C++ Functional Parameters

Chris Riesbeck
Last updated: July 3, 2009

Many of the STL generic algorithms take a functional argument. This is a very powerful feature, once you learn how to define such functions in all their glory.

For example, consider `find_if()`:

```
InputIterator find_if(InputIterator begin,
                      InputIterator end,
                      Predicate pred);
```

`find_if` returns an iterator pointing to the first value in the range satisfying the predicate. A predicate is any unary C++ function that returns true or false. The result is `end`, if the value was not found.

Suppose we wanted to find the first even number in some container. The easiest thing to do is to define an even number predicate:

```
bool isEven (int x) { return ((x % 2) == 0); }
```

Then, if `c` is some STL container of integers, we can search it by passing `find_if()` a **function pointer** pointing to the `isEven()` predicate.

```
result = find_if(c.begin(), c.end(), isEven);
if (result != c.end()) {
... do stuff with result ...
}
```

will set `result` to point to the first even number, if any.

From Function Pointers to Function Objects

Function pointers are nice and simple, but they has two disadvantages:

- **Inefficiency**: Every call to `isEven()` in the internal `find_if()` loop requires a pointer dereference. C can't optimize this away because you might change what the pointer points to. Not a big deal, but C programmers often object to any overhead.
- **Inflexibility**: Suppose we wanted to find the first element smaller than the value of some local variable `i`? There's no way to define such a function globally.

Fortunately, both problems can be overcome by using **function objects** instead of function pointers. A function object is an instance of a **function class**, where a function class is a class that overloads the `operator()`. To define a class of function objects that test for even integers, we write:

```
class IsEven {
  public:
    bool operator() (int x) const
    { return ((x % 2) == 0); }
};
```

`IsEven` is a class. The call to the constructor `IsEven()` creates an instance of this class. That instance can be passed as a function. Therefore, to find an even number, we write

```
result = find_if(c.begin(), c.end(), IsEven());
```

Though it's not at all apparent from the syntax, what' s being passed to `find_if()` is a function, not a pointer to a function, and therefore calls to this function inside `find_if()` can be inlined and run more efficiently.

Function Classes and Sets

There's another advantage to function classes. They can be used when constructing containers that sort their contents, like sets and maps. Such containers by default use `operator<`, which often is what you want. But suppose you want a set of objects sorted a different way? No problem: declare the set like this:

```
set<element-type, predicate-class > > s;
```

where *predicate-class* is the name of function class that implements `operator()` to take two parameters and return true if the first is "less" than the second, by some rule. For example, suppose we had a set of students in a course, and assume the class `Student` defines `operator<()` to compare by name. Then

```
set<Student> students;
```

would be a set of student that, when students are added to it, automatically -- and efficiently -- sorts them by name. But sometimes we might want a set of students sorted by their ID number, say for posting grades anonymously. An easy way to do that is to first define a function class to do the sorting we want:

```
class IDCompare
{
public:
   bool operator() ( const Student &s1, const Student &s2 ) const {
     return s1.getID() < s2.getID();
   }
};
```

Not much to that, is there? Now we can create a new set of students, for posting purposes, that has all the data in the other set, but sorted by ID number, with one short statement:

```
set<Student, IDCompare> posting( students.begin(), students.end() );
```

Functions with Local State

Even better though is that that function objects are instances of classes. Instances of classes can have private local variables. Those local variables can be very useful!

For example, consider the problem of finding the first number in some container `c` that is less than some value, say a value entered by the user. We'd like to write something like this:

```
cout << "Enter search value: " << endl;
int x;
cin >> x;
result = find_if(v.begin(), v.end(), ...);
```

What goes in the ...? We need a predicate of one argument that returns true if the argument is less than `x`. Such a predicate can't be defined until we know what `x` is. What to do?

Fortunately, the answer is easy. Look at this function class:

```
class IsLessThan {
  public:

    IsLessThan (double x = 0.0) : val(x) {}

    bool operator() (double x) const
    { return (x < val); }

  private:
    double val;
};
```

This defines a function class. Instances of the class are predicates that take one integer and return true if the argument is less than the value stored in the instance variable `val`. How does val get set? When you construct an instance of the class. The constructor `IsLessThan(`*number*`)` creates a function instance and saves *`number`* in `val`. Therefore

```
IsLessThan(5)
```

returns a predicate that is true when given any number less than 5.

So we can finish our example above with this line of code:

```
result = find_if(v.begin(), v.end(), IsLessThan(x));
```

Here's a more realistic example. Suppose you wanted to look at a container of XY coordinates and find the one closest to some point P.

The STL has this handy generic algorithm that returns the minimum element in a range, and we can pass it the predicate for comparing elements. So the simple solution is something like this:

```
template <class Iter>
Iter findClosest( Iter start, Iter stop, const Point & p )
{
  return min_element( start, stop, &minDist );
}
```

That is, "return a pointer to the element the least distance from `p`."

For this to work, `minDist()` must look something like this

```
bool minDist ( const Point & p1, const Point & p2 )
{
   return distance( p1, p ) < distance( p2, p );
}
```

which says `p1` is less than `p2` if `p1` is closer to `p` than `p2` is.

