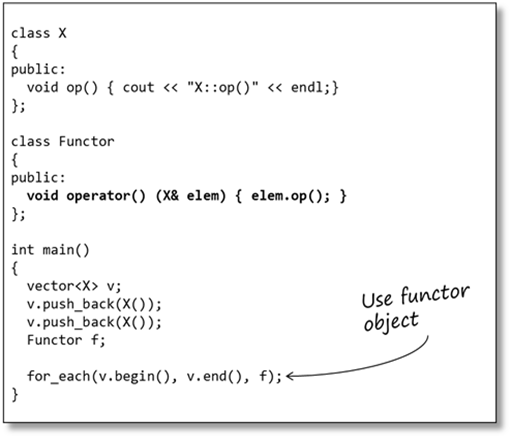
[Demystifying C++ lambdas](http://blog.feabhas.com/2014/03/demystifying-c-lambdas/" \o "Permalink to: Demystifying C++ lambdas)

**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.

[](http://blog.feabhas.com/wp-content/uploads/2014/02/image.png)

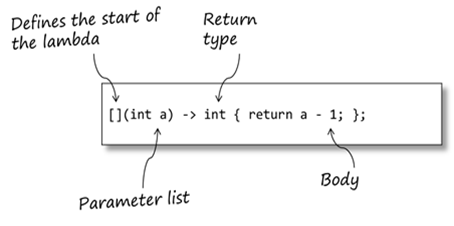
A functor is a class that provides an implementation of operator().

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.

[](http://blog.feabhas.com/wp-content/uploads/2014/02/image1.png)

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.

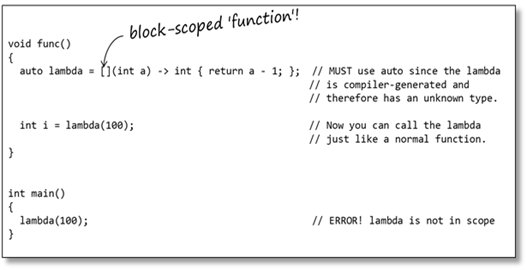
The body of the lambda is just a normal function body, and can be arbitrarily complex (although, as we’ll see, it’s generally good practice to keep lambda bodies relatively simple).

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:

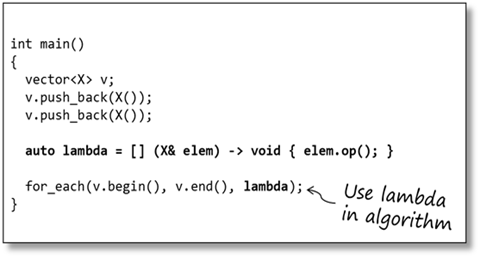
* The return type is void
* The compiler can deduce the return type (lambda body is return *<type>*)

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.

[](http://blog.feabhas.com/wp-content/uploads/2014/02/image2.png)

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:

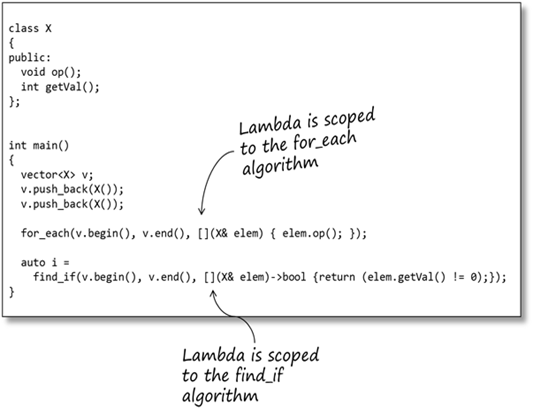
[](http://blog.feabhas.com/wp-content/uploads/2014/02/image3.png)

In the code above the for\_each algorithm calls the lambda with each element in the container it turn.

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.

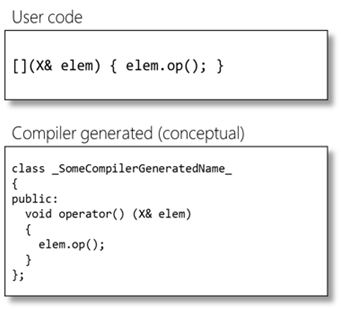
[](http://blog.feabhas.com/wp-content/uploads/2014/02/image4.png)

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).

