C++11 Lambda — Having Fun With Brackets!

Oct 12th, 2011 12:00 am

Lambda

Breaking a complex problem into smaller chunks has obvious advantages and this true with functions as it is with design. However, sanity turns into insanity when the sheer number of such small function tends to be overwhelming. In addition, the boilerplate code needed to use these functions such as a functor, tend to slow you down considerably.

Would it not be nice if there were a way to create functions easily on the fly at the point of call?

Anonymous functions (Lambda functions) have existed for long time in some of the popular languages such as Javascript, C#, python, ruby and most of the functional languages. C++ however, never had such concept of inner functions; and a workaround would be to define local classes (read classes defined in function scope) or to use the famous boost lambda library; but both of these approaches were workarounds to simulate lambda. The new standard C++11 specification includes Lambda Expressions and most of the commercial grade compilers support it. In this article I would be making a flying pass thru this new cool feature.

> ***Note:***
> Lambda expressions are supported in VS2010 SP1 & GCC 4.5 onwards

Hello Lambda

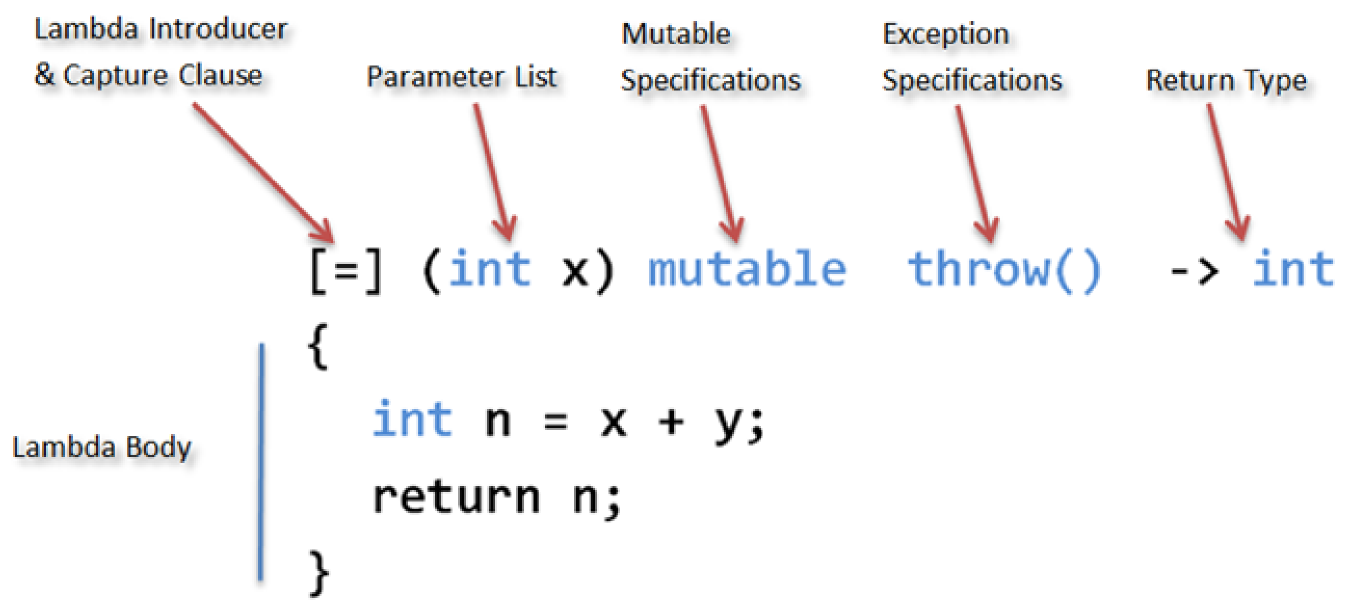
As has been the tradition let's start with hello lambda:

```
#include <string>
#include <iostream>

int main()
{
    //  signature of lambda is void lambda(const std::string& str);
    auto lambda = [](const std::string& str) -> void
    {
        std::cout << str << std::endl;
    };

    std::string hello("Hello Lambda!");
    lambda(hello);

    std::cin.ignore();
    return 0;
}
```

The highlighted piece of code is a lambda function definition. An astute user would notice that it is similar to a functor definition:

```
struct LambdaFunctor
{
    void operator()(const std::string& str)
    {
        std::cout << str << std::endl;
    }
};
LambdaFunctor lambda;
```

Lambda Expression Anatomy



```
[=] (int x) mutable throw() -> int
{
   int n = x + y;
   return n;
}
```

Capture Clause:

A lambda expression can access any variable that has automatic storage duration and that can be accessed in the enclosing scope. The capture clause specifies whether the body of the lambda expression accesses variables in the enclosing scope by value or by reference: variables that have the ampersand (&) prefix are accessed by reference and variables that do not have the & prefix are accessed by value. The empty capture clause, **[]**, indicates that the body of the lambda expression accesses no variables from the enclosing scope.

