Chapter 15

Container Classes

Simple data structures are found at the heart of almost all nontrivial computer programs. Example data structures include vectors, linked lists, stacks, queues, binary trees, sets, and dictionaries. Because data structures are so common, one would expect them to be ideal for development as reusable components. Indeed, it *is* possible to create such components, but there are subtle issues involved that can trap the unwary programmer.

An exploration of the problems in developing reusable container classes is for this reason a good illustration of how the features of a programming language influence the style of development, as well as a demonstration of some of the powers and some of the limitations of object-oriented techniques.

15.1 Containers in Dynamically Typed Languages

Producing reusable container abstractions is considerably easier in a dynamically typed language, such as Smalltalk, Clos or Objective-C, than they are in a statically typed language. Indeed, dynamically typed languages usually come with a large collection of data abstractions already developed, thus freeing the programmer from having to address the container problem.

As we saw in our earlier discussion on binding times, in a dynamically typed language it is a value itself that retains knowledge of its type, not the variable by which it is accessed. Thus any object can be placed into a container, and when it is removed it can be assigned to any variable. The following, for example, shows an integer and a string being placed into a Smalltalk array, and later removed:

```
anArray <- Array new: 2.
anArray at: 1 put: 'abc'.
anArray at: 2 put: 12.3.
theString <- anArray at: 1.
theNumber <- anArray at: 2.
theString <- theString characterAt: 2.
```

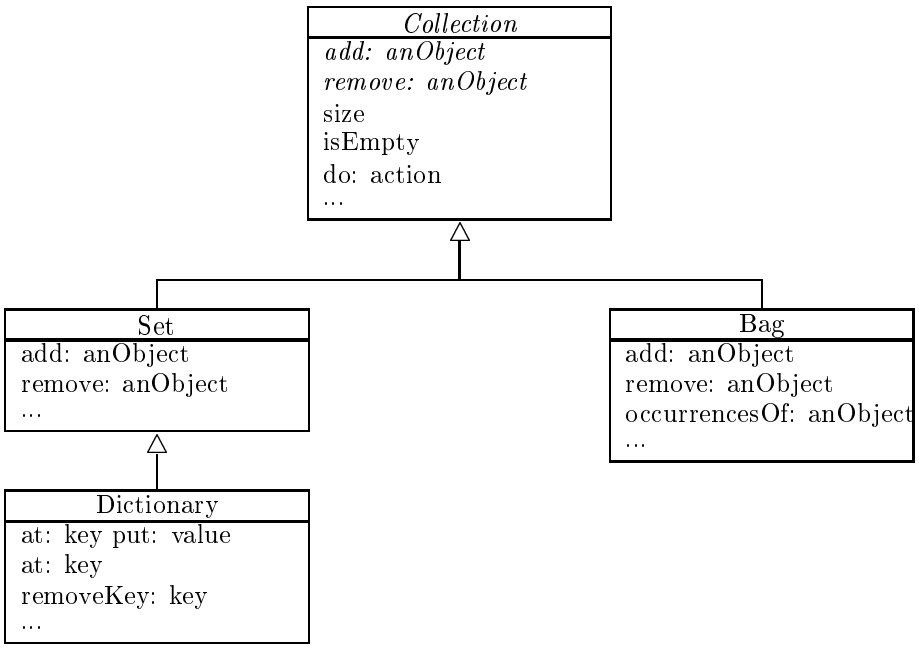


Figure 15.1 A Portion of the Collection Inheritance Hierarchy in Smalltalk-80

```
theNumber <- theNumber * 7.
```

Notice how a dynamic language permits values of different classes to be held in the same container. Such a collection is sometimes termed a *hetrogeneous* collection, as opposed to a *homogeneous* collection where values are all of the same type. Statically typed languages have difficulty forming truly hetrogeneous collections, although they can approximate them through the use of the principle of substitution.

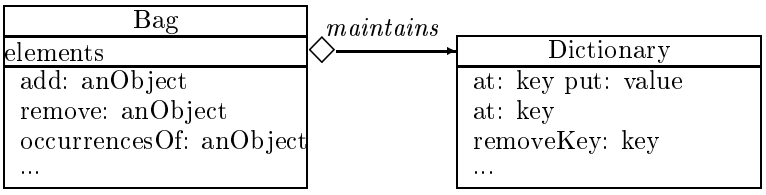
15.1.1 Containers in Smalltalk-80

The software reuse techniques of inheritance and composition can be used to advantage in the creation of collection classes. To illustrate, consider the Smalltalk-80 classes Collection, Set, Bag and Dictionary. These four classes are linked in the inheritance hierarchy shown in Figure 15.1.

