informIT

C++

Lambda Expressions and Closures, Part I

Last updated Jul 11, 2008.

The C++ standards committee approved the lambda expressions proposal at the Bellevue meeting of February 2008. The latest version of the proposal is very different from the initial draft that I presented here in 2005. The terminology has changed, as have the syntax, semantics, usage and underlying implementation of lambda expressions. In the first part of this series I introduce the fundamental concepts of the latest lambda proposal.

Why Use Lambda Expressions?

A *lambda expression* (also known as a *lambda function*) is nameless function defined at the place of call. As such, it's similar to a function object. Indeed, lambda expressions are automatically transformed into function objects. So why not use function objects in the first place? As useful it may be, creating a function object is a laborious task: you have to define a class with data members, an overloaded function call operator and a constructor. You then have to instantiate an object of that type at the place of all. This is very verbose and ånot well-suited for creating a one-time function object that is used "on the fly".

To demonstrate the usefulness of lambda expressions, suppose you need to find the first employee whose salary is within a given range. Using traditional and verbose function objects, you could first write a function class called `withinrange`:

```
class withinrange {
 double low, high;
public:
 withinrange(double l, double h) : low(l), high(h) { }
 bool operator()(const employee& emp) {
  return emp.salary() >= low && emp.salary() < high;

  }
};
```

Next, you use the `find_if` algorithm to locate the first employee whose salary is within the specified range from a sequence of employees:

```
double minimum_salary=1000;
std::find if(employees.begin(), employees.end(),
withinrange(minimum_salary, 1.25*  minimum_salary));
```

The third argument of the `find_if` call is a function object that in other programming languages is called a *closure*. A closure is an unnamed function object that stores the said function's *environment*. The environments consists of local variables that the function accesses. In this case, the data members `low` and `high` are the environment stored in the closure. In simpler words, a closure is the hypothetical function object that the compiler will generate for a given lambda expression.

Lambda Expression Usage

Using the new lambda proposal, the above `find_if` call can be rewritten as:

```
double minimum_salary = 1000;
double upper_limit = 1.25 * minimum_salary;
std::find if(employees.begin(), employees.end(),
[&](const employee& emp)  (emp.salary() >= minimum_salary && emp.salary() < upper_limit));
```

First, notice that the lambda expression is self-contained within a single line of code. You don't need a separate function class anymore.

A lambda expression begins with the *lambda introducer* `[]` (I will discuss the meaning of the `&` between the brackets in a different part of this series). The lambda expression's parameter list appears after the lambda introducer. In this example, the parameter list consists of the sole parameter `const employee&`. This entire lambda expression is said to be *monomorphic* because the types of its parameters are explicitly specified. Here, the type of the sole parameter `emp` is `const employee&`. A polymorphic version of the same expression would be:

```
[&](emp) (emp.salary() >= minimum_salary && emp.salary() < upper_limit)
```

The latter form requires that the parameter types shall be deduced from the context (the place of call). The current proposal focuses only on monomorphic lambda expressions so I will not discuss polymorphic lambdas in this series.

Implicit and Explicit Return Types

The last part of the previous lambda expression:

```
(emp.salary() >= minimum_salary && emp.salary() < upper_limit)
```

is the lambda expression's *body*. A lambda expression's body can consist of a single parenthesized expression. In that case, the return type of the lambda function is implicitly deduced from the expression itself. For example, the following expression yields a `bool` result:

```
(emp.salary() >= minimum_salary && emp.salary() < upper_limit)
```

Therefore, the return type of the above lambda expression is `bool`.

Technically speaking, if the return type isn't explicitly specified in a lambda expression, it's defined as `decltype(e)` where `e` is the body of the lambda expression.

You may specify the return type of a lambda expression explicitly, though. Using the new function declaration syntax, here's how you do it:

```
[&](emp) ->bool (emp.salary() >= minimum_salary && emp.salary() < upper_limit)
```

Lambda Expression Body

A lambda expression's body may contain more than one statement. In such cases, the entire body is enclosed within a pair of braces and must have an explicit return statement. The following lambda expression takes two parameters of type `int` and has a return type `int`. Its body consists of three statements enclosed in a `{}` block:

```
[](int x, int y) -> int { int z; z = x + y; return z; }
```

The return statement in this example is mandatory because the lambda expression's body consists of multiple statements. Likewise, the explicit return type after the parameter list is mandatory as well.

