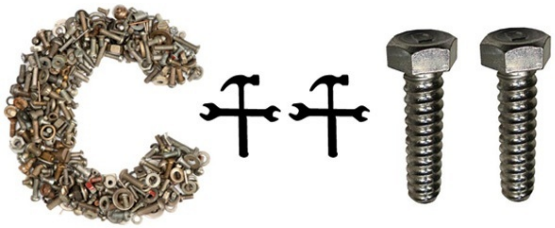
C++11 Tutorial: Lambda Expressions — The Nuts and Bolts of Functional Programming

November 1, 2012 by Danny Kalev 7 Comments

Share

One highlight of C++11 is lambda expressions: function-like blocks of executable statements that you can insert where normally a function call would appear. Lambdas are more compact, efficient, and secure than function objects. Danny Kalev shows you how to read lambda expressions and use them in C++11 applications.



Originally, functional programming in C consisted of defining a full-blown functions and calling them from other translation units. Pointers to functions and file-scope (i.e., static) functions were additional means of diversifying function usage in C.

C++ improved matters by adding inline functions, member functions and function objects. However, these improvements, particularly function objects, proved to be labor intensive.

At last, the final evolutionary stage of functional programming has arrived in the form of lambda expressions.

Lambda by Example

Note: *According to the C++11 standard, implementations* `#include <initializer_list>` *implicitly when necessary. Therefore, you're not supposed to* `#include` *this header in your programs. However, certain compilers (GCC 4.7 for example) aren't fully compliant with the C++11 standard yet. Therefore, if the new initialization notation causes cryptic compilation errors, add the directive* `#include <initializer_list>` *to your code manually.*

Before discussing the technicalities, let's look at a concrete example. The last line in the following code listing is a lambda expression that screens the elements of a vector according to a certain computational criterion:

```
//C++11

vector <accountant> emps {{"Josh", 2100.0}, {"Kate", 2900.0},
{"Rose",1700.0}};

const auto min_wage = 1600.0;

const auto upper_limit = 1.5*min_wage;
```


Popular Posts

Please Stop Saying Java Sucks

14 Ways to Contribute to Open Source without Being a Programming Geniu...

Understanding SOAP and REST Basics And Differences

A Taste of Salt: Like Puppet, But Less Frustrating

There Ain't No ROI in Software Testing

C11: A New C Standard Aiming at Safer Programming

Creating your own TestSteps in soapUI

5 Reasons It's Time to Ditch MySQL

```
//report which accountant has a salary that is within a specific range

std::find_if(emps.begin(), emps.end(),

[=](const accountant& a) {return a.salary()>=min_wage && a.salary() < upper_limit;});
```

As all lambda expressions, ours begins with the *lambda introducer* `[]`. Even without knowing the syntactic rules, you can guess what this lambda expression does: The executable statements between the braces look like an ordinary function body. That's the essence of lambdas; they function (pun unintended) as locally-defined functions. In our example, the lambda expression reports whether the current accountant object in the vector `emps` gets a salary that is both lower than the upper limit and higher than the minimum wage.

Inline computations such as this are ideal candidates for lambda expressions because they consist of only one statement.

Dissecting a Lambda Expression

Now let's dissect the syntax. A typical lambda expression looks like this:

```
[capture clause] (parameters) -> return-type {body}
```

As said earlier, all lambdas begin with a pair of balanced brackets. What's inside the brackets is the optional *capture clause*. I get to that shortly.

The lambda's parameters appear between the parentheses. Although you can omit the parentheses if the lambda takes no parameters, I recommend you leave them, for the sake of clarity.

Lambda expressions can have an explicit return type that's preceded by a `->` sign after the parameter list (find out more about this new functionality in Let Your Compiler Detect the Types of Your Objects Automatically). If the compiler can work out the lambda's return type (as was the case in the first example above), or if the lambda doesn't return anything, you can omit the return type.

Finally, the lambda's body appears inside a pair of braces. It contains zero or more statements, just like an ordinary function.

Back to the `find_if()` call. It includes a lambda expression that takes `const accountant&`. Where did this parameter come from? Recall that `find_if()` calls its predicate function with an argument of type `*InputIterator`. In our example, `*InputIterator` is `accountant`. Hence, the lambda's parameter is `const accountant&`. The `find_if()` algorithm invokes the lambda expression for every `accountant` in `emps`.