The parent class Collection is shown as abstract, since some of the methods (those shown in italics) must be redefined in child classes. The subclass Set represents an unordered collection of unique elements. It redefines the abstract methods for addition and removal that were inherited from Collection, as well as

adding new functionality (not shown).

The class Dictionary represents a collection of key/value pairs. Elements can be inserted into the dictionary using a specific key, and the user can search for the element associated with a key. The dictionary is implemented as a Set where each element of the set is an Association representing a single key/value pair. By defining Dictionary as a subclass of Set, the methods that do not deal with the keys directly can be inherited without change from the parent. (Using the categories from Chapter ?? this is subclassing for construction).



A third type of collection is a Bag. Conceptually, a Bag is similar to a Set, only values are allowed to be entered into the collection more than once. To implement this behavior, the class Bag uses composition, maintaining an internal value of type Dictionary. The keys in the dictionary represent the elements inserted into the Bag, whereas the value associated with the key is the number of times the item appears in the Bag. In this fashion only one instance of each element is actually stored in the container, but as items are inserted and removed the counts are updated appropriately.

15.2 Containers in Statically-Typed Languages

It is clear from the countless data structure textbooks that have appeared over the years that container abstractions can be written in almost any language, including statically-typed languages. The problem with statically typed languages is not that they preclude the development of container classes, but that static typing interfeers with software reuse. That is, it is difficult to write a container class in such a way that it can be easily carried from one project to the next and still retain the benefits of static typing.

In the following section we will first describe in detail the origin of this tension between static typing and software reuse. After considering the problem, we will then explore how object-oriented software techniques have been used to overcome this difficulty. In particular, we will consider three different solutions:

- Using the principle of substitution to store values in a container, and combine with downcasting (reverse polymorphism) when values are removed.
- Again using the principle of substitution, but avoiding downcasting through the use overriding.
- Using generics or templates.

15.2.1 The Tension between Typing and Reuse

To place the problem in perspective, we must first consider how data structures are typically implemented in a conventional language, such as C or Pascal. We will use a linked list of integers as our example abstraction. In Pascal a linked list might be formed out of two types of records. The first is the list header itself, which maintains a pointer to the first link:

```
type
    List = Record
            firstLink : ↑ Link;
            end;
```

A list header can be statically allocated, as the amount of storage it maintains (namely, one pointer) remains fixed throughout execution. The second record is used to maintain the actual values themselves. Each Link node maintains one integer value and a pointer to the next link:

```
type
    Link = Record
            value : integer;
            nextElement : ↑ Link;
        end;
```

Link nodes must be dynamically allocated and released, although such details can be largely hidden from the user of the list abstraction through the development of functions, such as a function to add a new value to the front of the list, return and remove the first element in a list, and so on.

```
procedure addToList (var aList : List, newVal : integer);
var         (* add a new value to a list *)
    newLink : ↑ Link;
begin
        (* create and initialize a new link *)
    new (newLink);
    newLink↑.value := newVal;
        (* place it at the front of the list *)
    newLink↑.nextElement := aList.firstLink;
    aList.firstLink = newLink;
end;

function firstElement (var aList : List) : integer;
var         (* remove and return first element from a list *)
    firstNode : ↑ Link;
begin
```

```
    firstNode := aList.firstLink;
    firstElement := firstNode↑.value;
    aList.firstLink := firstNode↑.nextElement;
    dispose (firstNode);
end;
```

Our concern here is not with the details of how a linked list might be implemented (such details can be found in any data structure textbook) but with the question of reusability. Suppose our programmer has implemented the linked-list abstraction given above and now wishes to maintain, in addition to a linked list of integers, a linked list of real numbers.

The problem is that the programming language is *too* strongly typed. The data type integer used for the value being held by the link is an intrinsic part of the definition. The only way it can be replaced by a different type is through the creation of a totally new data type, for example RealLink, as well as a totally new list header, RealList, and totally new routines for accessing and manipulating the data structures (addToRealList and firstElementInRealList, for example).

Now, it is true that something like a variant record (called a *union* in C) could be used to permit a single list abstraction to hold both integers and real numbers. Indeed, a variant record would permit one to define a heterogeneous list that contains both integers *and* real numbers. But variant records solve only part of the problem. It is not possible to define a function that returns a variant record, for example, so one still needs to write separate functions for returning the first element in a list. Furthermore, a variant record can have only a finite number of possible alternatives. What happens when the next project requires a totally new type of list, such as a list of characters?

In short, a language that is too strongly typed does not provide the facilities necessary to create and manipulate truly reusable container abstractions. The question then is, do the additional facilities provided by object-oriented languages yield any new way to overcome this problem? The principle new tool found in an object-oriented language that is not found in a conventional language is the principle of substitution. And indeed, the principle of substitution can be used in at least two different ways to overcome the problem of overly strong typing.

