# Chapter 9

# Templates and Containers

The template mechanism in C++ is perhaps one of the more complex features of the language that has no correspondence in Java. Templates allow class definitions or functions to be parameterized by types or values, in much the same way that a function definition can be executed with a variety of different values. The template mechanism is performed at compile time, and permits a great deal of type checking to be performed statically, and eliminates many of the run-time casts that typically populate Java programs (and consume Java execution time).

DEFINE *A template allows a class or function to be parameterized by a type*

A major use of templates is as a tool to develop a rich set of data structure, or container abstractions. In this chapter we will first explain the template mechanism, contrasting it with various different techniques in Java. The chapter will then conclude with a description of the Standard Template Library, or STL. The STL is the major data structure library used by C++ programs.

## 9.1 Template Classes

Template classes are perhaps best explained using an example. Consider the following definition, which is a generalization of the box data structure we developed in earlier chapters:

```
template <class T> class box {
public:
    box ( ) { }
    box (T v) : val(v) { }
    box (box<T> & right) : val(right.val) { }

    T value() { return val; }
```

```
    void operator = (T right) { val = right; }
    void operator = (box<T> & right) { val=right.val; }

private:
    T val;
};
```

The new box is a template class. That means that the type box itself is incomplete, it cannot by itself be used to create instances. Instead, the parameter (T, in this case) must be filled in with a specific type before an instance can be created. 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.

Within the class body the variable T can be used as a type name. Thus, we can declare variables of type T, have functions that return a T value, and so on. (Note that T is simply an identifier, and that any other identifier name could have been used.)

To create an object we must first specify a value for T. For example, the following creates a box that will hold an integer, and a box that will hold a double precision value:

```
box<int> ib;
box<double> db;
ib = 7;
db = 3.14;
box<int> ibtwo = 4; // can be initialized in constructor
ib = ibtwo;
int x = ib.value();
```

The types associated with template classes are scrupulously checked at compile time. An attempt to use a value incorrectly will result in a compile time error:

```
ib = 2.7; // error - cannot assign real to int
```

Probably the most common use for template classes, although by no means the only one, is to create container classes. The STL, described in Section 9.3, is one such collection of classes. For example, the list data structure represents the abstraction of a linked list, but does not itself specify the particular type of elements it will contain. Instead, the element type is specified by a template parameter:

```
list<int> ilist;     // create a list of integers
```

```
list<double> dlist:     // create a list of real numbers
list<Animal *> alist;     // create a list of pointers to animals

ilist.push_front(7);     // add an element to front of list
int x = ilist.front(); // extract first element from list
```

Contrast this with the way collections are implemented in Java. In Java, the values held by a collection class are stored in variables declared as Object. There are two major problems with the Java approach:

1. It means that non object values, such as primitive types (integer and the like) cannot be stored in Java collections. This is the major reason the Java language provides wrapper classes, such as Integer.

2. It means that when a value is removed from a Java collection, it must be cast back to the appropriate type. Notice that there are no cast operations in the above example. When we remove an element from the list ilist, the compiler knows already that it is an integer, and not a double or an animal or any other sort of value.

By using templates, the language allows truly reusable, general-purpose components to be created and manipulated with a minimum of difficulty and yet still retain type safety, which is an important goal of strongly typed languages. On the other hand, Java can easily maintain heterogeneous collections, that is, collections of values of various different types. Such collections are more difficult to represent in C++.

The keyword class in the template parameter list is somewhat misleading, since the value can be any type quantity, not simply a class value. (The slightly more descriptive keyword typename can be used instead of class, however this is a recent change to C++ and as yet not supported by many compilers). For example, we created box values using the int type as a template parameter, which is not a class type. Other primitive values can also be used as template parameters. For example, the following creates a bit array with a given number of bit values.