Since our lambda's body consists of the following Boolean expression:

```
{return a.salary()>= min_wage && a.salary() < upper_limit;}
```

The compiler figures out that the lambda's return type is `bool`. However, you may specify the return type explicitly, like this:

```
[=](const accountant& a)->bool

{return a.salary()>= min_wage && a.salary() < upper_limit;});
```

Capture Lists

15 Code Editors For the iPad – For Free or Very Cheap

Fabrice Bellard: Portrait of a Super-Productive Programmer

Our Latest Posts!

Is Your Website Getting Coal This Holiday? Get our Performance Checklist December 16, 2015

Code Review Throughout The Years December 10, 2015

Why You Can't Talk About Microservices Without Mentioning Netflix December 8, 2015

The Utopia of API Documentation December 7, 2015

What is your Agile Development Jedi Rank? December 7, 2015

Unlike an ordinary function, which can only access its parameters and local variables, a lambda expression can also access variables from the enclosing scope(s). Such a lambda is said to have *external references*. In our example, the lambda accesses, or *captures*, two variables from its enclosing scope: `min_wage` and `upper_limit`.

There are two ways to capture variables with external references:

- Capture by copy
- Capture by reference

The capture mechanism is important because it affects how the lambda expression manipulates variables with external references.

At this stage I'm compelled to divulge another behind-the-scenes secret. Conceptually, the compiler transforms every lambda expression you write into a function object, according to the following guidelines:

- The lambda's parameter list becomes the parameter list of the overloaded `operator()`.
- The lambda's body morphs into the body of the overloaded `operator()`.
- The captured variables become data members of the said function object.

The compiler-generated function object is called the *closure object*. The lambda's capture clause thus defines which data members the closure will have, and what their types will be. A variable captured by copy becomes a data member that is a copy of the corresponding variable from the enclosing scope. Similarly, a variable captured by reference becomes a reference variable that is bound to the corresponding variable from the enclosing scope.

A *default capture* specifies the mechanism by which all of the variables from the enclosing scope are captured. A default capture by copy looks like this:

```
[=] //capture all of the variables from the enclosing scope by
value
```

A default capture by reference looks like this:

```
[&]//capture all of the variables from the enclosing scope by
reference
```

There is also a third capture form that I will not discuss here for the sake of brevity. It's used in lambdas defined inside a member function:

```
[this]//capture all of the data members of the enclosing class
```

You can also specify the capture mechanism for individual variables. In the following example `min_wage` is captured by copy and `upper_limit` by reference:

```
[min_wage, &upper_limit](const accountant& a)->bool

{return a.salary()>= min_wage && a.salary() < upper_limit;});
```

Finally, a lambda with an empty capture clause is one with no external references. It accesses only variables that are local to the lambda:

```
[] (int i, int j) {return i*j;}
```

Let's look at a few examples of lambdas with various capture clauses:

```
vector<int> v1={0,12,4}, v2={10,12,14,16}; //read about the new
C++11 initialization notation

[&v1](int k) {vi.push_back(k); };  //capture v1 by reference

[&]  (int m) {v1.push_back(m); v2.push_back(m) };//capture vi and
vi2 by ref

[v1]() //capture v1 by copy

{for_each(auto y=v1.begin(), y!=v1.end(), y++) {cout<<y<<" ";}};
```

Name Me a Closure

In most cases, lambda expressions are ad-hoc blocks of statements that execute only once. You can't call them later because they have no names. However, C++11 lets you store lambda expressions in named variables in the same manner you name ordinary variables and functions. Here's an example:

```
auto factorial = [](int i, int j) {return i * j;};
```

This `auto`-declaration defines a *closure type* named `factorial` that you can call later instead of typing the entire lambda expression (a closure type is in fact a compiler-generated function class):

```
int arr{1,2,3,4,5,6,7,8,9,10,11,12};

long res = std::accumulate(arr, arr+12, 1, factorial);

cout<<"12!="<<res<<endl; // 479001600
```

On every iteration, the `factorial` closure multiplies the current element's value and the value that it has accumulated thus far. The result is the factorial of 12. Without a lambda, you'd need to define a separate function such like this one:

```
inline int factorial (int n, int m)

{

return n*m;

}
```

Now try to write a factorial function object and see how many more keystrokes that would require compared to the lambda expression!

