Chapter 17

Case Study: Container Classes in C++

In this chapter we develop several examples of data-structure classes in C++. Because these data structures are all used to maintain collections of elements, they are often known as *container classes*. Although these classes do have a utility all of their own, our major goal in the exposition will be to illustrate how subclass coupling and inheritance can be used to create general-purpose reusable software components even in the presence of strong typing.

17.1 The Problem of Lost Identity

To be useful, container classes must be reused across many different projects. This means the original programmer developing the container classes will have little or no information as to the type of objects the containers will eventually hold. In a strongly typed language, such as C++, this inevitable results in the container problem we discussed in Chapter 7.

Recall that the container problem is concerned with types. When a value is inserted into a container, under what type is the value stored? Similarly, when a value is removed from a container, how is the original type restored to the value? Remember, when the container class is written it is not known what type of value will be held by the class.

We will consider and contrast three different solutions to this problem:

1. The container maintains elements that must be derived from a common parent class.

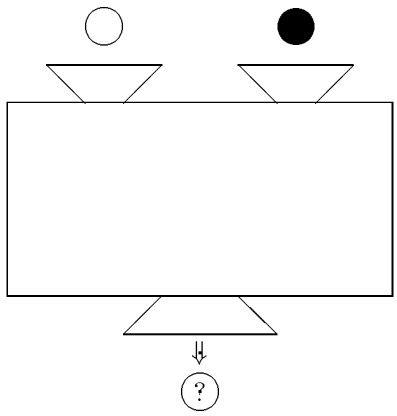

Figure 17.1: Type Information is Lost in a Container

2. The container holds only pointers to the elements.

3. Templates.

In each case, our vehicle for exposition will be a simple linked-list class. Other data structures can be easily defined in a similar manner, but will not be discussed here.

17.2 Inheritance from a Basic Element Type

The first solution is used in many collections that are modelled on the Smalltalk collection hierarchy. The basic idea is to create a class for the link elements, and to create specialized types of lists by using inheritance to subclass from this class.

Each link in our linked list abstraction is responsible for maintaining two data fields; a pointer to the next link and the value associated with the link element. The container problem is concerned only with the second of these two values – certainly the programmer developing the class abstraction knows the data-type associated with the pointer values.

One partial solution to the container problem is thus to define a general purpose "list" class, which makes use of a general purpose "link" class, such as that shown in Figure 17.2. Neither the list nor the links have any specific knowledge of the elements they hold; indeed, they hold no values whatsoever. Nevertheless, insertion of new elements to the front of the list

```
class List {
public:
        // constructor
    List() : firstElement(0) { }

        // operations
    void Add(Link * n)
        { n→next = firstElement; firstElement = n; }

    int isEmpty()
        { return firstElement == 0; }

    Link * First();
        { return firstElement; }

    void RemoveFirst();
        { Link * p = firstElement;
          firstElement = firstElement→ next;
          delete p; }

    int  Includes(Link * n)
        { for (Link * p = firstElement; p; p = p→next)
            if (*p == *n) return 1;
          return 0; }

private:
        // data field
    Link * firstElement;
};

class Link {
public:
        // constructors
    Link () : next(0) { }
    Link (Link * n) : next(n) { }

        // operations
    virtual int operator == (Link *) = 0;

        // data field
    Link * next;
};
```

Figure 17.2: Lists using Inheritance from a List Element

```
class DoubleLink : public Link {
public:
    // data area
    double theValue;

    // constructor
    DoubleLink (double v) : theValue(v) {}

    // operations
    int operator == (Link * d)
        { return thisValue == ((DoubleLink *) d)->theValue; }
};
```

Figure 17.3: Subclassing Links to maintain a specific value

and removal of the first element and other list-like operations can all be defined and implemented. Even a test to see if a specific value is contained in the class can be implemented, using a pure virtual function in class **Link** which must be overridden for each new type of element.

To be useful in any specific situation, the user of the List abstraction must create a new subclass that inherits from the **Link** class. For example, suppose the programmer wishes to create a list of double precision values. The programmer would create a new class, **DoubleLink**, which subclassed the **Link** class and added fields to maintain a double precision value (Figure 17.3).

