CHAPTER

5 Design Patterns

In the earlier chapters, we have seen that a central goal of object-oriented programming is the support of code reuse. This chapter examines the tangential, but equally important concept of idea reuse: the reuse of basic programming and design technique, rather than the reuse of actual code. Many of these techniques are known as *design patterns*.

In this chapter, we will see:

- Why design patterns are important
- Several examples of object-based design patterns, including the Functor, Adapter, Wrapper, Iterator, Composite, and Observer patterns.
- Some sophisticated C++ programming tricks
- A brief discussion of object-oriented (inheritance-based) design patterns.

5.1 What Is a Pattern?

Although software design and programming are often difficult challenges, many experienced software engineers will argue that software engineering really has only a relatively small set of basic problems. Perhaps this is an understatement, but it is true that many basic problems are seen over and over in software projects. Software engineers who are familiar with these problems, and in particular, the efforts of other programmers in solving these problems, have the advantage of not needing to "reinvent the wheel."

A *design pattern* describes a problem that occurs over and over in software engineering, and then describes the solution in a sufficiently generic manner as to be applicable in a wide variety of contexts.

The idea of a design pattern is to document a problem and its solution so that others can take advantage of the collective experience of the entire software engineering community. Writing a pattern is much like writing a recipe for a cookbook; many common patterns have been written and rather than expending energy reinventing the wheel, these patterns can be used to write better programs. Thus a *design pattern* describes a problem that occurs over and over in software engineering, and then describes the solution in a sufficiently generic manner as to be applicable in a wide variety of contexts.

Like a recipe, design patterns usually follow a format. There are several formats for designing a pattern. Typically, a design pattern consists of:

1. The pattern name.
2. The problem, including a specification of the problem the pattern solves, an explanation of why this problem is important, some applications of the problem, and examples of known uses.
3. The solution, including a description of the classes in the patterns, possibly with a structure diagram, a generic (language independent) implementation of the pattern, with language-specific issues as appropriate, and sample code.
4. Consequences, including the results and trade-offs, of using the pattern, and a discussion of related patterns.

In this chapter, we will discuss several patterns that are commonly used in object-based data structures. Our pattern specifications will not be as formal as discussed above, but where reasonable, we will provide some code, and pointers to the use of the pattern elsewhere in the text. Toward the end of the chapter, we will see some object-oriented patterns.

5.2 The Functor (Function Objects)

In Chapter 3, we saw how function templates can be used to write generic algorithms. As an example, the function template in Figure 5.1 can be used to find the maximum item in an array.

```
 1 // Generic findMax for Comparables; uses natural order.
 2 // Precondition: a.size( ) > 0.
 3 template <class Comparable>
 4 const Object & findMax( const vector<Comparable> & a )
 5 {
 6     int maxIndex = 0;
 7
 8     for( int i = 1; i < a.size( ); i++ )
 9         if( a[ maxIndex ] < a[ i ] )
10             maxIndex = i;
11
12     return a[ maxIndex ];
13 }
```

Figure 5.1 Generic `findMax` algorithm: works only for `Comparable` objects, and uses their natural order

```
 1 // A simple rectangle class.
 2
 3 class Rectangle
 4 {
 5   public:
 6     Rectangle( int len = 0, int wid = 0 )
 7       : length( len ), width( wid ) { }
 8
 9     int getLength( ) const
10       { return length; }
11
12     int getWidth( ) const
13       { return width; }
14
15     void print( ostream & out = cout ) const
16       { out << "Rectangle " << getLength( ) << " by "
17             << getWidth( ); }
18
19   private:
20     int length;
21     int width;
22 };
23
24 ostream & operator<< ( ostream & out, const Rectangle & rhs )
25 {
26     rhs.print( out );
27     return out;
28 }
```

Figure 5.2 `Rectangle` class that does not have a meaningful `operator<` function

However, the template has an important limitation. That is, it works only for objects that have an `operator<` function defined, and it uses that `operator<` as the basis for all comparison decisions. There are many situations in which this is not feasible. As an example, consider the `Rectangle` class in Figure 5.2.

The `Rectangle` class does not have an `operator<`. The main reason for this is that because there are many plausible alternatives, it is difficult to decide on a good meaning for `operator<`. We could base the comparison on area, perimeter, length, width, and so on. As we saw in the `Shape` example in Section 4.3, once we write `operator<` as a member function, we are stuck with it.

Even without `operator<` as a member function, we could still write a two-parameter nonmember `operator<`, and separate the comparison from the rectangle class. Thus each time we write a new program, the program can specify its own version of `operator<`. This is a slight improvement, but we can write only one `operator<` for the program. What if we want to have `findMax` work with several different comparison alternatives?

```
// Generic findMax, with a function object.
// Uses a named method in the function object.
// Precondition: a.size( ) > 0.
template <class Object, class Comparator>
const Object & findMax( const vector<Object> & a,
                        Comparator comp )
{
    int maxIndex = 0;

    for( int i = 1; i < a.size( ); i++ )
        if( comp.isLessThan( a[ maxIndex ], a[ i ] ) )
            maxIndex = i;

    return a[ maxIndex ];
}


// Compare object: ordering by length.
class LessThanByWidth
{
  public:
    bool isLessThan( const Rectangle & lhs,
                     const Rectangle & rhs ) const
      { return lhs.getWidth( ) < rhs.getWidth( ); }
};

int main( )
{
    ...
    cout << findMax( a, LessThanByWidth( ) ) << endl;
    ...
}
```

Figure 5.3 Example of a function object that does not use function call operator overloading

The solution to the problem is to pass the comparison function as a second parameter to `findMax`, and have `findMax` use the comparison function instead of assuming the existence of an `operator<`. Thus `findMax` will now have two parameters: a `vector` of `Object` (which need not have any `operator<` defined), and a comparison function.

The main issue left is how to pass the comparison function. Some languages, including C++, allow parameters to be functions (actually they are pointers to functions). However, this solution often has efficiency problems and is not available in all object-oriented languages.

Instead, a solution that works in all object-oriented languages is to use a function object. The function object is sometimes known as a *functor*.

```
 1 // Generic findMax, with a C++-style function object.
 2 // Precondition: a.size( ) > 0.
 3 template <class Object, class Comparator>
 4 const Object & findMax( const vector<Object> & a,
 5                          Comparator isLessThan )
 6 {
 7     int maxIndex = 0;
 8
 9     for( int i = 1; i < a.size( ); i++ )
10         if( isLessThan( a[ maxIndex ], a[ i ] ) )
11             maxIndex = i;
12
13     return a[ maxIndex ];
14 }
```

Figure 5.4 `findMax` with a C++-style function object.

The function object contains a method specified by the generic algorithm. An instance of the class is passed to the algorithm.

The function object often contains no data. It contains a single method, with a given name, that is specified by the generic algorithm (in this case `findMax`). The object, which is simply an instance of the single-method class is then passed to the algorithm, which in turn calls the single method of the function object. We can design different comparison functions by simply declaring new classes. Each new class contains a different implementation of the agreed-upon single method. An example is shown in Figure 5.3.

Functor is another name for a function object.

