Chapter 13: Functors

Consider a simple task. Suppose you have a `vector<string>` and you'd like to count the number of `string`s that have length less than five. You stumble upon the STL `count_if` algorithm, which accepts a range of iterators and a predicate function, then returns the number of elements in the range for which the function returns true. For example, you could use `count_if` as follows to count the number of even integers in a `vector`:

```
bool IsEven(int val) {
    return val % 2 == 0;
}

vector<int> myVector = /* ... */
int numEvens = count_if(myVector.begin(), myVector.end(), IsEven);
```

In our case, since we want to count the number of strings with length less than five, we could write a function like this one:

```
bool LengthIsLessThanFive(const string& str) {
    return str.length() < 5;
}
```

And then call `count_if(myVector.begin(), myVector.end(), LengthIsLessThanFive)` to get the number of short strings in the `vector`. Similarly, if we want to count the number of strings with length less than ten, we could write a `LengthIsLessThanTen` function like this one:

```
bool LengthIsLessThanTen(const string& str) {
    return str.length() < 10;
}
```

and then call `count_if(myVector.begin(), myVector.end(), LengthIsLessThanTen)`. In general, if we know in advance what length we want to compare the string lengths against, we can write a function that returns whether a particular string's length is less than that value, then pass it into `count_if` to get our result. This approach is legal C++, but is not particularly elegant. Every time we want to compare the string length against a particular value, we have to write an entirely new function to perform the comparison. Good programming practice suggests that we should instead just write *one* function that looks like this:

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

This more generic function takes in a string and a length, then returns whether the string's length is less than the requested length. This way, we can specify the maximum length as the second parameter rather than writing multiple instances of similar functions.

While this new function is more generic than the previous version, unfortunately we can't use it in conjunction with `count_if`. `count_if` requires a *unary* function (a function taking only one argument) as its final parameter, and the new `LengthIsLessThan` is a *binary* function. Our new `LengthIsLessThan` function, while more generic than the original version, is actually *less useful* in this context. There must be some way to compromise between the two approaches. We need a way to construct a function that takes

in only one parameter (the string to test), but which can be customized to accept an arbitrary maximum length. How can we do this?

This problem boils down to a question of data flow. To construct this hybrid function, we need to somehow communicate the upper bound into the function so that it can perform the comparison. So how can we give this data to the function? Recall that a function has access the following information:

- Its local variables.
- Its parameters.
- Global variables.

Is there some way that we can store the maximum length of the string in one of these locations? We can't store it in a local variable, since local variables don't persist between function calls and aren't accessible to callers. As mentioned above, we also can't store it in a parameter, since `count_if` is hardcoded to accept a unary function. That leaves global variables. We *could* solve this problem using global variables: we would store the maximum length in a global variable, then compare the string parameter length against the global. For example:

```
size_t gMaxLength; // Value to compare against

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

This approach works: if our `vector<string>` is called `v`, then we can count the number of elements less than some value by writing

```
gMaxLength = /* ... some value ... */
int numShort = count_if(v.begin(), v.end(), LengthIsLessThan);
```

But just because this approach works does not mean that it is optimal. This approach is deeply flawed for several reasons, a handful of which are listed here:

- **It is error-prone**. Before we use `LengthIsLessThan`, we must take care to set `gMaxLength` to the maximum desired length. If we forget to do so, then `LengthIsLessThan` will use the wrong value in the comparison and we will get the wrong answer. Moreover, because there is no formal relationship between the `gMaxLength` variable and the `LengthIsLessThan` function, the compiler can't verify that we correctly set `gMaxLength` before calling `LengthIsLessThan`, putting an extra burden on the programmer

- **It is not scalable**. If every time we encounter a problem like this one we create a new global variable, programs we write will begin to fill up with global variables that are used only in the context of a single function. This leads to *namespace pollution*, where too many variables are in scope and it is easy to accidentally use one when another is expected.

- **It uses global variables**. Any use of global variables should send a shiver running down your spine. Global variables should be avoided at all costs, and the fact that we're using them here suggests that something is wrong with this setup.

None of the options we've considered are feasible or attractive. There has to be a better way to solve this, but how?

Functors to the Rescue

The fundamental issue at heart here is that a unary function does not have access to enough information to answer the question we're asking. Essentially, we want a unary function to act like a binary function without taking an extra parameter. Using only the tools we've seen so far, this simply isn't possible. To solve this problem, we'll turn to a more powerful C++ entity: a *functor*. A functor (or *function object*) is an C++ class that acts like a function. Functors can be called using the familiar function call syntax, and can yield values and accept parameters just like regular functions. For example, suppose we create a functor class called `MyClass` imitating a function accepting an `int` and returning a `double`. Then we could "call" an object of type `MyClass` as follows:

```
MyClass myFunctor;
cout << myFunctor(137) << endl; // "Call" myFunctor with parameter 137
```

Although `myFunctor` is an object, in the second line of code we treat it as though it were a function by invoking it with the parameter 137.

At this point, functors might seem utterly baffling: why would you ever want to create an object that behaves like a function? Don't worry, we'll answer that question in a short while. In the meantime, we'll discuss the syntax for functors and give a few motivating examples.

To create a functor, we create an object that overloads the function call operator, `operator ()`. The name of this function is a bit misleading – it is a function called `operator ()`, not a function called `operator` that takes no parameters. Despite the fact that the name looks like "operator parentheses," we're not redefining what it means to parenthesize the object. Instead, we're defining a function that gets called if we invoke the object like a function. Thus in the above code,

```
cout << myFunctor(137) << endl;
```

is equivalent to

```
cout << myFunctor.operator()(137) << endl;
```

Unlike other operators we've seen so far, when overloading the function call operator, you're free to return an object of any type (or even `void`) and can accept any number of parameters. Remember that the point of operator overloading is to allow objects to act like built-in types, and since a regular function can have arbitrarily many parameters and any return type, functors are allowed the same freedom. For example, here's a sample functor that overloads the function call operator to print out a string:

```
class MyFunctor {
public:
    void operator() (const string& str) const {
        cout << str << endl;
    }
};
```

Note that in the function definition there are two sets of parentheses. The first group is for the function name – `operator ()` – and the second for the parameters to `operator ()`. If we separated the implementation of `operator ()` from the class definition, it would look like this:

```
class MyFunctor {
public:
    void operator() (const string& str) const;
};

void MyFunctor::operator() (const string& str) const {
    cout << str << endl;
}
```

Now that we've written `MyFunctor`, we can use it as follows:

```
MyFunctor functor;
functor("Functor power!");
```

This code calls the functor and prints out "Functor power!"

At this point functors might seem like little more than a curiosity. "Sure," you might say, "I can make an object that can be called like a function. But what does it buy me?" A lot more than you might initially suspect, it turns out. The key difference between a function and a functor is that a functor's function call operator is a *member function* whereas a raw C++ function is a *free function*. This means that a functor can access the following information when being called:

- Its local variables.
- Its parameters.
- Global variables.
- **Class data members**.

This last point is extremely important and is the key difference between a regular function and a functor. If a functor's `operator()` member function requires access to data beyond what can be communicated by its parameters, we can store that information as a data member inside the functor class. Since `operator()` is a member of the functor class, it can then access that data freely. For example, consider the following functor class:

```
class StringAppender {
public:
    /* Constructor takes and stores a string. */
    explicit StringAppender(const string& str) : toAppend(str) {}

    /* operator() prints out a string, plus the stored suffix. */
    void operator() (const string& str) const {
        cout << str << ' ' << toAppend << endl;
    }

private:
    const string toAppend;
};
```

This functor's constructor takes in a string and stores it for later use. Its `operator ()` function accepts a string, then prints that string suffixed with the string stored by the constructor. We can then use the `StringAppender` functor like this:

```
StringAppender myFunctor("is awesome");
myFunctor("C++");
```

This code will print out "C++ is awesome," since the constructor stored the string "is awesome" and we passed "C++" as a parameter to the function. If you'll notice, though, in the actual function call we only passed in one piece of information – the string "C++." This is precisely why functors are so useful. Like regular functions, functors are invoked with a fixed number of parameters. Unlike raw functions, however, functors can be constructed to store as much information is necessary to solve the task at hand.

Let's return to the above example with `count_if`. Somehow we need to provide a unary function that can return whether a string is less than an arbitrary length. To solve this problem, instead of writing a unary function, we'll create a unary *functor* whose constructor stores the maximum length and whose `operator ()` accepts a string and returns whether it's of the correct length. Here's one possible implementation:

```
class ShorterThan {
public:
    /* Accept and store an int parameter */
    explicit ShorterThan(size_t maxLength) : length(maxLength) {}

    /* Return whether the string length is less than the stored int. */
    bool operator() (const string& str) const {
        return str.length() < length;
    }

private:
    const size_t length;
};
```

In this code, the constructor accepts a single `size_t`, then stores it as the `length` data member. From that point forward, whenever the functor is invoked on a particular string, the functor's `operator()` function can compare the length of that string against `length` data member. This is exactly what we want – a unary function that knows what value to compare the parameter's length against. To tie everything together, here's the code we'd use to count the number of strings in the `vector` that are shorter than the specified value:

```
ShorterThan st(length);
count_if(myVector.begin(), myVector.end(), st);
```

Functors are incredible when combined with STL algorithms for this very reason – they look and act like regular functions, but have access to extra information. This is just our first taste of functors, as we continue our exploration of C++ you will recognize exactly how much they will influence your program design.

Look back to the above code with `count_if`. If you'll notice, we created a new `ShorterThan` object, then fed it to `count_if`. After the call to `count_if`, odds are we'll never use that particular `ShorterThan` object again. This is an excellent spot to use temporary objects, since we need a new `ShorterThan` for the function call but don't plan on using it afterwards. Thus, we can convert this code:

```
ShorterThan st(length)
count_if(myVector.begin(), myVector.end(), st);
```

Into this code:

```
count_if(myVector.begin(), myVector.end(), ShorterThan(length));
```

Here, `ShorterThan(length)` constructs a temporary `ShorterThan` functor with parameter `length`, then passes it to the `count_if` algorithm. Don't get tripped up by the syntax – `ShorterThan(length)` does *not* call the `ShorterThan`'s `operator ()` function. Instead, it invokes the `ShorterThan` constructor with the parameter `length` to create a temporary object. Even if we had written the `operator()` function to

take in an `int`, C++ would realize that the parentheses here means "construct an object" instead of "invoke `operator()`" from context.

Writing Functor-Compatible Code

In previous chapters, you've seen how to write code that accepts a function pointer as a parameter. For example, the following code accepts a function that takes and returns a `double`, then prints a table of some values of that function:

```
const double kLowerBound = 0.0;
const double kUpperBound = 1.0;
const int    kNumSteps   = 25;
const double kStepSize   = (kUpperBound - kLowerBound) / kNumSteps;

void TabulateFunctionValues(double function(double)) {
    for(double i = kLowerBound; i <= kUpperBound; i += kStepSize)
        cout << "f(" << i << ") = " << function(i) << endl;
}
```

For any function accepting and returning a `double`, we can call `TabulateFunctionValues` with that function as an argument. But what about functors? Can we pass them to `TabulateFunctionValues` as well? As an example, consider the following implementation of a `Reciprocal` functor, whose `operator()` takes in a `double` and returns the reciprocal of that `double`:

```
class Reciprocal {
public:
    double operator() (double val) const {
        return 1.0 / val;
    }
};
```

Given this class implementation, is the following code legal?

```
TabulateFunctionValues(Reciprocal());
```

(Recall that `Reciprocal()` constructs a temporary `Reciprocal` object for use as the parameter to `TabulateFunctionValues`.)

At a high level, this code seems perfectly fine. After all, `Reciprocal` objects can be called as though they were unary functions taking and returning `double`s, so it seems perfectly reasonable to pass a `Reciprocal` into `TabulateFunctionValues`. But despite the similarities, `Reciprocal` is *not* a function – it's a funct*or* – and so the above code will not compile. The problem is that C++'s static type system prevents function pointers from pointing to functors, even if the functor has the same parameter and return type as the function pointer. This is not without reason – the machine code for calling a function is very different from machine code for calling a functor, and if C++ were to conflate the two it would result either in slower function calls or undefined runtime behavior.

Given that this code doesn't compile, how can we fix it? Let's begin with some observations, then generalize to the optimal solution. The above code does not compile because we're trying to provide a `Reciprocal` object to a function expecting a function pointer. This suggests one option – could we rewrite the `TabulateFunctionValues` function such that it accepts a `Reciprocal` as a parameter instead of a function pointer? For example, we could write the following:

```
void TabulateFunctionValues(Reciprocal function) {
    for(double i = kLowerBound; i <= kUpperBound; i += kStepSize)
        cout << "f(" << i << ") = " << function(i) << endl;
}
```

Now, if we call the function as

```
TabulateFunctionValues(Reciprocal());
```

The code is perfectly legal because the argument has type `Reciprocal` and the `TabulateFunctionValues` function is specifically written to take in objects of type `Reciprocal`. But what if we have another functor we want to use in `TabulateFunctionValues`? For example, we might write a functor called `Arccos` that computes the inverse cosine of its parameter, as seen here:

```
class Arccos {
public:
    double operator() (double val) const {
        return acos(val); // Using the acos function from <cmath>
    }
};
```

Unfortunately, if we try to call `TabulateFunctionValues` passing in an `Arccos` object, as shown here:

```
TabulateFunctionValues(Arccos());
```

we'll get yet *another* compile-time error, this time because the `TabulateFunctionValues` function is hardcoded to accept a `Reciprocal`, but we've tried to provide it an object of type `Arccos`. Again, if we rewrite `TabulateFunctionValues` to only accept objects of type `Arccos`, we could alleviate this problem. Of course, in doing so, we would break all code that accepted objects of type `Reciprocal`. How can we resolve this problem? Fortunately, the answer is yes, thanks to a particularly ingenious trick. Below are three versions of `TabulateFunctionValues`, each of which take in a parameter of a different type:

```
void TabulateFunctionValues(double function(double)) {
    for(double i = kLowerBound; i <= kUpperBound; i += kStepSize)
        cout << "f(" << i << ") = " << function(i) << endl;
}
void TabulateFunctionValues(Reciprocal function) {
    for(double i = kLowerBound; i <= kUpperBound; i += kStepSize)
        cout << "f(" << i << ") = " << function(i) << endl;
}
void TabulateFunctionValues(Arccos function) {
    for(double i = kLowerBound; i <= kUpperBound; i += kStepSize)
        cout << "f(" << i << ") = " << function(i) << endl;
}
```

Notice that the only difference between the three implementations of `TabulateFunctionValues` is the type of the parameter to the function. The rest of the code is identical. This suggests a rather elegant solution using templates. Instead of providing multiple different versions of `TabulateFunctionValues`, each specialized for a particular type of function or functors, we'll write a single *template* version of `TabulateFunctionValues` parameterized over the type of the argument. This is shown here:

```
template <typename UnaryFunction>
    void TabulateFunctionValues(UnaryFunction function) {
    for(double i = kLowerBound; i <= kUpperBound; i += kStepSize)
        cout << "f(" << i << ") = " << function(i) << endl;
}
```

Now, we can pass any type of object to `TabulateFunctionValues` that we want, provided that the argument can be called with a single `double` as a parameter to produce a value. This means that we can pass in raw functions, `Reciprocal` objects, `Arccos` objects, and any other functor classes that happen to mimic functions from `double`s to `double`s. This hearkens back to our discussion of concepts in the previous chapter. By writing `TabulateFunctionValues` as a template function parameterized over an arbitrary type, we let clients provide objects of whatever type they see fit, as long as it can be called as a function taking a `double` and returning a `double`.

When writing functions that require a user-specified callback, you may want to consider parameterizing the function over the type of the callback instead of using function pointers. The resulting code will be more flexible and future generations of programmers will be much the better for your extra effort.

STL Algorithms Revisited

Now that you're armed with the full power of C++ functors, let's revisit some of the STL algorithms we've covered and discuss how to maximize their firepower.

The very first algorithm we covered was `accumulate`, defined in the `<numeric>` header. If you'll recall, `accumulate` sums up the elements in a range and returns the result. For example, given a `vector<int>`, the following code returns the sum of all of the `vector`'s elements:

```
accumulate(myVector.begin(), myVector.end(), 0);
```

The first two parameters should be self-explanatory, and the third parameter (zero) represents the initial value of the sum.

However, this view of `accumulate` is limited, and to treat `accumulate` as simply a way to sum container elements would be an error. Rather, *`accumulate` is a general-purpose function for transforming a collection of elements into a single value.*

There is a second version of the `accumulate` algorithm that takes a binary function as a fourth parameter. This version of `accumulate` is implemented like this:

```
template <typename InputIterator, typename Type, typename BinaryFn>
inline Type accumulate(InputIterator start,
                       InputIterator stop,
                       Type accumulator,
                       BinaryFn fn) {
   while(start != stop) {
      accumulator = fn(accumulator, *start);
      ++start;
   }
   return initial;
}
```

This `accumulate` iterates over the elements of a container, calling the binary function on the accumulator and the current element of the container and storing the result back in the accumulator. In other words, `accumulate` continuously updates the value of the accumulator based on its initial value and the values

contained in the input range. Finally, `accumulate` returns the accumulator. Note that the version of `accumulate` we encountered earlier is actually a special case of the above version where the provided callback function computes the sum of its parameters.

To see `accumulate` in action, let's consider an example. Recall that the STL algorithm `lower_bound` returns an iterator to the first element in a range that compares greater than or equal to some value. However, `lower_bound` requires the elements in the iterator range to be in sorted order, so if you have an unsorted `vector`, you cannot use `lower_bound`. Let's write a function `UnsortedLowerBound` that accepts a range of iterators and a lower bound, then returns the *value* of the least element in the range greater than or equal to the lower bound. For simplicity, let's assume we're working with a `vector<int>` so that we don't get bogged down in template syntax, though this approach can easily be generalized.

Although this function can be implemented using loops, we can leverage off of `accumulate` to come up with a considerably more concise solution. Thus, we'll define a functor class to pass to `accumulate`, then write `UnsortedLowerBound` as a wrapper call to `accumulate` with the proper parameters.
Consider the following functor:

```
class LowerBoundHelper {
public:
    explicit LowerBoundHelper(int lower) : lowestValue(lower) {}
    int operator() (int bestSoFar, int current) {
        return current >= lowestValue && current < bestSoFar?
               current : bestSoFar;
    }

private:
    const int lowestValue;
};
```

This functor's constructor accepts the value that we want to lower-bound. Its `operator ()` function accepts two `int`s, the first representing the lowest known value greater than `lowestValue` and the second the current value. If the value of the current element is greater than or equal to the lower bound and also less than the best value so far, `operator ()` returns the value of the current element. Otherwise, it simply returns the best value we've found so far. Thus if we call this functor on every element in the `vector` and keep track of the return value, we should end up with the lowest value in the `vector` greater than or equal to the lower bound. We can now write the `UnsortedLowerBound` function like this:

```
int UnsortedLowerBound(const vector<int>& input, int lowerBound) {
    return accumulate(input.begin(), input.end(),
                      numeric_limits<int>::max(),
                      LowerBoundHelper(lowerBound));
}
```

Our entire function is simply a wrapped call to `accumulate`, passing a specially-constructed `LowerBoundHelper` object as a parameter. Note that we've used the value `numeric_limits<int>::max()` as the initial value for the accumulator. `numeric_limits`, defined in the `<limits>` header, is a traits class that exports useful information about the bounds and behavior of numeric types, and its `max` static member function returns the maximum possible value for an element of the specified type. We use this value as the initial value for the accumulator since any integer is less than it, so if the range contains no elements greater than the lower bound we will get `numeric_limits<int>::max()` back as a sentinel.

If you need to transform a range of values into a single result (of any type you wish), use `accumulate`. To transform a range of values into another range of values, use `transform`. We discussed transform briefly in the chapter on STL algorithms in the context of `ConvertToUpperCase` and `ConvertToLowerCase`, but

such examples are just the tip of the iceberg. `transform` is nothing short of a miracle function, and it arises a whole host of circumstances.*

Higher-Order Programming

This discussion of functors was initially motivated by counting the number of short `string`s inside of an STL `vector`. We demonstrated that 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:

```
class ShorterThan {
public:
    explicit ShorterThan(size_t maxLength) : length(maxLength) {}
    bool operator() (const string& str) const {
        return str.length() < length;
    }

private:
    size_t length;
};

const size_t 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 is 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:

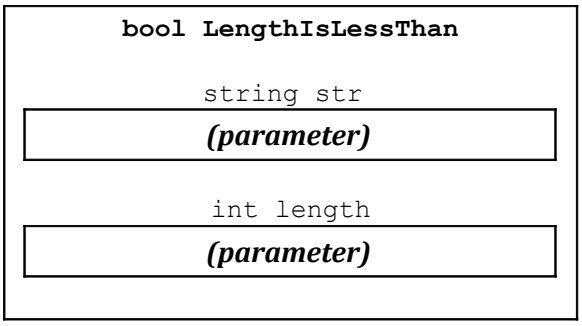
```
bool LengthIsLessThan(const string& str, int threshold) {
    return str.length() < threshold;
}
```

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

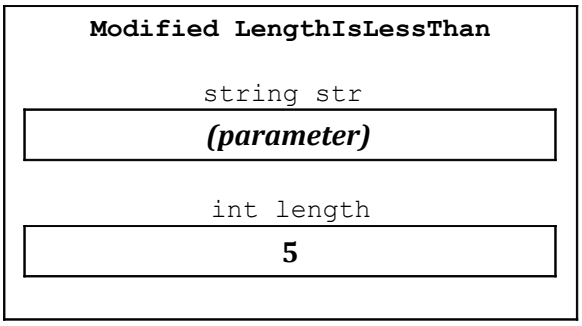
The fundamental problem is that the STL `count_if` algorithm requires a single-parameter function, but the function we want to use requires two pieces of data. We want the STL algorithms to use our two-parameter function `LengthIsLessThan`, but with the second parameter always having the same value. What if

* Those of you familiar with functional programming might recognize `accumulate` and `transform` as the classic higher-order functions Map and Reduce.

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 `str`), it would be as though we had called the initial version of `LengthIsLessThan`, passing as parameters the value of `str` and the stored value 5. This then returns whether the length of the `str` string is less than 5. Essentially, by binding the second parameter of the two-parameter `LengthIsLessThan` function, we end up with a one-parameter function that describes exactly the predicate function we want 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 locking 5 as 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 functors and the STL functional programming libraries, in many cases it is possible to write higher-order code in C++. In the remainder of this chapter, 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:

```
class MyFunction {
public:
    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 inheritance to import all of the information from `unary_function` into our `MyFunction` functor. 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:

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

We'll explore inheritance in more detail later, but for now just think of it as a way for importing information from class into another. 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:

```
class MyOtherFunction {
public:
    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

```
class MyOtherFunction: public binary_function<string, int, bool> {
public:
    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 convert 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.

`ptr_fun` is a useful but imperfect tool. Most notably, 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.

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. These functions convert member functions into unary functions that accept as input a receiver object, then invoke that member function on the receiver. The difference between `mem_fun` and `mem_fun_ref` is how they accept their parameters – `mem_fun` accepts a *pointer* to the receiver object, while `mem_fun_ref` accepts a *reference* to the receiver. 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));
```

Let's dissect this call to `transform`, since there's a lot going on. The first two parameters delineate the input range, in this case the full contents of `myVector`. The third parameter specifies where to put the output, and since here it's an `ostream_iterator` the output will be printed directly to the console instead of stored in some other location. The final parameter is `mem_fun_ref(&string::length)`, a function that accepts as input a `string` and then returns the value of the `length` member function called on that `string`.

`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

* `ptr_fun` is short for "pointer function", not "fun with pointers."

parameter to the function. When we cover binders in the next section, you should get a better feel for exactly how useful this is.

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 chapter, 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*.

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));
```

Compare this code to the functor-based approach illustrated at the start of this chapter. This version of the code is much, *much* shorter than the previous version. If you aren't beginning to appreciate exactly how much power and flexibility the `<functional>` library provides, skip ahead and take a look at the practice problems.

The `bind1st` function acts similarly to `bind2nd`, except that (as its name suggests) 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

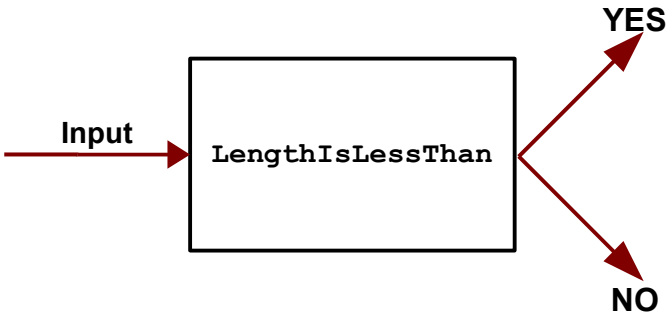
```
count_if(myVector.begin(), myVector.end(),
         bind1st(ptr_fun(LengthIsLessThan), "C++!"));
```

(Admittedly, this isn't the most practical use case for `bind1st`, but it does get the point across).

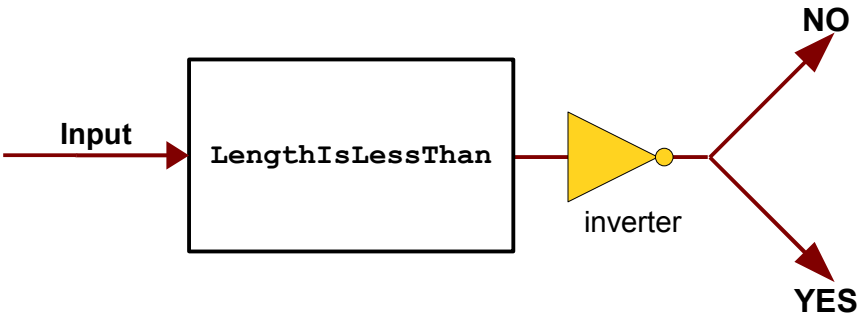
In the STL functional programming library, parameter binding is restricted only to binary functions. Thus you cannot bind a parameter in a three-parameter function to yield a new binary function, nor can you bind the parameter of a unary function to yield a zero-parameter ("nullary") function. For these operations, you'll need to create your own custom functors, as shown in the practice problems at the end of this chapter.

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 *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 dense, it 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 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, the designers of the STL functional library recognized how tedious it is to write out this sort of code, and so 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 construct a `plus<int>` object for `bind2nd`. Forgetting the parentheses can cause a major compiler error headache that can take a while to track down. Also notice that we don't need to use `ptr_fun` here, since `plus<int>` is already an adaptable function.

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

`plus`	`minus`	`multiplies`	`divides`	`modulus`	`negate`
`equal_to`	`not_equal_to`	`greater`	`less`	`greater_equal`	`less_equal`
`logical_and`	`logical_or`	`logical_not`			

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.

Unifying Functions and Functors

There are a huge number of ways to define a function or function-like object in C++, each of which has slightly different syntax and behavior. For example, suppose that we want to write a function that accepts as input a function that can accept an `int` and return a `double`. While of course we could accept a `double (*) (int)` – a pointer to a function accepting an `int` and returning a `double` – this is overly restrictive. For example, all of the following functions can be used as though they were functions taking in an `int` and returning a `double`:

```
double Fn1(const int&);    /* Accept by reference-to-const, yield double. */
int    Fn2(int);           /* Accept parameter as a int, return int. */
int    Fn3(const int&);    /* Accept reference-to-const int, return int. */
```

In addition, if we just accept a `double (*) (int)`, we also can't accept functors as input, meaning that neither of these objects below – both of which can accept an `int` and return a `double` – could be used:

```
/* Functor accepting an int and returning a double. */
class MyFunctor {
public:
    double operator() (int);
};

/* Adaptable function accepting double and returning a double. */
bind2nd(multiplies<int>(), 137);
```

Earlier in this chapter, we saw how we can write functions that accept any of the above functions using templates, as shown here:

```
template <typename UnaryFunction> void DoSomething(UnaryFunction fn) {
    /* ... */
}
```

If we want to write a *function* that accepts a function as input we can rely on templates, but what if we want to write a *class* that needs to store a function of any arbitrary type? To give a concrete example, suppose that we're designing a class representing a graphical window and we want the client to be able to control the window's size and position. The window object, which we'll assume is of type `Window`, thus allows the user to provide a function that will be invoked whenever the window is about to change size. The user's function then takes in an `int` representing the potential new width of the window and returns an `int` representing what the user wants the new window size to be. For example, if we want to create a window that can't be more than 100 pixels wide, we could pass in this function:

```
int ClampTo100Pixels(int size) {
    return min(size, 100);
}
```

Alternatively, if we want the window size to always be 100 pixels, we could pass in this function:

```
int Always100Pixels(int) {
    return 100; // Ignore parameter
}
```

Given that we need to store a function of an arbitrary type inside the `Window` class, how might we design `Window`? Using the approach outlined above, we could parameterize the `Window` class over the type of the function it stores, as shown here:

```
template <typename WidthFunction> class Window {
public:
   Window(WidthFunction fn, /* ... */);

    /* ... */

private:
    WidthFunction width;

    /* ... */
};
```

Given this implementation of `Window`, we could then specify that a window should be no more than 100 pixels wide by writing

```
Window<int (*)(int)> myWindow(ClampTo100Pixels);
```

This Window class lets us use any reasonable function to determine the window size, but has several serious drawbacks. First, it requires the `Window` client to explicitly parameterize Window over the type of callback being stored. When working with function pointers this results in long and convoluted variable declarations (look above for an example), and when working with library functors such as those in the STL `<functional>` library (e.g. `bind2nd (ptr_fun(MyFunction), 137)`)*, we could end up with a `Window` of such a complicated type that it would be infeasible to use with without the aid of `typedef`. But a more serious problem is that this approach causes two `Window`s that don't use the same type of function to compute width to have completely different types. That is, a `Window` using a raw C++ function to compute its size would have a different type from a `Window` that computed its size with a functor. Consequently, we couldn't make a `vector<Window>`, but instead would have to make a `vector<Window<int (*)(int)> >` or a `vector<Window<MyFunctorType> >`. Similarly, if we want to write a function that accepts a `Window`, we can't just write the following:

```
void DoSomething(const Window& w) { // Error - Window is a template, not a type
    /* ... */
}
```

We instead would have to write

```
template <typename WindowType>
    void DoSomething(const WindowType& w) { // Legal but awkward
    /* ... */
}
```

---

* As an FYI, the type of `bind2nd(ptr_fun(MyFunction), 137)` is

```
binder2nd<pointer_to_binary_function<Arg1, Arg2, Ret> >
```

where `Arg1`, `Arg2`, and `Ret` are the argument and return types of the `MyFunction` function.

It should be clear that templatizing the `Window` class over the type of the callback function does not work well. How can we resolve this problem? In the remainder of this chapter, see a beautiful solution to this problem that will unify our treatment of functors, inheritance, templates, operator overloading, copy functions, and conversion constructors. The result is amazingly elegant and hopefully will impress upon you exactly how powerful functors are as a technique.

Inheritance to the Rescue

Let's take a few minutes to think about the problem we're facing. We have a collection of different objects that each have similar functionality (they can be called as functions), but we don't know exactly which object the user will provide. This sounds exactly like the sort of problem we can solve using inheritance and virtual functions. But we have a problem – inheritance only applies to *objects*, but some of the values we might want to store are simple function pointers, which are primitives. Fortunately, we can apply a technique called the *Fundamental Theorem of Software Engineering* (or FTSE) to solve this problem:

> **Theorem *(*The Fundamental Theorem of Software Engineering*)*:** Any problem can be solved by adding enough layers of indirection.

This is a very useful programming concept that will prove relevant time and time again – make sure you remember it!

In this particular application, the FTSE says that we need to distance ourselves by one level from raw function pointers and functor classes. This leads to the following observation: while we might not be able to treat functors and function pointers polymorphically, we certainly can create a new class hierarchy and then treat that class polymorphically. The idea goes something like this – rather than having the user provide us a functor or function pointer which could be of any type, instead we'll define an abstract class exporting the callback function, then will have the user provide a subclass which implements the callback.

One possible base class in this hierarchy is shown below:

```
class IntFunction {
public:
    /* Polymorphic classes need virtual destructors. */
    virtual ~IntFunction() {}

    /* execute() actually calls the proper function and returns the value. */
    virtual int execute(int value) const = 0;
};
```

`IntFunction` exports a single function called `execute` which accepts an `int` and returns an `int`. This function is marked purely virtual since it's unclear exactly what this function should do. After all, we're trying to store an arbitrary function, so there's no clearly-defined default behavior for `execute`.

We can now modify the implementation of `Window` to hold a pointer to an `IntFunction` instead of being templatized over the type of the function, as shown here:

```
class Window {
public:
    Window(IntFunction* sizeFunction, /* ... */);

    /* ... */
private:
    IntFunction* fn;
};
```

Now, if we wanted to clamp the window to 100 pixels, we can do the following:

```
class ClampTo100PixelsFunction: public IntFunction {
public:
    virtual int execute(int size) const {
        return min(size, 100);
    }
};

Window myWindow(new ClampTo100PixelsFunction, /* ... */);
```

Similarly, if we want to have a window that's always 100 pixels wide, we could write

```
class FixedSizeFunction: public IntFunction {
public:
    virtual int execute(int size) const {
        return 100;
    }
};

Window myWindow(new FixedSizeFunction, /* ... */);
```

It seems as though we've solved the problem – we now have a `Window` class that allows us to fully customize its resizing behavior – what more could we possibly want?

The main problem with our solution is the sheer amount of boilerplate code clients of `Window` have to write if they want to change the window's resizing behavior. Our initial goal was to let class clients pass raw C++ functions and functors to the `Window` class, but now clients have to subclass `IntFunction` to get the job done. Both of the above subclasses are lengthy even though they only export a single function. Is there a simpler way to do this?

The answer, of course, is yes. Suppose we have a pure C++ function that accepts an `int` by value and returns an `int` that we want to use for our resizing function in the `Window` class; perhaps it's the `ClampTo100Pixels` function we defined earlier, or perhaps it's `Always100Pixels`. Rather than defining a new subclass of `IntFunction` for every single one of these functions, instead we'll build a single class that's designed to wrap up a function pointer in a package that's compatible with the `IntFunction` interface. That is, we can define a subclass of `IntFunction` whose constructor accepts a function pointer and whose `execute` calls this function. This is the Fundamental Theorem of Software Engineering in action – we couldn't directly treat the raw C++ function polymorphically, but by abstracting by a level we can directly apply inheritance.

Here's one possible implementation of the subclass:

```
class ActualFunction: public IntFunction {
public:
    explicit ActualFunction(int (*fn)(int)) : function(fn) {}

    virtual int execute(int value) const {
        return function(value);
    }

private:
    int (*function)(int);
};
```

Now, if we want to use `ClampTo100Pixels` inside of `Window`, we can write:

```
Window myWindow(new ActualFunction(ClampTo100Pixels), /* ... */);
```

There is a bit of extra code for creating the `ActualFunction` object, but this is a one-time cost. We can now use `ActualFunction` to wrap any raw C++ function accepting an `int` and returning an `int` and will save a lot of time typing out new subclasses of `IntFunction` for every callback.

Now, suppose that we have a functor class, which we'll call `MyFunctor`, that we want to use inside the `Window` class. Then we could define a subclass that looks like this:

```
class MyFunctorFunction: public IntFunction {
public:
    explicit MyFunctorFunction(MyFunctor fn) : function(fn) {}

    virtual int execute(int value) const {
        return function(value);
    }

private:
    MyFunctor function;
};
```

And could then use it like this:

```
Window myWindow(new MyFunctorFunction(MyFunctor(137)), /* ... */);
```

Where we assume for simplicity that the `MyFunctor` class has a unary constructor.

We're getting much closer to an ideal solution. Hang in there; the next step is pretty exciting.

Templates to the Rescue

At this point we again could just call it quits – we've solved the problem we set out to solve and using the above pattern our `Window` class can use any C++ function or functor we want. However, we are close to an observation that will greatly simplify the implementation of `Window` and will yield a much more general solution.

Let's reprint the two subclasses of `IntFunction` we just defined above which wrap function pointers and functors:

```
class ActualFunction: public IntFunction {
public:
    explicit ActualFunction(int (*fn)(int)) : function(fn) {}

    virtual int execute(int value) const {
        return function(value);
    }

private:
    int (*const function)(int);
};

class MyFunctorFunction: public IntFunction {
public:
    explicit MyFunctorFunction(MyFunctor fn) : function(fn) {}

    virtual int execute(int value) const {
        return function(value);
    }

private:
    MyFunctor function;
};
```

If you'll notice, besides the name of the classes, the only difference between these two classes is what type of object is being stored. This similarity is no coincidence – any callable function or functor would require a subclass with exactly this structure. Rather than requiring the client of `Window` to reimplement this subclass from scratch each time, we can instead create a *template class* that's a subclass of `IntFunction`. It's rare in practice to see templates and inheritance mixed this way, but here it works out beautifully. Here's one implementation:

```
template <typename UnaryFunction> class SpecificFunction: public IntFunction {
public:
    explicit SpecificFunction(UnaryFunction fn) : function(fn) {}

    virtual int execute(int value) const {
        return function(value);
    }

private:
    UnaryFunction function;
};
```

We now can use the `Window` class as follows:

```
Window myWindow(new SpecificFunction<int(*)(int)>(ClampTo100Pixels), /*...*/);
Window myWindow(new SpecificFunction<MyFunctor>(MyFunctor(137)), /*...*/);
```

The syntax here might be a bit tricky, but we've greatly reduced the complexity associated with the `Window` class since clients no longer have to implement their own subclasses of `IntFunction`.

One More Abstraction

This design process has consisted primarily of adding more and more abstractions on top of the system we're designing, and it's time for one final leap. Let's think about what we've constructed so far. We've built a class hierarchy with a single base class and a template for creating as many subclasses as we need. However, everything we've written has been hardcoded with the assumption that the `Window` class is the only class that might want this sort of functionality. Having the ability to store and call a function of any conceivable type is enormously useful, and if we can somehow encapsulate all of the necessary machinery to get this working into a single class, we will be able to reuse what we've just built time and time again. In this next section, that's exactly what we'll begin doing.

We'll begin by moving the code from `Window` that manages the stored function into a dedicated class called `Function`. The basic interface for `Function` is shown below:

```
class Function {
public:
    /* Constructor and destructor.  We'll implement copying in a bit. */
    Function(IntFunction* fn);
    ~Function();

    /* Function is a functor that calls into the stored resource. */
    int operator() (int value) const;

private:
    IntFunction* function;
};
```

We'll leave the `Function` constructor left as an implicit conversion constructor, since that way we can implicitly convert between a callable `IntFunction` pointer and a `Function` functor. We can then implement the above functions as follows:

```
/* Constructor accepts an IntFunction and stores it. */
Function::Function(IntFunction* fn) : function(fn) {
    // Handled in initializer list
}

/* Destructor deallocates the stored function. */
Function::~Function() {
    delete function;
}

/* Function call just calls through to the pointer and returns the result. */
int Function::operator() (int value) const {
    return function->execute(value);
}
```

Nothing here should be too out-of-the-ordinary – after all, this is pretty much the same code that we had inside the `Window` class.

Given this version of `Function`, we can write code that looks like this:

```
Function myFunction = new SpecificFunction<int (*)(int)>(ClampTo100Pixels);
cout << myFunction(137) << endl; // Prints 100
cout << myFunction(42) << endl; // Prints 42
```

If you're a bit worried that the syntax `new SpecificFunction<int (*)(int)>(ClampTo100Pixels)` is unnecessarily bulky, that's absolutely correct. Don't worry, in a bit we'll see how to eliminate it. In the meantime, however, let's implement the copy behavior for the `Function` class. After all, there's no reason that we shouldn't be able to copy `Function` objects, and defining copy behavior like this will lead to some very impressive results in a bit.

We'll begin by defining the proper functions inside the Function class, as seen here:

```
class Function {
public:
    /* Constructor and destructor. */
    Function(IntFunction* fn);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    int operator() (int value) const;

private:
    IntFunction* function;

    void clear();
    void copyOther(const Function& other);
};
```

Now, since the `Function` class contains only a single data member (the `IntFunction` pointer), to make a deep-copy of a `Function` we simply need to make a deep copy of its requisite `IntFunction`. But here we run into a problem. `IntFunction` is an abstract class and we can't tell at compile-time what type of object is actually being pointed at by the function pointer. How, then, can we make a deep-copy of the `IntFunction`? The answer is surprisingly straightforward – we'll just introduce a new virtual function to the `IntFunction` class that returns a deep copy of the receiver object. Since this function duplicates an existing object, we'll call it `clone`. The interface for `IntFunction` now looks like this:

```
class IntFunction {
public:
    /* Polymorphic classes should have virtual destructors. */
    virtual ~IntFunction() { }

    /* execute() actually calls the proper function and returns the value. */
    virtual int execute(int value) const = 0;

    /* clone() returns a deep-copy of the receiver object. */
    virtual IntFunction* clone() const = 0;
};
```

We can then update the template class `SpecificFunction` to implement `clone` as follows:

```
template <typename UnaryFunction> class SpecificFunction: public IntFunction {
public:
    explicit SpecificFunction(UnaryFunction fn) : function(fn) {}

    virtual int execute(int value) const {
        return function(value);
    }

    virtual IntFunction* clone() const {
        return new SpecificFunction(*this);
    }

private:
    UnaryFunction function;
};
```

Here, the implementation of `clone` returns a new `SpecificFunction` initialized via the copy constructor as a copy of the receiver object. Note that we haven't explicitly defined a copy constructor for `SpecificFunction` and are relying here on C++'s automatically-generated copy function to do the trick for us. This assumes, of course, that the `UnaryFunction` type correctly supports deep-copying, but this isn't a problem since raw function pointers can trivially be deep-copied as can all primitive types and it's rare to find functor classes with no copy support.

We can then implement the copy constructor, assignment operator, destructor, and helper functions for Function as follows:

```
Function::~Function() {
    clear();
}

Function::Function(const Function& other) {
    copyOther(other);
}

Function& Function::operator= (const Function& other) {
    if(this != &other) {
        clear();
        copyOther(other);
    }
    return *this;
}

void Function::clear() {
    delete function;
}

void Function::copyOther(const Function& other) {
    /* Have the stored function tell us how to copy itself. */
    function = other.function->clone();
}
```

Our `Function` class is now starting to take shape!

Hiding `SpecificFunction`

Right now our `Function` class has full deep-copy support and using `SpecificFunction<T>` can store any type of callable function. However, clients of `Function` have to explicitly wrap any function they want to

store inside `Function` in the `SpecificFunction` class. This has several problems. First and foremost, this breaks encapsulation. `SpecificFunction` is only used internally to the `Function` class, never externally, so requiring clients of `Function` to have explicit knowledge of its existence violates encapsulation. Second, it requires the user to know the type of every function they want to store inside the Function class. In the case of `ClampTo100Pixels` this is rather simple, but suppose we want to store `bind2nd(multiplies<int>(), 137)` inside of `Function`. What is the type of the object returned by `bind2nd(multiplies<int>(), 137)`? For reference, it's `binder2nd<multiplies<int> >`, so if we wanted to store this in a Function we'd have to write