15.2.2 Substitution and Downcasting

It is tempting to think that substitutability by itself can solve the container class problem for statically typed languages. Recall from Chapter 6 that the principle of substitution claims that a variable declared as maintaining some object type can, in fact, be assigned a value derived from a subclass of the variable's declared class.

The principle of substitution is most valuable in languages that have a single class at the root of the inheritance hierarchy. Recall that this was true for both Java (the root class is Object) and Delphi (the root class is TObject). So we

see that in both of these languages containers are provided that store elements in variables declared as the root class. A Java Vector, for example, stores its elements in an array of Object values.

While this purposeful supression of typing information solves one problem, it comes only at the cost of introducing another. As we have noted, any value can be assigned to a variable of type Object (in Java) or TObject (in Delphi). But when values are removed from the container the programmer typically wants them restored to their original type. Since the removal method can only declare its result as an Object, a casting expression must be used to restore the original data type, as in the following code fragment in Java:

```
Vector aVector = new Vector();
Cat felice = new Cat();
aVector.addElement(felice);
    ...
    // cast used to convert Object value to Cat
Cat animal = (Cat) aVector.elementAt(0);
```

A problem with this approach is the detection of typing errors. Suppose a programmer creates a container that they think will maintain values of a certain type, for example class Cat. By accident a value of the wrong type, for example a Dog, is placed into the container. The error cannot be discovered by any static compile time analysis of the program. Worse, the resulting run-time error will not be discovered at the point of insertion (which is where the logic error is being committed) but at the point of removal, when the attempt to perform the downcast will result in an casting exception being thrown:

```
    // make a collection of Cat values
Vector catCollection = new Vector();
Cat aCat = new Cat();
Dog aDog = new  Dog();
catCollection.addElement(aCat); // no problem
    // although the following incorrectly inserts
    // a value of type Dog into the collection,
    // no compiler error will ensue
catCollection.addElement(aDog);
    ...
    // it is only here, when the element is removed and an
    // attempt is made to convert to type Cat, that
    // a run-time error is detected.
Cat newCat = (Cat) catCollection.elementAt(1);
```

Heterogenous Collections

Because Java collections store their values in variables declared as type Object, in principle it is easy to create heterogenous collections. But in practice the problem is not placing the values into the collections, but taking them back out again. As we have noted, normally a value must be down cast to a more specific type after it is removed from the container. In a hetergeneous collection this type must first be tested before it can be cast, as shown in the following example:

```
    // make a stack that contains both cats and dogs
Stack stk = new Stack();
stk.addElement(new Cat());
stk.addElement(new Dog());
    // ... adding more values
    // now do something with Cat values
if (stk.peep() instanceof Cat) {
        // do conversion to Cat
    Cat aCat = (Cat) stk.pop();
    // .. also do something with cat values
    // now do something with Dog values
} else if (stk.peep() instanceof Dog) {
        // do conversion to Dog
    Dog aDog = (Dog) stk.pop();
    // .. do something with Dog value
}
```

Container Classes in Delphi

A popular container collection in Delphi, the Spider classes destributed by Interval Software, has an interesting solution to the error detection problem. Although values are still stored internally in variables of type TObject, a class value can be given as argument to the constructor when the collection is created. As each element is inserted, the class value is used to ensure the element matches the desired type. If it is not, an error exception is raised. Since this error occurs at the point of insertion, not at the point of removal, it makes the discovery of logic errors much easier:

```
var
    stack : Tstack;
    aCat : TCat;
    aDog : TDog;

begin
        // create a stack that can hold only TCat values
    stack := TStack.Create (TCat);
```

```
    stack.push (aCat); // ok
    stack.push (aDog); // will raise exception
    ...
end
```

Heterogeneous collections can be accomodated by using a more general class value. For example, if both cats and dogs must be held in the same list, the collection can be created using the class value TMammal (or TAnimal).

Although a check was performed as the value was inserted into the container, it is still necessary to cast the value back to the correct type when it is accessed or removed, as the declared results of these operations is only TObject. Delphi provides two different ways to peform this operation. The as operator performs a check to ensure that the conversion is valid:

```
    aCat := stack.Pop as TCat;
```

The alternative syntax uses the name of the child class as if it were a function call. This form, however, does not check the veracity of the cast, and so should only be used when you are abolutely certain no errors can occur:

```
    aCat := TCat(stack.Pop);
```

Storing Non-Object Data in Containers

Java containers can store any value that is ultimately derived from class Object. The Spider container classes in Delphi can store any value that is ultimately derived from TObject. Unforunately, in both of these languages the primitive values, such as integers and floating point numbers, are not objects in the technical sense. Thus, primitive values cannot be stored directly in a container in these languages.

