The STL <functional> Library

Introduction

In the previous handout, we explored functors, objects that implement the `operator ()` function, and demonstrated how they could be used to increase the expressive power of the STL algorithms. For example, by using `count_if` with a custom functor as the final parameter, we were able to write code that counted the number of elements in a `vector<string>` whose length was less than a certain value. But while this code solved the problem efficiently, we ended up writing so much code that any potential benefits of the STL algorithms were dwarfed by the time spent writing the functor. For reference, here was the code we used:

```
struct ShorterThan
{
    explicit ShorterThan(int maxLength) : length(maxLength) {}
    bool operator() (const string &str) const
    {
        return str.length() < length;
    }
private:
    int length;
};
const int myValue = GetInteger();
count_if(myVector.begin(), myVector.end(), ShorterThan(myValue));
```

Consider the following code which also solves the problem, but by using a simple `for` loop:

```
const int myValue = GetInteger();
int total = 0;
for(int i = 0; i < myVector.size(); ++i)
    if(myVector[i].length() < myValue) ++total;
```

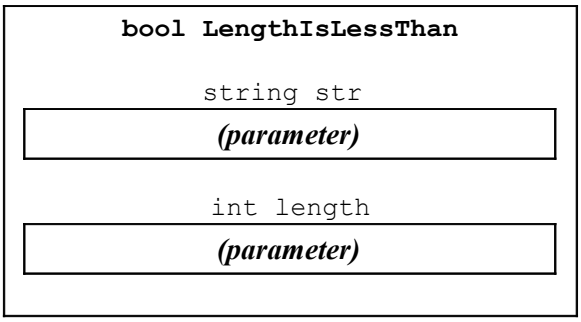
This code is considerably more readable than the functor version and is approximately a third as long. By almost any metric, this code would be considered superior to the earlier version.

If you'll recall, we were motivated to write this `ShorterThan` functor because we were unable to use `count_if` in conjunction with a traditional C++ function. Because `count_if` accepts as a parameter a unary function, we could not write a C++ function that could accept both the current container element and the value to compare its length against. However, we did note that were `count_if` to accept a binary function and extra client data, then we could have written a simple C++ function like this one:

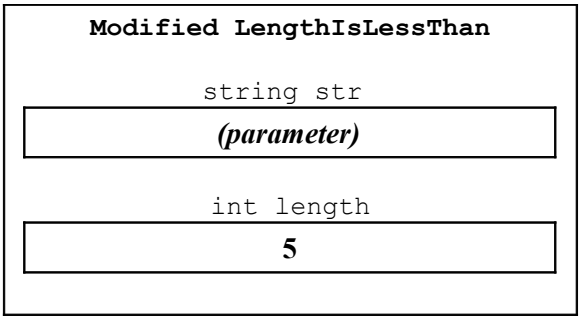
```
bool LengthIsLessThan(const string& str, int threshold)
{
    return str.length() < threshold;
}
```

And then passed it in, along with the cutoff length, to the `count_if` function.

The fundamental problem is that the STL `count_if` algorithm requires a single-parameter function, but the function we want to use requires two pieces of data. We want the STL algorithms to use our two-parameter function `LengthIsLessThan`, but with the second parameter always having the same value. What if somehow we could modify `LengthIsLessThan` by "locking in" the second parameter? In other words, we'd like to take a function that looks like this:



And transform it into another function that looks like this:



Now, if we call this special version of `LengthIsLessThan` with a single parameter (call it `first`), it would be as though we had called the initial version of `LengthIsLessThan`, passing as parameters the value of `first` and the stored value 5. This then returns whether the length of the `first` string is less than 5. Essentially, by binding the second parameter of the two-parameter `LengthIsLessThan` function, we ended up with a one-parameter function that described exactly the predicate function we wanted to provide to `count_if`. Thus, at a high level, the code we want to be able to write should look like this:

```
count_if(v.begin(), v.end(),
  the function formed by binding 5 to the second parameter of LengthIsLessThan);
```

This sort of programming, where functions can be created and modified just like regular objects, is known as *higher-order programming*. While by default C++ does not support higher-order programming, using the STL functional programming libraries, in many cases it is possible to write higher-order code in C++. In the remainder of this handout, we'll explore the STL functional programming libraries and see how to use higher-order programming to supercharge STL algorithms.