List operations are then implemented using **DoubleLink** fields. Since a **DoubleLink** inherits from class **Link**, the **List** class is perfectly happy to operate with these types. The following sequence of statements illustrates the declaration of a list type and the insertion of three new elements.[1]

```
List aList;

aList.Add(new DoubleLink(2.71828));
aList.Add(new DoubleLink(1.61803));
aList.Add(new DoubleLink(3.14159));
```

Unfortunately, when a value is *removed* from a list, the type yielded by the **First** function is simply a **Link**. Not to worry, since the programmer

[1] As we will note in the final section, one of the major issues glossed over in our discussion concerns memory management. In this case, we simply assume all memory allocation calls will be successfully honored.

```
class DoubleList : public List {
public:
   void Add(double v)
      { List::add(new DoubleLink(v)); }

   double First()
      { return firstElement→value; }

   int Includes(double v)
      { DoubleLink v1(v);
        return List::includes(v1); }
};
```

Figure 17.4: Subclassing Lists and well as Links

knows that only `DoubleLink` values have been inserted into the list, a cast can be applied to convert the result back into a `DoubleLink`.

```
DoubleLink * pi;

pi = (DoubleLink *) aList.First();
```

17.2.1 Hiding the Links

A significant problem with the linked list class we have developed is the constant necessity to manipulate instances of `DoubleLink`, instead of `Link`'s or `double`'s. This manifests itself in several ways. For example, the programmer must explicitly create and delete `DoubleLink` values. The necessity of using cast's is a troubling sign – cast's are a notorious source of programming errors if abused or improperly used.

One approach to solving both of these problems is to subclass a new class from the `List` class. Methods that need to know the element type for each value, such as addition and returning the first element, can be appropriately redefined (Figure 17.4). Casts can be safely hidden within this class, and need not appear in code that uses the data abstraction. Methods that do not care about the particulars of the data types being held, such as `IsEmpty` and `RemoveFirst`, are simply inherited unchanged.

With this change, the user of DoubleList class can then manipulate values without regard for links of any sort:

```
DoubleList aList;

aList.Add(2.71828);
aList.Add(1.61803);
aList.Add(3.14159);

double pi;

pi = aList.First();
aList.RemoveFirst();
```

Besides requiring the programmer to create two new subclasses in order to obtain type-safe collections, a slightly annoying aspect of this solution is that the equality testing operator in the sublcass `DoubleLink` must still declare the argument holding the test value as type `Link`. This must then be cast into a `DoubleLink` value before any meaningful comparison can be performed. We will return to this question and see how it is solved when we discuss the template solution to the container class problem.

17.3 Containers that hold only Pointers

Because only pointer or reference values can be used to create truly polymorphic variables in C++, in many applications collections naturally maintain pointer values. If we assume that a linked list will always be used to simply maintain pointer values, then a slightly simpler solution can be developed which only requires one class to be subclassed.

Instead of assuming the `Link` class will be subclassed in order to hold values, the `Link` class now maintains a pointer value, in addition to the pointer to the next link it already contained. Since the pointer value must of necessity be generic (not specific to any application), it is declared as a pointer to `void`. Any pointer value can be assigned to such a variable.

Once again, in order to be used in a type-safe manner the class `List` must be subclassed. This time, however, only the `List` class, and not the `Link` class, must be changed. Suppose, for example, that a list is maintaining a collection of pointers to a class named `Window`. A type-safe subclass can be created as show in Figure 17.6. Only the two methods that add and remove a value to and from a list need be changed, and these only to invoke the methods in class `List`.

In addition, our new class now satisfies the first rule of memory management; namely, the person responsible for making a mess (allocating dynamic storage) should be the person responsible for cleaning it up. In the first

```
class List {
public:
    ... // same as before

    void Add(void * v)
        { firstElement = new Link(v, firstElement); }

    void * First()
        { return firstElement→value; }

    ... // same as before
};

class Link {
        // constructors
    Link (void * v) : next(0), value(v) {}
    Link (void * v, Link * n) : next(n), value(v) {}

        // data areas
    Link * next;
    void * value;
};
```

Figure 17.5: Link Values for maintaining Pointers

```
class WindowList : public List {
public:
    void Add(Window * w)
        { List::add(w); }

    Window * First()
        { return (Window *) List::first(); }
};
```

Figure 17.6: A Type-Safe collection of Windows

list class we presented the programmer entering a new element into a class was responsible for creating a new link, but the list class itself deleted it. In the new structure both tasks are performed by the list class.