```
Function myFunction =
    new SpecificFunction<binder2nd<multiplies<int> > >(bind2nd(multiplies<int>(),137));
```

This is a syntactic nightmare and makes the `Function` class terribly unattractive.

Fortunately, however, this problem has a quick fix – we can rewrite the `Function` constructor as a template function parameterized over the type of argument passed into it, then construct the relevant `SpecificFunction` for the `Function` client. Since C++ automatically infers the parameter types of template functions, this means that clients of Function never need to know the type of what they're storing – the compiler will do the work for them. Excellent!

If we do end up making the Function constructor a template, we should also move the `IntFunction` and `SpecificFunction` classes so that they're inner classes of `Function`. After all, they're specific to the implementation of Function and the outside world has no business using them.

The updated interface for the `Function` class is shown here:

```
class Function {
public:
    /* Constructor and destructor. */
    template <typename UnaryFunction> Function(UnaryFunction fn);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    int operator() (int value) const;

private:
    class IntFunction { /* ... */ };
    template <typename UnaryFunction> class SpecificFunction { /* ... */ };

    IntFunction* function;

    void clear();
    void copyOther(const Function& other);
};
```

We can then implement the constructor as follows:

```
template <typename UnaryFunction> Function::Function(UnaryFunction fn) {
    function = new SpecificFunction<UnaryFunction>(fn);
}
```

Since we've left the `Function` constructor not marked explicit, this template constructor is a conversion constructor. Coupled with the assignment operator, this means that we can use `Function` as follows:

```
Function fn = ClampTo100Pixels;
cout << fn(137) << endl; // Prints 100
cout << fn(42) << endl; // Prints 42

fn = bind2nd(multiplies<int>(), 2);
cout << fn(137) << endl; // Prints 274
cout << fn(42) << endl; // Prints 84
```

This is exactly what we're looking for – a class that can store any callable function that takes in an `int` and returns an `int`. If this doesn't strike you as a particularly elegant piece of code, take some time to look over it again.

There's one final step we should take, and that's to relax the restriction that `Function` always acts as a function from `int`s to `int`s. There's nothing special about int, and by giving `Function` clients the ability to specify their own parameter and return types we'll increase the scope of what `Function` is capable of handling. We'll thus templatize `Function` as `Function<ArgType, ReturnType>`. We also need to make some minor edits to `IntFunction` (which we'll rename to `ArbitraryFunction` since `IntFunction` is no longer applicable), but in the interest of brevity we won't reprint them here.

The final interface for `Function` thus looks like this:

```
template <typename ArgType, typename ReturnType> class Function {
public:
    /* Constructor and destructor. */
    template <typename UnaryFunction> Function(UnaryFunction);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    ReturnType operator() (ArgType value) const;

private:
    class ArbitraryFunction { /* ... */ };
    template <typename UnaryFunction> class SpecificFunction { /* ... */ }

    ArbitraryFunction* function;

    void clear();
    void copyOther(const Function& other);
};
```

To conclude our discussion of `Window`, using the new `Function` type we could rewrite the `Window` class using `Function` as follows:

```
class Window {
public:
    Window(const Function<int, int>& widthFn, /* ... */
    /* ... other member functions ... */

private:
    Function<int, int> widthFunction;
};
```

Now, clients can pass any unary function (or functor) that maps from `int`s to `int`s as a parameter to `Window` and the code will compile correctly.

External Polymorphism

The `Function` type we've just developed is subtle in its cleverness. Because we can convert any callable unary function into a `Function`, when writing code that needs to work with some sort of unary function, we can have that code use `Function` instead of any specific function type. This technique of abstracting away from the particular types that provide a behavior into an object representing that behavior is sometimes known as *external polymorphism*. As opposed to *internal polymorphism*, where we explicitly define a set of classes containing virtual functions, external polymorphism "grafts" a set of virtual functions onto any type that supports the requisite behavior.

Virtual functions can be slightly more expensive than regular functions because of the virtual function table lookup required. External polymorphism is implemented using inheritance and thus also incurs an overhead, but the overhead is slightly greater than regular inheritance. Think for a minute how the `Function` class we just implemented is designed. Calling `Function::operator()` requires the following:

1. Following the `ArbitraryFunction` pointer in the `Function` class to its virtual function table.

2. Calling the function indicated by the virtual function table, which corresponds to the particular `SpecificFunction` being pointed at.

3. Calling the actual function object stored inside the `SpecificFunction`.

This is slightly more complex than a regular virtual function call, and illustrates the cost associated with external polymorphism. That said, in some cases (such as the `Function` case outlined here) the cost is overwhelming offset by the flexibility afforded by external polymorphism.

Implementing the `<functional>` Library

Now what we've seen how the `<functional>` library works from a client perspective, let's discuss how the library is put together. What's so special about adaptable functions? How does `ptr_fun` convert a regular function into an adaptable one? How do functions like `bind2nd` and `not1` work? This discussion will be highly technical and will push the limits of your knowledge of templates, but by the time you're done you should have an excellent grasp of how template libraries are put together. Moreover, the techniques used here are applicable beyond just the `<functional>` library and will almost certainly come in handy later in your programming career.

Let's begin by looking at exactly what an adaptable function is. Recall that adaptable functions are functors that inherit from either `unary_function` or `binary_function`. Neither of these template classes are particularly complicated; here's the complete definition of `unary_function`:*

```
template <typename ArgType, typename RetType> class unary_function {
public:
    typedef ArgType argument_type;
    typedef RetType result_type;
};
```

This class contains no data members and no member functions. Instead, it exports two `typedef`s – one renaming `ArgType` to `argument_type` and one renaming `RetType` to `result_type`. When you create an adaptable function that inherits from `unary_function`, your class acquires these `typedef`s. For example, if we write the following adaptable function:

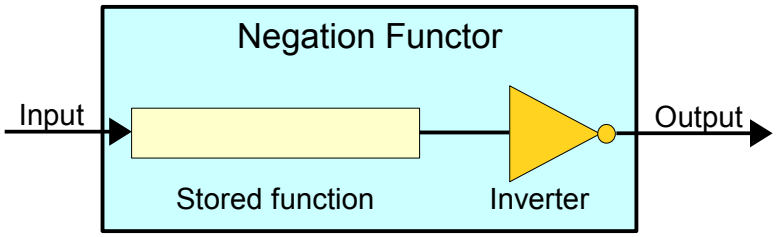
```
class IsPositive: public unary_function<double, bool> {
public:
    bool operator() (double value) const {
        return value > 0.0;
    }
};
```

The statement `public unary_function<double, double>` imports two `typedef`s into `IsPositive`: `argument_type` and `return_type`, equal to `double` and `bool`, respectively. Right now it might not be apparent how these types are useful, but as we begin implementing the other pieces of the `<functional>` library it will become more apparent.

Implementing `not1`

To begin our behind-the-scenes tour of the `<functional>` library, let's see how to implement the `not1` function. Recall that `not1` accepts as a parameter a unary adaptable predicate function, then returns a new adaptable function that yields opposite values as the original function. For example, `not1(IsPositive())` would return a function that returns whether a value is *not* positive.

Implementing `not1` requires two steps. First, we'll create a template functor class parameterized over the type of the adaptable function to negate. This functor's constructor will take as a parameter an adaptable function of the proper type and store it for later use. We'll then implement its `operator()` function such that it calls the stored function and returns the negation of the result. Graphically, this is shown here:



Once we have designed this functor, we'll have `not1` accept an adaptable function, wrap it in our negating functor, then return the resulting object to the caller. This means that the return value of `not1` is an adaptable unary predicate function that returns the opposite value of its parameter, which is exactly what we want.

---

* Technically speaking `unary_function` and `binary_function` are `struct`s, but this is irrelevant here.

Let's begin by writing the template functor class, which we'll call `unary_negate` (this is the name of the functor class generated by the `<functional>` library's `not1` function). We know that this functor should be parameterized over the type of the adaptable function it negates, so we can begin by writing the following:

```
template <typename UnaryPredicate> class unary_negate {
public:
    explicit unary_negate(const UnaryPredicate& pred) : p(pred) {}

    /* ... */
private:
    UnaryPredicate p;
};
```

Here, the constructor accepts an object of type `UnaryPredicate`, then stores it in the data member `p`.

Now, let's implement the `operator()` function, which, as you'll recall, should take in a parameter, feed it into the stored function `p`, then return the inverse result. The code for this function looks like this:

```
template <typename UnaryPredicate> class unary_negate {
public:
    explicit unary_negate(const UnaryPredicate& pred) : p(pred) {}

    bool operator() (const /* what goes here? */& param) const {
        return !p(param); // Call function and return the opposite result.
    }
private:
    UnaryPredicate p;
};
```

We've almost finished writing our `unary_negate` class, but we have a slight problem – what is the type of the parameter to `operator()`? This is where adaptable functions come in. Because `UnaryPredicate` is adaptable, it must export a type called `argument_type` corresponding to the type of its argument. We can thus define our `operator()` function to accept a parameter of type `typename UnaryPredicate::argume nt_type` to guarantee that it has the same parameter type as the `UnaryPredicate` class.* The updated code for `unary_negate` looks like this:

```
template <typename UnaryPredicate> class unary_negate {
public:
    explicit unary_negate(const UnaryPredicate& pred) : p(pred) {}

    bool
    operator() (const typename UnaryPredicate::argument_type& param) const {
        return !p(param); // Call stored function and return opposite result.
    }
private:
    UnaryPredicate p;
};
```

That's quite a mouthful, but it's exactly the solution we're looking for. If it weren't for the fact that `UnaryPredicate` is an adaptable function, we would not have been able to determine the parameter type for the `operator()` member function, and code like this would not have been possible.

---

* Remember that the type is `typename UnaryPredicate::argument_type`, not `UnaryPredicate::argument_type`. `argument_type` is nested inside `UnaryPredicate`, and since `UnaryPredicate` is a template argument we have to explicitly use `typename` to indicate that `argument_type` is a type.

There's one step left to finalize this functor class, and that's to make the functor into an adaptable function by having it inherit from the proper `unary_function`. Since the functor's argument type is `typename UnaryPredicate::argument_type` and its return type is `bool`, we'll inherit from `unary_function <typename UnaryPredicate::argument_type, bool>`. The final code for `unary_negate` is shown here:

```
template <typename UnaryPredicate>
class unary_negate:
    public unary_function<typename UnaryPredicate::argument_type, bool>
{
public:
    explicit unary_negate(const UnaryPredicate& pred) : p(pred) {}

    bool
    operator() (const typename UnaryPredicate::argument_type& param) const {
        return !p(param); // Call stored function and return opposite result.
    }
private:
    UnaryPredicate p;
};
```

We've now finished writing our functor class to perform the negation, and all that's left to do is write `not1`. `not1` is much simpler than `unary_negate`, since it simply has to take in a parameter and wrap it in a `unary_negate` functor. This is shown here:

```
template <typename UnaryPredicate>
    unary_negate<UnaryPredicate> not1(const UnaryPredicate& pred) {
    return unary_negate<UnaryPredicate>(pred);
}
```

That's all there is to it – we've successfully implemented `not1`!

You might be wondering why there are two steps involved in writing `not1`. After all, once we have the functor that performs negation, why do we need to write an additional function to create it? The answer is *simplicity*. We don't need `not1`, but having it available reduces complexity. For example, using the `IsPositive` adaptable function from above, let's suppose that we want to write code to find the first nonpositive element in a `vector`. Using the `find_if` algorithm and `not1`, we'd write this as follows:

```
vector<double>::iterator itr =
    find_if(v.begin(), v.end(), not1(IsPositive()));
```

If instead of using `not1` we were to explicitly create a `unary_negate` object, the code would look like this:

```
vector<double>::iterator itr =
    find_if(v.begin(), v.end(), unary_negate<IsPositive>(IsPositive()));
```

That's quite a mouthful. When calling the template function `not1`, the compiler automatically infers the type of the argument and constructs an appropriately parameterized `unary_negate` object. If we directly use `unary_negate`, C++ will not perform type inference and we'll have to spell out the template arguments ourselves. The pattern illustrated here – having a template class and a template function to create it – is common in library code because it lets library clients use complex classes without ever having to know how they're implemented behind-the-scenes.

Implementing `ptr_fun`

Now that we've seen how `not1` works, let's see if we can construct the `ptr_fun` function. At a high level `ptr_fun` and `not1` work the same way – they each accept a parameter, construct a special functor class based on the parameter, then return it to the caller. The difference between `not1` and `ptr_fun`, however, is that there are two different versions of `ptr_fun` – one for unary functions and one for binary functions. The two versions work almost identically and we'll see how to implement them both, but for simplicity we'll begin with the unary case.

To convert a raw C++ unary function into an adaptable unary function, we need to wrap it in a functor that inherits from the proper `unary_function` base class. We'll make this functor's `operator()` function simply call the stored function and return its value. To be consistent with the naming convention of the `<functional>` library, we'll call the functor `pointer_to_unary_function` and will parameterize it over the argument and return types of the function. This is shown here:

```
template <typename ArgType, typename RetType>
class pointer_to_unary_function: public unary_function<ArgType, RetType>
{
public:
    explicit pointer_to_unary_function(ArgType fn(RetType)) : function(fn) {}

    RetType operator() (const ArgType& param) const {
        return function(param);
    }
private:
    ArgType (*function)(RetType);
};
```

There isn't that much code here, but it's fairly dense. Notice that we inherit from `unary_function<ArgType, RetType>` so that the resulting functor is adaptable. Also note that the argument and return types of `operator()` are considerably easier to determine than in the `unary_negate` case because they're specified as template arguments.

Now, how can we implement `ptr_fun` to return a correctly-constructed `pointer_to_unary_function`? Simple – we just write a template function parameterized over argument and return types, accept a function pointer of the appropriate type, then wrap it in a `pointer_to_unary_function` object. This is shown here:

```
template <typename ArgType, typename RetType>
    pointer_to_unary_function<ArgType, RetType>
    ptr_fun(RetType function(ArgType)) {
        return pointer_to_unary_function<ArgType, RetType>(function);
    }
```

This code is fairly dense, but gets the job done.

The implementation of `ptr_fun` for binary functions is similar to the implementation for unary functions. We'll create a template functor called `pointer_to_binary_function` parameterized over its argument and return types, then provide an implementation of `ptr_fun` that constructs and returns an object of this type. This is shown here:

```
template <typename Arg1, typename Arg2, typename Ret>
class pointer_to_binary_function: public binary_function<Arg1, Arg2, Ret> {
public:
    explicit pointer_to_binary_function(Ret fn(Arg1, Arg2)) : function(fn) {}

    Ret operator() (const Arg1& arg1, const Arg2& arg2) const {
        return function(arg1, arg2);
    }
private:
    Ret (*function)(Arg1, Arg2);
};

template <typename Arg1, typename Arg2, typename Ret>
pointer_to_binary_function<Arg1, Arg2, Ret> ptr_fun(Ret function(Arg1, Arg2)) {
    return pointer_to_binary_function<Arg1, Arg2, Ret>(function);
}
```

Note that we now have *two* versions of `ptr_fun` – one that takes in a unary function and one that takes in a binary function. Fortunately, C++ overloading rules allow for the two functions to coexist, since they have different signatures.

Implementing `bind1st`

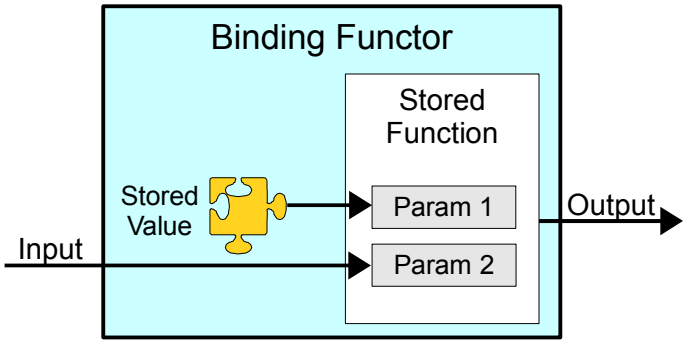
To wrap up our tour of the `<functional>` library, let's see how to implement `bind1st`. If you'll recall, `bind1st` takes in a binary adaptable function and a value, then returns a new unary function equal to the input function with the first parameter locked in place. We'll follow the pattern of `not1` and `ptr_fun` by writing a template functor class called `binder1st` that actually does the binding, then having `bind1st` construct and return an object of the proper type.

Before proceeding with our implementation of `binder1st`, we need to take a quick detour into the inner workings of the `binary_function` class. Like `unary_function`, `binary_function` exports `typedef`s so that other parts of the `<functional>` library can recover the argument and return types of adaptable functions. However, since a binary function has two arguments, the names of the exported types are slightly different. `binary_function` provides the following three `typedef`s:

- **`first_argument_type`**, the type of the first argument,
- **`second_argument_type`**, the type of the second argument, and
- **`result_type`**, the function's return type.

We will need to reference each of these type names when writing `bind1st`.

Now, how do we implement the `binder1st` functor? Here is one possible implementation. The `binder1st` constructor will accept and store an adaptable binary function and the value for its first argument. `binder1st` then provides an implementation of `operator()` that takes a single parameter, then invokes the stored function passing in the function parameter and the saved value. This is shown here:



Let's begin implementing `binder1st`. The functor has to be a template, since we'll be storing an arbitrary adaptable function and value. However, we only need to parameterize the functor over the type of the binary adaptable function, since we can determine the type of the first argument from the adaptable function's `first_argument_type`. We'll thus begin with the following implementation:

```
template <typename BinaryFunction> class binder1st {
    /* ... */
};
```

Now, let's implement the constructor. It should take in two parameters – one representing the binary function and the other the value to lock into place. The first will have type `BinaryFunction`; the second, `typename BinaryFunction::first_argument_type`. This is shown here:

```
template <typename BinaryFunction> class binder1st {
public:
    binder1st(const BinaryFunction& fn,
              const typename BinaryFunction::first_argument_type& arg) :
                  function(fn), first(arg) {}

