Using C++ Lambdas

David Kieras, EECS Department, University of Michigan

February 27, 2015

Overview

This tutorial deals with C++ *lambda* facility (C++11 and later) that allows one to write un-named functions "in place", which makes the Standard Library algorithms much more usable. Moreover, part of the lambda concept is that variables from the local context can be "captured" and used in the function without being passed in as parameters.

While a lambda doesn't have to be used with one of the Standard Library algorithms, this is by far its most common purpose. Let's start with a simple example of what you had to do pre-lambda to apply a simple function to a container with an algorithm and what you can do post-lambda. Suppose `int_list` is a `std::list` of integers, and you want to print them out in an unusual "custom" fashion, with a colon before and after each value. Pre-lambda, a typical way would be the following:

```
// define a special-purpose custom printing function
void print_it (int i)
{
    cout << ":" << i << ":";
}
...
// apply print_it to each integer in the list
for_each(int_list.begin(), int_list.end(), print_it);
cout << endl;
```

This seems simple enough. However, if `print_it` is never used anywhere else, it seems like a bad idea to have this special-case function elevated to the status equivalent to all the other functions. Conceptually, it doesn't deserve to have the status of a full-fledged function with its own name, callable from anywhere in the present module. Surely it would be better to have a way to write code that is like a function, but without the normal function declaration coding work that also makes it an "official" function with its own name! Lambdas provide an ability to write a un-named function "in place" in the code where it is called. For this example, all we need is:

```
for_each(int_list.begin(), int_list.end(), [](int i){cout << ":" << i << ":";} );
cout << endl;
```

The lambda expression is bold-face above. It starts with the square open/close brackets, followed by a function-like parameter list and a function-like body in curly braces. This whole thing is in the "function" slot of the `for_each` call, meaning you can place a lambda anywhere you can place a function name. In effect, you get to write the function "in place" in the code that calls it, instead of declaring and defining it separately. The lambda behaves like a function in the `for_each`, but it did not have to be declared or defined anywhere else, and it doesn't have a name, and does not exist outside the scope of the `for_each`. The kinky syntax is an attempt to making writing a lambda as simple and compact as possible, with no new keywords needed in the language. ("Hooray!" say the C++ gurus!)

First, why is it called "lambda"? This is a term from the dawn of programming and computer science theory. In the LISP language (which was actually one of the first high-level languages), you could define something very much like the above un-named "in place" function, using the LISP keyword `lambda` which in turn was taken from mathematical recursive function theory, where λ was used to denote a function.

A lambda expression in fact creates a thing that can be saved and treated like a function pointer or function object; this is how the lambda in the above example works in the "function" slot of the `for_each` function template. Although it would be unusual to do so, you can call the lambda in the same statement that creates it, by following it with the function call syntax of open/close parentheses around an optional argument list:

```
[](int i){cout << ":" << i << ":";} (42);
```

The function call operator `(42)` at the end calls the lambda function with 42 as the value for the parameter `i`, producing the output ":42:"

Equally unusually, you can save the lambda in a variable like a function pointer. Fortunately, you do not have to know the mysterious exact type of the variable, thanks to the C++11 `auto` keyword, which allows you to declare a variable that has the same type as its initializing value. So we could store the above example lambda in a variable, and then call it using the syntax that we would also use a function pointer or function object, as follows:

```
auto func1 = [](int i) {cout << ":" << i << ":";};
func1(42);
```

This will output ":42:". The examples below will use some lambdas stored in variables to help illustrate the concepts. But almost always in practice, lambdas are "bare" - their key ability is to allow you to define a function "in place", and thus storing a lambda in a named variable is usually beside the point and wastes time and space. If the function is too complicated to be readable when written as a bare lambda, it would be better to make it a normal named function or function object class.

Lambda Basics

Basic Lambda Syntax

A lambda expression consists of the following:

```
    [capture list] (parameter list) {function body}
```

The capture list and parameter list can be empty, so the following is a valid lambda:

```
[](){cout << "Hello, world! << endl;}
```

The parameter list is just like a function parameter list - a sequence of parameter types and variable names, and follows the same rules as for an ordinary function. The function body is likewise an ordinary function body and a return value is possible. What is the type of the return value? If there is no `return` statement in the function body, the return type is assumed to be `void`. If the function body consists of only a `return` statement (which is very common), the return type is assumed to be the same as the type of the value being returned.