Adaptable Functions

To provide higher-order programming support, standard C++ provides the `<functional>` library. `<functional>` exports several useful functions that can transform and modify functions on-the-fly to yield new functions more suitable to the task at hand. However, because of several language limitations,

the <functional> library can only modify specially constructed functions called "adaptable functions," funct*ors* (not regular C++ functions) that export information about their parameter and return types. Fortunately, any one- or two-parameter function can easily be converted into an equivalent adaptable function. For example, suppose you want to make an adaptable function called MyFunction that takes a string by reference-to-const as a parameter and returns a bool, as shown below:

```
struct MyFunction
{
    bool operator() (const string &str) const
    {
        /* Function that manipulates a string */
    }
};
```

Now, to make this function an adaptable function, we need to specify some additional information about the parameter and return types of this functor's operator () function. To assist in this process, the functional library defines a helper template class called unary_function, which is prototyped below:

```
template<typename ParameterType, typename ReturnType>
   class unary_function;
```

The first template argument represents the type of the parameter to the function; the second, the function's return type.

Unlike the other classes you have seen before, the unary_function class contains no data members and no member functions. Instead, it performs some behind-the-scenes magic with the typedef keyword to export the information expressed in the template types to the rest of the functional programming library. Since we want our above functor to also export this information, we'll use a C++ technique called *inheritance* to tell C++ to import all of the information from unary_function into our MyFunction functor. We have not covered inheritance in CS106L yet, but fortunately the syntax used to inherit the information from unary_function is rather straightforward. Because MyFunction accepts as a parameter an object of type string and returns a variable of type bool, we will have MyFunction inherit from the type unary_function<string, bool>. The syntax to accomplish this is shown below:

```
struct MyFunction : public unary_function<string, bool>
{
    bool operator() (const string &str) const
    {
        /* Function that manipulates a string */
    }
};
```

In case you haven't seen inheritance yet, the code "public unary_function<string, bool>" simply tells C++ to import all of the data from the class unary_function<string, bool> into this class. Don't worry if you don't fully understand exactly how this code works – you'll see inheritance later in both CS106X and CS106L.

Note that although the function accepts as its parameter a const string&, we chose to use a unary_function specialized for the type string. The reason is somewhat technical and has to do with how unary_function interacts with other functional library components, so for now just remember that you should not specify reference-to-const types inside the unary_function template parametrization.

The syntax for converting a binary functor into an adaptable binary function works similarly to the above code for unary functions. Suppose that we'd like to make an adaptable binary function that accepts a `string` and an `int` and returns a `bool`. We begin by writing the basic functor code, as shown here:

```
struct MyOtherFunction
{
    bool operator() (const string &str, int val) const
    {
        /* Do something, return a bool. */
    }
};
```

To convert this functor into an adaptable function, we'll have it inherit from `binary_function`. Like `unary_function`, `binary_function` is a template class that's defined as

```
template<typename Param1Type, typename Param2Type, typename ResultType>
    class binary_function;
```

Thus the adaptable version of `MyOtherFunction` would be

```
struct MyOtherFunction: public binary_function<string, int, bool>
{
    bool operator() (const string &str, int val) const
    {
        /* Do something, return a bool. */
    }
};
```

While the above approach for generating adaptable functions is perfectly legal, it's a bit clunky and we still have a high ratio of boilerplate code to actual logic. Fortunately, the STL functional library provides the powerful but cryptically named `ptr_fun`* function that transforms a regular C++ function into an adaptable function. `ptr_fun` can converts both unary and binary C++ functions into adaptable functions with the correct parameter types, meaning that you can skip the hassle of the above code by simply writing normal functions and then using `ptr_fun` to transform them into adaptable functions. For example, given the following C++ function:

```
bool LengthIsLessThan(string myStr, int threshold)
{
    return myStr.length() < threshold;
}
```

If we need to get an adaptable version of that function, we can write `ptr_fun(LengthIsLessThan)` in the spot where the adaptable function is needed. You'll see some other examples of `ptr_fun` later in this handout.