`findMax` now takes two parameters. The second parameter is the function object. As shown on line 11, `findMax` expects that the function object implements a method named `isLessThan` (specifically, it is expecting that `isLessThan` takes two parameters, both of type compatible with `Object`, for whatever `Object` turns out to be). Since the name of the function object parameter is `comp`, line 11 calls the comparison function by `comp.isLessThan`. What is the type of `comp`? `comp` can be any type (that has an `isLessThan` method), so we know its type is a second template parameter. We use the symbolic name `Comparator` to signify its role. (`Functor` would work also, but `Comparator` is a more specific name for this function object.)

Once `findMax` is written, it can be called in `main`. To do so, we need to pass to `findMax` a `vector` of `Rectangle` objects and a function object that has an `isLessThan` method. We implement a new class `LessThanByWidth`, which contains the required method. That method returns a Boolean indicating if the first rectangle is less than the second rectangle on the basis of widths. `main` simply passes an *instance* of `LessThanByWidth` to `findMax`.

Function objects can be passed using call by value. Their calls can be inlined.

Observe that the `LessThanByWidth` object has no data members. This is usually true of function objects, and so function objects are often passed using call by value. Observe also that when the `findMax` template is expanded, the type of `comp` is known. Therefore, the definition of `comp.isLessThan` is also known. An aggressive compiler can perform the inline optimization of replacing the function call to `comp.isLessThan` with the actual definition. In many

cases, the ability of the compiler to perform this inline optimization can significantly decrease the running time.

The function object technique is an illustration of a pattern that we see over and over again, not just in C++, but in any language that has objects. In C++, there is an additional coding trick that makes the code look much prettier in the generic algorithm, and only slightly uglier in the function object's class.

The prettier `findMax` is shown in Figure 5.4. There are two basic changes:

- Line 10 now has the call to `isLessThan`, apparently without a controlling object.
- `isLessThan` is the parameter to `findMax`.

So it appears that we are passing the method directly to `findMax` and bypassing the function object. That is not what is actually happening. Instead, we have used the C++ trick of overloading the function call operator in the function object itself. What we mean by this is that since `isLessThan` is an object, line 10, which reads as

```
if( isLessThan( a[ maxIndex ], a[ i ] ) )
```

is actually interpreted as

```
if( isLessThan.operator() ( a[ maxIndex ], a[ i ] ) )
```

```
 1 // Compare object: ordering by length.
 2 class LessThanByLength
 3 {
 4   public:
 5     bool operator( ) ( const Rectangle & lhs,
 6                        const Rectangle & rhs ) const
 7       { return lhs.getLength( ) < rhs.getLength( ); }
 8 };
 9
10
11 // Compare object: ordering by area.
12 class LessThanByArea
13 {
14   public:
15     bool operator( ) ( const Rectangle & lhs,
16                        const Rectangle & rhs ) const
17       { return lhs.getLength( ) * lhs.getWidth( ) <
18                rhs.getLength( ) * rhs.getWidth( ); }
19 };
```

Figure 5.5 Two C++-style function objects to compare `Rectangle` objects

`operator()` is the function call operator, much like `operator[]` is the array indexing operator. With this mind-blowing piece of C++ syntax, we can implement various function objects in Figure 5.5. Figure 5.6 shows a simple `main` that places some `Rectangle` objects in a `vector` and then, using the function objects of types defined in Figure 5.5, finds the maximum `Rectangle`, first on the basis of length, and then on the basis of area.

`operator()` is the function call operator

5.3 Adapters and Wrappers

When we implement algorithms, often we run into a language typing problem: we have an object of one type, but the language syntax requires an object of a different type. Sometimes a type conversion works. Often we need a little more.

For instance, in Section 4.3, we found that sorting pointers to shapes did not quite work because `operator<` is already defined for pointers. Consequently, we needed to define a new class that stored the pointer and defined a meaningful `operator<`.

```
 1 #include <iostream>
 2 #include <vector>
 3 using namespace std;
 4
 5 // main: create four rectangles.
 6 // find the max, using two different criteria.
 7 int main( )
 8 {
 9     vector<Rectangle> a;
10
11     a.push_back( Rectangle( 1, 10 ) );
12     a.push_back( Rectangle( 10, 1 ) );
13     a.push_back( Rectangle( 5, 5 ) );
14     a.push_back( Rectangle( 4, 6 ) );
15
16     cout << "Largest length:\n\t"
17          << findMax( a, LessThanByLength( ) ) << endl;
18     cout << "Largest area:\n\t"
19          << findMax( a, LessThanByArea( ) ) << endl;
20
21     return 0;
22 }
```

Figure 5.6 Sample program that illustrates `findMax` with two different comparison functions

A *wrapper class* stores a primitive type, and adds operations that the primitive type does not support correctly.
An *adapter class* is used when the interface of a class is not exactly what is needed.

This technique illustrates the basic theme of a *wrapper class*. One typical use is to store a primitive type, and add operations that the primitive type either does not support or does not support correctly. A similar concept is an *adapter class*. An adapter class is typically used when the interface of a class is not exactly what is needed, and provides a wrapping effect, while changing the interface (in a manner similar to electrical adapters).

One difference between wrappers and adapters is that often an adapter is implemented by private inheritance. Notice that the implementation via private inheritance is often not available for wrapper classes that wrap a primitive (and therefore, non-class) type. This section illustrates some useful examples of wrapper classes and also provides an example of an adapter class.

5.3.1 Wrapper for Pointers

This subsection illustrates two wrapper templates for pointers. Our first template is based on the shape example. We will use it when we discuss indirect sorting in Section 9.9. The second example illustrates a casual implementation of the newly standardized `auto_ptr` class. This wrapper is used to add limited destructor-like behavior to pointers.

Pointers for Sorting

The `Pointer` will store, as a data member, a pointer to a `Comparable`.

Our first wrapper is the class template, `Pointer`, shown in Figure 5.7. The `Pointer` will store, as a data member, a pointer to a `Comparable`. We can then provide a comparison operator for the `Pointer` type.

```
 1 // Class that wraps a pointer variable for sorting.
 2
 3 template <class Comparable>
 4 class Pointer
 5 {
 6   public:
 7     explicit Pointer( Comparable *rhs = NULL )
 8       : pointee( rhs ) { }
 9     bool operator<( const Pointer & rhs ) const
10       { return *pointee < *rhs.pointee; }
11     operator Comparable * ( ) const
12       { return pointee; }
13     Comparable * get( ) const
14       { return pointee; }
15
16   private:
17     Comparable *pointee;
18 };
```

Figure 5.7 `Pointer` class, which wraps a pointer variable for sorting

The data member, `pointee`, is declared in the `private` section at line 17. The constructor for the `Pointer` class requires an initial value for `pointee` (which defaults to `NULL` if omitted); this is shown at line 7.

Classes that encapsulate the behavior of a pointer are sometimes called *smart pointer classes*. This class is smarter than a plain pointer because it automatically initializes itself to `NULL`, if no initial value is provided.

Classes that encapsulate the behavior of a pointer are called *smart pointer classes*.

Implementing `operator<` is identical to the `PtrToShape` class in Section 4.18. As was done there, we just apply the `<` operator to the `Comparable` objects that are being pointed at. Note carefully that this is not circular logic. The (template) `operator<` at line 9 in class `Pointer` compares two `Pointer` types; the call at line 10 will compare two `Comparable` types.