For example, with this lambda, the compiler assumes that the return type is `void`, so calling it without any use of the return value is legal:

```
[](){cout << "Hello from trivial lambda!" << endl;} ();
```

However, trying to use the return type of the call by outputting it is not legal - there is no return value, so the following won't compile:

```
// cout << [](){cout << "Hello from trivial lambda!" << endl;} () << endl;
```

The following lambda takes two integers as parameters and returns a `bool` value which is true if the first integer is half the value of the second. The compiler knows a `bool` is returned from the lambda function because that's what the return statement returns:

```
if([](int i, int j){return 2*i == j;}(12, 24))
    cout << "It's true!" << endl;
else
    cout << "It's false!" << endl;
```

Thanks to these rules, in many cases, you define your lambda in place by typing only a tiny bit of code. That's a good thing, right? But what happens if you need to specify a conversion of the returned value from one type to another, or your function body consists of more than just a single return statement? In these cases you have to specify the return type using a bit of wacky syntax borrowed from elsewhere in the language (no new keywords or operators! hooray!). In the space between the parameter list and the function body, you use the arrow operator followed by the return type. For example, here we have a somewhat complicated function body with multiple returns; we need to help the compiler out by specifying the return type:

```
cout << "This lambda returns " <<
    [] (int x, int y) -> int {
        if(x > 5)
            return x + y;
        else if (y < 2)
            return x - y;
        else
            return x * y;
        }
    (4, 3) << endl;
```

In the following lambda, we tell the compiler that an `int` needs to be returned, even though the return statement provides a `double`.

```
cout << "This lambda returns " <<
    [](double x, double y) -> int {return x + y;} (3.14, 2.7) << endl;
```

The output is "This lambda returns 5".

Lambdas have a Super Power - Capturing the Context

So far, it looks like lambdas are a handy way to write a simple function in place. But actually, they are a lot smarter than just a function; lambdas have some capabilities that are like function objects. That is, a lambda can store information about variables that are in the local block scope at the point of the lambda's creation, and the lambda function body can refer to those variables using the same name as in the surrounding scope. This behaves something like having a nested scope done with curly brackets:

```
{
    int int_var = 42;
    double dbl_var = 3.14;
    {
        int i = 7;       // inside a scope nested within the outer scope
        cout << int_var << ' ' << dbl_var << ' ' << i << endl;
    }
}
```

Notice how `int_var` and `dbl_var`, declared in the outer scope, are known inside the inner scope. You can't get this effect with a function in C or C++ because you aren't allowed to define a function inside another function. But you can get a similar effect with a lambda by "capturing" variables that are currently in local block scope[1] when the lambda is created. For example, the following lambda almost corresponds to the above example.

```
{
    int int_var = 42;
    double dbl_var = 3.14;
    [int_var, dbl_var] ()
    {
        int i = 7;
        cout << int_var << ' ' << dbl_var << ' ' << i << endl;
    } ();
}
```

The output would be the same as the previous code, namely "42 3.14 7". The stuff inside the square brackets names the two variables that we want to "capture" and use inside the lambda. Notice how they have the same names in the capture specification and in the lambda function body as they have in the enclosing scope - so these are not parameter values, but rather a way to allow the function body to refer to variables that would otherwise be outside the function scope.

How is this done? It is not done by breaking a fundamental concept of C/C++ and allowing you to write functions or function bodies inside other functions. Instead, a lambda is actually implemented like a function object which the compiler cleverly creates using the same variable names. Here is a sketch that shows what this would look like if the compiler generated ordinary C++ code for the above lambda. The concept is that the compiler reads your lambda expression, and then replaces it with code that declares, creates and initializes, and then calls a function object. The function object stores the captured variable values in member variables, and these are initialized when the function

[1] By variables in "local block scope" is meant that you can capture variables that are on the function call stack, which includes not only variables declared and defined within the function body (or sub-scopes within the function body) but also the function parameters. But you can't capture global variables, local static-lifetime variables, or member variables in a class member function (see later in this document about member variables).

object is created.[2] So the effect is like the following where everything after the "under the hood" comment is generated by the compiler in response to your lambda expression:

```
  {
  int int_var = 42;
  double dbl_var = 3.14;
  // the following is "under the hood"
  class The_lambda {
  public:
     The_lambda(int int_var_, double dbl_var_) :
        int_var(int_var_), dbl_var(dbl_var_) {}
     void operator() const // see footnote3
        {
           int i = 7;
           cout << int_var << ' ' << dbl_var << ' ' << i << endl;
        }
  private:
     int int_var;
     double dbl_var;
  };
  // create the function object, initializing with the captured values
  The_lambda x(int_var, dbl_var);
  // call the lambda
  x();
}
```