`ptr_fun` is a useful but imperfect tool. You cannot use `ptr_fun` on functions that accept parameters as reference-to-`const`. `ptr_fun` returns a `unary_function` object, and as mentioned above, you cannot specify reference-to-`const` as template arguments to `unary_function`. Also, because of the way that the C++ compiler generates code for functors, code that uses `ptr_fun` can be a bit slower than code using functors.

---

* `ptr_fun` is short for "pointer function", since you're providing as a parameter a function pointer. It should not be confused with "fun with pointers."

For situations where you'd like to convert a member function into an adaptable function, you can use the `mem_fun` or `mem_fun_ref` functions. `mem_fun_ref` converts a nullary (zero-parameter) member function into an adaptable unary functor that accepts as a parameter a receiver object and applies the function to that object. For example, given a `vector<string>`, the following code will print out the lengths of all of the strings in the `vector`:

```
transform(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, "\n"),
          mem_fun_ref(&string::length));*
```

`mem_fun_ref` can also be used to convert unary (one-parameter) member functions into adaptable binary functions that take as a first parameter the object to apply the function to and as a second parameter the parameter to the function. This can be useful with binders, as you'll see in the next section.

The `mem_fun` adapter works like `mem_fun_ref`, except that the adaptable functions it creates accepts the receiving object by pointer instead of by reference. To get the lengths of elements in a `vector<string *>`, you'd use `mem_fun` in the same way we used `mem_fun_ref` in the above example.

Binding Parameters

Now that we've covered how the STL functional library handles adaptable functions, let's consider how we can use them in practice.

At the beginning of this handout, we introduced the notion of *parameter binding*, converting a two-parameter function into a one-parameter function by locking in the value of one of its parameters.† To allow you to bind parameters to functions, the STL functional programming library exports two functions, `bind1st` and `bind2nd`, which accept as parameters an adaptable function and a value to bind and return new functions that are equal to the old functions with the specified values bound in place. For example, given the following implementation of `LengthIsLessThan`:

```
bool LengthIsLessThan(string str, int threshold)
{
    return str.length() < threshold;
}
```

We could use the following syntax to construct a function that's `LengthIsLessThan` with the value five bound to the second parameter:

```
bind2nd(ptr_fun(LengthIsLessThan), 5);
```

The line `bind2nd(ptr_fun(LengthIsLessThan), 5)` first uses `ptr_fun` to generate an adaptable version of the `LengthIsLessThan` function, then uses `bind2nd` to lock the parameter 5 in place. The result is a new unary function that accepts a `string` parameter and returns if that string's length is less than 5, the value we bound to the second parameter. Since `bind2nd` is a function that accepts a function as a parameter and returns a function as a result, `bind2nd` is a function that is sometimes referred to as a *higher-order function*.

---

* Recall that the `transform` function accepts four parameters. The first two parameters define an iterator range, the second the start of a destination range, and the fourth a function to apply to each of the elements.

† The terminology "parameter binding" is preferred inside the C++ community. However, in theoretical computer science, the proper term is *currying*, named after logician Haskell Curry.

