16

The `string` Class and the Standard Template Library

In this chapter you'll learn about the following:

- The standard C++ `string` class
- The `auto_ptr`, `unique_ptr`, and `shared_ptr` templates
- The Standard Template Library (STL)
- Container classes
- Iterators
- Function objects (functors)
- STL algorithms
- The `initializer_list` template

By now you are familiar with the C++ goal of reusable code. One of the big payoffs is when you can reuse code written by others. That's where class libraries come in. There are many commercially available C++ class libraries, and there are also libraries that come as part of the C++ package. For example, you've been using the input/output classes supported by the `ostream` header file. This chapter looks at other reusable code available for your programming pleasure.

You've already encountered the `string` class, and this chapter examines it more extensively. Then the chapter looks at "smart pointer" template classes that make managing dynamic memory a bit easier. Next, the chapter looks at the Standard Template Library (STL), a collection of useful templates for handling various kinds of container objects. The STL exemplifies the programming paradigm called *generic programming*. Finally, the chapter looks at the `initializer_list` template class, the C++11 addition that enables using initializer-list syntax with STL objects.

The string Class

Many programming applications need to process strings. C provides some support with its string.h (cstring in C++) family of string functions, and many early C++ implementations provide home-grown classes to handle strings. Chapter 4, "Compound Types," introduced the ANSI/ISO C++ string class. Chapter 12, "Classes and Dynamic Memory Allocation," with its modest String class, illustrates some aspects of designing a class to represent strings.

Recall that the string class is supported by the string header file. (Note that the string.h and cstring header files support the C library string functions for C-style strings, not the string class.) The key to using a class is knowing its public interface, and the string class has an extensive set of methods, including several constructors, overloaded operators for assigning strings, concatenating strings, comparing strings, and accessing individual elements, as well as utilities for finding characters and substrings in a string, and more. In short, the string class has lots to offer.

Constructing a String

Let's look at the string constructors. After all, one of the most important things to know about a class is what your options are when creating objects of that class. Listing 16.1 uses seven of the string constructors (labeled ctor, the traditional C++ abbreviation for *constructor*). Table 16.1 briefly describes the constructors. The table begins with the seven constructors used in Listing 16.1, in that order. It also lists a couple of C++11 additions. The constructor representations are simplified in that they conceal the fact that string really is a typedef for a template specialization basic_string<char> and that they omit an optional argument relating to memory management. (This aspect is discussed later this chapter and in Appendix F, "The string Template Class.") The type size_type is an implementation-dependent integral type defined in the string header file. The class defines string::npos as the maximum possible length of the string. Typically, this would equal the maximum value of an unsigned int. Also the table uses the common abbreviation NBTS for null-byte-terminated string—that is, the traditional C string, which is terminated with a null character.

Table 16.1 **string Class Constructors**

Constructor	Description
string(const char * *s*)	Initializes a string object to the NBTS pointed to by *s*.
string(size_type *n*, char *c*)	Creates a string object of *n* elements, each initialized to the character c.
string(const string & *str*)	Initializes a string object to the string object *str* (copy constructor).

Table 16.1 **string Class Constructors**

Constructor	Description
`string()`	Creates a default `string` object of 0 size (default constructor).
`string(const char * s, size_type n)`	Initializes a `string` object to the NBTS pointed to by *s* and continues for *n* characters, even if that exceeds the size of the NBTS.
`template<class Iter> string(Iter begin, Iter end)`	Initializes a `string` object to the values in the range [*begin*, *end*), where *begin* and *end* act like pointers and specify locations; the range includes *begin* and is up to but not including *end*.
`string(const string & str, size_type pos, size_type n = npos)`	Initializes a `string` object to the object *str*, starting at position *pos* in *str* and going to the end of *str* or using *n* characters, whichever comes first.
`string(string && str) noexcept` (C++11)	Initializes a `string` object to the `string` object *str*; *str* may be altered (move constructor).
`string(initializer_list<char> il)` (C++11)	Initializes a `string` object to the characters in the initializer list *il*.

Listing 16.1 **str1.cpp**

```
// str1.cpp -- introducing the string class
#include <iostream>
#include <string>
// using string constructors

int main()
{
    using namespace std;
    string one("Lottery Winner!");     // ctor #1
    cout << one << endl;               // overloaded <<
    string two(20, '$');               // ctor #2
    cout << two << endl;
    string three(one);                 // ctor #3
    cout << three << endl;
    one += " Oops!";                   // overloaded +=
    cout << one << endl;
    two = "Sorry! That was ";
    three[0] = 'P';
    string four;                       // ctor #4
```

```
    four = two + three;               // overloaded +, =
    cout << four << endl;
    char alls[] = "All's well that ends well";
    string five(alls,20);             // ctor #5
    cout << five << "!\n";
    string six(alls+6, alls + 10);    // ctor #6
    cout << six  << ", ";
    string seven(&five[6], &five[10]); // ctor #6 again
    cout << seven << "...\n";
    string eight(four, 7, 16);        // ctor #7
    cout << eight << " in motion!" << endl;
    return 0;
}
```

The program in Listing 16.1 also uses the overloaded += operator, which appends one string to another, the overloaded = operator for assigning one string to another, the overloaded << operator for displaying a string object, and the overloaded [] operator for accessing an individual character in a string.

Here is the output of the program in Listing 16.1:

```
Lottery Winner!
$$$$$$$$$$$$$$$$$$$$
Lottery Winner!
Lottery Winner! Oops!
Sorry! That was Pottery Winner!
All's well that ends!
well, well...
That was Pottery in motion!
```

Program Notes

The start of the program in Listing 16.1 illustrates that you can initialize a string object to a regular C-style string and display it by using the overloaded << operator:

```
string one("Lottery Winner!");    // ctor #1
cout << one << endl;              // overloaded <<
```

The next constructor initializes the string object two to a string consisting of 20 $ characters:

```
string two(20, '$');              // ctor #2
```

The copy constructor initializes the string object three to the string object one:

```
string three(one);                // ctor #3
```

The overloaded += operator appends the string " Oops!" to the string one:

```
one += " Oops!";                  // overloaded +=
```

This particular example appends a C-style string to a string object. However, the += operator is multiply overloaded so that you can also append string objects and single characters:

```
one += two;   // append a string object (not in program)
one += '!';   // append a type char value (not in program)
```

Similarly, the = operator is overloaded so that you can assign a string object to a string object, a C-style string to a string object, or a simple char value to a string object:

```
two = "Sorry! That was "; // assign a C-style string
two = one;                // assign a string object (not in program)
two = '?';                // assign a char value (not in program)
```

Overloading the [] operator, as the String example in Chapter 12 does, permits access to individual characters in a string object by using array notation:

```
three[0] = 'P';
```

A default constructor creates an empty string that can later be given a value:

```
string four;                      // ctor #4
four = two + three;               // overloaded +, =
```

The second line here uses the overloaded + operator to create a temporary string object, which is then assigned, using the overloaded = operator, to the four object. As you might expect, the + operator concatenates its two operands into a single string object. The operator is multiply overloaded, so the second operand can be a string object or a C-style string or a char value.

The fifth constructor takes a C-style string and an integer as arguments, with the integer indicating how many characters to copy:

```
char alls[] = "All's well that ends well";
string five(alls,20);             // ctor #5
```

Here, as the output shows, just the first 20 characters ("All's well that ends") are used to initialize the five object. As Table 16.1 notes, if the character count exceeds the length of the C-style string, the requested number of characters is still copied. So replacing 20 with 40 in the preceding example would result in 15 junk characters being copied at the end of five. (That is, the constructor would interpret the contents in memory following the string "All's well that ends well" as character codes.)

The sixth constructor has a template argument:

```
template<class Iter> string(Iter begin, Iter end);
```

The intent is that *begin* and *end* act like pointers pointing to two locations in memory. (In general, *begin* and *end* can be iterators, generalizations of pointers extensively used in the STL.) The constructor then uses the values between the locations pointed to by *begin* and *end* to initialize the string object it constructs. The notation [*begin*, *end*), borrowed from mathematics, means the range includes *begin* but doesn't include *end*.