External References

Lambda expressions are divided into two major categories: lambda expressions with no external references and those with external references. The latter access variables that are defined outside the lambda expression's parameter list, as opposed to lambda expressions that do not access variables defined outside the lambda expression's parameter list. Here's a lambda expression with no external references:

```
[](int x, int y) -> int { return x + y; }
```

Here's one with external references:

```
int z;
myfunc([](int x, int y) -> int { return x + y + z; } );//pseudo code
```

References to local variables declared outside of the lambda function bodies have been debated for a long time. The problem is that any local variable referenced in a lambda function body, e.g., `z` in the previous example, must somehow be stored in the resulting closure. How these variables are exactly stored in the closure is the disputed issue. Some proposed that copies of the external variables shall be stored in the closure. Copying however can be inefficient in some cases and might also lead to slicing and iterator invalidation. The other solution was to store references to the external variables in the closure. This approach can also be problematic as it could lead to dangling references. In the second part of this series I will show how the latest proposal solved the external references problem and how external references are represented in a closure.

informIT

C++

Lambda Expressions and Closures, Part II

Last updated Jul 18, 2008.

The second part of this series will discuss the mechanism by which lambda functions access variables that are declared in their enclosing scope and how the compiler-generated closure object associated with a lambda expression is formed.

Lambda Captures

Recall that every lambda function is preceded by the lambda introducer `[]`. A lambda introducer may contain a *lambda capture* i.e., a description of the environment that the lambda expression will access. An empty lambda introducer indicates that the lambda expression doesn't have external references; the lambda expression in this case accesses only variables declared in its parameter list and variables declared inside the lambda expression's body.

Let's look back at an example of a lambda expression which references variables defined outside the lambda parameter list:

```
double minimum_salary = 1000;
double upper_limit = 1.25 * minimum_salary;
std::find if(employees.begin(), employees.end(),
[&](const employee& emp)  (emp.salary() >= minimum_salary && emp.salary() < upper_limit));
```

The lambda introducer here includes the *capture-default &*. The presence of a capture of any type means that the lambda expression refers to variables declared in its enclosing scope. Indeed, this lambda expression accesses the variables `upper_limit` and `minimum_salary`.

As you already know, every lambda expression is translated to a *closure* -- an anonymous function object. Closures are instances of local classes that the compiler implicitly declares in the enclosing scope of the lambda expression. There's a correlation between the lambda expression's external references and the corresponding local class: each variable with an external reference is represented as a data member of the local class:

```
class __local_lambda_expression //1 ompiler-generated local class
{
 double upper_limit_&, minimum_salary_&;
public:
 __local_lambda_expression(double upper, double minimum) :  upper_limit(upper)_, minimum_salary(minimum) {}
//the lambda expression's body corresponds to the overloaded () operator:

 auto operator()(const employee& emp) const->decltype(emp.salary() >= minimum_salary && emp.salary() < upper_limit)
 {
   return  emp.salary() >= minimum_salary && emp.salary() < upper_limit;
 }
};
```

At the place of call (i.e., where the lambda expression is defined), the compiler instantiates a closure, initializes it with the two variables declared in the enclosing scope and invokes the overloaded `()` operator like this:

```
std::find if(employees.begin(), employees.end(),
__local_lambda_expression(upper_limit, salary)(employees)); //pseudo C++ code
```

In summary, the lambda expression and its corresponding local class and anonymous function object are related to one another in the following manners:

- Variables with external references correspond to data members of the local class.
- The lambda parameter list corresponds to the argument(s) passed to the overloaded `()` operator call.
- The lambda function's body corresponds more or less to the body of the overloaded operator `()` defined in the local class.
- The return type of the overloaded `()` operator is automatically deduced from a decltype expression.

Capture-Defaults

Earlier drafts of the lambda proposals hesitated quite a bit about the way in which external references should be represented in a closure. The authors decided eventually to avoid a default argument passing mechanism, forcing the programmer to specify explicitly whether the closure's data members are references to variables defined in the enclosing scope or copies of those variables.