Line 11 shows bizarre C++ syntax at its finest. This is the type conversion operator; in particular, this method defines a type conversion from `Pointer<Comparable>` to `Comparable *`. The implementation is simple enough; we just return `pointee` at line 12. This allows us to get at the pointer. Although we could have used a named member function, such as `get`, at line 13, this type conversion simplifies the `largeObjectSort` algorithm in Figure 9.9.

It is important to remark that these type conversions are great when they work, but they can cause unexpected problems. Consider the code in Figure 5.8. We have provided `operator!=`, and to avoid any suspicious compiler bugs, we write it as a real function, instead of a function template.

Type conversions can cause unexpected problems.

Suppose that the constructor at line 7 in Figure 5.7 was not declared using `explicit`. In this case line 13 would not compile! This is because an ambiguity would be created. We can either convert `q` to an `int *`, and use the `!=` operator that is defined for primitive pointer variables, or we can promote `p` to a `Pointer<int>`, using the constructor, and then use the `!=` defined for `Pointer<int>`.

Avoid dual-direction implicit conversions in any non-trivial class.

```
 1 // Define != for two Pointer<int> objects.
 2 bool operator!= ( const Pointer<int> & lhs,
 3                   const Pointer<int> & rhs )
 4 {
 5     return lhs < rhs || rhs < lhs;
 6 }
 7
 8 int main( )
 9 {
10     int *p = new int( 3 );
11     Pointer<int> q( new int( 4 ) );
12
13     if( q != p )  // Compare pointed-at objects???
14         cout << "q and p are different" << endl;
15     return 0;
16 }
```

Figure 5.8 Ambiguity with dual-direction type conversions

There are lots of ways out of this quandary, but suffice it to say, because of this, you should generally avoid dual-direction implicit conversions in any non-trivial class. If you always use `explicit`, or you never use type conversion operators, you won't have this problem.

Auto-pointers

The recently adopted C++ standard has added a new wrapper class called the `auto_ptr`. This class is intended to help solve three common C++ pointer problems.

The `auto_ptr` wraps a pointer and its destructor calls `delete`.

Recall that if an object is dynamically allocated by a call to `new`, then it must eventually be freed by a call to `delete`. Otherwise, we can have a memory (or other resource) leak. The `auto_ptr` is intended to help do this automatically by wrapping the pointer inside the `auto_ptr` class, and then having the `auto_ptr` destructor call `delete`. There are three scenarios that the `auto_ptr` is designed for.

Scenario #1: We need to `delete` a local dynamically allocated object.

In the first scenario, inside a function, a local pointer variable allocates, via `new`, an object that has local scope only. When the function returns, it is expected that the object is freed by a call to `delete`. Typical code looks like:

```
void func( )
{
    Object *obj = new Object( ... );
     ...
    delete obj;
}
```

This code seems simple enough, but there are dangers lurking. If `func` has multiple returns, then we must be careful that `delete` is always reached. And if `func` throws an uncaught exception, the `delete` never occurs. However, since destructors are always called when a function exits (even if via an uncaught exception), if the pointer is wrapped inside an object whose destructor calls `delete`, the memory will be reclaimed.

Scenario #2: We need to `delete` an object created and returned from a completed function call.

The second scenario is a function that allocates an object and returns a pointer to it. The caller is now expected to call `delete` when it no longer needs the object. Typical code looks like:

```
Object *func1( )
{
    Object *obj1 = new Object( ... );
     ...
    return obj1;
}
```

```
void func( )
{
    Object *obj = func1( );
     ...
    delete obj;
}
```

This has similar problems as the first scenario (we must make sure that we reach the `delete` in all cases), except that we must make sure `delete` is not called in `func1`.

Scenario #3: We need to `delete` a dynamically allocated object created by the calling function.

A third scenario is a function that allocates an object and then calls another function, with the expectation that the other function will clean up, as shown below:

```
void func( Object *obj )
{
     ...
    delete obj;
}

void func1( )
{
    Object *obj1 = new Object( ... );
     ...
    func( obj1 );
     ...
}
```

Do not use `auto_ptr` for more than is intended.

We remark that this all the `auto_ptr` is expected to be used for. Fancier automatic garbage collection requires more sophisticated logic.

The basic property of the `auto_ptr` wrapper is as follows. It will wrap the pointer variable (thus it stores the value of the pointer as the `pointee` data member). It will also maintain an indication of whether or not it is the owner of the pointer (as a Boolean `isOwner`). If it is the owner, then when its destructor is called, it must apply the `delete` operator to the pointer. When a copy is performed ownership is transferred.

The destructor calls `delete` if it owns the pointee.

Figure 5.9 illustrates our version, the `AutoPointer`. The basic constructor is shown starting at line 7. It simply sets `pointee` and `isOwner`. The destructor calls the member function `free`, which itself calls `delete` if the `AutoPointer` is the owner (of `pointee`). Lines 37 and 38 implement some of the same logic that was seen in the `Pointer` class earlier in this section. The code in lines 33 to 36 implements the dereferencing operator (which is the equivalent of `*pointee`), and `->` (which is the equivalent of `pointee`).

The `release` method gives up ownership of the pointee.

Only three other methods are left: `release`, the copy constructor, and `operator=`. The `release` method (lines 39 and 40) gives up ownership, but otherwise behaves like `get`. The intent is that it can be called when control is transferred by a copy operation.

```
1  // Class that wraps a local pointer variable.
2
3  template <class Object>
4  class AutoPointer
5  {
6    public:
7      explicit AutoPointer( Object *rhs = NULL )
8        : isOwner( rhs != NULL ), pointee( rhs ) { }
9
10     AutoPointer( AutoPointer & rhs ) : isOwner( rhs.isOwner )
11       { pointee = rhs.release( ); }
12
13     ~AutoPointer( )
14       { free( ); }
15
16     const AutoPointer & operator= ( AutoPointer & rhs )
17     {
18         if( this != & rhs )
19         {
20             Object *other = rhs.get( );
21             if( other != pointee )
22             {
23                 free( );
24                 pointee = other;
25             }
26             else if( rhs.isOwner )
27                 isOwner = true;
28             pointee = rhs.release( );
29         }
30         return *this;
31     }
32
33     Object & operator* ( ) const
34       { return *get( ); }
35     Object * operator-> ( ) const
36       { return get( ); }
37     Object * get( ) const
38       { return pointee; }
39     Object * release( )
40       { isOwner = false; return pointee; }
41
42   private:
43     Object *pointee;
44     bool isOwner;
45
46     void free( )
47       { if( isOwner ) delete pointee; }
48 };
```

Figure 5.9 The `auto_ptr` class (we use a different name)

Thus, in the copy constructor at lines 10 and 11, instead of copying the pointer value with `pointee=rhs.pointee`, `rhs.pointee.release()` is used on the right-hand side. Ownership is transferred via the initializer list.

The most complicated routine is the copy assignment operator, shown at lines 16 to 31. The main complication is that before copying into the target, we must call `delete` on the target's current `pointee` if the target owns it (and the target's current `pointee` is different than the new `pointee`).