That is, *end* points to a location one past the last value to be used. Consider the following statement:

```
string six(alls+6, alls + 10);    // ctor #6
```

Because the name of an array is a pointer, both `alls + 6` and `alls + 10` are type `char *`, so the template is used with `Iter` replaced by type `char *`. The first argument points to the first `w` in the `alls` array, and the second argument points to the space following the first `well`. Thus, `six` is initialized to the string `"well"`. Figure 16.1 shows how the constructor works.

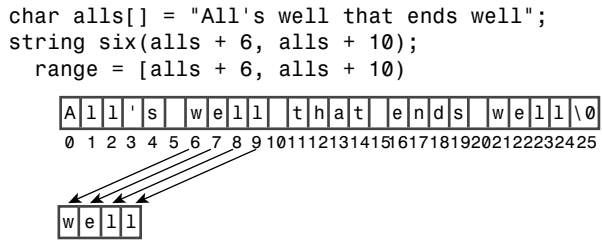


Figure 16.1 A `string` constructor using a range.

Now suppose you want to use this constructor to initialize an object to part of another `string` object—say, the object `five`. The following does not work:

```
string seven(five + 6, five + 10);
```

The reason is that the name of an object, unlike the name of an array, is not treated as the address of an object, hence `five` is not a pointer and `five + 6` is meaningless. However, `five[6]` is a `char` value, so `&five[6]` is an address and can be used as an argument to the constructor:

```
string seven(&five[6], &five[10]);// ctor #6 again
```

The seventh constructor copies a portion of one `string` object to the constructed object:

```
string eight(four, 7, 16);          // ctor #7
```

This statement copies 16 characters from `four` to `eight`, starting at position 7 (the eighth character) in `four`.

C++11 Constructors

The `string(string && str) noexcept` constructor is similar to the copy constructor in that the new `string` is a copy of `str`. However, unlike the copy constructor, it doesn't guarantee that `str` will be treated as `const`. This form of constructor is termed a *move constructor*. The compiler can use it in some situations instead of the copy constructor to

optimize performance. Chapter 18, "Visiting with the New C++ Standard," discusses this topic in the section "Move Semantics and the rvalue Reference."

The `string(initializer_list<char> il)` constructor enables list-initialization for the `string` class. That is, it makes declarations like the following possible:

```
string piano_man = {'L', 'i', 's','z','t'};
string comp_lang {'L', 'i', 's', 'p'};
```

This may not be that useful for the `string` class because using C-style strings is easier, but it does satisfy the intent to make the list-initialization syntax universal. This chapter will discuss the `initializer_list` template further later on.

The `string` Class Input

Another useful thing to know about a class is what input options are available. For C-style strings, recall, you have three options:

```
char info[100];
cin >> info;             // read a word
cin.getline(info, 100);  // read a line, discard \n
cin.get(info, 100);      // read a line, leave \n in queue
```

For `string` objects, recall, you have two options:

```
string stuff;
cin >> stuff;            // read a word
getline(cin, stuff);     // read a line, discard \n
```

Both versions of `getline()` allow for an optional argument that specifies which character to use to delimit input:

```
cin.getline(info,100,':');    // read up to :, discard :
getline(stuff, ':');          // read up to :, discard :
```

The main operational difference is that the `string` versions automatically size the target `string` object to hold the input characters:

```
char fname[10];
string lname;
cin >> fname;      // could be a problem if input size > 9 characters
cin >> lname;      // can read a very, very long word
cin.getline(fname, 10);  // may truncate input
getline(cin, fname);     // no truncation
```

The automatic sizing feature allows the `string` version of `getline()` to dispense with the numeric parameter that limits the number of input characters to be read.

A design difference is that the C-style string input facilities are methods of the `istream` class, whereas the `string` versions are standalone functions. That's why `cin` is an invoking object for C-style string input and a function argument for `string` object input. This applies to the `>>` form, too, which is evident if the code is written in function form:

```
cin.operator>>(fname);     // ostream class method
operator>>(cin, lname);    // regular function
```

Let's examine the `string` input functions a bit more closely. Both, as mentioned, size the target string to fit the input. There are limits. The first limiting factor is the maximum allowable size for a string, represented by the constant `string::npos`. This, typically, is the maximum value of an `unsigned int`, so it doesn't pose a practical limit for ordinary, interactive input. It could be a factor, however, if you attempt to read the contents of an entire file into a single `string` object. The second limiting factor is the amount of memory available to a program.

The `getline()` function for the `string` class reads characters from the input and stores them in a `string` object until one of three things occurs:

- The end-of-file is encountered, in which case `eofbit` of the input stream is set, implying that both the `fail()` and `eof()` methods will return `true`.
- The delimiting character (`\n`, by default) is reached, in which case it is removed from the input stream but not stored.
- The maximum possible number of characters (the lesser of `string::npos` and the number of bytes in memory available for allocation) is read, in which case `failbit` of the input stream is set, implying that the `fail()` method will return `true`.

(An input stream object has an accounting system to keep track of the error state of the stream. In this system, setting `eofbit` registers detecting the end-of-file; setting `failbit` registers detecting an input error; setting `badbit` registers some unrecognized failure, such as a hardware failure; and setting `goodbit` indicates that all is well. Chapter 17, "Input, Output, and Files," discusses this further.)

The `operator>>()` function for the `string` class behaves similarly, except that instead of reading to and discarding a delimiting character, it reads up to a white space character and leaves that character in the input queue. A white space character is a space, newline, or tab character or more generally, any character for which `isspace()` returns `true`.

So far in this book, you've seen several examples of console `string` input. Because the input functions for `string` objects work with streams and recognize the end-of-file, you can also use them for file input. Listing 16.2 shows a short example that reads strings from the file. It assumes that the file contains strings separated by the colon character and uses the `getline()` method of specifying a delimiter. It then numbers and displays the strings, one string to an output line.

Listing 16.2 **strfile.cpp**

```
// strfile.cpp -- read strings from a file
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
int main()
```

```
{
     using namespace std;
     ifstream fin;
     fin.open("tobuy.txt");
     if (fin.is_open() == false)
     {
        cerr << "Can't open file. Bye.\n";
        exit(EXIT_FAILURE);
     }
     string item;
     int count = 0;
     getline(fin, item, ':');
     while (fin)  // while input is good
     {
        ++count;
        cout << count <<": " << item << endl;
        getline(fin, item,':');
     }
     cout << "Done\n";
     fin.close();
     return 0;
}
```

Here is a sample `tobuy.txt` file:

```
sardines:chocolate ice cream:pop corn:leeks:
cottage cheese:olive oil:butter:tofu:
```

Typically, for the program to find the text file, the text file should be in the same directory as the executable program or sometimes in the same directory as the project file. Or you can provide the full path name. On a Windows system, keep in mind that in a C-style string the escape sequence `\\` represents a single backslash:

```
fin.open("C:\\CPP\\Progs\\tobuy.txt"); // file = C:\CPP\Progs\tobuy.txt
```

Here is the output of the program in Listing 16.2:

```
1: sardines
2: chocolate ice cream
3: pop corn
4: leeks
5:
cottage cheese
6: olive oil
7: butter
8: tofu
9:

Done
```

Note that with : specified as the delimiting character, the newline character becomes just another regular character. Thus, the newline character at the end of the first line of the file becomes the first character of the string that continues as "cottage cheese". Similarly, the newline character at the end of the second input line, if present, becomes the sole content of the ninth input string.

Working with Strings

So far, you've learned that you can create string objects in a variety of ways, display the contents of a string object, read data into a string object, append to a string object, assign to a string object, and concatenate two string objects. What else can you do?

