experience, mostly in high-integrity systems for the defence and aerospace industry.

He specialises in C++, UML, software modelling, Systems Engineering and process development.



This entry was posted in [C/C++ Programming](#) and tagged [C++0x](#), [C++98](#), [functors](#), [lambda](#), [Standard Library](#), [std::function](#), [STL](#). Bookmark the [permalink](#).

31 Responses to Demystifying C++ lambdas



Dan says:

March 9, 2014 at 3:44 pm

This is one of the best write-ups on lambdas I've seen, thank you.

That said, and I know this is extremely subjective, I'm not the inline lambda syntax improves readability (e.g. `for_each`). It's completely possible that it's just so new that my brain is still fighting it, I'll concede that. But I could quickly see a lambda becoming more complex, and soon the `for_each` looks like someone sneezed out a bunch of code (I realize lambdas should only be used for small snippets, but we all know how well code ages as new & different developers come on board).

Anyway, thanks again, this one is definitely bookmarked. I haven't found too many of the C++11 features to be compelling for deeply embedded systems, but this post has encouraged me to at least give lambdas another look.



**Antoine** says:

August 20, 2014 at 10:44 pm

Very nice. Stright to the point, informative and pedagogical.
May I ask what software you used to produce the nice code snapshots with the comments and all?
Thanks.

**glennan** says:

August 21, 2014 at 5:01 am

Thanks Antoine!
The code snippets are done with PowerPoint 😊 They're taking directly from our training course materials.

**Ioannis** says:

August 21, 2014 at 10:27 am

Great Work!

I think you are missing a -> on the last picture

```
int main(){  
SimpleCallback callback([]() -> {cout << "Lambda" << endl;});  
callback.execute();  
}
```

Other than that, excellent work!

**FipS** says:

August 21, 2014 at 4:23 pm

Very well written and nicely presented! Thanks for sharing.

**Paul Jurczak** says:

August 22, 2014 at 3:22 am

Nice article, thank you. You may want to revisit your Filter example though. `std::remove` and `std::remove_if` are the most confusingly named STL algorithms - they don't remove anything, they just rearrange order of elements in the container and create a bunch of undefined values. You must follow them with `erase`, in order to achieve desired result.



Henri Tuhola says:

August 22, 2014 at 3:58 pm

C++ - lots of carcinogenic abstractions



Steve Little says:

August 25, 2014 at 1:43 pm

"It is relatively easy to see now why capturing the context has potential overheads: for every object captured by value a copy of the original is made; for every object captured by reference a reference is stored. (Therefore you might want to think twice about capturing everything in the context by value (using [=]))"

I'm not sure that's correct. The C++ language standard section [expr.prim.lambda] says:

"If a lambda-expression has an associated capture-default and its compound-statement odr-uses (3.2) this or a variable with automatic storage duration and the odr-used entity is not explicitly captured, then the odr-used entity is said to be implicitly captured; such entities shall be declared within the reaching scope of the lambda expression."

Based on my reading of that, a compiler is **not required** to capture all objects (when using a capture-default) just all objects that are ODR-used in the body of the lambda-expression.

Some implementations **may** choose to copy everything, but that would be the fault of an inefficient implementation, and not necessarily a good reason to avoid using a capture-default.



Michael Hamilton says:

August 25, 2014 at 10:59 pm

@Ioannis: No, the author has it correct. Lambda declarations only require -> when the return type cannot be automatically deduced. In the case of no return statement it is trivially "void".



KrzaQ says:

August 26, 2014 at 3:08 pm

I liked the article, but I have one nit-pick: unless your lambda is mutable, its `operator()` should take `const this`.





Andy Prowl says:

August 26, 2014 at 6:02 pm

Nice article 😊 One minor thing: the lambda's call operator generated by the compiler is const-qualified, unless "mutable" is used.



glennan says:

August 26, 2014 at 9:05 pm

Well spotted, Andy.

I've updated the article to reflect this.



glennan says:

August 26, 2014 at 9:06 pm

Very good point Steve.

I've updated the article to reflect this.

Thanks.