NOTE *The keyword* typename *is a recent addition to the C++ language*

```
template <int s> class bitSet {
public:
    set (int index) { ... }
    test (int index) { ... }

    void operator = (bitSet<s> & right);

protected:
        // assume 16 bits per word
    int data [ (s + 15)/ 16 ];
```

```
};
```

To manipulate a bit array we simply fill the template argument value with an integer quantity:

```
bitSet<25> a;
a.set(17); // set position 17
if (a.test(i)) ...
```

A bit array can be assigned to another bit array of the same size, but not to an array with a smaller number of values:

```
bitSet<25> b;
bitSet<30> c;

a = b; // ok, will execute assignment operator
a = c; // produces compile time error, sizes don't match
```

### 9.1.1 Template Methods

When template methods are written separately from the class definition, they must also be be parameterized by the template argument:

```
template <int s>
void bitSet<s>::operator = (bitSet<s> & right)
{
        // first compute the data vector size
    int max = (s + 15) / 16;
        // then copy all the data fields into our array
    for (int i = 0; i < max; i++)
        data[i] = right.data[i];
}
```

Notice that the class name bitSet has been qualified by the template argument `s`. This is not simply a method in class bitSet, but is a method in bitSet<s>.

## 9.2 Template Functions

In addition to classes, ordinary functions can also be given template definitions. A simple example is the following, which is a function to determine the maximum of two quantities:

```
template <class T>
T max (T left, T right)
{
        // return largest value
    if (left < right)
        return right;
    else
        return left;
}
```

The function max can be used with any data type that implements the < operator. Since the < operator can be overloaded for user defined data types, this is potentially an infinite set of possibilities.

NOTE
*Template function types will be inferred from the argument values, and need not be specified by the programmer*

```
int i = max(3, 4);
double d = max(3.14, 4.7);

    // assume comparison has been defined for class anObject
AnObject a, b;
AnObject c = max(a, b);

    // mixing types will not work
int i = max(2, a); // will produce compiler error
```

A feature to note is that it is not necessary to explicitly declare the types that will be used with an invocation of a template function, as it is with template classes. Instead, the necessary types are *inferred* from the types given by the arguments. If a single unambiguous meaning is not possible, as in the last statement shown above, then the compiler will produce an error message.

WARNING
*Errors in template functions are often difficult to trace back to the originating statement*

Template functions are not expanded until they are used, at which point an instance of the function with the correct argument types will be created. Often template functions operate by using other template functions, and so on many levels deep. An error, for example, using argument values that do not support all the necessary operations (for instance, using max with arguments that do not recognize the < operator), will be reported is relation to the expanded template function, not in reference to the function invocation that caused the template to be expanded. Because of this, an error in a template function may be very difficult to trace back to the originating statement.

## 9.3 The Standard Template Library

The Standard Template Library, or STL, is a collection of useful container classes for common data structures, such as lists and stacks. It includes the following:

| | |
|---|---|
| vector | Resizeable array |
| list | Linked list |
| deque | Double ended vector |
| set and multiset | Ordered set |
| map and multimap | Keyed dictionary |
| stack | last-in, first out collection |
| queue | first-in, first out collection |
| priority queue | ordered access collection |

Manipulation of the containers is facilitated by a tool called an iterator. An iterator is a generaization of a memory pointer, used to access elements in a container without knowledge of the internal representation for the container, similar to the class Enumeration in Java. Finally, the STL is unique in providing a large collection of generic algorithms, which are functions manipulated by means of iterators, not tied to any single container. Each of these features will be described in the sections that follow.

### 9.3.1 Containers

Space does not permit a complete exposition of all the STL classes, instead in this section we will simply contrast those containers which are most closely similar to the standard containers in Java, and outline the major areas of difference.

#### Vectors

WARNING *The vector data type has lower case* v, *unlike the Vector data type in Java*

Like the Java Vector data type, the class vector (note lower case v) represents a dynamically resizable array. Unlike the Java class, the C++ class must be provided with a template parameter that describes the element type:

```
vector<int> a;
vector<double> b(10); // initially ten elements
```

Note that data structures are generally declared as ordinary variables. It is not necessary to allocate a container class using the new operator except in special circumstances, such as when the container must outlive the context in which it is declared.

