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:"