    /* ... */

private:
    BinaryFunction function;
    typename BinaryFunction::first_argument_type first;
};
```

Phew! That's quite a mouthful, but is the reality of much library template code. Look at the declaration of the `first` data member. Though it may seem strange, this is the correct way to declare a data member whose type is a type nested inside a template argument.

We now have the constructor written and all that's left to take care of is `operator()`. Conceptually, this function isn't very difficult, and if we ignore the parameter and return types have the following implementation:

```
template <typename BinaryFunction> class binder1st {
public:
    binder1st(const BinaryFunction& fn,
              const typename BinaryFunction::first_argument_type& arg) :
                  function(fn), first(arg) {}

    /* ret */ operator() (const /* arg */& param) const {
        return function(first, param);
    }

private:
    BinaryFunction function;
    typename BinaryFunction::first_argument_type first;
};
```

What are the argument and return types for this function? Well, the function returns whatever object is produced by the stored function, which has type `typename BinaryFunction::result_type`. The function accepts a value for use as the second parameter to the stored function, so it must have type `typename BinaryFunction::second_argument_type`. This results in the following code:

```
template <typename BinaryFunction> class binder1st {
public:
    binder1st(const BinaryFunction& fn,
              const typename BinaryFunction::first_argument_type& arg) :
                  function(fn), first(arg) {}

    typename BinaryFunction::result_type
    operator()  (const typename BinaryFunction::second_argument_type& param) const {
        return function(first, param);
    }
private:
    BinaryFunction function;
    typename BinaryFunction::first_argument_type first;
};
```

We're almost finished, and all that's left for `binder1st` is to make it adaptable. Using the logic from above, we'll have it inherit from the proper instantiation of `unary_function`, as shown here:

```
template <typename BinaryFunction> class binder1st :
   public unary_function<typename BinaryFunction::second_argument_type,
                         typename BinaryFunction::result_type> {
public:
    binder1st(const BinaryFunction& fn,
              const typename BinaryFunction::first_argument_type& arg) :
                  function(fn), first(arg) {}

    typename BinaryFunction::result_type
    operator()  (const typename BinaryFunction::second_argument_type& param) const {
        return function(first, param);
    }
private:
    BinaryFunction function;
    typename BinaryFunction::first_argument_type first;
};
```

That's it for the `binder1st` class. As you can see, the code is dense and does a lot of magic with `typename` and nested types. Without adaptable functions, code like this would not be possible.

To finish up our discussion, let's implement `bind1st`. This function isn't particularly tricky, though we do need to do a bit of work to extract the type of the value to lock in place:

```
template <typename BinaryFunction>
binder1st<BinaryFunction>
   bind1st(const BinaryFunction& fn,
           const typename BinaryFunction::first_argument_type& arg) {
    return binder1st<BinaryFunction>(fn, arg);
}
```

We now have a complete working implementation of `bind1st`. If you actually open up the `<functional>` header and peek around inside, the code you'll find will probably bear a strong resemblance to what we've written here.

Limitations of the Functional Library

While the STL functional library is useful in a wide number of cases, the library is unfortunately quite limited. `<functional>` only provides 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. For example, it will provide a general function called `bind` that lets you bind as many values as you'd like to a function of arbitrary arity. Keep your eyes peeled for the next release of C++ – it will be far more functional than the current version!

Practice Problems

We've covered a lot of programming techniques in this chapter and there are no shortage of applications for the material. Here are some problems to get you thinking about how functors and adaptable functions can influence your programming style:

1. What is a functor?
2. What restrictions, if any, exist on the parameter or return types of `operator()`?
3. Why are functors more powerful than regular functions?
4. How do you define a function that can accept both functions and functors as parameters?
5. What is an adaptable function?
6. How do you convert a regular C++ function into an adaptable function?
7. What does the `bind1st` function do?
8. What does the `not2` function do?
9. The STL algorithm `for_each` accepts as parameters a range of iterators and a unary function, then calls the function on each argument. Unusually, the return value of `for_each` is the unary function passed in as a parameter. Why might this be?

10. Using the fact that `for_each` returns the unary function passed as a parameter, write a function `MyAccumulate` that accepts as parameters a range of `vector<int>::iterator`s and an initial value, then returns the sum of all of the values in the range, starting at the specified value. Do not use any loops – instead, use `for_each` and a custom functor class that performs the addition.

11. Write a function `AdvancedBiasedSort` that accepts as parameters a `vector<string>` and a `string` "winner" value, then sorts the range, except that all strings equal to the winner are at the front of the `vector`. Do not use any loops. *(Hint: Use the STL `sort` algorithm and functor that stores the "winner" parameter.)*

12. Modify the above implementation of `AdvancedBiasedSort` so that it works over an arbitrary range of iterators over strings, not just a `vector<string>`. Then modify it once more so that the iterators can iterate over any type of value.

13. The STL `generate` algorithm is defined as `void generate(ForwardIterator start, ForwardIterator end, NullaryFunction fn)` and iterates over the specified range storing the return value of the zero-parameter function `fn` as it goes. For example, calling `generate(v.begin(), v.end(), rand)` would fill the range [`v.begin()` to `v.end()`) with random values. Write a function `FillAscending` that accepts an iterator range, then sets the first element in the range to zero, the second to one, etc. Do not use any loops.

14. Write a function `ExpungeLetter` that accepts four parameters – two iterators delineating an input range of `string`s, one iterator delineating the start of an output range, and a character – then copies the strings in the input range that do not contain the specified character into the output range. The function should then return an iterator one past the last location written. Do not use loops. *(Hint: Use the `remove_copy_if` algorithm and a custom functor).*

15. The *standard deviation* of a set of data is a measure of how much the data varies from its average value. Data with a small standard deviation tends to cluster around a point, while data with large standard deviation will be more spread out.

    The formula for the standard deviation of a set of data $\{x_1, x_2, ..., x_n\}$ is

    $$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2}$$

    Here, $\bar{x}$ is the average of the data points.

    To give a feeling for this formula, given the data points 1, 2, 3, the average of the data points is 2, so the standard deviation is given by

    $$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2}=\sqrt{\frac{1}{3}\sum_{i=1}^{3}(x_i-2)^2}=\sqrt{\frac{1}{3}((1-2)^2+(2-2)^2+(3-2)^2)}=\sqrt{\frac{1}{3}(1+0+1)}=\sqrt{\frac{2}{3}}$$

    Write a function `StandardDeviation` that accepts an input range of iterators over `double`s (or values implicitly convertible to `double`s) and returns its standard deviation. Do not use any loops – instead use the `accumulate` function to compute the average, then use `accumulate` once more to compute the sum. *(Hint: To get the number of elements in the range, you can use the `distance` function)*

16. Write a function `ClearAllStrings` that accepts as input a range of iterators over `string`s that sets each string to be the empty string. If you harness the `<functional>` library correctly here, the function body will be only a single line of code.

17. 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 256 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.

18. Write a template function `CapAtValue` that accepts a range of iterators and a value by reference-to-`const` and replaces all elements in the range that compare greater than the parameter with a copy of the parameter. *(Hint: use the `replace_if` algorithm)*

19. One piece of functionality missing from the `<functional>` library is the ability to bind the first parameter of a unary function to form a nullary function. In this practice problem, we'll implement a function called `BindOnly` that transforms a unary adaptable function into a nullary function.

    a. Write a template functor class `BinderOnly` parameterized whose constructor accepts an adaptable function and a value to bind and whose `operator()` function calls the stored function passing in the stored value as a parameter. Your class should have this interface:

```
template <typename UnaryFunction> class BinderOnly {
public:
    BinderOnly(const UnaryFunction& fn,
               const typename UnaryFunction::argument_type& value);
    RetType operator() () const;
};
```

    b. Write a template function `BindOnly` that accepts the same parameters as the `BinderOnly` constructor and returns a `BinderOnly` of the proper type. The signature for this function should be

```
template <typename UnaryFunction>
    BinderOnly<UnaryFunction>
        BindOnly(const UnaryFunction &fn,
                 const typename UnaryFunction::argument_type& value);
```

20. Another operation not supported by the <functional> library is *function composition*. For example, given two functions **f** and **g**, the composition **g** ○ **f** is a function such that **g** ○ **f**(x) = **g**(**f**(x)). In this example, we'll write a function `Compose` that lets us compose two unary functions of compatible types.

    a. Write a template functor `UnaryCompose` parameterized over two adaptable function types whose constructor accepts and stores two unary adaptable functions and whose `operator()` accepts a single parameter and returns the composition of the two functions applied to that argument. Make sure that `UnaryCompose` is an adaptable unary function.

    b. Write a wrapper function `Compose` that takes in the same parameters as `UnaryCompose` and returns a properly-constructed `UnaryCompose` object.

    c. Explain how to implement `not1` using `Compose` and `logical_not`, a unary adaptable function exported by `<functional>` that returns the logical inverse of its argument.