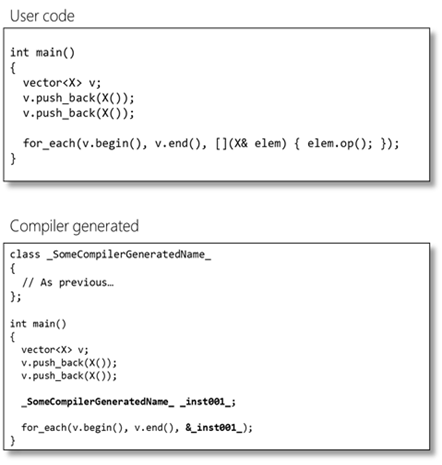
From a readability perspective we have put the functionality right where it is being used; unlike a functor, which may well be defined in another module (that the programmer has to go and find, and/or understand out of context to where it’s being used).

**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)

[](http://blog.feabhas.com/wp-content/uploads/2014/02/image5.png)

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.

[](http://blog.feabhas.com/wp-content/uploads/2014/02/image6.png)

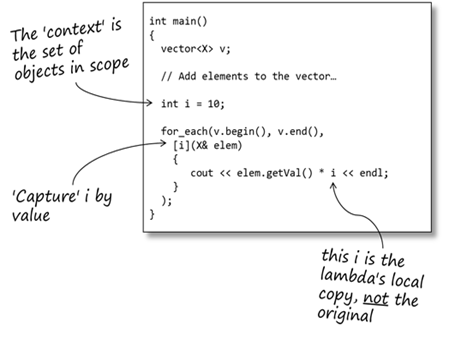
**Capturing the context**

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?)

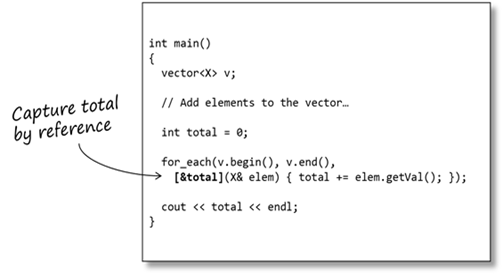
If you were writing your own functor you could do this by passing in the appropriate parameters to the constructor of the functor. C++ provides a convenient mechanism for achieving this with lambdas called ‘capturing the context’.

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.

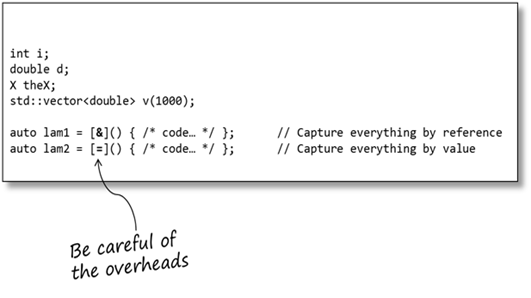
[](http://blog.feabhas.com/wp-content/uploads/2014/02/image7.png)

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.

[](http://blog.feabhas.com/wp-content/uploads/2014/02/image8.png)

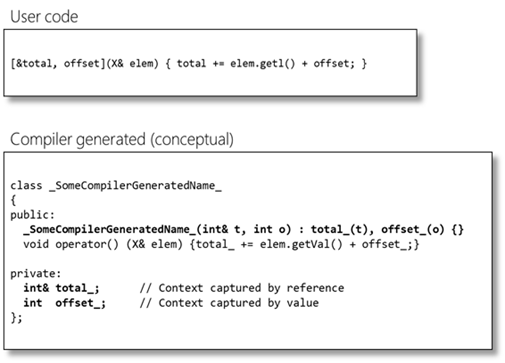
A word of warning here:  A lambda is, as we’ve seen, just an object and, like other objects it may be copied, passed as a parameter, stored in a container, etc. The lambda object has its own scope and lifetime which may, in some circumstances, be different to those objects it has ‘captured’.  Be very 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 (but be careful of the overheads of doing so) – the compiler must make copies of all objects (including copy constructors), or keep references for every object that is currently in scope.

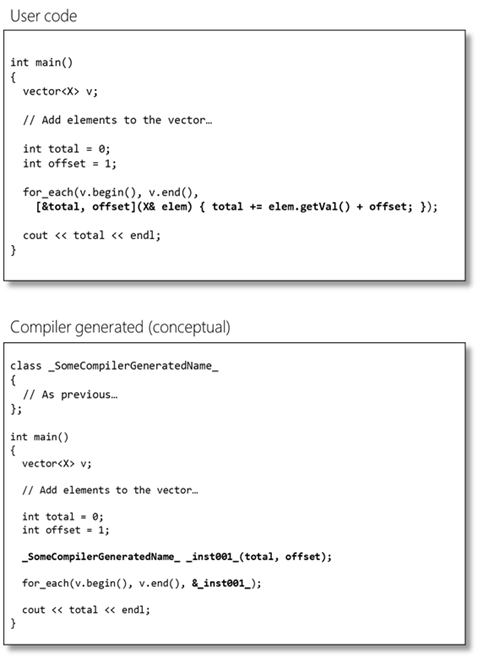
[](http://blog.feabhas.com/wp-content/uploads/2014/02/image9.png)