Marcel Wid says:

August 27, 2014 at 9:54 am

Your code examples in section Under the hood (again) are wrong:

1. "It is unspecified whether additional unnamed non-static data members are declared in the closure type for entities captured by reference." (§5.1.2p15)
2. This is a serious mistake! The closure type (your class `_SomeCompilerGeneratedName_`) is local to main, i.e. it is defined inside main! And you don't pass a pointer to `for_each`. The object `_inst001_` is never created as lvalue. The correct compiler generated code is `"for_each(v.begin(), v.end(), _SomeCompilerGeneratedName_(offset));"`
3. To be even more precise, the closure type has a deleted default constructor.



glennan says:

August 27, 2014 at 5:25 pm

Marcel,

Good points, there.

I don't disagree with you. The idea of the 'conceptual equivalence' example was to echo the earlier functor example, to help illustrate to the reader approximately what was happening. It was never intended to be a literal 'what the compiler generates'.

What you've written is, of course, quite correct. Apologies for the confusion.



Marcel Wid says:

August 28, 2014 at 11:31 am

Well, I see.

But it is even conceptually wrong to pass a pointer to a functor (&_inst001_) to for_each. If you want the analogy to functors then you should pass (the lvalue) _inst001_ itself as you did in the functor example where you pass f and not &f.



glennan says:

August 28, 2014 at 11:52 am

Quite right; and updated. 😊



panovr says:

September 3, 2014 at 1:57 pm

"This means I can replace my manually-created functor in the STL algorithm with a lambda:"

```
auto lambda = [](X& elem) -> void { elem.op(); }
```

should be:

```
auto lambda = [](X& elem) -> void { elem.op(); };
```



glennan says:

September 3, 2014 at 2:42 pm

It is now! 😊

Thank you!



**panovr** says:

September 4, 2014 at 2:21 pm

You wrote: "Callable object is a generic name for any object that can be called like a function:

1. A member function (pointer)
2. A free function (pointer)
3. functor
4. lambda"

And there are three examples using `std::function` with functor, free function and lambda.

How about a class member function? Can it be used with `std::function`?

**Andrei Zissu** says:

September 5, 2014 at 9:28 pm

panovr,

You can bind a member function indirectly, as you also have to bind to a specific object instance (unless it's a static member function, which is just like any other non-member function). You can accomplish that by either providing a lambda which itself makes the actual call, or by using `std::bind`.



Pingback: [Using C++11 Lambdas with ELLCC | The ELLCC Embedded Compiler Collection](#)

**dorodnic** says:

December 9, 2014 at 7:40 am

Hi,

Great article, thank you!

I was curious how the following code works under the hood:

```
function createFunc(int x)
{
    return [=]() { return x; };
}
```

In particular, where the generated object is stored?

**JayArby** says:

May 23, 2015 at 5:19 pm

Thoroughly demystifying. Thank you!



Andreas says:

July 22, 2015 at 3:47 pm

would you mind telling how you created your code images? I really like the the pseudo hand-writing.



Glennan Carnie says:

July 22, 2015 at 3:52 pm

All the code snippets are created in PowerPoint (yup, really!).

The font used for the code is Consolas

The hand-written font is Segoe Print.



shashank says:

October 11, 2015 at 10:29 am

one of the best writeups on lambdas I ve found! really helpful , thanks!



Sharath says:

February 7, 2018 at 6:26 pm

This is exactly the detailed documentation I was looking for . Kudos for writing up this !



Glennan Carnie says:

February 7, 2018 at 6:35 pm

Really glad it was useful.

I'll be delivering this material as a webinar next month, so please sign up! 😊



Enrico says:

September 6, 2018 at 10:03 am

A question about "Lambdas within member functions": what if I need to capture "this" in a static member function?

Use case is I have a library which exposes a setCallback() and so the function I define need to be static. However, what if I want to register a member function instead of a "classic" function? Examples around the web are not enlightening... 😊



Like (0)



Dislike (0)

Pingback: [\[Перевод\] Лямбды: от C++11 до C++20. Часть 1 – СЕРА website](#)

Sticky Bits – Powered by Feabhas

 Proudly powered by WordPress.