In both cases the solution is to provide a series of auxillary classes that do little more than act as a box that can hold a single primitive value. In Java these are called Wrapper classes, while in the Spider Delphi containers these are called Bucket classes. The following illustrates how a double precision number can be stored and later removed from a Java Vector:

```
Vector aVector = new Vector();
    // create a wrapper to hold a real number
aVector add: (new Double(12.34));
    // ...
    // later we first find the Double object
Double dwrap = (Double) aVector.elementAt(0);
    // then unwrap to get original value
double dval = dwrap.doubleValue();
```

	Java	**Delphi**
boolean	Boolean	TBooleanBucket
byte	Byte	TByteBucket
char	Character	TCharBucket
double	Double	TRealBucket
int	Integer	TIntegerBucket
long integer	Long	TLongIntBucket
short integer	Short	TShortIntBucket

Table 15.1 Auxillary classes used to store primitive types

Table 15.1 gives the wrapper classes for Java and for the Spider data structure classes in Delphi associated with the more common primitive types.

15.2.3 Using Substitution and Overriding

A cast expression is often considered to be not truly object-oriented, since it requires the programmer to name an explicit type in the code. In many situations explicit casts can be avoided through the use of substitution combined with method overriding. However, in the case of container classes this is possible only when the original developer knows how an object will be used, even if they do not know what type of value will be stored in the container. Thus, this technique is applicable only in a few restricted situations.

One example is found in the code in Java used to respond to user initiated events, such as mouse presses. In Java events are handled by creating a *listener* object and attaching it to a window. When an event occurs in the given window, all the registered listeners are notified of the event. A listener must match a fixed specification. There are a number of different types of specifications, corresponding to the variety of events that can occur:

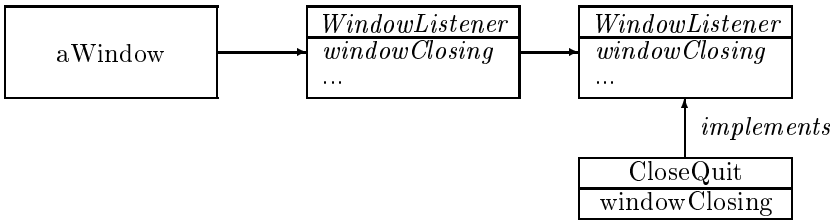
ActionListener	change in graphical component state
ItemListener	changes to selected item component
KeyListener	key press events
MouseListener	mouse presses and releases
MouseMotionListener	mouse motions
TextListener	text component changes
WindowListener	window actions

Because many listeners are used for a large number of different actions, the Java library also provides a collection of *adapters* that implement the interface, and define an empty action for each possibility. To create a listener the Java programmer defines a class that implements this interface, and overrides key

methods. An example is the following, which subclasses from the WindowAdapter class (which in turn implements the WindowListener interface) and overrides the method windowClosing.

```
public class CloseQuit extends WindowAdapter {
        // execute when the user clicks in the close box
    public void windowClosing (WindowEvent e) {
        System.exit(0); // halt the program
    }
}
```

All the listeners attached to a window are stored in a linked list. The Window class maintains the view that these values are all instances of WindowListener (or one of the other listener hierarhcies). In reality, they are instances of user-defined classes that implement the WindowListener interface and are only stored on the list through the principle of substitution. When an event occurs, the Window passes a message to each listener, "thinking" that it is an instance of WindowListener. But the method is overridden, and the message is actually handled by the user defined class.



Notice how this achives the desired effect without the need to explictly cast the listener value to a new type. On the negative side, this technique is only applicable when the programmer has precise information concerning how a value stored in the container will be used, even if they do not know the type for the value.

15.2.4 Parameterized Classes

The previous two solutions to the container abstraction problem both employed the principle of substitution. However, this technique is only suitable if there is a parent class that can be used as the basis for the substitution. If a language has a single root as the ultimate ancestor of all classes, as does Java, then that is the logical candidate for the parent type. But what about a language such as C++, where there is no single root class?

The language C++ gets around this difficulty by introducting a new language feature, which in turn permits an entirely different solution to the container class problem. This new feature is the ability to define classes that are *parameterized*

by type arguments. Such classes are called *templates* in C++, and *generics* in some other languages. (Generics are also found in the object-oriented language Eiffel, and in some non-object-oriented langauges, such as Ada).

A class template gives the programmer the ability to define a data type in which some type information is purposely left unspecified, to be filled in at a later time. One way to think of this is that the class definition has been parameterized in a manner similar to a procedure or function. Just as several different calls on the same function can all pass different argument values through the parameter list, different instantiations of a parameterized class can fill in the type information in different ways.