Because the result of the above call to `bind2nd` is a unary function that determines if a string has length less than five, we can use the `count_if` algorithm to count the number of values less than five by using the following code:

```
count_if(container.begin(), container.end(),
         bind2nd(ptr_fun(LengthIsLessThan), 5));
```

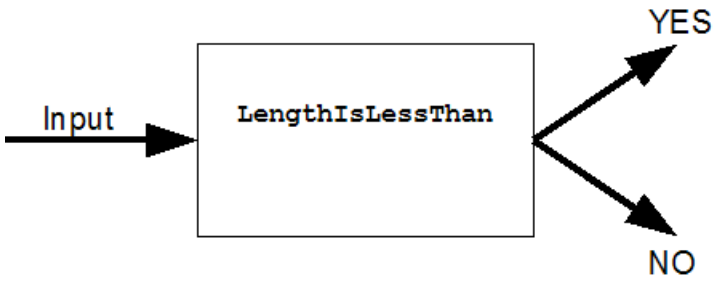
The `bind1st` function acts similarly to `bind2nd`, except that it binds the first parameter of a function. Returning to the above example, given a `vector<int>`, we could count the number of elements in that `vector` smaller than the length of string "C++!" by writing

```
count_if(myVector.begin(), myVector.end(),
         bind1st(ptr_fun(LengthIsLessThan), "C++!"));
```

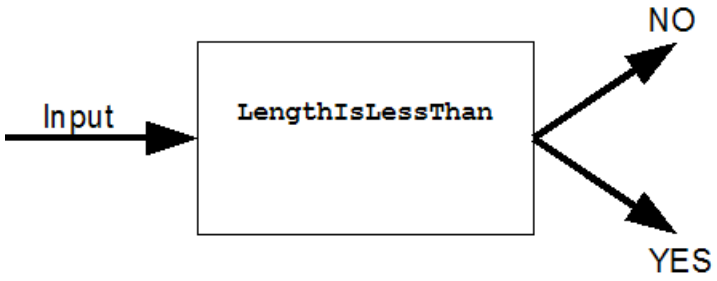
In the STL functional programming library, parameter binding is restricted only to binary functions. Thus you cannot bind a parameter in a three-parameter function to yield a new binary function, nor can you bind the parameter of a unary function to yield a zero-parameter ("nullary") function. For these operations, you'll need to use functors, as shown in the last handout.

Negating Results

Suppose that given a function `LengthIsLessThan`, we want to find the number of strings in a container that are *not* less than a certain length. While we could simply write another function `LengthIsNotLessThan`, it would be much more convenient if we could somehow tell C++ to take whatever value `LengthIsLessThan` returns and to use the opposite result. That is, given a function that looks like this:



We'd like to change it into a function that looks like this:



This operation is called *negation* – constructing a new function whose return value has the opposite value of the input function. There are two STL negator functions – `not1` and `not2` – that return the negated result of a unary or binary predicate function, respectively. Thus, the above function that's a negation of

`LengthIsLessThan` could be written as `not2(ptr_fun(LengthIsLessThan))`. Since `not2` returns an adaptable function, we can then pass the result of this function to `bind2nd` to generate a unary function that returns whether a string's length is at least a certain threshold value. For example, here's code that returns the number of `string`s in a container with length at least 5:

```
count_if(container.begin(), container.end(),
         bind2nd(not2(ptr_fun(LengthIsLessThan)), 5));
```

While this line is incredibly dense, it quite elegantly solves the problem at hand by combining and modifying existing code to create entirely different functions. Such is the beauty and simplicity of higher-order programming – why rewrite code from scratch when you already have all the pieces individually assembled?

Operator Functions

Let's suppose that you have a container of `int`s and you'd like to add 137 to each of them. Recall that you can use the STL `transform` algorithm to apply a function to each element in a container and then store the result. Because we're adding 137 to each element, we might consider writing a function like this one:

```
int Add137(int param)
{
    return param + 137;
}
```

And then writing

```
transform(container.begin(), container.end(), container.begin(), Add137);
```

While this code works correctly, this approach is not particularly robust. What if later on we needed to increment all elements in a container by 42, or perhaps by an arbitrary value? Thus, we might want to consider replacing `Add137` by a function like this one:

```
int AddTwoInts(int one, int two)
{
    return one + two;
}
```

And then using binders to lock the second parameter in place. For example, here's code that's equivalent to what we've written above:

```
transform(container.begin(), container.end(), container.begin(),
    bind2nd(ptr_fun(AddTwoInts), 137));
```