Figure 9.1 compares methods in the vector data type to the Java equivalents. The C++ class provides a variety of different constructors, including ones that set the initial size, and a constructor that sets the initial size and provides a default initial value. Element access and modification in the C++ version is both provided by the subscript operator, whereas

| **operation** | C++ | Java equivalent |
|---|---|---|
| Creation | vector<T> v;<br>vector<T> v(int size);<br>vector<T> v(size, initial value);<br>vector<T> v(oldvector); | v = new Vector(); |
| Element access | v[index] | elementAt(index) |
| First element<br>Last element | front()<br>back() | firstElement()<br>lastElement() |
| size<br>empty test<br>set size<br>capacity<br>set capacity | size()<br>empty()<br>resize(newsize)<br>capacity()<br>reserve(newsize) | size()<br>isEmpty()<br>setSize(newsize)<br>capacity()<br>ensureCapacity(newsize) |
| add to front<br>add to back<br>insert at position<br>change at position | push_front(value)<br>push_back(value)<br>insert(iterator, value)<br>v[position] = value | insertElementAt(value, 0)<br>addElement(value)<br>insertElementAt(value, position)<br>setElementAt(value, position) |
| Remove last element<br>Remove from position | pop_back()<br>erase(iterator) | *none*<br>removeElementAt(position) |
| copy | vector<T> v(oldvector); | clone() |
| create iterator | begin()<br>end() | elements() |

Figure 9.1: Comparison of vector methods

Java uses two different methods for these two activities. In the methods that refer to an internal position within the vector the Java methods generally use an integer index, while the C++ versions use an iterator value (see Section 9.3.2). A copy, or clone, is formed in Java using an explicit method, while in C++ the same action is performed using a copy constructor.

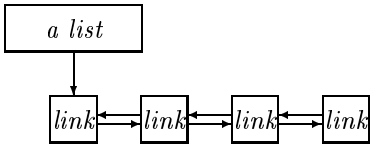
WARNING *Like arrays and strings, vectors do not check for out of range index values*

One important difference between the Java abstraction and the C++ version is that attempting to access an element that is out of range will always raise an exception in Java, however the C++ data abstraction performs no run-time checks. It is possible that an out of range index value would not be detected, and garbage values would be returned on access, or an unknown location modified on element set. An alternative method, at, does perform a run-time range check, and generates a sf out_of_range error for illegal index values.

The Java class provides a method (contains) that can be used to determine whether or not a specific value is held by the container, and another (removeElement(val)) to remove an element by giving its value, rather than position. The C++ version has no similar methods, although one of the generic algorithms (see Section 9.3.3) can be used for this purpose. If this operation is common, the set data abstraction is a preferable alternative to using a vector.

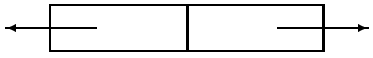
### Linked list

The list proves most of the same features as the vector data type, adding methods that allow insertions and removals from the front of the container, as well as from the back. In addition, insertions or removals from the middle are performed in constant time, rather than the $O(n)$ time required by the vector data type.




### Deque

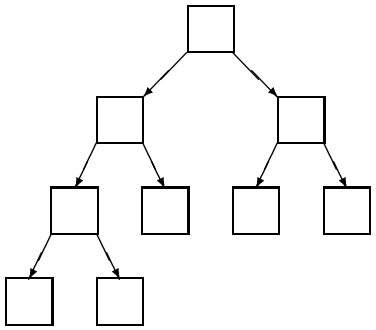
The deque is an interesting data abstraction, that can be thought of as a pair of vectors placed back to back, moving in opposite directions. This arrangement permits efficient (constant time) insertion to either end, however slower linear insertion into the middle. However, the deque is a more space efficient structure than is a list.