Member templates allow us to declare member function templates, using additional template types.

The main difference between this implementation and the implementation in the C++ standard is behavior under inheritance. Specifically, consider classes `Base` and `Derived`, as usual. Although a `Derived*` value can be copied into a `Base*` value, an `AutoPointer<Derived>` object cannot be copied into a `AutoPointer<Base>` object in our implementation. Adding this functionality requires using member templates. *Member templates* allow us to declare member function templates, using additional template types. For instance, here is how the copy constructor would look:

```
template<class OtherObject>
AutoPointer( AutoPointer<OtherObject> & rhs )
  : isOwner( rhs.isOwner ) { pointee = rhs.release( ); }
```

The main difference between this implementation and the implementation in the C++ standard is behavior under inheritance.

In this scenario, we define the construction (no longer a copy constructor) of an `AutoPointer` of one type with an `AutoPointer` of any other type. However, the assignment statement will generate a compiler error if the underlying pointees are not type compatible.

While this is a slick solution, unfortunately, member templates are a recent language addition that is not supported on all compilers.

5.3.2 A Constant Reference Wrapper

A reference variable must reference an object. Sometimes this is unfortunate.

Reference variables in C++ are different from pointer variables in several ways. One important difference is that whereas pointer variables can point at either an object or `NULL`, a reference variable must reference an object. Sometimes this is unfortunate. For instance, when we search for an object in an arbitrary container, we may want to return a reference to it. But what if the object is not found?

`Cref` wraps the behavior of a reference variable inside a class.

The solution, as usual, is to wrap the behavior of a reference variable inside a class. Our class, `Cref`, shown in Figure 5.10, mimics a constant reference variable. (Alternatively, we could implement a simple reference class, and even use inheritance to make the reference and constant reference classes type compatible. This is left as Exercise 5.18.) The implementation is straightforward.

`Cref` provides an `isNull` method that returns true if the null reference is being represented.

We store a pointer to the referenced object as a private data member. The pointer is initialized in the constructor, but is `NULL` if `Cref` is constructed with no parameters. We provide a `get` method that returns the constant reference and an `isNull` method that returns true if the null reference is being represented.

```
// Class that wraps a constant reference variable.
// Useful for return value from a container find method.

template <class Object>
class Cref
{
  public:
    Cref( ) : obj( NULL ) { }
    explicit Cref( const Object & x ) : obj( &x ) { }

    const Object & get( ) const
    {
        if( isNull( ) )
            throw NullPointerException( );
        else
            return *obj;
    }

    bool isNull( ) const
      { return obj == NULL; }

  private:
    const Object *obj;
};
```

Figure 5.10 Constant reference wrapper

5.3.3 Adapters: Changing an Interface

The adapter pattern is used to change the interface of an existing class to conform to another.

The adapter pattern is used to change the interface of an existing class to conform to another. Sometimes it is used to provide a simpler interface, either with fewer methods, or easier-to-use methods. Other times it is used simply to change some method names. In either case, the implementation technique is similar.

As an example, our `MemoryCell` class in Section 3.4 uses `read` and `write`. But what if we wanted the interface to use `get` and `put` instead? There are two reasonable alternatives. One is to use composition. However, this means, for instance, that a call to `get` will then call `read`, thus adding an extra layer of overhead. The other alternative is to use private inheritance.

We use private inheritance to implement the new class, `StorageCell`, in Figure 5.11. Its methods are implemented by calls to the base class methods. As discussed in Section 4.4.5, in private inheritance, public methods in the base class are private in the derived class. Thus, as Figure 5.12 illustrates, the only visible methods are the `StorageCell` constructor, `get`, and `put`.

```
// A class for simulating a memory cell.

template <class Object>
class StorageCell : private MemoryCell<Object>
{
  public:
    explicit StorageCell( const Object & initialValue
                                                = Object( ) )
      : MemoryCell<Object>( initialValue ) { }

    const Object & get( ) const
      { return read( ); }
    void put( const Object & x )
      { write( x ); }
};
```

Figure 5.11 An adapter class that changes the `MemoryCell` interface to use `get` and `put`.

```
int main( )
{
    StorageCell<int>    m1;
    StorageCell<string> m2( "hello" );

    m1.put( 37 );
    m2.put( m2.get( ) + " world" );
    cout << m1.get( ) << endl << m2.get( ) << endl;

    // The next line does not compile if uncommented.
    // cout << m1.read( ) << endl;
    return 0;
}
```

Figure 5.12 Illustration of private inheritance: the `MemoryCell` methods are no longer visible

5.4 Iterators

An *iterator* object controls iteration of a collection.

Consider the problem of printing the elements in a collection. Typically, the collection is an array, so assuming that the object `v` is an expanded `vector` template, its contents are easily printed with code like:

```
for( int i = 0; i < v.size( ); i++ )
    cout << v[ i ] << endl;
```

In this loop, `i` is an *iterator* object, because it is the object that is used to control the iteration. However, using the integer `i` as an iterator constrains the design: We can only store the collection in an array-like structure. A more flexible alternative is to design an iterator class that encapsulates a position inside of a collection. The iterator class provides methods to step through the collection.

When we program to an interface, we write code that uses the most abstract methods. These methods will be applied to the actual concrete types.

The key is the concept of programming to an interface: We want the code that performs access of the container to be as independent of the type of the container as possible. This is done by using only methods that are common to all containers and their iterators.

There are many different possible iterator designs. We describe three designs, in increasing order of complexity. Iterators are a core component of the STL, and its design is similar to our second design (but of course, is slightly more complex). In Chapter 7, we discuss STL iterators. They are used throughout the case studies in Part III and some implementations of these iterators and the collections that they iterate over are provided in Part IV.

5.4.1 Iterator Design #1

`getIterator` returns an appropriate iterator for the collection.

The first iterator design uses only three methods. The container class is required to provide a `getIterator` method. `getIterator` returns an appropriate iterator for the collection. The iterator class has the other two methods, `hasNext` and `next`. `hasNext` returns true if the iteration has not yet been exhausted. `next` returns the next item in the collection (and in the process, advances the notion of the current position). This iterator interface matches one that is provided in the Java programming language.

To illustrate the implementation of this design, we write the collection and iterator class templates, `MyVector` and `VectorIterator`, respectively. Their use is shown in Figure 5.13. `MyVector` is written in Figure 5.14. To simplify matters, we inherit from the `vector` class. The only difference between `MyVector` and `vector` is its `getIterator` method. (The use of inheritance here has nothing to do with the iterator pattern.)

```
int main( )
{
    MyVector<int> v;

    v.push_back( 3 );
    v.push_back( 2 );

    cout << "Vector contents: " << endl;

    VectorIterator<int> itr = v.getIterator( );
    while( itr.hasNext( ) )
        cout << itr.next( ) << endl;

    return 0;
}
```

Figure 5.13 `main` method to illustrate iterator design #1

```
template <class Object>
class VectorIterator;

// Same as the vector, but has a getIterator method.
// No extra data, no overridden methods, so non-virtual
// destructor in original vector is OK!

template <class Object>
class MyVector : public vector<Object>
{
  public:
    explicit MyVector( int size = 0 )
      : vector<Object>( size ) { }

    VectorIterator<Object> getIterator( ) const
      { return VectorIterator<Object>( this ); }
};
```

Figure 5.14 The `MyVector` class, designs #1 and #2

```
1  // A passive iterator class. Steps through its MyVector.
2
3  template <class Object>
4  class VectorIterator
5  {
6    public:
7      VectorIterator( const MyVector<Object> *v )
8        : owner( v ), count( 0 ) { }
9
10     bool hasNext( ) const
11       { return count != owner->size( ); }
12
13     const Object & next( )
14       { return (*owner)[ count++ ]; }
15
16   private:
17     const MyVector<Object> *owner;
18     int count;
19 };
```

Figure 5.15 Implementation of the `VectorIterator`, design #1

The iterator is constructed with a pointer to the container that it iterates over.