There are two types of *capture-defaults*: *&* and =. They correspond to references and copies, respectively. The introducer `[&]` indicates that the closure's data members are reference variables whereas `[=]` indicates copy semantics. The following lambda expression:

```
std::find if(employees.begin(), employees.end(),
[=](const employee& emp)  (emp.salary() >= minimum_salary && emp.salary() < upper_limit));
```

Will cause the compiler to declare the following local class:

```
class __local_lambda_expression
{
 double upper_limit_, minimum_salary_;//data members are copies of the external variables
public:
 __local_lambda_expression(double upper, double minimum) :  upper_limit(upper)_, minimum_salary(minimum) {}
 auto operator()(const employee& emp) const->decltype(emp.salary() >= minimum_salary && emp.salary() < upper_limit)
 {
  return  emp.salary() >= minimum_salary && emp.salary() < upper_limit;
 }
};
```

Capture Lists and `this`

In addition to the two capture-defaults, you may spell out explicitly the passing mechanism of each externally referenced variable in a *capture list*. A capture list contains a list of variables with external references that are accessed by the lambda expression:

```
int z;
double d;
myfunc([z, &d](int x, int y) -> int { return x + y + z +d; } )
```

Here, the capture list `[z, &d]` declares `z` and `d`. These declarations refer to the variables declared in the enclosing scope of the lambda expression. `z` is passed by value and `d` is passed by reference. Consequently, the closure associated with this lambda expression will be:

```
class __local_lambda_expression //local class
{
 int z, double &d;
public:
 __local_lambda_expression(int __z, double & __d) : z(__z)_, d(__d) {}
 auto operator()(int x, int y) const->int //return value specified explicitly in lambda expression
 {
  return x + y + z +d;
 }
};
myfunc(__local_lambda_expression(z, d) (x,y)); //closure
```

If a lambda function is defined in a member function, the body of the lambda function may contain occurrences of this, either implicitly or explicitly. When the compiler translates the code, these occurrences should refer to the object in whose member function the lambda was defined (not to be confused with the closure object that has been generated for the lambda expression). Consider:

```
class C
{
double a;
public:
double func(int n) {
 [this](int n) (return this->a*n;)
 }
};
```

Here the closure has a data member which by convention is called `__this`:

```
class __local_lambda_expression //local class corresponding to the lambda exp above
{
 C *const __this;
public:
 auto
 __local_lambda_expression(C* athis) : __this(athis){}
 auto operator()(int n) const->decltype(/*..*/)
 {
  return __this->a*n;
 }
};
```

You probably have noticed that the compiler-generated closures for all the lambda expression I've shown declare their overloaded () operator as a const member function. This raises two questions: is this the default behavior? If so, is there a mechanism that allows programmers to override the default? In the next part of this series I will answer these questions and discuss other technical aspects of closures..

informIT

C++

Lambda Expressions and Closures, Part III

Last updated Jul 25, 2008.

The third part of this series will discuss the implementation of closures with respect to the constness of lambda functions, and address a minor technical issue concerning the `decltype` expression that indicates a lambda expression's return type.

Implementation of Closures

By now you know that the compiler translates every lambda expression to an anonymous function object known as a closure. The closure's data members serve as the environment for the lambda function. When the lambda capture uses pass-by-reference the closure's data members are all reference variables. In that case, the closure can store a single pointer to the stack frame where the lambda function is defined, and access the externally referenced variables via that pointer instead of initialization of a long list of data members. This optimization also saves reduces the memory footprint of the closure object.

Constness of Lambda Functions

The compiler-generated closures I've shown so far declare their overloaded () operator as a const member function. The latest lambda proposal states that by default, the closure's overloaded `()` operator shall be const. Effectively, this means that a lambda expression cannot modify the resulting closure's data members. Let's look back at an example of a lambda expression which accesses variables that are defined outside the lambda parameter:

```
double minimum_salary = 1000;
double upper_limit = 1.25 * minimum_salary;
std::find if(employees.begin(), employees.end(),
[=](const employee& emp)  (emp.salary() >= minimum_salary && emp.salary() < upper_limit));

class __local_lambda_expression
{
 double u_limit, m_salary;
public:
 __local_lambda_expression(double upper, double minimum) :  u_limit(upper)_, m_salary(minimum) {}
//the lambda expression's body corresponds to the overloaded () operator:

 auto operator()(const employee& emp) const->decltype(emp.salary() >= m_salary && emp.salary() < u_limit)
 {
   return  emp.salary() >= m_salary && emp.salary() < u_limit;
 }
};
```

The overloaded `()` operator cannot modify the data members `u_limit` and `m_salary` because they are implicitly converted to `const double` when accessed from within a const member function (recall that a const member function of class `T` gets a this pointer that is of type `const T* const`, as opposed to `T* const` in a non-const member function). Preventing the lambda function from modifying the closure's data members has one advantage -- it allows the invocation of a closure object regardless of whether the object is const.