A parameterized class definition for a linked list abstraction might be written in C++ in the following way:

```
template<class T> class List {
public:
    void    addElement (T newValue);
    T    firstElement ();
    ListIterator<T> iterator();
private:
    Link<T> * firstLink;
};

template<class T> class Link {
public:
    T    value;
    Link *    nextLink;

    Link (T, Link *);
};
```

Within the class template, the template argument (T, in this case) can be used as a type name. Thus, one can declare variables of type T, have functions return values of type T, and so on.

Member functions that define template operations must also be declared as template:

```
template<class T>
void List<T>::addElement (T newValue)
{
    firstLink = new Link<T> (newValue, firstLink);
}

template<class T>
T List<T>::firstElement ()
{
```

```
    Link * first = firstLink;
    T result = first->value;
    firstLink = first->nextLink;
    delete first;
    return result;
}

template<class T>
Link<T>::Link(T v, Link * n) : value(v), nextLink(n)
{ }
```

The user creates different types of lists by filling in the parameterized type values with specific types. For example, the following creates a list of integer values as well as a list of real numbers.

```
List<int>        integerList;
List<double>    doubleList;
```

In this fashion, homogeneous lists of any type can be created.

A template is an elegant solution to the container class problem. It allows truly reusable, general-purpose components to be created and manipulated with a minimum of difficulty and yet still retain the type safety, which is the goal of statically typed languages. On the other hand, there are drawbacks to the use of templates. They do not permit the definition of heterogeneous lists, as all elements must match the declared type. More importantly, implementations of the template mechanism vary greatly in their ease of use and the quality of code they generate. Most implementations act as little more than sophisticated macros, generating for each new type of element an entirely new class definition as well as entirely new method bodies. Needless to say, if several different element types are used in the same program, this can result in a considerable growth in code size.

Nevertheless, because templates free the programmer from so much conceptual drudgery (namely, rewriting data structure classes in every new program), their appeal is widespread. In the next chapter we will examine one such library.

Bounded Genericity

Templates as they are implemented in C++ do not place any explicit restriction on the template argument values, instead type restrictions are defined implicitly by the method body. This is illustrated by the following example:

```
template <class A, class B>
int countAll (A value, B collection)
{
    int count = 0;
```

```
    A element = B.firstValue();
    while (element != null) {
        if (value.equals(element))
            count++;
        element = B.nextValue();
    }
}
```

A careful examination of the body of the function will reveal that instances of the class A need to understand the method equals, while instances of the class B need to implement the methods firstValue and nextValue. However, it is only the statements in the code, and nothing in the function header that indicates this fact.

Other programming langauges that support genericity, such as the programming language Eiffel, allow the programmer to place restrictions on the type parameters, in much the same way that value parameters can be typed. For example, in Eiffel a hash table might be described as follows:

```
class
    HASH_TABLE [ H -> HASHABLE ]
...
```

The arrow indicates that the argument can only be filled with a subtype of HASHABLE (that is, a class that inherits from HASHABLE if it is a class, or implements the HASHABLE interface if it is an interface.) Bounding the type arguments allows for slightly better type checking, as the legality of argument values can be determined at compile time.

15.3 Restricting Element Types

Container classes can be divided into three major groups that are differentiated by the requirements they place on their element types. The simplest are containers such as linked lists or vectors. These require only that elements have the ability to be compared against other elements for equality. Slightly more complicated are the ordered containers, such as binary search trees or sorted lists. These require that elements have the ability to be compared against other elements for ordering. A third category of container are hash tables. These require that every element have the ability to determine an integer value, called the hash of the element.

Once again we have the situation where there is a simple interface (the relational test or the hash function) and a wide range of implementations (the technique used to determine a hash value for a character, for example, will be very different from that used to compute the hash value for a complex number). Languages and libraries exhibit a wide range of solutions to this problem.

In languages, such as Smalltalk or Java, that have a single root class at the top of the inheritance hierarchy, it is common for operations to have a default implementation in the root class, and allow for this default implementation to be overridden in child classes. Thus in Smalltalk, for example, the class Object contains the methods == and hash. In Java the corresponding methods are equals and hashValue. Since these methods are defined in Object they can be applied to every object value. Since they can be overridden, classes can supply their own specialized meaning.

Nevertheless, it is useful to allow the programmer to supply their own comparison algorithm for sorting elements in an ordered container. In Smalltalk this is accomplished by passing a block to the instance creation method:

```
aCollection <- SortedCollection sortBlock: [ :a :b | a <= b ]
```

In Java, the progrmmer can specify ordering by defining a class that implements the Comparator interface:

```
public interface Comparator {
    public int compare (Object left, Object right);
}
```

The method compare returns the integer -1 if the left argument is smaller than the right, 0 if they are equal, and 1 if the left is larger than the right. The user must create a class that implements this interface. The following, for example, is a comparator that will test two instances of the wrapper class Double. Note how the arguments are declared as Object, and must be down cast to the appropriate type before the actual comparison can be performed.

```
public class DoubleCompare implements Comparator {
    public int compare (Object left, Object right) {
            // first down cast the arguments
        Double dleft = (Double) left;
        Double dright = (Double) right;
            // then do the comparison
        if (dleft.doubleValue() == dright.doubleValue())
            return 0;
        if (dleft.doubleValue() < dright.doubleValue())
            return -1;
        return 1;
    }
}
```

A comparator object is then passed to the constructor when an ordered collection is created:

```
    // create a new ordered collection
SortedSet aCollection = new TreeSet(new DoubleCompare());
```

The Delphi Spider classes use a similar technique.