But the variable `p` is undefined inside `minDist()`! `minDist()` needs 3 values, p1, p2, and the point `p`, but `minElement()` is only going to pass it 2 points, from the container. How does `minElement()` get the other point?

Answer: make a functor class that holds the 3rd value, like this

```
class MinDist {
public:
  MinDist( const Point & p ) : myPoint( p ) {}

  bool operator() ( const Point & p1, const Point & p2 )
  {
    return distance( p1, myPoint ) < distance( p2, myPoint );
  }

 private:
  Point myPoint;
}
```

Notice that most of the code is just like what we wrote for `minDist()` before. All we added was a class with (1) a private data member to hold the extra value, and (2) a constructor to initialize that value.

Now our `findClosest()` looks like this:

```
template <class Iter>
Iter findClosest( Iter start, Iter stop, const Point & p )
{
  return min_element( start, stop, MinDist( p ) );
}
```

The expression `MinDist( p )` constructs an instance of `MinDist`. That instance has `p` stored in `myPoint`. The instance is passed to `min_element()`. Because the instance has `operator()` defined, it can be called like a function. When called, it does the comparison we want.

Function Makers

It turns out that classes like `IsLessThan` are so commonly needed that you don't actually have to define them by hand. There are two tools provided by the STL for creating them on the fly as needed.

First, there are templates for making binary functions out of common binary operators.

For example, suppose you wanted to multiply all the numbers in a vector. It seems like `accumulate()` should do this, except that instead of adding all the numbers, with 0 as an initial sum, we want to multiply, with 1 as the initial product.

Unfortunately,

```
result = accumulate(v.begin(), v.end(), 1.0, *);
```

is not legal C++. `operator*` is not a function, and, besides, which `operator*` do you mean? The one for integers, doubles, user classes, ...?

Fortunately, the STL, in `<functional>`, defines templates that make function classes for all the common operators. In particular,

```
multiplies<type>()
```

returns a function that calls *`type`*'s `operator*`. So we can multiply a container full of numbers with

```
result = accumulate(v.begin(), v.end(), 1.0,
                    multiplies<double>());
```

`<functional>` defines the following templates:

- `plus` for `x + y`
- `minus` for `x - y`
- `multiplies` for `x * y` (formerly called `times`)
- `divides` for `x / y`
- `modulus` for `x % y`
- `negate` for `- x`
- `equal_to` for `x == y`
- `not_equal_to` for `x != y`
- `greater` for `x > y`
- `less` for `x < y`
- `greater_equal` for `x >= y`
- `less_equal` for `x <= y`
- `logical_and` for `x && y`
- `logical_or` for `x || y`
- `logical_not` for !x

All are used similarly to `multiplies`, e.g., `greater<int>()` returns a binary predicate that tests for greater than.

Function Binders

But wait! There's more. Often what we want is to convert a binary two-argument predicate, e.g., less than, into a unary predicate, by replacing one argument with a fixed value. This is called "binding" one of the arguments. In the less-than case given early, we bound the second argument. Another example might be a function that multiplies an input number by a constant value.

This is a very common operation and so there's a convenience template for it. We can convert a binary function to a unary function in two ways:

```
bind2nd(function, value)
```

makes a unary function by binding the second argument of *function* to *value*. Guess what

```
bind1st(function, value)
```

does. So

```
bind2nd(less<int>(), 5)
```

creates a unary function that tests for "integer less than 5," and

```
bind2nd(multiplies<double>(), 1.2)
```

creates a unary function that multiplies its argument by 1.2.

So we could rewrite our original `find_if()`example using

```
result = find_if(v.begin(), v.end(), bind2nd(less<double>(), x);
```

and we don't need to define the `IsLessThan` class at all.

Exercise for the reader: What does

```
result = find_if(v.begin(), v.end(), bind1st(less<double>(), x);
```

search for?

Function Adapters

There's one small problem with `bind1st` and `bind2nd`. In order to work, they need the extra class information that things like `less<>` provide. As a result, something simple like

```
bind2nd(pow, 2)
```

won't work because it won't know what to do with `pow`. A similar problem arises when trying to pass any member function, e.g., `Employee::name()`, to any generic algorithm.

Fortunately, there are two adapters that you can wrap around pointers to regular functions and to member functions when needed.

Use `ptr_fun` when you want to pass a regular function to something like `bind2nd`. E.g., to make a squaring function:

```
bind2nd(ptr_fun(pow), 2)
```

If you have a container full of instances of a class, and want to call a class member function on those instances, use `mem_fun_ref` plus the address (`&`) of the fully-scoped member function. For example, to collect all the employee names from a container of `Employee` records:

```
vector<Employee> emps;
vector<string> names;
... initialize emps ...
transform(emps.begin(), emps.end(), back_inserter(names),
          mem_fun_ref(&Employee::name));
```

Through several layers of C++ magic, `mem_fun_ref()` will create and return something equivalent to the following function:

```
string some_funny_name( Employee & x )
{
   return x.name();
}
```

If your container has pointers to instances, you use `mem_fun`, which creates a function like this:

```
string some_funny_name( Employee * x )
{
   return x->name();
}
```

Note: I didn't invent the names `mem_fun` and `mem_fun_ref`. I just work here.

Comments? Let me know!