`getIterator` simply returns a new iterator; notice that the iterator must have information about the container that it is iterating over. Thus the iterator is constructed with a pointer to the `vector`. The only other method in `MyVector` is the constructor, which simply calls the base-class constructor. Because we are using public inheritance, it would be proper to change the `vector` class' destructor to be virtual, if possible. However, in this case it does not matter, since `MyVector` adds no extra data members. As a result, this turns out to be a nice example of public inheritance.

```
1  // A passive iterator class. Steps through its MyVector.
2
3  template <class Object>
4  class VectorIterator
5  {
6    public:
7      VectorIterator( const MyVector<Object> *v )
8        : owner( v ), count( 0 ) { }
9
10     void reset( )
11       { count = 0; }
12
13     bool isValid( ) const
14       { return count < owner->size( ); }
15
16     void advance( )
17       { count++; }
18
19     const Object & retrieve( ) const
20       { return (*owner)[ count ]; }
21
22   private:
23     const MyVector<Object> *owner;
24     int count;
25 };
26
27 int main( )
28 {
29     MyVector<int> v;
30
31     v.push_back( 3 );
32     v.push_back( 2 );
33     VectorIterator<int> itr = v.getIterator( );
34
35     for( int i = 0; i < 2; i++ )
36     {
37         cout << "Vector contents: " << endl;
38         for( itr.reset( ); itr.isValid( ); itr.advance( ) )
39             cout << itr.retrieve( ) << endl;
40     }
41
42     return 0;
43 }
```

Figure 5.16 A new `VectorIterator`, with some additional functionality, and a test program

Incomplete class declarations are necessary when two or more classes refer to each circularly.

Lines 1 and 2 represent an *incomplete class declaration*. This particular declaration states that `VectorIterator` is a class template, but does not provide the class definition. However, this is enough to make line 16 compile. Incomplete class declarations are necessary when two or more classes refer to each circularly.

Figure 5.15 shows the `VectorIterator`. The iterator keeps a variable (`count`) that represents the current position in the vector, and a pointer to the vector. The implementation of the constructor and two member functions is straightforward. The constructor directly initializes the data members in the initializer list, `hasNext` simply compares the current position with the vector size, and `next` uses the current position to index the array (and then advances the current position). Notice the use of `const` throughout, to ensure that this iterator makes no attempt to modify the container.

5.4.2 Iterator Design #2

The second design puts more functionality in the iterator

A limitation of the first iterator design is the relatively limited interface. Observe that it is impossible to reset the iterator back to the beginning, and that the `next` method couples access of an item with advancing. A second design, shown in Figure 5.16, puts more functionality in the iterator. It leaves the `MyVector` class completely unchanged.

The STL iterator has some commonality with our second design, but also some important differences.

The STL iterator has some commonality with our second design, but also some important differences. It is similar in that the methods to advance and retrieve are separate. It is different in that the iterator can make changes to the underlying collection.

```
template <class Object>
class Iterator;

template <class Object>
class VectorIterator;

// Same as the vector, but has a getIterator method.
// No extra data, no overridden methods, so non-virtual
// destructor in original vector is OK!

template <class Object>
class MyVector : public vector<Object>
{
  public:
    explicit MyVector( int size = 0 )
      : vector<Object>( size ) { }

    Iterator<Object> *getIterator( ) const
      { return new VectorIterator<Object>( this ); }
};
```

Figure 5.17 Inheritance-based iterator design. `getIterator` returns a pointer to an iterator

```
// A passive iterator class protocol.
// Steps through its container.

template <class Object>
class Iterator
{
  public:
    virtual ~Iterator( ) { }

    virtual bool hasNext( ) const = 0;
    virtual const Object & next( ) = 0;
};


// A concrete implementation of the iterator.
// Could have been nested inside of MyVector!

template <class Object>
class VectorIterator : public Iterator<Object>
{
  public:
    VectorIterator( const MyVector<Object> *v )
      : owner( v ), count( 0 ) { }

    bool hasNext( ) const
      { return count != owner->size( ); }

    const Object & next( )
      { return (*owner)[ count++ ]; }

  private:
    const MyVector<Object> *owner;
    int count;
};
```

Figure 5.18 The iterator abstract class and a concrete implementation

Another difference is that the STL iterator does not have `isValid` or `reset` methods. Instead, the container has a method to return an invalid iterator and a method to return an iterator representing the starting point. We can test if an iterator is in an invalid state by comparing it with the invalid iterator given by the container. We can reset the iterator by copying the starting point iterator into it. The STL also makes extensive use of operator overloading. For example, `advance` is replaced with `operator++`. Details on using STL iterators can be found in Chapter 7.

5.4.3 Inheritance-based Iterators and Factories

The iterators designed so far manage to abstract the concept of iteration into an iterator class. This is good, because it means that if the collection changes from an array-based collection to something else, the basic code such as lines 38 and 39 in Figure 5.16 does not need to change.

While this is a significant improvement, changes from an array-based collection to something else require that we change all the declarations of the iterator. For instance, in Figure 5.16, we would need to change line 33. We discuss an alternative in this section.

An inheritance-based iteration scheme defines an iterator abstract base class. Clients program to this interface.

Our basic idea is to define an abstract base class `Iterator`. Corresponding to each different kind of contain is an iterator that implements the `Iterator` protocol. In our example, this gives three classes: `MyVector`, `Iterator`, and `VectorIterator`. The relationship that holds is `VectorIterator` IS-A `Iterator`. The reason we do this is that each container can now create an appropriate iterator, but pass it back as an abstract `Iterator`.

The inheritance-based scheme introduces pointers (or references). The iterators are now allocated by `new`.

Figure 5.17 shows `MyVector`. The new `MyVector` returns a pointer to an `Iterator` object that is dynamically constructed by calling `new`. Since `VectorIterator` IS-A `Iterator`, this is safe to do. Notice that the use of inheritance and polymorphism will require that we introduce pointers or references. As we will see, this muddies the code a bit, which is one reason why the STL is template-based, rather than inheritance-based.

A *factory method* creates a new concrete instance, but returns it using a pointer (or reference) to the abstract class.

Because `getIterator` creates and returns a new `Iterator` object, whose actual type is unknown, it is commonly known as a *factory method*. The iterator classes are shown in Figure 5.18. First, we have the abstract class `Iterator`, which serves simply to establish the protocol by which all subclasses of `Iterator` can be accessed. The protocol is specified with pure virtual functions. As usual, we have a virtual destructor.