### Set

The set data type maintains elements in order, and thereby permits very efficient (that is, logarithmic) time insertion, removal, and inclusion test for values. Internally, it is implemented by a balanced binary tree.

NOTE
*While a set in mathematics does not imply order, the* set *data type maintains its values in order*



### Map

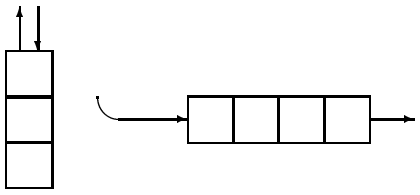
A map is a key/value structure, similar to the Java Dictionary or Hashtable data types. The map data type is parameterized by two template arguments, one for the key type and a second for the value type. Operations on maps are implemented using a data type called a pair, which is a key/value combination. Iterators, for example, yield pair values. The key element in the pair is obtained using the function first, and the value field is found using the method second. An optional third argument (required in some C++ compiler implementations) is a function object used to compare key values to each other. (Function objects will be discussed in Section 9.3.4.)

DEFINE *A map is an indexed collection, similar to the Java Dictionary*

The case study in graph manipulation, Chapter 15, illustrates the use of the map data type.

### Stack and Queue

A stack is a linear structure that allows insertions and removals only from one end, while a queue inserts elements from one end and removes them from the other:



DEFINE
*An adapter is a software component that changes the interface to another component*

The stack and queue data structures in the STL are interesting in that they are *adapters*, built on top of an underlying data type, such as a vector or a linked list. The template argument used in the constructor specifies the underlying container:

```
stack< vector<int> > stackOne;
stack< list<anObject *> > stackTwo;
queue< deque<double> > queueOne;
```

Note the separating space between the two right angle brackets, many C++ compilers will report spurious compiler errors if the space is omitted (confusing the angle brackets for the right shift operator). The method names for the stack abstraction are similar to those used by the Java Stack class.

#### Priority Queue

The priority queue data type provides rapid access to the largest element in a collection, and rapid removal of this element. Like the stack, it is built as an adaptor on top of another container, typically a vector or a list. Two template arguments are used with a priority queue. The first is the underlying container, while the second is a function object that is used to compare elements.

### 9.3.2 Iterators

The concept of an iterator in the STL is similar in purpose to the idea of an Enumeration in Java, but differs in the particulars of use. This is perhaps best illustrated by an example. Imagine that v is a vector of integer values. We could compute the sum of the values in Java using the following code fragment:

```
int sum = 0;
for (Enumeration e = v.elements(); e.hasMoreElements(); ) {
    Object val = e.nextElement();
    Integer iv = (Integer) val;
    sum += iv.intValue();
}
```

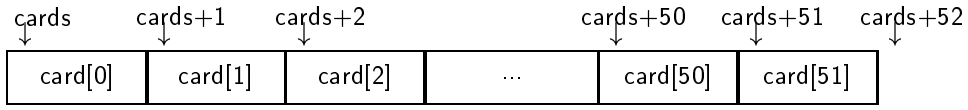
The same idea would be written using iterators as follows:

```
int sum = 0;
vector<int>::iterator start = v.begin();
vector<int>::iterator stop = v.end();
for ( ; start != stop; ++start)
    sum += *start;
```