The point. The variables inside the lambda function body *look like* they are the same variables in the outer scope, because they have the same names, but actually they are member variables that hold a *copy* of the values from the outer scope when the lambda expression is created. If the lambda is only called once, you wouldn't see any difference between the outer values and these stored inner values. But if the outer value changes after the lambda object has been initialized, those changes won't affect what the lambda function does. To illustrate, first let's create a simple lambda with a capture and store it in a variable so we can call the same lambda object more than once:

```
int int_var = 42;
auto lambda_func = [int_var](){cout <<
   "This lambda has a copy of int_var when created: " << int_var << endl;};
lambda_func();
// output:
This lambda has a copy of int_var when created: 42
```

So far, no surprise. The variable `lambda_func` is actually holding something like a whole function object, not just something like a function pointer. Let's play some games here by modifying `int_var`, and calling multiple times the very same function-object-like lambda that we've already created:

```
for(int i = 0; i < 3; i++) {
   int_var++;
   lambda_func();
   }
// output:
This lambda has a copy of int_var when created: 42
This lambda has a copy of int_var when created: 42
This lambda has a copy of int_var when created: 42
```

Whoa! The output is the same all three times, and the same as in the first call of `lambda_func` above. The fact that `int_var` is being incremented in the loop is irrelevant - the lambda is using a stored copy of the value of

[2] Such a lambda object that holds captured values is also called a *closure*.

[3] The function call operator for a lambda is defined as a const member function by default, so if you want the lambda code to modify the captured value inside the lambda for some reason, you need to declare the lambda as mutable, as in:
`[int_var]() mutable { /* code */}`

int_var when it was created. So a good way to describe the capture process is that it *captures and saves the value that a variable had at the time the lambda object was created*.

Now for the really super power. You can also specify capture by reference, which means that instead of storing a copy of the value, the lambda object stores a *reference* to the actual variable. Let's redo the same example, but this time capture int_var by reference:

```
int int_var = 42;
auto lambda_func = [&int_var] () {cout <<
    "This lambda captures int_var by reference: " << int_var << endl;};
for(int i = 0; i < 3; i++) {
    int_var++;
    lambda_func();
    }
```

The capture-by-reference is specified with the & in front of the variable name in the square brackets. The output:

```
This lambda captures int_var by reference: 43
This lambda captures int_var by reference: 44
This lambda captures int_var by reference: 45
```

How about that! Since the lambda contains a reference to the outer int_var, every time the function body is executed, the current value of that variable is looked up and used. The lambda could also modify the outer variable through that reference, say like this:

```
auto lambda_func = [&int_var] () {cout <<
    "This lambda is modifying int_var by adding 5" << endl; int_var += 5;};
```

This will change the outer scope int_var by adding 5 every time we call this lambda object.[4]

Here is an example to make it clear that we don't need to store the lambda object in a variable to use a reference capture, and also that we can use lambda parameters at the same time:

```
    int sum = 0;
    for(int i = 0; i < 3; i++)
        [&sum](int x){sum += x;}(i);
```

When the loop is complete, sum will contain the sum of 0 + 1 + 2. Pretty neat!

A gotcha. Be careful with reference captures. A captured reference is only valid if the referred-to variable still exists, so if you have a lambda with a reference capture that ends up getting used after the original variable goes out of scope, the results are undefined.

Shortcuts. There are two shortcut capture specifications that might be useful on occasion:

[=] capture all variables currently in local block scope by copy of their current values

[&] capture all variables currently in local block scope by reference

Lambdas haven't been available long enough for the community to have evolved recommendations on usage of these shortcuts, but it seems generally better to capture the specific variables needed inside the lambda instead of these "capture everything" options.

Standard Library Algorithms and Lambdas - Great Together!

In C++98, using the Standard Library algorithms was often a painful experience because you had to define helper functions or function object classes just so you could write a supposedly elegant one-line bit of code. Thanks to modern C++, this problem has gone away; almost all of these clumsy special-purpose functions or classes can be replaced with either a lambda, the bind template (a powerful function-object generator), or in simple cases with the easy-as-pie mem_fn adapter.

[4] We don't have to declare this lambda as mutable because its function call operator is not changing a member variable of the function object. The function object contains a reference-type member variable, which is a reference to the actual variable which is outside the function object; the reference-type member variable does not in fact change (it can't). See the Savem example on page 7 below to see what this looks like when written as a function object class.