This version of `VectorIterator` is essentially identical to the original implementation in Figure 5.18, except that it is a derived class of `Iterator`.

```
 1 int main( )
 2 {
 3     MyVector<int> v;
 4
 5     v.push_back( 3 );
 6     v.push_back( 2 );
 7
 8     cout << "Vector contents: " << endl;
 9
10     Iterator<int> *itr = v.getIterator( );
11     while( itr->hasNext( ) )
12         cout << itr->next( ) << endl;
13
14     delete itr;
15
16     return 0;
17 }
```

Figure 5.19 Illustration of the use of iterators in inheritance-based design

Nowhere in `main` is there any mention of the actual iterator type.

Figure 5.19 demonstrates how the inheritance-based iterators are used. At line 10, we see the declaration of `itr`: it is now a pointer to an `Iterator`. Nowhere in `main` is there any mention of the actual `VectorIterator` type. In fact, we could have written `VectorIterator` as a nested class in the *private* section of `MyVector`. The fact that a `VectorIterator` exists is not used by any clients of the `MyVector` class. This is a very slick design, and illustrates nicely the idea of hiding an implementation and programming to an interface.

The changes to `main` are relatively minimal. Lines 11 and 12 change because we must use `operator->` instead of the dot operator.

The dynamically allocated iterator must be reclaimed by calling `delete`.

Line 14 illustrates a down side: the dynamically allocated iterator must be reclaimed by calling `delete`. Remembering to do this all the time is annoying. However, if we examine this closely, we see a classic application of scenario #2 for the `auto_ptr`. Recall from Section 5.3.1 (page 232) that in this scenario, an object is allocated inside a function (in this case, `getIterator`), and is returned to the caller. The caller is responsible for handling the `delete`. This is exactly the situation in Figure 5.19. Thus although inconvenient, there is some support in the language to make our life easier.

By the way, recall that at the end of Section 5.3.1 (beginning on page 235), we explained that our version of `AutoPointer` differed from the STL `auto_ptr` because the STL version allows any compatible pointers to be wrapped, while our version requires an exact type match. We also explained that member templates could be used to loosen the requirement of an exact type match. If we look at how the `auto_ptr` would be used here, we see that we would need to do the following:

1. In Figure 5.19, line 10, `itr` would be an `auto_ptr`, and line 14 would disappear.
2. In Figure 5.17, `getIterator` would be rewritten to return an `auto_ptr` (line 18) and the result of `new` would be wrapped inside an `auto_ptr` (line 19).

These changes to `getIterator`, give the following implementation:

```
auto_ptr<Iterator<Object> > getIterator( ) const
  { return auto_ptr<Iterator<Object> >( new
                    VectorIterator<Object>( this ) ); }
```

The construction of an `auto_ptr<Iterator<Object> >`, with a `VectorIterator<Object>` pointer, implies that we need the STL version.

5.5 Composite (Pair)

A common design pattern is to return two objects as a *pair*.

In most languages, a function can return only a single object. What do we do if we need to return two or more things? There are several possibilities. One alternative is to use reference variables. The other is to combine the objects into a single `struct` (or `class`). The most common situation in which multiple objects need to be returned is the case of two objects. So a common design pattern is to return the two objects as a *pair*. This is the Composite pattern.

Pairs are useful for implementing key-value pairs in maps and dictionaries.

In addition to the situation described above, pairs are useful for implementing maps and dictionaries. In both these abstractions, we maintain key-value pairs: the pairs are added into the map or dictionary, and then we search for a key, returning its value. One common way to implement a map as to use a set. In a set, we have a collection of items, and search for a match. If the items are pairs, and the match criterion is based exclusively on the key component of the pair, then it is easy to write an adapter class that constructs a map on the basis of a set. We will see this idea explored in more detail in Chapter 19.

The STL defines a `pair` class.

The Standard Template Library defines a `pair` class. An implementation is shown in Figure 5.20. Notice that it is a class only in the technical sense; the data members are public.

5.6 Observer

The Observer pattern involves a *subject* and a set of *observers*. The observers are to be informed whenever something interesting happens to the subject.

Our last pattern is the Observer pattern. The typical use of the Observer pattern involves a *subject* and a set of *observers*. For this to be interesting, the number of observers can vary as the application runs, and can be large. The basic premise is that the observers are to be informed whenever something interesting happens to the subject.

A simplistic example of this could be a windowing operating system, such as Windows or Mac O/S. When a window is created or maximized, it covers other windows. These windows that are covered are now observers in the sense that they want to be informed when the newly created window is minimized or destroyed, or simply moved, as this might require that the previously hidden windows be redrawn.

```
 1 // Class (more like a struct) that stores
 2 // a pair of objects.
 3
 4 template <class Type1, class Type2>
 5 class pair
 6 {
 7   public:
 8     Type1 first;
 9     Type2 second;
10     pair( const Type1 & f = Type1( ),
11           const Type2 & s = Type2( ) )
12       : first( f ), second( s ) { }
13 };
```

Figure 5.20 The `pair` class; basically the same as the STL version

Another example might be a class that wraps pointers. Several wrapper objects may be sharing a pointee that is pointing at some dynamically allocated object. If one instance of the wrapper calls `delete` on the dynamically allocated object, then the other wrappers have stale pointers, which gives undefined behavior that can be hard to detect. A solution is that when the `delete` is performed, we inform all the wrappers that are looking at that deleted object, perhaps setting their pointees to `NULL` to insure defined behavior.

Figure 5.21 contains an abstract base class `Observer` and base class (which could be abstract, but is not) for `Subject`.

The `Observer` abstract class specifies a protocol: when something interesting happens, the `Observer` is told about it by having its `update` method called. A subclass can override `update` to handle the interesting occurrence. For instance, in our windowing example, the observer windows will have their `update` method called when the window that was covering it is no longer in the way. The `update` method could redraw an appropriate portion of the screen.

The `Observer` abstract class specifies a protocol: when something interesting happens, the `Observer` is told about it by having its `update` method called.

```
1  class Subject;
2
3  class Observer
4  {
5    public:
6      virtual ~Observer( ) { }
7      virtual void update( Subject *observee ) = 0;
8  };
9
10 class Subject
11 {
12   public:
13     virtual ~Subject( ) { }
14
15       // Add obs to the set of observers; see online code.
16     virtual void addObserver( Observer *obs );
17
18       // Remove obs from the set of observers.
19     virtual void removeObserver( Observer *obs );
20
21       // Call the update method for all observers.
22     virtual void notifyAll( )
23     {
24         for( int i = 0; i < observers.size( ); i++ )
25             observers[ i ]->update( this );
26     }
27
28   private:
29     vector<Observer *> observers;
30 };
```

Figure 5.21 Abstract classes for `Observer`, and base class for observee (`Subject`).

The `Subject` class is not abstract (but see Exercise 5.7). Instead it defines methods to add an observer, remove an observer, and notify all observers that something has happened. It does this by keeping an internal list of all registered observers (possibly in a `vector`). The implementation of `addObserver` and `removeObserver` can be found in the online code.