> ***Tip**: When you define a named closure, the compiler generates a corresponding function class for it. Every time you call the lambda through its named variable, the compiler instantiates a closure object at the place of call. Therefore, named closures are useful for reusable functionality (factorial, absolute value, etc.), whereas unnamed lambdas are more suitable for inline ad-hoc computations.*

In Conclusion

Unquestionably, the rising popularity of functional programming will make lambdas widely-used in new C++ projects. It's true that lambdas don't offer anything you haven't been able to do before with function objects. However, lambdas are more convenient than function objects because the tedium of writing boilerplate code for every function class (a constructor, data members and an overloaded `operator()` among the rest) is relegated to compiler. Additionally, lambdas tend to be more

efficient because the compiler is able to optimize them more aggressively than it would a user-declared function or class. Finally, lambdas provide a higher level of security because they let you localize (or even hide) functionality from other clients and modules.

With respect to compiler support, Microsoft's Visual Studion 11 and GCC 4.5 support the most up-to-date specification of lambdas. EDG, Clang, and Intel's compilers also support slightly outdated versions of the lambda proposal.

About the author:

Danny Kalev is a certified system analyst and software engineer specializing in C++. Kalev has written several C++ textbooks and contributes C++ content regularly on various software developers' sites. He was a member of the C++ standards committee and has a master's degree in general linguistics.

See also:

- The Biggest Changes in C++11 (and Why You Should Care)
- C11: A New C Standard Aiming at Safer Programming

Webinar On-demand: Secrets of High Quality Development
Insight on Agile best practices from Scrum co-creator, Jeff Sutherland.
View Webinar Now

Click here to subscribe to our blog!

The Biggest Changes in C++11 (and Why You Should Care)
In "C++"

C++11 Tutorial: Introducing the Move Constructor and the Move Assignment Operator
In "C++"

Using constexpr to Improve Security, Performance and Encapsulation in C++
In "C++"

Filed Under: C++, Programming, Programming Languages

Comments

Gustavo **says:**
December 7, 2012 at 7:07 AM

So in summary Lambda Expressions are a way of turning C++ into a lazy and hard to maintain C code?

Reply

Deki **says:**
October 9, 2014 at 10:13 AM

No, but it helps in parallel computation on vectors.

Reply

Jeff Meyer **says:**
June 25, 2013 at 11:48 AM

Great article, Danny. These intro C++11 articles have been an excellent introduction to the language changes incorporated into C++11. A very big help to me.

One request to the SmartBear Blog webmaster and/or Danny: Many of the code examples (i.e., in red text) are often being truncated on the right-hand side.

E.g., the initial code example in the “Lambda by Example” section.

I’ve tried several different browsers (Firefox, Safari, IE) on different systems, and no luck. Nor does decreasing or increasing the size of the text.

I’ve had to go to the HTML source code to extract the examples.

If someone could fix that, it would be appreciated. Apologies for reporting the problem here — I couldn’t find an address for the SmartBear Blog webmaster.

Reply

reza nezami **says:**
October 28, 2015 at 2:25 AM

In chrome it works fine highlighting the code part and copy it out. It copies all the code not just what is visible in the window.

Reply

Rahul Gire **says:**
December 12, 2014 at 7:29 AM

input as a 453 and i want out-put as a three five four
is this possible using inline() in c++……
plz reply
thanks

Reply

Phil Derome **says:**
April 2, 2015 at 8:46 PM

Very superficial minor typos in some of the examples, but the article is great.

[&v1](int k) {vi.push_back(k); }; //capture v1 by reference [CONFUSING vi with v1, and body of lambda can end with } no need for extra ;]
[&] (int m) {v1.push_back(m); v2.push_back(m) };//capture vi and vi2 by ref [PLACEMENT of } and last ;]

Reply

Speak Your Mind

Name *

Email *

Website

Post Comment

Contact Information
Email:
Daniel.Giordano@smartbear.com

Search this website … Search

Stay Connected With Us