In the previous section we saw how template container classes also restrict the type of values they can handle. Unbounded template classes, such as those found in C++, define implicitly the requirements for element types. This implicit requirement derives from the functions used in the body of the methods for the container. Bounded generics, such as are found in Eiffel, explicitly place restrictions on the types of elements that containers can hold.

Some developers of data structure classes perfer to place responsibility for comparisons and hash values in the objects themselves, rather than in the container. This is made more difficult if there is no single root class or if, as in Delphi, the root class does not provide all the necessary functionality. A developer of data structure classes in Delphi, for example, might insist that to be held elements must implement an interface such as the following:

```
type
    TContainable = interface
        public
            function compareTo (const right : TContainable) : integer;
            function hashValue : integer;
    end;
```

This is in some respects a combination of the techniques described in Sections 15.2.2 and 15.2.3. The container itself can invoke the methods compareTo and hashValue without needing to execute a cast. However, the user must still cast values to their correct type when they are accessed or removed from the container.

15.4 Iteration

Regardless of whether a language is statically typed or dynamically typed, another difficult problem that must be handled in order to create truly useful container abstractions is the task of iteration. The problem of iteration is best understood in the context of a multi-person development project. Suppose there are two programmers, named programmer-one and programmer-two. Programmer one must create a data abstraction, for example a set implemented using a red-black tree, and programmer two is going to use the abstraction. Programmer two need only know the interface in order to add elements to the container and remove elements from the container. But now imagine that programmer two wants to write a loop that will iterate over the elements of the container. How can programmer two perform this task without any explicit knowledge of the internal structure of the container class?

We can see the problem in concerete terms by again considering the Pascal linked list data type we introduced earlier in Section 15.2.1. A typical loop that prints the values in a list might be written as follows:

```
var
    aList : List;    (* the list being manipulated *)
    p : Link;    (* a pointer for the loop *)

begin
    ...
    p := aList.firstLink;
    while (p <> nil) do begin
        writeln (p.value);
        p := p^.nextElement;
    end;
```

Note that to create a loop it was necessary to introduce an extraneous variable, here named p. Furthermore, this variable had to be of type Link, a data type we were taking pains to hide, and the loop itself required access to the link fields in the list, which we were also attempting to hide.

Once again we can ask whether the new mechanisms provided by object-oriented languages permit a solution to this problem that was not available in more conventional languages. And once again, the answer is yes. There are two solutions we will examine.

- An *iterator* makes use of the property that in an object-oriented language it is possible to have many different implementations for the same interface. An iterator is an object that implements an interface designed specifically for forming a loop.
- A *visitor* is an alternative approach that is possible when the programmer language provides an easy way to encapsulate a series of actions and hand them to the container.

15.4.1 Iterator loops

The concept of an iterator relies on the ability to have many different implementations match the same interface. The iterator interface is designed to be easy to remember, and flexible enough to work with a wide variety of containers. In Java, for example, the iterator interface (called an Enumeration) consists of just two methods. The method hasMoreElements returns true if the loop should continue, and the method nextElement yields the next element in the sequence. A typical loop looks like the following:

```
    // create the iterator object
Enumeration e = aList.elements();
```

```
    // then do the loop
while (e.hasMoreElements()) {
    Object obj = e.nextElement();
    // ... do something with obj
}
```

Every container class in the Java library implements a method named elements, which returns a value that matches the specification defined by the class Enumeration. In fact, however, the actual value returned will differ from one collection to another, as each different type of collection requires its own set of actions to perform an enumeration. Thus a LinkedList, for example, will return a ListIterator, which is a data type that is derived from Enumeration. Because many different implementations can match the same specification, the loop used to access the elements in a container will look exactly the same, regardless of the type of container being examined.