The `Subject` class defines methods to add an observer, remove an observer, and notify all observers that something has happened

An illustration of the Observer pattern in action is shown in Figure 5.22. Here we have a subject, which is a `Timer` object, and we have `Echo` objects which

are the interested observers. The `Timer` object has a `tick` method; whenever the `tick` method is called, any `Echo` objects that are observing the `Timer` object are notified, and their `update` method is called. In our example, the `update` method simply prints a message, so we can see that the notification occurred.

```
1  // Timer class: tick method calls notify
2
3  class Timer : public Subject
4  {
5    public:
6      void tick( )
7        { notifyAll( ); }
8  };
9
10
11 // Echo class: this in an observer.
12 // It is constructed with a Timer object; when
13 // the Timer object ticks, update is
14 // automatically called.
15
16 class Echo : public Observer
17 {
18   public:
19     Echo( int id, Timer *t ) : myId( id ), observee( t )
20       { observee->addObserver( this ); }
21
22     ~Echo( )
23       { observee->removeObserver( this ); }
24
25     void update( Subject *s )
26       { if( observee == s ) cout << myId << endl; }
27
28   private:
29     int myId;
30     Subject *observee;
31 };
```

Figure 5.22 Concrete classes: `Echo` observes a `Timer` object, and reacts when the `Timer`'s `tick` method is called

```
 1 // A test program.
 2 void testEcho12( Timer & t)
 3 {
 4     Echo e1( 1, &t );
 5     Echo e2( 2, &t );
 6
 7     cout << "Expecting 1 and 2 to respond." << endl;
 8     t.tick( );
 9     cout << "1 and 2 disappear." << endl;
10 }
11
12 int main( )
13 {
14     Timer t;
15
16     testEcho12( t );   // 1 and 2 should respond
17
18     Echo e3( 3, &t );
19     Echo e4( 4, &t );
20
21     Timer other;
22     Echo e5( 5, &other );  // registered with other, not t
23
24     cout << "Only 3 and 4 are currently observable." << endl;
25     cout << "Expecting 3 and 4 to respond." << endl;
26     t.tick( );
27
28     return 0;
29 }
```

Figure 5.23 Program to test the `Timer` and `Echo` classes

The `Timer` class is often used to implement callback functions. A *callback* function is a function that is registered to be called at a later time.

A *callback* function is a function that is registered to be called at a later time.

Here, the `update` method in the various `Echo` objects are called back, when `tick` occurs. In a typical application, we have a set of actions that are to occur regularly, perhaps every hour. Each action registers itself with the `Timer`, and when the clock strikes a new hour, all registered actions are executed.

Thus, a `Timer` object simply calls its inherited `notifyAll` whenever tick is executed. The `Echo` class is somewhat more complicated. `Echo` objects are constructed by providing a `Timer` object. In other words, an observer is constructed by passing in the subject that it is observing. The observer is then added to the subject's list of interested parties (line 20). When the observer is no longer active, it is removed from the subject's list. This situation is handled in the `Echo` destructor. As mentioned earlier, we then provide the required implementation for the `Observer` protocol, which in our case means that we implement `update`. Our `update` simply prints a message.

A `Timer` object simply calls its inherited `notifyAll` whenever tick is executed. `Echo` objects are constructed by providing a `Timer` object.

A program that tests all of this is shown in Figure 5.23. First we declare a `Timer` object `t`. Occasionally, we will call `t.tick()`, and when this occurs, any `Echo` object that is actively registered with `t` will have its `update` method called.

Thus, when `testEcho12` is called, since `e1` and `e2` are both constructed with `t` as their subject, when `t`'s `tick` method is called, we will see output for objects `e1` and `e2`. Notice also that when `testEcho12` returns, `e1` and `e2` no longer exist, and their destructors make sure that they are no longer registered as observers for `t`. `e3`, `e4`, and `e5` are then constructed as new observers. Notice however, that `e5` is listening for `other.tick()`. Thus when `t`'s `tick` method is called at line 26, only `e3` and `e4` will respond.

Summary

The chapter illustrates several common design patterns that we will use throughout the text. Although we concentrate mostly on object-based design patterns, as we see from some of the patterns toward the end of the chapter, inheritance plays a central role in many patterns. Unfortunately, because of memory management issues, inheritance adds some complication to our implementations, and as a result we have chosen to use it sparingly. However, some of these patterns can give you a feel for the power of inheritance and the techniques that it introduces.

This chapter concludes the first part of the text, which provided an overview of C++ and object-oriented programming. We will now go on to look at algorithms and the building blocks of problem-solving programming.

Objects of the Game

Adapter A class that is typically used when the interface of another class is not exactly what is needed. The adapter provides a wrapping effect, while changing the interface. (230)

`auto_ptr` A class in the STL that is intended to help automatically `delete` dynamically allocated objects. (232)

callback function A function that is registered to be called at a later point. In the Observer pattern, `update` is such as function. (251)

Composite The pattern in which we store two or more objects in one entity. (247)

design pattern Describes a problem that occurs over and over in software engineering, and then describes the solution in a sufficiently generic manner as to be applicable in a wide variety of contexts. (223)

dual-direction implicit conversion The circumstance in which implicit conversions between two types are defined in both directions. Often this leads to ambiguities, and thus should be avoided. (231)

factory method A method that creates new concrete instances, but returns them using a pointer (or reference) to an abstract class. (245)
function object An object passed to a generic function with the intention of having its single method used by the generic function. (227)
Functor A function object. (227)
incomplete class declaration Used to inform the compiler of the existence of a class. Incomplete class declarations are necessary when two or more classes refer to each circularly. (242)
Iterator An object that is used to control iteration over a container. (237)
member template A member function that is itself a template. Member templates can occur inside of class templates, in which case, they have different template parameters. (235)
Observer The pattern that involves a subject and a set of observers. The observers are informed whenever something interesting happens to the subject. (248)
Pair The composite pattern with two objects. (247)
programming to an interface The technique of using classes by writing in terms of the most abstract interface. Attempts to hide even the name of the concrete class that is being operated on. (245)
smart pointer A generic name for a pointer wrapper class. (231)
Wrapper A class that is used to store a primitive type, and add operations that the primitive type either does not support or does not support correctly. (230)

Common Errors

1. When writing `operator()`, remember that you still must have a parameter list.
2. When you send a function object as a parameter, you must send a constructed object, and not simply the name of the class.
3. Using dual-direction implicit type conversions is dangerous because they can lead to ambiguities.
4. Using `auto_ptr` in a setting more complex than it was designed for is sure to lead to trouble, because the pointee will be destroyed if ownership is transferred to an owner that exits scope earlier than you think.
5. The adapter pattern often implies private, rather than public inheritance. If you are changing, rather than simply augmenting the class interface, then you do not want public inheritance.
6. Many design patterns that have cooperating classes will require incomplete class declarations because the class declarations refer to each other circularly.

7. In C++, inheritance based design patterns tend to require that the programmer deal with reclaiming dynamically allocated objects, often leading to memory leaks, stale pointers, and other assorted bugs.
8. Using too few classes can be a sign of poor design. Often adding a class can clean up the design.

On the Internet

Rectangle.cpp The `Rectangle` class in Figures 5.2, and 5.4 to 5.6.