A variable is explicitly captured if it appears in capture clause, else it is implicitly captured. The body of lambda uses the default capture mode to access variables that are implicitly captured.

Parameter List:

Parameter list for lambda is same as that for any function but with few constraints:

1. No default arguments
2. No variable length arguments
3. No unnamed parameters

Parameter list for lambdas is optional if you are not going to pass-in any parameter, but keeping it there ensures sane code.

Mutable Specification:

It enables the function body to modify the values that have been captured by values. It is optional.

Exception Specification:

As with functions you can have exception specifications with lambdas, I would recommend not using them, but it's really your call whether you want them. It is optional.

Return Type:

Return type for lambda is specified by ***-> return_type***, it is optional only if lambda body contains exactly one statement. However it is good practice to specify return types.

Lambda Body:

It's just like regular function body, can contain anything that an ordinary function would contain; you can define nested lambda expressions as well. Lambda expression can access:

1. Parameters from parameter list
2. Captured variables from enclosing scope
3. Class data members, when defined inside a class (behaves like a member function)
4. Any global variables

Lambdas cannot be a template.

Lambdas all the way

Lambda – currying, captures

In hello world example lambda did not referred to any variable, thus its capture clause was empty. Capture clause is really powerful, flexible as it lets you capture on variables from enclosing scope and is the most important factor for lambda's coolness. Let's see a contrived example where you have been given an array of numbers and have been asked to find if some number exists in it. Here is the first take, off course we will use STL, let's assume we are looking for number 10:

```
struct Comparator
{
    bool operator()(double a) const
    {
        return a == 10;
    }
};
auto iter = std::find_if(numbers.begin(), numbers.end(), Comparator());
```

This works, but is not flexible; we will have to code comparator for each number that we want to find. Second approach is to make comparator take two double values but we still want to use STL:

```
struct Comparator2 : std::binary_function<double, double, bool>
{
    bool operator()(double a, double b) const
    {
        return a == b;
    }
};

auto numberToFind = 10.0;
```

```
Comparator2 comparator;
auto iter2 = std::find_if(numbers.begin(), numbers.end(),
              std::bind1st(comparator, numberToFind));
```

find_if function takes predicate, here comparator object, which takes only one parameter but our Comparator takes 2 parameters, for flexibility. Somehow we need to convert this comparator to take only 1 parameter. This is where bind1st comes into picture, what it does is it creates a wrapper functor taking 1 parameter, and binds 1st argument of comparator to the specified value. Note for this to work you must derive Comparator2 from binary_function. You can imagine how tedious this will become when your comparator takes 3 or more parameters.

Binder were first shot at introducing some kind of closure (variable capturing) support with STL. There are quite a few examples and libraries, like Boost.Lambda and FC++ for higher order function programming, however lambda expressions are coolest as they are language feature. Well this is how the above code will look with lambda:

```
auto numberToFind = 10.0;

//  numberToFind is captured by value from enclosing scope
//  this is an excellent example of closures, as the lambda has two variables
//  one from input argument and other from enclosing scope
//  as the enclosing scope variable is kind of fixed, lambda expression
//  is said to be closed for that variable and open to input argument
//
auto lambda = [numberToFind](double a) -> bool
{
    return a == numberToFind;
};
auto iter_l = std::find_if(numbers.begin(), numbers.end(), lambda);
```

This is so elegant to code. No need to write boiler-plate functors just to enable such semantics.