The language C++ also uses the concept of an iterator. However, iterators in C++ are manipulated in pairs, in much the same fashion as pointers. (This is perhaps to be expected, since pointers are such an important part of the language). The first iterator value specifies the current element, while the second iterator specifies the end of the loop. The interface for iterators includes the following three operators:

operation	purpose	example
==	compare two iterators for equality	start == stop
++	advance iterator to next element	start++
*	return value referenced by iterator	*start

A typical iterator loop looks something like the following:

```
    // create starting and stopping iterators
list<string>::iterator start = aList.begin();
list<string>::iterator stop  = aList.end();
    // then do the loop
for ( ; start != stop; start++ ) {
    string value = *start; // get the value
    // ... do something with the value
}
```

Although the interfaces are different, in both languages the key idea is that each container can provide an implementation of the iterator iterface that is specific to the container. A method in the container class returns a value that is more specialized than its type signature might indicate. The methods begin and end in each of the C++ STL containers returns an iterator appropriate to the container.

It is indeed true that these more specialized classes must have intimate knowledge of the container over which they are looping. An iterator for a linked list, for

example, must know about the link classes that are used in the implementation. But as the iterator classes are written by the same programmer who developed the abstraction, and these internal details are not exposed by the interface, the key principle of information hiding is not being voilated. (Frequently techniques such as friends or inner classes, both of which are discussed in Chapter xx, are necessary in order to link a container and its iterator).

15.4.2 The Visitor Approach

An alternative solution to the problem of iteration is possible if the programming language provides a way to bundle a sequence of actions and hand them to the container, for example in the form of a function. The container can then take the bundle, and execute the actions on each element of the collection in turn.

This technique is used in the language Smalltalk. A Block in Smalltalk is a series of statements enclosed in square brackets, which can optionally begin with a sequence of argument values. In essence, a block is a simple way to create an unnamed function. To iterate over a collection, the programmer uses the method do:, passing as argument a one-argument block containing the action to be performed:

```
aList do: [ :x | ('element is ' + x) print ]
```

The container executes the block repeatedly, passing each element in the collection as argument in turn.

The same idea is also possible in C++, as an alternative to the use of iterators in that language. A *function object* is an object that implements the parenthesis operator, and hence can be used both as an object (for example, it can be stored in a variable) and as a function. For example, a simple function object might just print its argument:

```
class printingObject {
public:
    void operator () (int x)
    {
        cout << "value is " << x << endl;
    }
};
```

The generic function for_each takes a pair of iterators and a function object. It executes the function object on each element specified by the iterator:

```
printingObject printer;  // create an instance of the function object
for_each (aList.begin(), aList.end(), printer);
```

Often the argument will be specified by a nameless temporary, using the

ability in C++ to create a new value by simply naming the class:

```
for_each (aList.begin(), aList.end(), printingObject());
```

In the Spider classes in Delphi looping is performed in a similar fashion, using the method ForEachCallMethod. The following is an example:

```
var aList : LinkedList;

procedure PrintingObject
        (const Obj : TObject, const additionalData : LongInt);
begin
    writeln("Value is ", Obj);
end

begin
    ...
    aList.ForEachCallMethod (TObject, LongInt(0));
end;
```

The second argument can be used to pass additional data from one invocation to the next. Use of this argument frequently eliminates the necessity of introducing global variables.

Premature Termination and Parallel Looping

It is natual to compare the two different approachs to looping (iterators and visitors) and to ask if there are problems that are more easily addressed using one form than with the other. And indeed two common problems can be identified that are both more easily addressed using the iterator approach than using the visitor technique.

The first situation arizes when it is desirable to halt a loop before it has enumerated the entire range of values. This might occur, for example, if one wanted to find the first element in a collection that satisfied a given condition. Such a loop is easy to write using an enumerator and the ability to break a loop before it has completed:

```
Enumeration e = aList.elements();
while (e.hasMoreElements()) {
    Object obj = e.nextElement();
    if (... obj satifies condition ...)
        break; // break out of loop
}
```

None of the visitor mechanisms allow the user to halt an iteration prema-

turely, although the C++ STL library does provide a specialized form of visitor designed for just this type of search (see following section).

The second common situation in which iterators seem to have an edge over visitors occurrs when it is necessary to iterate over two collections in parellel, operating on them element by element. This is easily accomplished by simply combining the ending conditions for two iterators:

```
Enumeration e = listOne.elements();
Enumeration f = listTwo.elements();
while (e.hasMoreElements() && f.hasMoreElements()) {
    Object objOne = e.nextElement();
    Object objTwo = f.nextElement();
    // ... operate on objOne and objTwo
}
```

The equivalent action cannot be achived using visitors without writing a special-purpose parallel visitor routine.

Other Loop-like Activities

Languages that use the visitor mechanism, such as Smalltalk and C++, frequently extend the model to provide other functionality that is based on looping. For example, in Smalltalk it is simple to create a computation in which every element is operated on in turn to produce a single final result. An example of such a computation might be a summation of the elements of the collection. To form this expression, the base element (the identity, such as zero for a summation) is combined with a two-argument block that defines the computation used to generate the intermediate values:

```
sum <- aList inject: 0 into: [:x :y | x + y ].
```

Each element if the collection (here, a list) is considered in turn. The block is evaluated using the current result (initially, the identity argument) and the collection element. The final result will be the value yielded by the block after the last element is considered.