You can compare strings. All six relational operators are overloaded for string objects, with one object being considered less than another if it occurs earlier in the machine collating sequence. If the machine collating sequence is the ASCII code, that implies that digits are less than uppercase characters and uppercase characters are less than lowercase characters. Each relational operator is overloaded three ways so that you can compare a string object with another string object, compare a string object with a C-style string, and compare a C-style string with a string object:

```
string snake1("cobra");
string snake2("coral");
char snake3[20] = "anaconda";
if (snake1 < snake 2)          // operator<(const string &, const string &)
    ...
if (snake1 == snake3)          // operator==(const string &, const char *)
    ...
if (snake3 != snake2)          // operator!=(const char *, const string &)
    ...
```

You can determine the size of a string. Both the size() and length() member functions return the number of characters in a string:

```
if (snake1.length() == snake2.size())
    cout << "Both strings have the same length.\n"
```

Why two functions that do the same thing? The length() member comes from earlier versions of the string class, and size() was added for STL compatibility.

You can search a string for a given substring or character in a variety of ways. Table 16.2 provides a short description of four variations of a find() method. Recall that string::npos is the maximum possible number of characters in a string, typically the largest unsigned int or unsigned long value.

Table 16.2 **The Overloaded `find()` Method**

Method Prototype	Description
`size_type find(const string & str, size_type pos = 0) const`	Finds the first occurrence of the substring *str*, starting the search at location *pos* in the invoking string. Returns the index of the first character of the substring if found and returns `string::npos` otherwise.
`size_type find(const char * s, size_type pos = 0) const`	Finds the first occurrence of the substring *s*, starting the search at location *pos* in the invoking string. Returns the index of the first character of the substring if found and returns `string::npos` otherwise.
`size_type find(const char * s, size_type pos = 0, size_type n)`	Finds the first occurrence of the substring consisting of the first *n* characters in *s*, starting the search at location *pos* in the invoking string. Returns the index of the first character of the substring if found and returns `string::npos` otherwise.
`size_type find(char ch, size_type pos = 0) const`	Finds the first occurrence of the character *ch*, starting the search at location *pos* in the invoking string. Returns the index of the character if found and returns `string::npos` otherwise.

The `string` library also provides the related methods `rfind()`, `find_first_of()`, `find_last_of()`, `find_first_not_of()`, and `find_last_not_of()`, each with the same set of overloaded function signatures as the `find()` method. The `rfind()` method finds the last occurrence of a substring or character. The `find_first_of()` method finds the first occurrence in the invoking string of any of the characters in the argument. For example, the following statement would return the location of the `r` in `"cobra"` (that is, the index 3) because that's the first occurrence of any of the letters in `"hark"` in `"cobra"`:

```
int where = snake1.find_first_of("hark");
```

The `find_last_of()` method works the same, except it finds the last occurrence. Thus, the following statement would return the location of the `a` in `"cobra"`:

```
int where = snake1.last_first_of("hark");
```

The `find_first_not_of()` method finds the first character in the invoking string that is not a character in the argument. So the following would return the location of the `c` in `cobra` because `c` is not found in `hark`:

```
int where = snake1.find_first_not_of("hark");
```

(You'll learn about `find_last_not_of()` in an exercise at the end of this chapter.)

There are many more methods, but these are enough to put together a sample program that's a graphically impaired version of the word game Hangman. The game stores a list of words in an array of `string` objects, picks a word at random, and lets you guess letters in the word. Six wrong guesses, and you lose. The program uses the `find()` function to check your guesses and the `+=` operator to build a `string` object to keep track of your wrong guesses. To keep track of your good guesses, the program creates a word the same length as the mystery word but consisting of hyphens. The hyphens are then replaced by correct guesses. Listing 16.3 shows the program.

Listing 16.3 **hangman.cpp**

```
// hangman.cpp -- some string methods
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
#include <cctype>
using std::string;
const int NUM = 26;
const string wordlist[NUM] = {"apiary", "beetle", "cereal",
    "danger", "ensign", "florid", "garage", "health", "insult",
    "jackal", "keeper", "loaner", "manage", "nonce", "onset",
    "plaid", "quilt", "remote", "stolid", "train", "useful",
    "valid", "whence", "xenon", "yearn", "zippy"};

int main()
{
    using std::cout;
    using std::cin;
    using std::tolower;
    using std::endl;
    std::srand(std::time(0));
    char play;
    cout << "Will you play a word game? <y/n> ";
    cin >> play;
    play = tolower(play);
    while (play == 'y')
    {
        string target = wordlist[std::rand() % NUM];
        int length = target.length();
        string attempt(length, '-');
        string badchars;
        int guesses = 6;
        cout << "Guess my secret word. It has " << length
            << " letters, and you guess\n"
            << "one letter at a time. You get " << guesses
```

```
    << " wrong guesses.\n";
cout << "Your word: " << attempt << endl;
while (guesses > 0 && attempt != target)
{
    char letter;
    cout << "Guess a letter: ";
    cin >> letter;
    if (badchars.find(letter) != string::npos
        || attempt.find(letter) != string::npos)
    {
        cout << "You already guessed that. Try again.\n";
            continue;
    }
    int loc = target.find(letter);
    if (loc == string::npos)
    {
        cout << "Oh, bad guess!\n";
        --guesses;
        badchars += letter; // add to string
    }
    else
    {
        cout << "Good guess!\n";
        attempt[loc]=letter;
        // check if letter appears again
        loc = target.find(letter, loc + 1);
        while (loc != string::npos)
        {
            attempt[loc]=letter;
            loc = target.find(letter, loc + 1);
        }
   }
    cout << "Your word: " << attempt << endl;
    if (attempt != target)
    {
        if (badchars.length() > 0)
            cout << "Bad choices: " << badchars << endl;
        cout << guesses << " bad guesses left\n";
    }
}
if (guesses > 0)
    cout << "That's right!\n";
else
    cout << "Sorry, the word is " << target << ".\n";
```

```
        cout << "Will you play another? <y/n> ";
        cin >> play;
        play = tolower(play);
    }

    cout << "Bye\n";

    return 0;
}
```

Here's a sample run of the program in Listing 16.3:

```
Will you play a word game? <y/n> y
Guess my secret word. It has 6 letters, and you guess
one letter at a time. You get 6 wrong guesses.
Your word: ------
Guess a letter: e
Oh, bad guess!
Your word: ------
Bad choices: e
5 bad guesses left
Guess a letter: a
Good guess!
Your word: a--a--
Bad choices: e
5 bad guesses left
Guess a letter: t
Oh, bad guess!
Your word: a--a--
Bad choices: et
4 bad guesses left
Guess a letter: r
Good guess!
Your word: a--ar-
Bad choices: et
4 bad guesses left
Guess a letter: y
Good guess!
Your word: a--ary
Bad choices: et
4 bad guesses left
Guess a letter: i
Good guess!
Your word: a-iary
Bad choices: et
4 bad guesses left
Guess a letter: p
```

```
Good guess!
Your word: apiary
That's right!
Will you play another? <y/n> n
Bye
```

Program Notes

In Listing 16.3, the fact that the relational operators are overloaded lets you treat strings in the same fashion you would treat numeric variables:

```
while (guesses > 0 && attempt != target)
```

This is easier to follow than, say, using `strcmp()` with C-style strings.

The program uses `find()` to check whether a character was selected earlier; if it was already selected, it will be found in either the `badchars` string (bad guesses) or in the `attempt` string (good guesses):

```
if (badchars.find(letter) != string::npos
    || attempt.find(letter) != string::npos)
```

The `npos` variable is a static member of the `string` class. Its value, recall, is the maximum allowable number of characters for a `string` object. Therefore, because indexing begins at zero, it is one greater than the largest possible index and can be used to indicate failure to find a character or a string.

The program makes use of the fact that one of the overloaded versions of the `+=` operator lets you append individual characters to a string:

```
badchars += letter;  // append a char to a string object
```

The heart of the program begins by checking whether the chosen letter is in the mystery word:

```
int loc = target.find(letter);
```

If `loc` is a valid value, the letter can be placed in the corresponding location in the answer string:

```
attempt[loc]=letter;
```

However, a given letter might occur more than once in the mystery word, so the program has to keep checking. The program uses the optional second argument to `find()`, which specifies a starting place in the string from which to begin the search. Because the letter was found at location `loc`, the next search should begin at `loc + 1`. A `while` loop keeps the search going until no more occurrences of that character are found. Note that `find()` indicates failure if `loc` is after the end of the string:

```
// check if letter appears again
loc = target.find(letter, loc + 1);
while (loc != string::npos)
{
```

```
    attempt[loc]=letter;
    loc = target.find(letter, loc + 1);
}
```

What Else Does the `string` Class Offer?