Let's demo lambda's contribution with a simple but realistic scenario. First we have a `Thing` class, only sketched out here, which has a member function with a reference parameter:

```
class Thing {
public:
    // various public member functions
    void save(ostream& os) const; // write our data to the provided output stream
private:
    // member variables
};
```

Let's suppose we created a bunch of Thing objects with `new`, and saved the pointers in

```
 list<Thing *> thing_ptrs;
```

Now let's tell each Thing in the list to write its output to a file stream object named `out_file` using Thing's `save` function. First, we'll open the file and then use an explicit `for` loop to get the iterator pointing to each Thing pointer, then dereference to get the pointer, and call the `Thing::save` function:

```
ofstream out_file("output.txt"); // open a file for output
for(auto it = thing_ptrs.begin(); it != thing_ptrs.end(); ++it)
    (*it)->save(out_file);
```

This works. Can we do the same thing with the `for_each` algorithm instead? We need something to go into the "function" slot that can be called with the dereferenced iterator. It turns out that due to various technical limitations in C++98, the only Standard way to do this was to define a custom function object class like `Savem` shown below. The `Savem` constructor saves a reference to the output stream in a reference-type member variable (one of the rare cases where you declare a variable of reference type that isn't a function parameter). `Savem's` function call operator takes the dereferenced iterator as a parameter and then provides the saved reference in the call to the `save` member function:

```
ofstream out_file("output.txt"); // open a file for output

// a custom function object class
class Savem {
public:
    Savem(ostream& output_stream_) : output_stream(output_stream_) {}
    void operator() (Thing * ptr) const
    {
        ptr->save(output_stream);
    }
private:
    ostream& output_stream;
};
```

Finally, we are ready to write the `for_each`, by creating an unnamed `Savem` object initialized with the output stream:

```
// tell each thing to save its data to the out_file stream
for_each(thing_ptrs.begin(), thing_ptrs.end(), Savem(out_file));
```

While the last line of code is elegant, look at all the function object code we had to write in order to write that elegant line! Often, `for_each` (and other algorithms) hardly seemed worthwhile to use. In C++11 we don't need the `Savem` function object class; instead we can ask `bind` to create a function object for us (see the `bind` handout), or we can use a lambda to write the relevant function code in place in the `for_each`, capturing the local variable for the output stream by reference:

```
for_each(thing_ptrs.begin(), thing_ptrs.end(),
    [&out_file](Thing * ptr){ptr->save(out_file);} );
```

Notice how the lambda code is mostly what we had to write for the function call operator in the custom function object class, and very little more than we wrote in the explicit `for` loop. So using a lambda in a `for_each` really reduces the amount of code you need to type!

Accessing Member Variables in a lambda that is in a Member Function

If you try to use a lambda inside a member function, and try to capture the member variables of that class, you are in for some potentially puzzling error messages. Let's look at a specific example with a class; only the relevant member functions are shown:

```
class Gizmo {
public:
    // returns the number of ints greater than criterion
    int get_count_over_criterion();
private:
    std::list<int> ints;
    int criterion;
    int count;
};
```

First, let's implement the member function without using a lambda:

```
int Gizmo::get_count_over_criterion()
{
    count = 0;
    for(int i : ints)
        if(i > criterion)
            count++;
    return count;
}
```

That was easy! Now let's do the same thing with a for_each and a lambda, doing what just comes naturally, but which in fact won't work:

```
int Gizmo::get_count_over_criterion()
{
    // this lambda is wrong!
    count = 0;
    for_each(ints.begin(), ints.end(),
        [&count, criterion](int i) {if (i > criterion) count++;} );
    return count;
}
```

The compiler gets upset here and can produce some mind-boggling error messages. Although we are used to writing member variable names in a member function without thinking about it, member variables are not local block scope (stack) variables! These are member variables of the class, in a different scope all together because they are actually being referred to through the `this` pointer! They are not local variables!

Fortunately, it is easy to fix this problem. The `this` pointer is a local variable (although hidden, it is a function parameter) so you can capture it, and capturing by value is fine since it can't be modified anyway. Then in the body of the lambda, the compiler knows that `count` and `criterion` are member variables because it has seen the class declaration, and rewrites the references to them using the `this` pointer which thanks to the capture, it knows about within the lambda body. So all we have to do is write:

```
int Gizmo::get_count_over_criterion()
{
    count = 0;
    for_each(ints.begin(), ints.end(),
        [this](int i) {if (i > criterion) count++;} );
    return count;
}
```