Lambda – inside a member function

As discussed earlier, you can define lambdas in member function and they will act as if they were member functions! Here is another contrived Sample class that calculates hypotenuse of array of numbers using lambda:

```
#include <vector>
#include <algorithm>
#include <iostream>

class Sample
{
    std::vector<double> hypotenuse;
public:
    Sample() : hypotenuse()
    {}

    template<class Func>
    void ApplyToAll(Func func) const
    {
        std::for_each(hypotenuse.begin(), hypotenuse.end(), func);
    }

    void operator()(const std::vector<double>& x,
                    const std::vector<double>& y)
    {
        //  'this' is captured by value
        //  x and y are capture by reference
        //  no-implicit capture
        //
        auto evaluate = [this, &x, &y]()->void
        {
            if( x.size() != y.size())
            {
                return;
            }

            this->hypotenuse.reserve(x.size());
            for( auto idx = 0U; idx < x.size(); ++idx )
            {
                auto xx = x[idx];
                auto yy = y[idx];

                //  capture implicitly by value
                //  since it is single statement I have
                //  skipped the return type
                //
                auto hypotenuseFunc = [=](){return xx*xx + yy*yy;};
                this->hypotenuse.push_back(hypotenuseFunc());
            }
        };

        evaluate();
    }
};

int main()
{

    static const size_t NumElements = 10;
    std::vector<double> x(NumElements);
    std::vector<double> y(NumElements);
    std::vector<double> z(NumElements);

    for( auto idx = 0U;idx < x.size(); ++idx )
    {
        x[idx] = idx;
        y[idx] = idx;
    }

    Sample smpl;
    smpl(x,y);

    smpl.ApplyToAll([](double dbl) ->void
                    {std::cout << dbl << std::endl;});

    std::cin.ignore();
    return 0;
}
```

In this class, **evaluate** is the lambda expression, it captures “this” by value and x,y by reference. Lines 68-69 demonstrate anonymous function; the function being passed to ApplyToAll is un-named function.

Few observations related to usage of lambdas in member functions:

1. They mimic member functions, but are not member functions
2. They cannot be defined virtual, so can’t be overridden
3. Lambda defined in const function cannot mutate, or change, any member variable, unless that member variable is marked mutable explicitly
4. Can be passed to any other higher order function, a function that takes function as one of the input parameters, as they are not member functions

Recursion:

Lambdas can be recursive as ordinary functions are; however you need to name lambdas to make recursion work. Here is simple example of recursive lambda to calculate factorial:

```
#include <iostream>
#include <functional>

int main()
{
    //  function factorial needs to be captured as we are going to call it
    //  recursively
    //
    //  we could have code like:
    //  auto factorial = lambda_expression
    //
    //  but standard requires auto to be defined completely before it is
    //  used, and in case of lambda expression definition is not complete
    //  till you define the entire body, which cannot use itself again
    //  unless it is named thus we capture lambda expression with a
    //  std::function object and then use it
    //
    std::function<int(int)> factorial = [&factorial](int val)->int
    {
        if( val <= 1 )
        {
            return 1;
        }
        else
        {
            return val * factorial(val-1);
        }
    };

    for( auto idx = 0; idx < 10; ++idx )
    {
        //  either we can just capture everything, read [&], or just the required stuff,
        //  in this case read [&factorial], either is ok
        //  I prefer to be explicit on what needs to be captured
        //  so that there are no surprises
        //
        auto printFn = [&](int idx){std::cout << "Factorial of "
                                << idx << " is "<< factorial(idx) << std::endl;};
        printFn(idx);
    }

    std::cin.ignore();
    return 0;
}
```

Summary

Lambdas (Lambda Calculus) were first introduced by Alonzo Church as a formal system for function definition, application and recursion. It was proved that Lambda Calculus is Turing complete, there by making it theory of computation and then it made its way into functional languages.

Since then Lambda Expressions have been extensively studied, pure functional languages are all lambda based. However hybrid languages, like javascript, lisp and perl, did incorporate functional features like function composition (higher order functions), captures, currying.

C++ STL did introduce some functional concepts like higher order functions, std::find_if taking a predicate, but it has been of very limited form. With C++11 language support now for the first time function objects are truly first class concept in C++.

I have tried to cover most of the aspects of lambda expressions in C++11, but this is not all. I will post as and when I find something to share.

Reading Material:

- Lambda Expressions and Closures (open-std.org)
- A proposal to add lambda functions to the C++ standard (open-std.org)
- Lambda expressions and closures for C++ (open-std.org)
- New wording for C++0x Lambdas (open-std.org)
- lambda functions proposal (comp.std.c++)
- Lambda parameters must have names (comp.std.c++)
- functional programming(nullptr.me)
- callable entity(nullptr.me)
- C++0x: Protected/private member access from lambda functions (comp.std.c++)
- lambda non-parameter variable binding (comp.std.c++)
- Lambda abstractions in C++ vs. Scheme (comp.lang.functional)
- lambda Functions??? (comp.lang.c++.moderated)
- Lambda expressions (i.e. anonymous functions) in C++ (comp.lang.c++.moderated)
- Templated Lambda Functions? (comp.lang.c++.moderated)

Authored by Sarang Baheti Oct 12th, 2011 12:00 am general

Tweet G+1 0 Like 8

« history of digital storage apple product design over 35 years »

Copyright © 2015 - Sarang Baheti - Powered by Octopress | Themed with Whitespace