At this point, our code is completely correct, but it can get a bit annoying to have to write a function `AddTwoInts` that simply adds two integers. Moreover, if we then need code to increment all `double`s in a container by 1.37, we would need to write another function `AddTwoDoubles` to avoid problems from typecasts and truncations. Fortunately, because this is a common use case, the STL functional library provides a large number of template adaptable function classes that simply apply the basic C++ operators to two values. For example, in the above code, we can use the adaptable function class `plus<int>` instead of our `AddTwoInts` function, resulting in code that looks like this:

```
transform(container.begin(), container.end(), container.begin(),
          bind2nd(plus<int>(), 137));
```

Note that we need to write `plus<int>()` instead of simply `plus<int>`, since we're using the temporary object syntax to create a temporary `plus<int>` object. Forgetting the parentheses can cause a major compiler error headache that can take a while to track down.

For reference, here's a list of the common "operator functions" exported by `<functional>`:

`plus`	`multiplies`	`divides`	`minus`	`modulus`	`equal_to`
`less`	`less_equal`	`greater`	`greater_equal`	`not_equal_to`	

To see an example that combines the techniques from the previous few sections, let's consider a function that accepts a `vector<double>` and converts each element in the `vector` to its reciprocal (one divided by the value). Because we want to convert each element with value x to the value $1/x$, we can use a combination of binders and operator functions to solve this problem by binding the value 1.0 to the first parameter of the `divides<double>` functor. The result is a unary function that accepts a parameter of type `double` and returns the element's reciprocal. The resulting code looks like this:

```
transform(v.begin(), v.end(), v.begin(), bind1st(divides<double>(), 1.0));
```

This code is concise and elegant, solving the problem in a small space and making explicit what operations are being performed on the data.

Limitations of the Functional Library

While the STL functional library is useful in a wide number of cases, the library is surprisingly limited. For example, the libraries only provide support for adaptable unary and binary functions, but commonly you'll encounter situations where you will need to bind and negate functions with more than two parameters. In these cases, one of your only options is to construct functor classes that accept the extra parameters in their constructors. Similarly, there is no support for function composition, so we could not create a function that computes $2x + 1$ by calling the appropriate combination of the `plus` and `multiplies` functors. However, the next version of C++, nicknamed "C++0x," promises to have more support for functional programming of this sort. Keep your eyes peeled for the next release of C++ – it will be far more functional than the current version!

More to Explore

Although this is a C++ class, if you're interested in functional programming, you might want to consider learning other programming languages like Scheme or Haskell. Functional programming is an entirely different way of thinking about computation, and if you're interested you should definitely consider expanding your horizons with other languages. If you're interested, this Spring the CS department is offering a new class, CS209, that is entirely dedicated to functional programming. The ever-popular class CS242 (Programming Languages) is also a great place to look.

Practice Problems

Each of the problems below can be solved remarkably concisely using functional programming (in fact, the first four have one-line solutions). Try to use the `<functional>` header as much as possible in these exercises to see exactly how tight you can make your code.

1. Write a function `ClearAllStrings` that accepts a `vector<char *>` of C strings and sets each string to be the empty string.
2. The ROT128 cipher is a weak encryption cipher that works by adding 128 to the value of each character in a string to produce a garbled string. Since `char` can only hold 255 different values, two successive applications of ROT128 will produce the original string. Write a function `ApplyROT128` that accepts a `string` and returns the `string`'s ROT128 cipher equivalent.
3. Write a template function `CapAtValue` that accepts a `vector<ElemType>` by reference and an `ElemType` by reference-to-`const` and replaces all elements in the `vector` that compare greater than the parameter with a copy of the parameter. *(Hint: use the `replace_if` algorithm)*
4. The *standard deviation* of a set of data is defined as

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}.$$

Where **N** is the number of elements in the data set, x_i represents the ith number in the data set, and $\overline{x}$ is the average of the elements in the data set. Write a function `StandardDeviation` that accepts a `vector<double>` and returns the standard deviation of the elements in the `vector`.