17.4 Templates

The third alternative to the container class problem is the best solution of all. It involves the use of a relatively new feature in C++, the idea of a *template*. A template parameterizes a class description, in much the same way that a function can take parameters. In the case of templates, however, the parameter is a *type*, and not a value. The template argument can be considered to be a legitimate type name throughout the template description. To create a specific instance of such a class, the programmer must supply a value to the template argument, much in the same way that arguments must be supplied prior to executing a function. The arguments to the template, however, must be a type and are written in angle brackets following the class name. A template description of our `List` class is shown in Figure 17.7. The following declares a list of double precision numbers, adds several values to the list and then removes one.

```
List<double> aList;

aList.Add(2.71828);
aList.Add(1.61803);
aList.Add(3.14159);

double pi;

pi = aList.First();
aList.removeFirst();
```

The obvious advantage of the template solution is that it does not require the programmer who uses the list abstraction to create any new subclasses. Multiple lists of many different types can be quickly created and manipulated. The major disadvantages of templates stem from their status as a relative recent addition to the C++ language. Not all compilers will support this feature. Often, the implementation techniques used to support templates force some odd quirks in their use. For example, one major compiler only permits template declarations in a global scope, not as local variables. Another compiler does not permit methods in template classes to be declared as inline functions. Ordering issues can also cause

```
template <class T>
class List {
        // constructors
    List();

        // operations
    void Add(T v)
        { firstElement = new Link<T>(v, firstElement); }

    int isEmpty()
        { return firstElement == 0; }

    T First()
        { return firstElement→value; }

    void RemoveFirst()
        { Link<T> * p = firstElement;
          firstElement = p→next;
          delete p; }

    int Includes(T v)
        { for (Link<T> * p = firstElement; p; p = p→next)
            if (v == p→value)
                return 1;
          return 0; }
};

template <class T>
class Link {
    // constructor
    Link (T v, Link * n) : value(v), next(n) { }

    // data fields
    Link * next;
    T value;
};
```

Figure 17.7: A Template version of a List class

confusion if mutually recursive templates are defined. Finally, many template implementations generate an internal class description for each type of template argument. Thus, code size expansion can occur if a class is instantiated several times with different types of arguments. It is likely that many of these problems will be reduced as the implementation techniques for templates become better understood.

17.5 Issues not considered

Our purpose in this chapter has been to explore how strong typing in C++ impinges upon the problem of creating reusable container classes. Because we have been concerned only with the typing issue, there are a number of related topics we have not dealt with. The following are the most significant of these issues:

- **Iteration**. We have not considered the problem of how to iterate over an entire list of elements without destroying the list in the process. Recall that the programmer using the list abstraction was assumed to be different from the programmer who originally developed the list classes. Presumably the user programmer need not be privy to unimportant implementation details of the class, and thus, such details should not be required in order to form loops. The most common solution in this case is for the developer of the list class to form a new class, called an `iterator`, which is designed specifically to allow looping.

- **Assignment and Identity**. When a value is placed into a collection, is the value an independent copy of the original, or a reference (alais) to the original? In the case of an alais, what happens if the original is then deleted? Similar questions can also be raised concerning how values are compared for equality, although the programmer has slightly more control over this issue since the equality testing operator can be easily redefined to give whatever meaning is desired.

- **Memory Management**. Collection classes of arbitrary size, such as our linked list classes, must almost of necessity use dynamic memory management. Should be management of memory for the values themselves be the responsibility of the collection class itself, the user of the class, or a combination of both? In part, this is tied to the answer to the preceding question, whether objects held in collections are copies or references. Techniques such as reference counting are often used to solve complex memory management issues.