Another example, this time from C++, is the generic function find_if. Just as with for_each, this function takes as argument a pair of iterators and a function object. This time, however, the function object must return a boolean (true/false) value. Each element of the collection is tested in turn. When the first element for which the function object returns true is encountered the function will halt and the corresponding iterator will be returned. In this way the first element that satisfies a property can be found. If no element satisfying the property is found the ending iterator is returned.

```
class BiggerThan12 {
```

```
// function object that finds a value larger than 12
public:
    bool operator () (int x)
    {
        return x > 12;
    }
}

list<int>::iterator start = aList.begin();
list<int>::iterator stop = aList.end();
start = find_if (start, stop, BiggerThan12());
if (start != stop) // found it
    ...
```

Summary

The development of reusable container abstractions illustrates both the power and the limitation of object-oriented techniques. Container classes are relatively easy to define in dynamically typed languages, but as is true of many other features of such languages the dynamic typing hinders the detection of typing errors at compile time. Statically typed languages have better static error detection abilities, but the static typing interfers with the development of reusable abstractions.

	advantages	**disadvantages**
dynamically typed language	easy to define reusable classes	poor static error detection
statically typed language	good static error detection	strong typing complicates developing reusable abstractions

One way to resolve the conflict between static typing and reusability is to use the principle of substitution. In this chapter we have examined two different approaches that both make use of this mechanism. The first stores values in variables of type Object (or TObject in Delphi) which is the root of the inheritance hierarhcy. By the principle of substitution any object value can be stored in such a variable, but must be downcast to the correct type when it is accessed or removed from the container. The second approach stores elements in a specific class type, and uses substitution combined with method overriding to specialize the behavior.

	advantages	**disadvantages**
Substitution and down casting	Works for most objects	Specific types required in cast expressions
Substitution and method overriding	No cast expressions	Only works with methods known in advance

There are negatives to both approaches. Cast expressions, required for down-casting, require putting explicit types into code. Having to name explicit types is often considered to violate the spirit of the object-oriented philosophy. On the other hand, using method overriding is only possible if the developer of the container abstraction can predict ahead of time how the objects stored in the container will be used.

An alternative approach that does not use the principle of substitution is the mechanism of template, or generic classes. A template can be thought of as a type parameter. Using templates the developer of a container abstraction need not know the type of elements that will be stored in the container. The final element types must then be specified by the user of the containers. Conceptually templates provide an elegant solution to the container class problem, however in practice the implementation of the template mechanism tends to be exceedingly complex, and the error messages that result from incorrect usage are often cryptic and misleading.

	advantages	**disadvantages**
Template (or generic)	works with all data types	implementation is complex error messages often cryptic

An entirely different problem that must be addressed in the creation of reusable container abstractions is the issue of iteration. How can the developer of a container class allow users to form a loop that will iterate over the elements in the container without exposing the inner implementation details for the container.

In the object-oriented languages we are considering there are two broad categories of solution to this problem. The first is to form iterators. An iterator is a specialized object whose sole purpose is to provide a means of forming a loop. Using the fact that many different implementations can be provided for the same interface, containers can each define a specialized iterator that implements a common interface in a unique way. The same type of loop can then be written for any container.

An alternative to an iterator is a visitor. The visitor mechanism bundles the actions to be performed, and passes them to the container, which in turn executes the actions using each element as argument in turn. The visitor is not as general as an iterator, and requires the ability to encapsulate a sequence of statements into a bundle.

Further Information

The data structure textbooks in C++ and in Java that I am most familiar with are [Budd 1998] and [Budd 2000], respectively. Collection classes in Smalltalk-80 are described in [Goldberg 1983]. A discussion of how bounded generics are used in Eiffel data structures is presented in [Meyer 1994].

Study Questions

Exercises

1. Argue whether container classes represent a success or a failure of object-oriented programming techniques.

2. Data structures can be divided into those that are characterized by their implementation (linked lists, trees) and those that are characterized by their purpose (stacks, sets). Describe how object-oriented programming techniques can be used to simplify the latter, hiding the implementation details. Give an illustration of a data structure with one interface and two very different implementations.

3. Give an example application of a heterogeneous container–that is, one with many different types of values.

4. The Smalltalk approach to iteration is to bundle the action to be performed and hand it to the data structure; in contrast, an iterator is a data structure that hands values one by one back to a statement performing a certain action. Would it be possible to implement the Smalltalk approach in a different programming language, such as Object Pascal or C++? Does static typing get in the way?

5. Give an example application for templates that is not associated with container classes.