However, this rule is somewhat restricting. Consider the following example:

```
vector<int> vi;
//..
int fac = 0;
transform(vi.begin(), vi.end(), vi.begin(), [fac](int n) { return fac += n; }); //compilation error
```

The compiler-generated closure for the above lambda function is similar to this:

```
class __local_lambda_expression
{
 int _fac;
public:
//.. constructor
 int operator()(int n) const { return _fac += n; }
};
```

The const overloaded `()` operator attempts to modify the data member `_fac`, hence you're getting a compilation error. As a workaround, you can store the state outside of the closure object by changing the capture to:

```
[&fac](int n) { return fac += n; }); //now OK
```

In this case, the data member `_fac` becomes a reference that is bound to the variable `fac` declared right before the `transform()` call. A const member function is allowed to modify a variable that isn't a data member of its object, including by modifying a reference data member. This workaround isn't ideal though. Changing the value of `fac` may not be an expected side effect, particularly when closures are invoked in a multithreaded program.

`mutable` Lambda Expressions

A closure's overloaded () operator is const by default. To override this default the authors propose that the lambda expression shall be explicitly declared as `mutable`. The place for the `mutable` qualifier is right after the parameter list of the lambda expression and before its optional exception specification (yes, lambdas can have an exception specification just like any ordinary function, not that exception specifications have much use in C++ today). The absence or presence of the `mutable` qualification determines whether the closure's overloaded `()` operator is const or non-const, respectively:

```
[fac](int n)  mutable { return fac += n; }); //OK
```

The resulting closure is now:

```
class __local_lambda_expression
{
 int _fac;
public:
//.. constructor
 int operator()(int n) { return _fac += n; } //no longer a const
};
```

`decltype` Issues

The first part of this series it was said that if the return type isn't explicitly specified in a lambda expression, it's implicitly defined as `decltype(e)` where `e` is the body of the lambda expression. Technically speaking, this isn't always entirely correct but if you're not particularly interested in C++ trivia, you may skip this discussion about the `decltype` of lambda expressions.

First let's see what the problem is with `decltype(e)`. In following lambda expression:

```
double arr[] = { 1.1, 2.2, 3.3};
double sum = 0;
int factor = 4;
for_each(arr, arr + 3, [&sum, factor] (double d)( sum += factor*d; ));
```

The implicit return type should seemingly be defined as:

```
struct __local_lambda_expression {
 double& _sum;
 int _factor;
 __local_lambda_expression (double& d, int n): _sum(d), _factor(n) {}
auto operator()(double d) const -> decltype(_sum +=_factor*d) //pseudo code
 { return _sum += factor*d;}
};
```

In reality, the return type cannot be `decltype(_sum += _factor*d)` because the data members `_sum` and `_factor` cannot be used outside the body of the overloaded () operator. Presently, compilers may use the following `decltype` expression instead in order to deduce the return type of the above lambda expression:

```
decltype(fake<double&>() += fake<int&>()*d)
```

In other words, each use of `_sum` and `_factor` is replaced with an expression that has the same types as these variables. The expression `fake<T>()` has the type `T` for any type `T`. `fake()` can be defined as:

```
template <typename T> T fake();
```

In Conclusion

Only a few years ago, lambda expressions were considered experimental, not to say avant-garde. Although some of the details of lambda expressions are still subjected to changes and extensions (for example, polymorphic lambda expressions are yet to be added to C++), it's already obvious that lambda functions will be used extensively in C++0x code both by library implementers and end-users.

The main advantage of lambda expressions is that they eliminate the manual tedium of declaring a class with data members, a constructor and an overloaded `()` operator, and later instantiating a function object. These are the compiler's job from now on. With respect to performance, suffice it to say that good compilers are already able to optimize away much of that code. After all, they've been doing this for years with user-defined function objects.