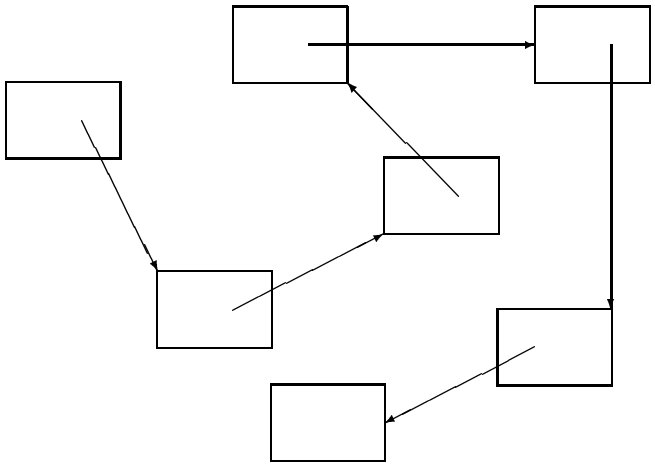
Several differences should be noted. Because the STL containers use template definitions, it is not necessary to cast the object to the proper type after it is removed from the container. The template property of the STL also means that containers can store primitive types, such as integers, and do not need the wrapper classes necessitated by the Java version. A different iterator data type is provided by each container, thus to create an iterator it is necessary to first specify the container type. Most importantly, while enumerations work as a single value, iterators must always be manipulated in pairs, using a beginning and an ending iterator.

One way to understand iterators is to note that they are designed to be equivalent, and compatible, with conventional pointers. Just as pointers can be used in a variety of ways in traditional programming, iterators are also used for a number of different purposes. An iterator can be used to denote a specific value, just as a pointer can be used to reference a specific memory location. On the other hand, a *pair* of iterators can be used to describe a *range* of values, in a manner analogous to the way in which two pointers can be used to describe a contiguous region of memory.

Imagine, for example, an array that is being used to represent a deck of playing cards. Two pointer values can be used to denote the beginning and ending of the deck:




If we need to represent the beginning and end of the memory space, we can use the values cards and cards+52. In the case of iterators, however, the values being described are not necessarily physically in sequence, but rather are logically in sequence, because they are derived from the same container, and the second follows the first in the order elements are maintained by the collection.



The convention used by the container classes in the standard library is to return, in response to the member function named begin(), an iterator that accesses the first element in the collection. An iterator denoting the end of the collection is yielded by the member function end().

Conventional pointers can sometimes be *null*, that is, they point at nothing. Iterators, as well, can fail to denote any specific value. Just as it is a logical error to dereference and use a null pointer, it is an error to dereference and use an iterator that is not denoting a value.

NOTE *Iterators produced by containers often come in pairs. The beginning iterator is returned by the function* begin, *the ending iterator by the function* end

When two pointers that describe a region in memory are used in a C++ program, it is conventional that the ending pointer is *not* considered to be part of the region. We see this in the picture of the cards array, where the array is described as extending from cards to cards+52, even though the element at cards+52 is not part of the array. Instead, the pointer value cards+52 is the *past-the-end* value – the element that is the next value *after* the end of the range being described. Iterators are used to describe a range in the same manner. The second value is not considered to be part of the range being denoted. Instead, the second value is a *past-the-end* element, describing the next value in sequence after the final value of the range. Sometimes, as with pointers to memory, this will be an actual value in the container. Other times it may be a special value, specifically constructed for the purpose. The value returned by the member function end() is usually of the latter type, being a special value that does not refer to any element in the collection. In either case, it is never legal to try to dereference an iterator that is being used to specify the end of a range. (An iterator that does not denote a location, such as an end-of-range iterator, is often called an *invalid* iterator).

An examination of a typical algorithm will help illustrate how iterators are used. The generic function named find() can be used to determine whether or not a value occurs in a collection. It is implemented as follows:

```
template <class iterator, class T>
iterator find (iterator first, iterator last, T & value)
{
    while (first != last && *first != value)
        ++first;
    return first;
}
```

The following shows how we could use this algorithm to search for a value being held by a conventional C++ array:

```
int data[100];
    ...
int * where = find(data, data+100, 7);
```

Alternatively, the following declares a new variable, then searches for the value 7 in a list of integers, assigning the resulting iterator to the variable:

```
list<int>::iterator where = find(aList.begin(), aList.end(), 7);
```

The resulting value is either the end-of-list iterator (equal to the value returned by the function end()) or it represents the location of the first 7 in the list.