Wrapper.h Contains both the `Cref` and `Pointer` classes.

Ambiguity.cpp Illustrates the problem with dual-direction implicit type conversions, shown in Figure 5.8. To see the problem, you must remove the `explicit` directive in the `Pointer` constructor in **Wrapper.h**, and you must be using a compiler that understands `explicit` (some simply ignore it).

AutoPointer.cpp Contains an implementation of `AutoPointer` shown in Figure 5.9, with a test program.

StorageCell.h The `StorageCell` adapter, shown in Figure 5.11.

TestStorageCell.cpp Tests the `StorageCell` adapter, as shown in Figure 5.12.

Iterator1.cpp The complete iterator class, with a test program, as shown in Section 5.4.1.

Iterator2.cpp The complete iterator class, with a test program, as shown in Section 5.4.2.

Iterator3.cpp The complete iterator class, with a test program, as shown in Section 5.4.3.

pair.h The `pair` class, shown in Figure 5.20.

Observer.cpp The complete set of classes, with a test program, for the Observer pattern discussed in Section 5.6.

Exercises

In Short

5.1. What is a design pattern?

5.2. Describe how function objects are implemented in C++.

5.3. Explain the Adapter and Wrapper patterns. How do they differ?

5.4. What are two common ways to implement adapters? What are the trade-offs between these implementation methods?

5.5. Describe in general the following patterns:

a. Iterator

b. Observer

5.6. Give an example of the Observer pattern that is not discussed in the text. Use a real-life example, rather than a programming example.

In Theory

5.7. A class is abstract if it has at least one abstract method. The `Subject` class in Figure 5.21 is therefore not abstract. A little-known feature of C++ allows abstract methods to have an implementation; this feature is primarily used on the destructor. Verify that this feature exists by making the `Subject` destructor a pure virtual function, while leaving its implementation in tact.

In Practice

5.8. Templates can be used to write generic function objects.

a. Write a class template function object named `less`. In the class template, overload `operator()` to call the `operator<` for its two `Comparable` parameters.

a. Rewrite the `findMax` function template in Figure 5.1 to call the `findMax` function template shown in Figure 5.4. Your rewrite may contain only one line in the function body: a call to the two-parameter `findMax`, that sends a compatible function object from part (a).

b. Instead of writing a separate one-parameter `findMax`, can we simply rewrite the two-parameter `findMax` function template shown in Figure 5.4 to accept a default parameter that is a compatible function object from part (a)? Verify your answer by compiling code.

5.9. This exercise asks you to write a generic `countMatches` function. Your function will take two parameters. The first parameter is an array of `int`. The second parameter is a function object that returns a Boolean.

a. `countMatches` returns the number of array items for which the function object returns true. Implement `countMatches`.

a. Test `countMatches` by writing a function object, `EqualsZero`, that overloads `operator()` to accept one parameter and returns true if the parameter is equal to zero. Use an `EqualsZero` function object to test `countMatches`.

a. Rewrite `countMatches` (and `EqualsZero`) using templates.

5.10. Although the function objects we have looked at store no data, this is not a requirement.

a. Write a function object `EqualsK`. `EqualsK` contains one data member (`k`). `EqualsK` is constructed with a single parameter (default is zero) that is used to initialize `k`. Its one parameter `operator()` returns true if the parameter is equal to `k`.

a. Use `EqualsK` to test `countMatches` in Exercise 5.9 (if you did part (c), then make `EqualsK` a template).

5.11. Add `operator*` and `operator->` to the `Pointer` class template in Figure 5.7.

5.12. Is it safe to apply `insertionSort` (Figure 3.4) to a vector of `auto_ptr` objects?

5.13. The `StorageClass` adapter is implemented by private inheritance. Redo the implementation in Figure 5.11 using composition. Test your program by calling a significant number of `StorageClass` methods. Is there a significant difference in speed?

5.14. Add both the `previous` and `hasPrevious` methods to:
   a. Iterator design #1 (Section 5.4.1)
   b. Iterator design #3 (Section 5.4.3)

5.15. In Java, iterator design #1 (Section 5.4.1) is an `Enumeration`. Assume that only design #2 has been written and that we want to write an iterator that follows the `Enumeration` interface.
   a. What pattern describes the problem we are trying to solve?
   b. Write the class, using the name `Enumeration`, in terms of the iterator defined in design #2. Add a `getEnumeration` method to `MyVector` to do this.

5.16. Redo iterator design #3 (Section 5.4.3) using nested classes.

5.17. This exercise illustrates the idea of programming to an interface. Recall from Section 4.4.1 (and other chapters) that we often implement `operator<<` by calling a class' `print` method.
   c. Write an abstract class `Printable`, with one named method: `print`.
   d. Should `Printable` define any other member functions? Why? (Hint: recall Section 4.2.7.)
   e. Implement an overloaded `operator<<` that takes a `Printable` object as a second parameter.
   f. Suppose `Shape` is a class. Describe what properties `Shape` must have in order to have the overloaded method in part (b) called for all `Shape` objects.
   g. An alternative scheme is to simply make `operator<<` a function template. Compare and contrast the two approaches. Is there any difference?

Programming Projects

5.18. Suppose in addition to a `Cref` class, we want a `Ref` class (that gives a simple reference). Keep in mind that `Ref<Obj>` and `Cref<Obj>` should be type-compatible in one direction. Recall that any function that is expecting a constant reference can receive either a constant reference or a

reference, but not vice-versa. This suggests that there should be an inheritance relationship between `Ref` and `Cref`.

a. Which class is the base class and which is the derived class?

a. One of the two classes will need to overload `get` with both an accessor and mutator version. Which one?

a. Implement the `Ref` and `Cref` classes.

a. Since `Cref<Base>` and `Cref<Derived>` are also type-compatible, if your compiler supports member templates, modify your classes to work when the template types are related by inheritance. See the discussion at the end of Section 5.3.2 (on page 235)

5.19. A reference-counted pointer class keeps track of how many pointers are pointing at the pointee, and does not `delete` the pointee until there are no longer any pointers. To implement the class you will need a second class, `RefCount`.

a. Write a class template called `RefCount`. `RefCount` stores a pointer to the pointee and a count of how many pointers are looking at the pointee. It should provide a method to add to the count, and to decrement the count. If the count goes to zero, it can `delete` the pointee, and then set the pointer to `NULL` (just to be safe). Carefully provide a constructor and if you feel it appropriate, an implementation of the big three. The semantics of these methods may depend on how you implement the rest of this design.

a. Write a class template called `RefPointer`. `RefPointer` stores a pointer to a `RefCount` object. If constructed with a plain pointer, then a new `RefCount` object is created, with count 1. If constructed with a copy constructor, then it shares the `RefCount` object, but adds one to the count in the `RefCount` object. If destructed, it drops the count in the `RefCount` object. You will need to work out the details for what happens in `operator=`. In addition to `get`, `RefPointer` should provide overloaded operators such as `operator*`, `operator->`, and perhaps a one-way conversion.

References

The classic reference on design patterns is [2]. This book describes 23 standard patterns, many of which are inheritance-based. [1] provides C++ implementations for several patterns, and coins the term *functor*.

1. J. O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading, MA, 1992.

2. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.