**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.

[](http://blog.feabhas.com/wp-content/uploads/2014/02/image10.png)

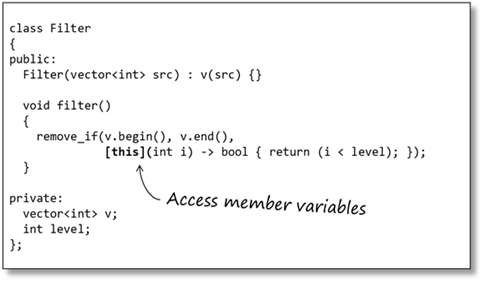
It is relatively easy to see now why capturing the context has potential overheads: for *every* object captured by value a copy of the original is made; for every object captured by reference a reference is stored. (Therefore you might want to think twice about capturing everything in the context by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

[](http://blog.feabhas.com/wp-content/uploads/2014/02/image11.png)

**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).

[](http://blog.feabhas.com/wp-content/uploads/2014/02/image12.png)

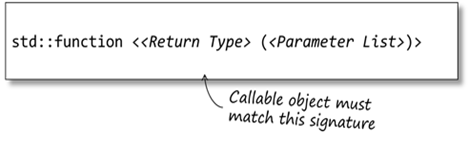
**Callable objects**

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

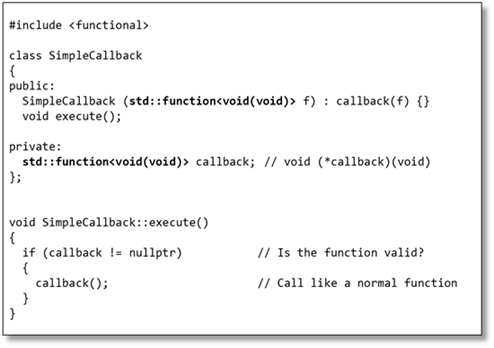
* A member function (pointer)
* A free function (pointer)
* A functor
* A lambda

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.

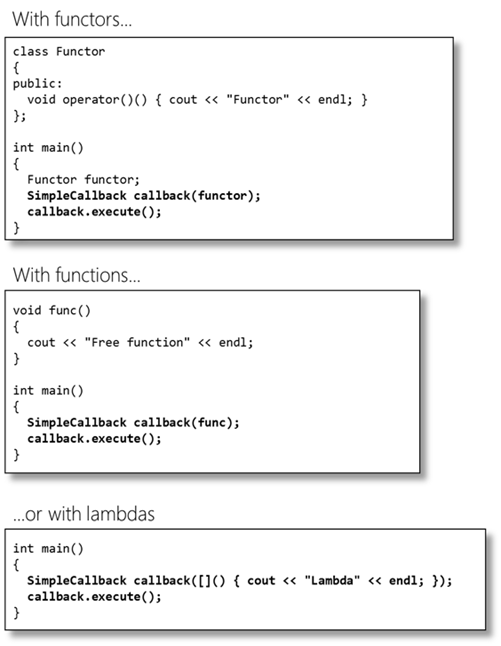
[](http://blog.feabhas.com/wp-content/uploads/2014/02/image13.png)

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).

[](http://blog.feabhas.com/wp-content/uploads/2014/02/image14.png)

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.

[](http://blog.feabhas.com/wp-content/uploads/2014/02/image15.png)

**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.

Posted on March 7th, 2014  
» [Feed to this thread](http://blog.feabhas.com/2014/03/demystifying-c-lambdas/feed/)  
» [Trackback](http://blog.feabhas.com/2014/03/demystifying-c-lambdas/trackback/)

**1 Comment a “Demystifying C++ lambdas”**

1. http://1.gravatar.com/avatar/5e3946b1dc751bbb13bad4f72f07bc18?s=40&d=http%3A%2F%2F1.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D40&r=G***Dan*** says:   
   [March 9th, 2014 at 3:44 pm](http://blog.feabhas.com/2014/03/demystifying-c-lambdas/#comment-4352)

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.