As with conventional pointers, the fundamental operation used to modify an iterator is the increment operator (operator ++). When the increment operator is applied to an iterator that denotes the final value in a sequence, it will be changed to the "past the end" value. An iterator j is said to be *reachable* from an iterator i if, after a finite sequence of applications of the expression ++i, the iterator i becomes equal to j.

WARNING
*When used as pairs, the second iterator must always be reachable from the first*

Ranges can be used to describe the entire contents of a container, by constructing an iterator to the initial element and a special "ending" iterator. Ranges can also be used to describe subsequences within a single container, by employing two iterators to specific values. Whenever two iterators are used to describe a range it is assumed, but not verified, that the second iterator is reachable from the first. Errors can occur if this expectation is not satisfied.

The find() algorithm illustrates three requirements for an iterator:

- An iterator can be compared for equality to another iterator. They are equal when they point to the same position, and are otherwise not equal.
- An iterator can be dereferenced using the * operator, to obtain the value being denoted by the iterator. Depending upon the type of iterator and variety of underlying container, this value can also sometimes be used as the target of an assignment in order to change the value being held by the container.

- An iterator can be incremented, so that it refers to the next element in sequence, using the operator ++.

What makes iterators possible is the fact that these characteristics can all be provided with new meanings in a C++ program, since the behavior of the given functions can all be modified by *overloading* the appropriate operators (see Chapter 7).

There are two major categories of iterator constructed by the containers in the standard library. The types list, set and map produce *bi-directional* iterators. These iterators recognize the increment and decrement operators (the latter moving the iterator *backwards* one element), but cannot be randomly accessed. The types vector, string and deque, on the other hand, generate *random-access* iterators, which permit the subscript operator and the addition of integer values to an iterator (analogous to adding an integer value to a pointer, as with the expression cards+52). Some of the generic algorithms depend upon this subscripting ability, and therefore cannot be used with lists or sets.

### 9.3.3 Generic Algorithms

DEFINE *A generic algorithm is a software algorithm that can be used with many different collection classes*

One of the most interesting features of the STL is the separation between the container abstractions themselves and algorithms that can be used with the containers. By separating the two and providing a rich collection of algorithms that work only through iterators, the same algorithms can be used with a variety of different containers, or indeed with normal arrays and regular memory pointers. These functions are termed *generic algorithms*, since they are generic to a wide variety of uses. Once again, the template mechanism is the key to specializing the generic algorithm for use in any particular situation.

We saw one example generic algorithm in the find procedure described earlier. This algorithm performs a linear search to locate a value within a collection. Other algorithms are used to initialize the elements in a container, to perform a variety of different searches, to transform the values in a container in place, to remove elements, or to reduce a collection to a single value.

An example algorithm that produces an in-place transformation is the function random_shuffle. This randomly rearranges the values in the collection. Using this, we could randomly shuffle the card values described earlier as follows:

```
random_shuffle (cards, cards+52, randomizer);
```

The randomizer used by this algorithm must be a random number generator, written in the form of a function object, to be described in the next section.

### 9.3.4 Function Objects

DEFINE *A Function Object is an object that can be used in the fashion of a function*

Functions are not really first class values in C++ (or in many other languages, for that

matter). One cannot have a variable that holds a function, for example[1]. Yet many of the generic algorithms must be specialized by passing a function as an argument.

The STL gets around this little problem in an interesting fashion. Function invocation is considered by C++ to be just another operator, in this case the parenthesis operator. Like almost all operators, this can be overloaded by a class. Thus, the programmer can make an *object* that can be used as if it were a *function*. One example is the random number generator used in the example above. This could be written as follows:

```
class randomInteger {
public:
    unsigned int operator () (unsigned int max) {
        // compute rand value between 0 and max
        unsigned int rval = rand();
        return rval % max;
    }
};

randomInteger randomizer; // create an instance of class
```

The parenthesis operator defines a "function-like" interface that takes a single integer argument. Using this value, and a real system-provided random number generator named rand,[2] the function computes a positive random number between 0 and the maximum value.

Another example will further illustrate this idea. The generic algorithm find_if locates the first element in a collection that satisfies a predicate supplied by the user. Suppose we wish to find the first value larger than 12. We could write a special "larger than 12" function, but let us generalize this to a "larger than x" function, where the value x is specified when an instance of the class is created. We can do this as follows:

```
class LargerThan {
public:
        // constructor
    LargerThan (int v) { val = v; }

        // the function call operator
    bool operator () (int test)
        { return test > val; }

private:
```

[1] It is important to be precise here. One can, in C++, have a pointer to a function, but that is not the same as having a function value.

[2] See Section A.8 in Appendix A.2 for a discussion of the standard libraries.

```
    int val;
};
```

Creating an instance of LargerThan gives us a function-like object that will test an argument value to see if it is larger than whatever value was specified by the constructor. Using this, we could find the first element in a list that is larger than 12 using the following function invocation:

```
LargerThan tester(12); // create the predicate function
list<int>::iterator found =
    find_if (aList.begin(), aList.end(), tester);
if (found != aList.end())
    printf("element is %d", *found); // found such a value
else
    printf("no element larger than 12");
```

The find_if generic algorithm takes a collection specified by a pair of iterators, and returns an iterator that indicates the first element that matches the specification, returning the end of range iterator is no such element is found. By testing the result against the ending iterator we can tell whether or not the search was a success. If it was successful, we can dereference the resulting iterator to find the actual value.

If the only use for a function object is as an argument to a generic algorithm, as in the example shown above, then the creation of the function object can often be replaced by the creation of a *nameless temporary* value. A nameless temporary is created by simply naming the class type for the temporary values along with the arguments to be used in the constructor that will initialize the temporary:

```
LargerThan(12) // creates an instance of LargerThan
```

The creation of this temporary can be performed directly in the argument list for the generic algorithm, yielding a very concise description:

```
list<int>::iterator found =
    find_if (aList.begin(), aList.end(), LargerThan(12));
```

Another common use for function objects is in the template parameter list for containers. For the map and priority_queue data types, as well as others, an optional template argument describes the algorithm to be used in comparison between elements. This algorithm must be described as a function object. For example, in Chapter 15 we present a case study that uses a map to represent a graph data type. For keys this data type uses primitive C++ strings, i.e., pointers to characters. Since the default implementation of pointer comparison

is not what we wish in this case, another algorithm must be defined. This is provided by the following class description:

```
class charCompare { // compare two character literal values
public:
    bool operator () (const char * left, const char * right) const
    {
        return strcmp(left, right) < 0;
    }
};
```

A charCompare takes two character pointer values, and compares the strings they reference. Using this, the desired data type is then declared as a structure that uses character pointers as keys, holds integers as values, and uses the charCompare function object to compare key values:

```
typedef map <const char *, unsigned int, charCompare> cityInfo;
```

Working together, containers, iterators, algorithms and function objects provide a set of powerful tools that can find use in almost any nontrivial program.

## Test Your Understanding

1. How is the description of a template class different from the description of a normal C++ class?
2. How is the creation of an instance of a template class different from the creation of a normal C++ value?
3. Using a container class in Java frequently necessitates the use of run-time casts. How does the template mechanism eliminate the need for these casts?
4. What is a template function? How are the template argument types for a template function determined?
5. What do the initials STL stand for?
6. What are some of the ways that the vector data type in C++ differs from the Vector data type in Java? In what ways are they similar?
7. Why are the stack, queue, and priority_queue data types known as adaptors?
8. What is a past-the-end value? How is such a value used in an iterator loop?

9. What is a generic algorithm?

10. What is a function object? How is such a value created? How is it used in conjunction with generic algorithms?