The `string` library supplies many other facilities. There are functions for erasing part or all of a string, for replacing part or all of one string with part or all of another string, for inserting material into a string or removing material from a string, for comparing part or all of one string with part or all of another string, and for extracting a substring from a string. There's a function for copying part of one string to another string, and there's a function for swapping the contents of two strings. Most of these functions are overloaded so that they can work with C-style strings as well as with `string` objects. Appendix F describes the string library function briefly, but let's talk about a few more features here.

First, think about the automatic sizing feature. In Listing 16.3, what happens each time the program appends a letter to a string? It can't necessarily just grow the string in place because it might run into neighboring memory that is already in use. So it may have to allocate a new block and then copy the old contents to a new location. It would be inefficient to do this a lot, so many C++ implementations allocate a block of memory larger than the actual string, giving the string room to grow. Then if the string eventually exceeds that size, the program allocates a new block twice the size to afford more room to grow without continuous resizing. The `capacity()` method returns the size of the current block, and the `reserve()` method allows you to request a minimum size for the block. Listing 16.4 shows an example that uses these methods.

Listing 16.4 **str2.cpp**

```
// str2.cpp -- capacity() and reserve()
#include <iostream>
#include <string>
int main()
{
    using namespace std;
    string empty;
    string small = "bit";
    string larger = "Elephants are a girl's best friend";
    cout << "Sizes:\n";
    cout << "\tempty: " << empty.size() << endl;
    cout << "\tsmall: " << small.size() << endl;
    cout << "\tlarger: " << larger.size() << endl;
    cout << "Capacities:\n";
    cout << "\tempty: " << empty.capacity() << endl;
    cout << "\tsmall: " << small.capacity() << endl;
    cout << "\tlarger: " << larger.capacity() << endl;
    empty.reserve(50);
```

```
    cout << "Capacity after empty.reserve(50): "
         << empty.capacity() << endl;
    return 0;
}
```

Here is the output of the program in Listing 16.4 for one C++ implementation:

```
Sizes:
        empty: 0
        small: 3
        larger: 34
Capacities:
        empty: 15
        small: 15
        larger: 47
Capacity after empty.reserve(50): 63
```

Note that this implementation uses a minimum capacity of 15 characters and seems to use 1 less than multiples of 16 as standard choices for capacities. Other implementations may make different choices.

What if you have a `string` object but need a C-style string? For example, you might want to open a file whose name is in a `string` object:

```
string filename;
cout << "Enter file name: ";
cin >> filename;
ofstream fout;
```

The bad news is that the `open()` method requires a C-style string argument. The good news is that the `c_str()` method returns a pointer to a C-style string that has the same contents as the invoking `string` object. So you can use this:

```
fout.open(filename.c_str());
```

String Varieties

This section treats the `string` class as if it were based on the `char` type. In fact, as mentioned earlier, the string library really is based on a template class:

```
template<class charT, class traits = char _traits<charT>,
        class Allocator = allocator<charT> >
basic_string {...};
```

The `basic_string` template comes with four specializations, each of which has a `typedef` name:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
typedef basic_string<char16_t> u16string;   // C++11
typedef basic_string<char32_t> u32string ;  // C++11
```

This allows you to use strings based on the wchar_t, char16_t, and char32_t types as well as the char type. You could even develop some sort of character-like class and use the basic_string class template with it, provided that your class met certain requirements. The traits class describes specific facts about the chosen character type, such as how to compare values. There are predefined specializations of the char_traits template for the char, wchar_t, char16_t, and char32_t types, and these are the default values for traits. The Allocator class represents a class to manage memory allocation. There are predefined specializations of the allocator template for the various character types, and these are the defaults. They use new and delete.

Smart Pointer Template Classes

A *smart pointer* is a class object that acts like a pointer but has additional features. Here we'll look at three smart pointer templates that can help with managing the use of dynamic memory allocation. Let's begin by taking a look at what might be needed and how it can be accomplished. Consider the following function:

```
void remodel(std::string & str)
{
    std::string * ps = new std::string(str);
    ...
    str = ps;
    return;
}
```

You probably see its flaw. Each time the function is called, it allocates memory from the heap but never frees the memory, thus creating a memory leak. You also know the solution—just remember to free the allocated memory by adding the following statement just before the return statement:

```
delete ps;
```

However, a solution involving the phrase "just remember to" is seldom the best solution. Sometimes you won't remember. Or you will remember but accidentally remove or comment out the code. And even if you do remember, there can still be problems. Consider the following variation:

```
void remodel(std::string & str)
{
    std::string * ps = new std::string(str);
    ...
    if (weird_thing())
        throw exception();
    str = *ps;
    delete ps;
    return;
}
```

If the exception is thrown, the `delete` statement isn't reached, and again there is a memory leak.

You can fix that oversight, as illustrated in Chapter 14, "Reusing Code in C++," but it would be nice if there were a neater solution. Let's think about what is needed. When a function such as `remodel()` terminates, either normally or by throwing an exception, local variables are removed from the stack memory—so the memory occupied by the pointer `ps` is freed. It would be nice if, in addition, the memory pointed to by `ps` was freed. If `ps` had a destructor, that destructor could free the pointed-to memory when `ps` expires. Thus, the problem with `ps` is that it is just an ordinary pointer and not a class object having a destructor. If it were an object, you could have its destructor delete the pointed-to memory when the object expires. And that is the idea behind `auto_ptr`, `unique_ptr`, and `shared_ptr`. The `auto_ptr` template is the C++98 solution. C++11 deprecates `auto_ptr` and provides the other two as alternatives. However, although deprecated, `auto_ptr` has been used for years and may be your only choice if your compiler doesn't support the other two.

Using Smart Pointers

These three smart pointer templates (`auto_ptr`, `unique_ptr`, and `shared_ptr`) each defines a pointer-like object intended to be assigned an address obtained (directly or indirectly) by `new`. When the smart pointer expires, its destructor uses `delete` to free the memory. Thus, if you assign an address returned by `new` to one of these objects, you don't have to remember to free the memory later; it will be freed automatically when the smart pointer object expires. Figure 16.2 illustrates the behavioral difference between `auto_ptr` and a regular pointer. The `shared_ptr` and `unique_ptr` share the same behavior in this situation.

To create one of these smart pointer objects, you include the `memory` header file, which includes the template definitions. Then you use the usual template syntax to instantiate the kind of pointer you require. The `auto_ptr` template, for instance, includes the following constructor:

```
template<class X> class auto_ptr {
public:
    explicit auto_ptr(X* p =0) throw();
...};
```

(The `throw()` notation, recall, means this constructor doesn't throw an exception. Like `auto_ptr`, it is deprecated.) Thus, asking for an `auto_ptr` object of type `X` gives you an `auto_ptr` object that points to type `X`:

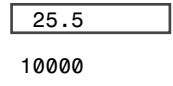
```
auto_ptr<double> pd(new double);  // pd an auto_ptr to double
                                  // (use in place of double * pd)
auto_ptr<string> ps(new string);  // ps an auto_ptr to string
                                  // (use in place of string * ps)
```

```
void demo1()
{
     double * pd = new double; // #1
     *pd = 25.5;               // #2
     return;                   // #3
}
```

#1: Creates storage for pd and a double value, saves address:

#2: Copies value into dynamic memory:

#3: Discards pd, leaves value in dynamic memory:




```
void demo2()
{
     auto_ptr<double> ap(new double); // #1
     *ap = 25.5;                      // #2
     return;                          // #3
}
```

#1: Creates storage for ap and a double value, saves address:

#2: Copies value into dynamic memory:

#3: Discards ap, and ap's destructor frees dynamic memory.

Figure 16.2 A regular pointer versus auto_ptr.

Here new double is a pointer returned by new to a newly allocated chunk of memory. It is the argument to the auto_ptr<double> constructor; that is, it is the actual argument corresponding to the formal parameter p in the prototype. Similarly, new string is also an actual argument for a constructor. The other two smart pointers use the same syntax:

```
unique_ptr<double> pdu(new double);  // pdu an unique_ptr to double
shared_ptr<string> pss(new string);  // pss a shared_ptr to string
```

Thus, to convert the `remodel()` function, you would follow these three steps:

1. Include the `memory` header file.
2. Replace the pointer-to-`string` with a smart pointer object that points to `string`.
3. Remove the `delete` statement.

Here's the function with those changes made using `auto_ptr`:

```
#include <memory>
void remodel(std::string & str)
{
    std::auto_ptr<std::string> ps (new std::string(str));
    ...
    if (weird_thing())
        throw exception();
    str = *ps;
    // delete ps;  NO LONGER NEEDED
    return;
}
```

Note that smart pointers belong to the `std` namespace. Listing 16.5 presents a simple program using all three of these smart pointers. (Your compiler will need to support the C++11 `shared_ptr` and `unique_ptr` classes.) Each use is placed inside a block so that the pointer expires when execution leaves the block. The `Report` class uses verbose methods to report when an object is created or destroyed.

Listing 16.5 **smrtptrs.cpp**

```
// smrtptrs.cpp -- using three kinds of smart pointers
// requires support of C++11 shared_ptr and unique_ptr
#include <iostream>
#include <string>
#include <memory>

class Report
{
private:
    std::string str;
public:
    Report(const std::string s) : str(s)
           { std::cout << "Object created!\n"; }
    ~Report() { std::cout << "Object deleted!\n"; }
    void comment() const { std::cout << str << "\n"; }
};

int main()
{
```

```
    {
        std::auto_ptr<Report> ps (new Report("using auto_ptr"));
        ps->comment();   // use -> to invoke a member function
    }
    {
        std::shared_ptr<Report> ps (new Report("using shared_ptr"));
        ps->comment();
    }
    {
        std::unique_ptr<Report> ps (new Report("using unique_ptr"));
        ps->comment();
    }
    return 0;
}
```

Here is the output:

```
Object created!
using auto_ptr
Object deleted!
Object created!
using shared_ptr
Object deleted!
Object created!
using unique_ptr
Object deleted!
```

Each of these classes has an `explicit` constructor taking a pointer as an argument. Thus, there is no automatic type cast from a pointer to a smart pointer object:

```
shared_ptr<double> pd;
double *p_reg = new double;
pd = p_reg;                          // not allowed (implicit conversion)
pd = shared_ptr<double>(p_reg);      // allowed (explicit conversion
shared_ptr<double> pshared = p_reg; // not allowed (implicit conversion)
shared_ptr<double> pshared(p_reg);  // allowed (explicit conversion)
```

The smart pointer template classes are defined so that in most respects a smart pointer object acts like a regular pointer. For example, given that `ps` is a smart pointer object, you can dereference it (`*ps`), use it to access structure members (`ps->puffIndex`), and assign it to a regular pointer that points to the same type. You can also assign one smart pointer object to another of the same type, but that raises an issue that the next section faces.

But first, here's something you should avoid with all three of these smart pointers:

```
string vacation("I wandered lonely as a cloud.");
shared_ptr<string> pvac(&vacation);  // NO!
```

When `pvac` expires, the program would apply the `delete` operator to non-heap memory, which is wrong.

If Listing 16.5 represents the pinnacle of your programming aspirations, any of these three smart pointers will serve your purposes. But there is more to the story.

Smart Pointer Considerations

Why three smart pointers? (Actually, there are four, but we won't discuss `weak_ptr`.) And why is `auto_ptr` being deprecated?

Begin by considering assignment:

```
auto_ptr<string> ps (new string("I reigned lonely as a cloud."));
auto_ptr<string> vocation;
vocation = ps;
```

What should the assignment statement accomplish? If `ps` and `vocation` were ordinary pointers, the result would be two pointers pointing to the same `string` object. That is not acceptable here because the program would wind up attempting to delete the same object twice—once when `ps` expires, and once when `vocation` expires. There are ways to avoid this problem:

- Define the assignment operator so that it makes a deep copy. This results in two pointers pointing to two distinct objects, one of which is a copy of the other.
- Institute the concept of *ownership*, with only one smart pointer allowed to own a particular object. Only if the smart pointer owns the object will its destructor delete the object. Then have assignment transfer ownership. This is the strategy used for `auto_ptr` and for `unique_ptr`, although `unique_ptr` is somewhat more restrictive.
- Create an even smarter pointer that keeps track of how many smart pointers refer to a particular object. This is called *reference counting*. Assignment, for example, would increase the count by one, and the expiration of a pointer would decrease the count by one. Only when the final pointer expires would `delete` be invoked. This is the `shared_ptr` strategy.

The same strategies we've discussed for assignment, of course, would also apply to the copy constructors.

Each approach has its uses. Listing 16.6 shows an example for which `auto_ptr` is poorly suited.

Listing 16.6 **fowl.cpp**

```
// fowl.cpp  -- auto_ptr a poor choice
#include <iostream>
#include <string>
#include <memory>

int main()
{
    using namespace std;
    auto_ptr<string> films[5] =
```

```
    {
        auto_ptr<string> (new string("Fowl Balls")),
        auto_ptr<string> (new string("Duck Walks")),
        auto_ptr<string> (new string("Chicken Runs")),
        auto_ptr<string> (new string("Turkey Errors")),
        auto_ptr<string> (new string("Goose Eggs"))
    };
    auto_ptr<string> pwin;
    pwin = films[2];   // films[2] loses ownership

    cout << "The nominees for best avian baseball film are\n";
    for (int i = 0; i < 5; i++)
        cout << *films[i] << endl;
    cout << "The winner is " << *pwin << "!\n";
    cin.get();
    return 0;
}
```

Here is some sample output:

```
The nominees for best avian baseball film are
Fowl Balls
Duck Walks
Segmentation fault (core dumped)
```

The "core dumped" message should help fix in your memory that a misused `auto_ptr` can be a problem. (The behavior for this sort of code is undefined, so you might encounter different behavior, depending on your system.) Here the problem is that the following statement transfers ownership from `films[2]` to `pwin`:

```
pwin = films[2];   // films[2] loses ownership
```

That causes `films[2]` to no longer refer to the string. After an `auto_ptr` gives up ownership of an object, it no longer provides access to the object. When the program goes to print the string pointed to by `films[2]`, it finds the null pointer, which apparently is an unpleasant surprise.

Suppose you go back to Listing 16.6 but use `shared_ptr` instead of `auto_ptr`. (You'll need a compiler that supports the C++11 `shared_ptr` class.) Then the program runs fine and gives this output:

```
The nominees for best avian baseball film are
Fowl Balls
Duck Walks
Chicken Runs
Turkey Errors
Goose Eggs
The winner is Chicken Runs!
```

The difference occurs in this part of the program:

```
shared_ptr<string> pwin;
pwin = films[2];
```

This time both `pwin` and `films[2]` point to the same object, and the reference count is upped from 1 to 2. At the end of the program, `pwin`, which was declared last, is the first object to have its destructor called. The destructor decreases the reference count to 1. Then the members of the array of `shared_ptr`s are freed. The destructor for `films[2]` decrements the count to 0 and frees the previously allocated space.

So with `shared_ptr`, Listing 16.6 runs fine. With `auto_ptr` it experienced a runtime crash. What happens if you use `unique_ptr`? Like `auto-ptr`, `unique_ptr` incorporates the ownership model. Yet instead of crashing, the `unique_ptr` version yields a compile-time error objecting to this line:

```
pwin = films[2];
```

Clearly, it is time to look further into the differences between these last two types.

Why `unique_ptr` Is Better than `auto_ptr`

Consider the following statements:

```
auto_ptr<string> p1(new string("auto");  //#1
auto_ptr<string> p2;                     //#2
p2 = p1;                                 //#3
```

When, in statement #3, `p2` takes over ownership of the `string` object, `p1` is stripped of ownership. This, recall, is good because it prevents the destructors for both `p1` and `p2` from trying to delete the same object. But it also is bad if the program subsequently tries to use `p1` because `p1` no longer points to valid data.

Now consider the `unique_ptr` equivalent:

```
unique_ptr<string> p3(new string("auto");  //#4
unique_ptr<string> p4;                     //#5
p4 = p3;                                   //#6
```

In this case, the compiler does not allow statement #6, so we avoid the problem of `p3` not pointing to valid data. Hence, `unique_ptr` is safer (compile-time error versus potential program crash) than `auto_ptr`.

But there are times when assigning one smart pointer to another doesn't leave a dangerous orphan behind. Suppose we have this function definition:

```
unique_ptr<string> demo(const char * s)
{
    unique_ptr<string> temp(new string(s));
    return temp;
}
```

And suppose we then have this code:

```
unique_ptr<string> ps;
ps = demo("Uniquely special");
```

Here, `demo()` returns a temporary `unique_ptr`, and then `ps` takes over ownership of the object originally owned by the returned `unique_ptr`. Then the returned `unique_ptr` is destroyed. That's okay because `ps` now has ownership of the `string` object. But another good thing has happened. Because the temporary `unique_ptr` returned by `demo()` is soon destroyed, there's no possibility of it being misused in an attempt to access invalid data. In other words, there is no reason to forbid assignment in this case. And, miraculously enough, the compiler does allow it!

In short, if a program attempts to assign one `unique_ptr` to another, the compiler allows it if the source object is a temporary rvalue and disallows it if the source object has some duration:

```
using namespace std;
unique_ptr< string> pu1(new string "Hi ho!");
unique_ptr< string> pu2;
pu2 = pu1;                                  //#1 not allowed
unique_ptr<string> pu3;
pu3 = unique_ptr<string>(new string "Yo!");  //#2 allowed
```

Assignment #1 would leave a dangling `unique_ptr` behind (that is, `pu1`), a possible source of mischief. Assignment #2 leaves no `unique_ptr` behind because it invokes the `unique_ptr` constructor, which constructs a temporary object that is destroyed when ownership is transferred to `pu3`. This selective behavior is one indication that `unique_ptr` is superior to `auto_ptr`, which would allow both forms of assignment. It's also the reason that `auto_ptr`s are banned (by recommendation, not by the compiler) from being used in container objects, whereas `unique_ptr`s are allowed. If a container algorithm tries to do something along the lines of assignment #1 to the contents of a container of `unique_ptr`s, you get a compile-time error. If the algorithm tries to do something like assignment #2, that's okay, and the program proceeds. With `auto_ptr`s, cases like assignment #1 could lead to undefined behavior and mysterious crashes.

Of course, it could be possible that you really want to do something like statement #1. The assignment is unsafe only if you use the abandoned smart pointer in an unsmart manner, such as dereferencing it. But you could safely reuse the pointer by assigning a new value to it. C++ has a standard library function called `std::move()` that lets you assign one `unique_ptr` to another. Here is an example using the previously defined `demo()` function, which returns a `unique_ptr<string>` object:

```
using namespace std;
unique_ptr<string> ps1, ps2;
ps1 = demo("Uniquely special");
ps2 = move(ps1);                // enable assignment
ps1 = demo(" and more");
cout << *ps2 << *ps1 << endl;
```

You may be wondering how `unique_ptr`, unlike `auto_ptr`, is able to discriminate between safe and possibly unsafe uses. The answer is that it uses the C++11 additions of move constructors and rvalue references, as discussed in Chapter 18.

Also `unique_ptr` has another advantage over `auto_ptr`. It has a variant that can be used with arrays. Remember, you must pair `delete` with `new` and `delete []` with `new []`. The `auto_ptr` template uses `delete`, not `delete []`, so it can only be used with `new`, not with `new []`. But `unique_ptr` has a `new[]`, `delete[]` version:

```
std::unique_ptr< double[]>pda(new double(5));  // will use delete []
```

> **Caution**
>
> You should use an `auto_prt` or `shared_ptr` object only for memory allocated by `new`, not for memory allocated by `new []`. You should not use `auto_ptr`, `shared_ptr`, or `unique_ptr` for memory not allocated via `new` or, in the case of `unique_ptr`, via `new` or `new[]`.

Selecting a Smart Pointer

Which pointer type should you use? If the program uses more than one pointer to an object, `shared_ptr` is your choice. For instance, you may have an array of pointers and use some auxiliary pointers to identify particular elements, such as the largest and the smallest. Or you could have two kinds of objects that contain pointers to the same third object. Or you might have an STL container of pointers. Many of the STL algorithms include copy or assignment operations that will work with `shared_ptr` but not with `unique_ptr` (you'll get a compiler warning) or `auto_ptr` (you'll get undefined behavior). If your compiler doesn't offer `shared_ptr`, you can get a version from the Boost library.

If the program doesn't need multiple pointers to the same object, `unique_ptr` works. It's a good choice for the return type for a function that returns a pointer to memory allocated by `new`. That way, ownership is transferred to the `unique_ptr` assigned the return value, and that smart pointer takes on the responsibility of calling `delete`. You can store `unique_ptr` objects in an STL container providing you don't invoke methods or algorithms, such as `sort()`, that copy or assign one `unique_ptr` to another. For example, assuming the proper `includes` and `using` statements, code fragments like the following could be used in a program:

```
unique_ptr<int> make_int(int n)
{
    return unique_ptr<int>(new int(n));
}
void show(unique_ptr<int> & pi)             // pass by reference
{
    cout << *a << ' ';
}
int main()
{
...
```

```
    vector<unique_ptr<int> > vp(size);
    for (int i = 0; i < vp.size(); i++)
        vp[i] = make_int(rand() % 1000);    // copy temporary unique_ptr
    vp.push_back(make_int(rand() % 1000))   // ok because arg is temporary
    for_each(vp.begin(), vp.end(), show);   // use for_each()
...
}
```

The call to `push_back()` works because it passes a temporary `unique_ptr` to be assigned to a `unique_ptr` in `vp`. Also note the `for_each()` statement would fail if `show()` were passed an object by value instead of by reference because then it would be necessary to initialize `pi` to a nontemporary `unique_ptr` from `vp`, which isn't allowed. As mentioned before, the compiler will catch attempts to use `unique_ptr` inappropriately.

You can assign a `unique_ptr` to `shared_ptr` under the same conditions that you can assign one `unique_ptr` to another—the source has to be an rvalue. As previously, in the following code `make_int()` is a function with a `unique_ptr<int>` return type:

```
unique_ptr<int> pup(make_int(rand() % 1000);  // ok
shared_ptr<int> spp(pup);                     // not allowed, pup an lvalue
shared_ptr<int> spr(make_int(rand() % 1000);  // ok
```

The `shared_ptr` template includes an explicit constructor for converting an rvalue `unique_ptr` to a `shared_ptr`. The `shared_ptr` takes over ownership of the object previously owned by the `unique_ptr`.

You would use `auto_ptr` in the same situation as `unique_ptr`, but the latter is preferred. If your compiler doesn't provide `unique_ptr`, you might consider the BOOST library `scoped_ptr`, which is similar.

The Standard Template Library

The STL provides a collection of templates representing containers, iterators, function objects, and algorithms. A container is a unit, like an array, that can hold several values. STL containers are homogeneous; that is, they hold values all of the same kind. Algorithms are recipes for accomplishing particular tasks, such as sorting an array or finding a particular value in a list. Iterators are objects that let you move through a container much as pointers let you move through an array; they are generalizations of pointers. Function objects are objects that act like functions; they can be class objects or function pointers (including function names because a function name acts as a pointer). The STL lets you construct a variety of containers, including arrays, queues, and lists, and it lets you perform a variety of operations, including searching, sorting, and randomizing.

Alex Stepanov and Meng Lee developed STL at Hewlett-Packard Laboratories, releasing the implementation in 1994. The ISO/ANSI C++ committee voted to incorporate it as a part of the C++ Standard. The STL is not an example of object-oriented programming. Instead, it represents a different programming paradigm called *generic programming*. This makes STL interesting both in terms of what it does and in terms of its approach.

There's too much information about the STL to present in a single chapter, so we'll look at some representative examples and examine the spirit of the generic programming approach. We'll begin by looking at a few specific examples. Then, when you have a hands-on appreciation for containers, iterators, and algorithms, we'll look at the underlying design philosophy and then take an overview of the whole STL. Appendix G, "The STL Methods and Functions," summarizes the various STL methods and functions.

The `vector` Template Class

Chapter 4 touched briefly on the `vector` class. We'll look more closely at it now. In computing, the term *vector* corresponds to an array rather than to the mathematical vector discussed in Chapter 11, "Working with Classes." (Mathematically, an N-dimensional mathematical vector can be represented by a set of N components, so in that aspect, a mathematical vector is like an N-dimensional array. However, a mathematical vector has additional properties, such as inner and outer products, that a computer vector doesn't necessarily have.) A computing-style vector holds a set of like values that can be accessed randomly. That is, you can use, say, an index to directly access the 10th element of a vector without having to access the preceding 9 elements first. So a `vector` class would provide operations similar to those of the `valarray` and `ArrayTP` classes introduced in Chapter 14 and to those of the `array` class introduced in Chapter 4. That is, you could create a `vector` object, assign one `vector` object to another, and use the `[]` operator to access `vector` elements. To make the class generic, you make it a template class. That's what the STL does, defining a `vector` template in the `vector` (formerly `vector.h`) header file.

To create a `vector` template object, you use the usual `<type>` notation to indicate the type to be used. Also the `vector` template uses dynamic memory allocation, and you can use an initialization argument to indicate how many vector elements you want:

```
#include vector
using namespace std;
vector<int> ratings(5);       // a vector of 5 ints
int n;
cin >> n;
vector<double> scores(n);     // a vector of n doubles
```

After you create a `vector` object, operator overloading for `[]` makes it possible to use the usual array notation for accessing individual elements:

```
ratings[0] = 9;
for (int i = 0; i < n; i++)
    cout << scores[i] << endl;
```

Allocators Again

Like the string class, the various STL container templates take an optional template argument that specifies what allocator object to use to manage memory. For example, the vector template begins like this:

```
template <class T, class Allocator = allocator<T> >
    class vector {...
```

If you omit a value for this template argument, the container template uses the `allocator<T>` class by default. This class uses `new` and `delete`.

Listing 16.7 uses this class in an undemanding application. This particular program creates two `vector` objects, one an `int` specialization and one a `string` specialization; each has five elements.

Listing 16.7 **vect1.cpp**

```
// vect1.cpp -- introducing the vector template
#include <iostream>
#include <string>
#include <vector>

const int NUM = 5;
int main()
{
    using std::vector;
    using std::string;
    using std::cin;
    using std::cout;
    using std::endl;

    vector<int> ratings(NUM);
    vector<string> titles(NUM);
    cout << "You will do exactly as told. You will enter\n"
         << NUM << " book titles and your ratings (0-10).\n";
    int i;
    for (i = 0; i < NUM; i++)
    {
        cout << "Enter title #" << i + 1 << ": ";
        getline(cin,titles[i]);
        cout << "Enter your rating (0-10): ";
        cin >> ratings[i];
        cin.get();
    }
    cout << "Thank you. You entered the following:\n"
          << "Rating\tBook\n";
    for (i = 0; i < NUM; i++)
    {
        cout << ratings[i] << "\t" << titles[i] << endl;
    }

    return 0;
}
```

Here's a sample run of the program in Listing 16.7:

```
You will do exactly as told. You will enter
5 book titles and your ratings (0-10).
Enter title #1: The Cat Who Knew C++
Enter your rating (0-10): 6
Enter title #2: Felonious Felines
Enter your rating (0-10): 4
Enter title #3: Warlords of Wonk
Enter your rating (0-10): 3
Enter title #4: Don't Touch That Metaphor
Enter your rating (0-10): 5
Enter title #5: Panic Oriented Programming
Enter your rating (0-10): 8
Thank you. You entered the following:
Rating  Book
6       The Cat Who Knew C++
4       Felonious Felines
3       Warlords of Wonk
5       Don't Touch That Metaphor
8       Panic Oriented Programming
```

All this program does is use the `vector` template as a convenient way to create a dynamically allocated array. The next section shows an example that uses more of the class methods.

Things to Do to Vectors

Besides allocating storage, what else can the `vector` template do for you? All the STL containers provide certain basic methods, including `size()`, which returns the number of elements in a container, `swap()`, which exchanges the contents of two containers, `begin()`, which returns an iterator that refers to the first element in a container, and `end()`, which returns an iterator that represents past-the-end for the container.

What's an iterator? It's a generalization of a pointer. In fact, it can be a pointer. Or it can be an object for which pointer-like operations such as dereferencing (for example, `operator*()`) and incrementing (for example, `operator++()`) have been defined. As you'll see later, generalizing pointers to iterators allows the STL to provide a uniform interface for a variety of container classes, including ones for which simple pointers wouldn't work. Each container class defines a suitable iterator. The type name for this iterator is a class scope `typedef` called `iterator`. For example, to declare an iterator for a type `double` specialization of `vector`, you would use this:

```
vector<double>::iterator pd;  // pd an iterator
```

Suppose `scores` is a `vector<double>` object:

```
vector<double> scores;
```

Then you can use the iterator pd in code like the following:

```
pd = scores.begin();  // have pd point to the first element
*pd = 22.3;           // dereference pd and assign value to first element
++pd;                 // make pd point to the next element
```

As you can see, an iterator behaves like a pointer. By the way, here's another place the C++11 automatic type deduction can be useful. Instead of, say,

```
vector<double>::iterator pd = scores.begin();
```

you can use this:

```
auto pd = scores.begin();   // C++11 automatic type deduction
```

Returning to the example, what's *past-the-end*? It is an iterator that refers to an element one past the last element in a container. The idea is similar to the idea of the null character being one element past the last actual character in a C-style string, except that the null character is the value in the element, and past-the-end is a pointer (or iterator) to the element. The end() member function identifies the past-the-end location. If you set an iterator to the first element in a container and keep incrementing it, eventually it will reach past-the-end, and you will have traversed the entire contents of the container. Thus, if scores and pd are defined as in the preceding example, you can display the contents with this code:

```
for (pd = scores.begin(); pd != scores.end(); pd++)
    cout << *pd << endl;;
```

All containers have the methods just discussed. The vector template class also has some methods that only some STL containers have. One handy method, called push_back(), adds an element to the end of a vector. While doing so, it attends to memory management so that the vector size increases to accommodate added members. This means you can write code like the following:

```
vector<double> scores;  // create an empty vector
double temp;
while (cin >> temp && temp >= 0)
    scores.push_back(temp);
cout << "You entered " << scores.size() << " scores.\n";
```

Each loop cycle adds one more element to the scores object. You don't have to pick the number of element when you write the program or when you run the program. As long as the program has access to sufficient memory, it will expand the size of scores as necessary.

The erase() method removes a given range of a vector. It takes two iterator arguments that define the range to be removed. It's important that you understand how the STL defines ranges using two iterators. The first iterator refers to the beginning of the range, and the second iterator is one beyond the end of the range. For example, the

following erases the first and second elements—that is, those referred to by `begin()` and `begin() + 1`:

```
scores.erase(scores.begin(), scores.begin() + 2);
```

(Because `vector` provides random access, operations such as `begin() + 2` are defined for the `vector` class iterators.) If `it1` and `it2` are two iterators, the STL literature uses the notation [`p1`, `p2`) to indicate a range starting with `p1` and going up to, but not including, `p2`. Thus, the range [`begin()`, `end()`) encompasses the entire contents of a collection (see Figure 16.3). Also the range [`p1`, `p1`) is empty. (The [) notation is not part of C++, so it doesn't appear in code; it just appears in documentation.)

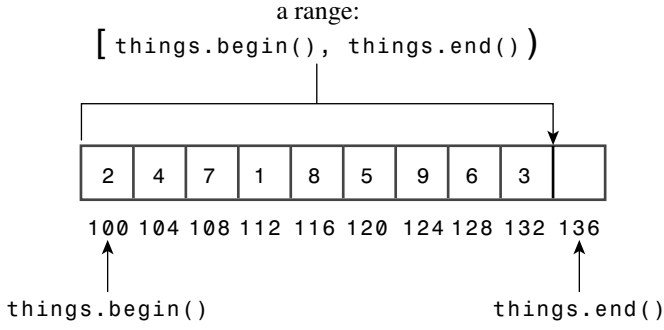


Figure 16.3 The STL range concept.

Note

A range [`it1`, `it2`) is specified by two iterators `it1` and `it2`, and it runs from `it1` up to, but not including, `it2`.

An `insert()` method complements `erase()`. It takes three iterator arguments. The first gives the position ahead of which new elements are to be inserted. The second and third iterator parameters define the range to be inserted. This range typically is part of another container object. For example, the following code inserts all but the first element of the `new_v` vector ahead of the first element of the `old_v` vector:

```
vector<int> old_v;
vector<int> new_v;
...
old_v.insert(old_v.begin(), new_v.begin() + 1, new_v.end());
```

Incidentally, this is a case where having a past-the-end element is handy because it makes it simple to append something to the end of a vector. In this code the new material is inserted ahead of `old.end()`, meaning it's placed *after* the last element in the vector:

```
old_v.insert(old_v.end(), new_v.begin() + 1, new_v.end());
```

Listing 16.8 illustrates the use of `size()`, `begin()`, `end()`, `push_back()`, `erase()`, and `insert()`. To simplify data handling, the `rating` and `title` components of Listing 16.7 are incorporated into a single `Review` structure, and `FillReview()` and `ShowReview()` functions provide input and output facilities for `Review` objects.

Listing 16.8 **vect2.cpp**

```
// vect2.cpp -- methods and iterators
#include <iostream>
#include <string>
#include <vector>

struct Review {
    std::string title;
    int rating;
};
bool FillReview(Review & rr);
void ShowReview(const Review & rr);

int main()
{
    using std::cout;
    using std::vector;
    vector<Review> books;
    Review temp;
    while (FillReview(temp))
        books.push_back(temp);
    int num = books.size();
    if (num > 0)
    {
        cout << "Thank you. You entered the following:\n"
            << "Rating\tBook\n";
        for (int i = 0; i < num; i++)
            ShowReview(books[i]);
        cout << "Reprising:\n"
            << "Rating\tBook\n";
        vector<Review>::iterator pr;
        for (pr = books.begin(); pr != books.end(); pr++)
            ShowReview(*pr);
        vector <Review> oldlist(books);     // copy constructor used
        if (num > 3)
        {
            // remove 2 items
            books.erase(books.begin() + 1, books.begin() + 3);
            cout << "After erasure:\n";
            for (pr = books.begin(); pr != books.end(); pr++)
                ShowReview(*pr);
```

```
            // insert 1 item
            books.insert(books.begin(), oldlist.begin() + 1,
                         oldlist.begin() + 2);
            cout << "After insertion:\n";
            for (pr = books.begin(); pr != books.end(); pr++)
                ShowReview(*pr);
        }
        books.swap(oldlist);
        cout << "Swapping oldlist with books:\n";
        for (pr = books.begin(); pr != books.end(); pr++)
            ShowReview(*pr);
    }
    else
        cout << "Nothing entered, nothing gained.\n";
    return 0;
}

bool FillReview(Review & rr)
{
    std::cout << "Enter book title (quit to quit): ";
    std::getline(std::cin,rr.title);
    if (rr.title == "quit")
        return false;
    std::cout << "Enter book rating: ";
    std::cin >> rr.rating;
    if (!std::cin)
        return false;
    // get rid of rest of input line
    while (std::cin.get() != '\n')
        continue;
    return true;
}

void ShowReview(const Review & rr)
{
    std::cout << rr.rating << "\t" << rr.title << std::endl;
}
```

Here is a sample run of the program in Listing 16.8:

```
Enter book title (quit to quit): The Cat Who Knew Vectors
Enter book rating: 5
Enter book title (quit to quit): Candid Canines
Enter book rating: 7
Enter book title (quit to quit): Warriors of Wonk
Enter book rating: 4
Enter book title (quit to quit): Quantum Manners
```

```
Enter book rating: 8
Enter book title (quit to quit): quit
Thank you. You entered the following:
Rating  Book
5       The Cat Who Knew Vectors
7       Candid Canines
4       Warriors of Wonk
8       Quantum Manners
Reprising:
Rating  Book
5       The Cat Who Knew Vectors
7       Candid Canines
4       Warriors of Wonk
8       Quantum Manners
After erasure:
5       The Cat Who Knew Vectors
8       Quantum Manners
After insertion:
7       Candid Canines
5       The Cat Who Knew Vectors
8       Quantum Manners
Swapping oldlist with books:
5       The Cat Who Knew Vectors
7       Candid Canines
4       Warriors of Wonk
8       Quantum Manners
```

More Things to Do to Vectors

There are many things programmers commonly do with arrays, such as search them, sort them, randomize the order, and so on. Does the vector template class have methods for these common operations? No! The STL takes a broader view, defining *nonmember* functions for these operations. Thus, instead of defining a separate `find()` member function for each container class, it defines a single `find()` nonmember function that can be used for all container classes. This design philosophy saves a lot of duplicate work. For example, suppose you had 8 container classes and 10 operations to support. If each class had its own member function, you'd need 8×10, or 80, separate member function definitions. But with the STL approach, you'd need just 10 separate nonmember function definitions. And if you defined a new container class, provided that you followed the proper guidelines, it too could use the existing 10 nonmember functions to find, sort, and so on.

On the other hand, the STL sometimes defines a member function even if it also defines a nonmember function for the same task. The reason is that for some actions, there is a class-specific algorithm that is more efficient than the more general algorithm. Therefore, the vector `swap()` will be more efficient than the nonmember `swap()`. On the other

hand, the nonmember version will allow you swap contents between two different kinds of containers.

Let's examine three representative STL functions: `for_each()`, `random_shuffle()`, and `sort()`. The `for_each()` function can be used with any container class. It takes three arguments. The first two are iterators that define a range in the container, and the last is a pointer to a function. (More generally, the last argument is a function object; you'll learn about function objects shortly.) The `for_each()` function then applies the pointed-to function to each container element in the range. The pointed-to function must not alter the value of the container elements. You can use the `for_each()` function instead of a `for` loop. For example, you can replace the code

```
vector<Review>::iterator pr;
for (pr = books.begin(); pr != books.end(); pr++)
    ShowReview(*pr);
```

with the following:

```
for_each(books.begin(), books.end(), ShowReview);
```

This enables you to avoid dirtying your hands (and code) with explicit use of iterator variables.

The `random_shuffle()` function takes two iterators that specify a range and rearranges the elements in that range in random order. For example, the following statement randomly rearranges the order of all the elements in the `books` vector:

```
random_shuffle(books.begin(), books.end());
```

Unlike `for_each`, which works with any container class, this function requires that the container class allow random access, which the `vector` class does.

The `sort()` function, too, requires that the container support random access. It comes in two versions. The first version takes two iterators that define a range, and it sorts that range by using the `<` operator defined for the type element stored in the container. For example, the following sorts the contents of `coolstuff` in ascending order, using the built-in `<` operator to compare values:

```
vector<int> coolstuff;
...
sort(coolstuff.begin(), coolstuff.end());
```

If the container elements are user-defined objects, then there has to be an `operator<()` function defined that works with that type of object in order to use `sort()`. For example, you could sort a vector containing `Review` objects if you provided either a `Review` member function or a nonmember function for `operator<()`. Because `Review` is a structure, its members are public, and a nonmember function like this would serve:

```
bool operator<(const Review & r1, const Review & r2)
{
    if (r1.title < r2.title)
        return true;
```

```
    else if (r1.title == r2.title && r1.rating < r2.rating)
        return true;
    else
        return false;
}
```

With a function like this in place, you could then sort a vector of `Review` objects (such as `books`):

```
sort(books.begin(), books.end());
```

This version of the `operator<()` function sorts in lexicographic order of the title members. If two objects have the same title members, they are then sorted in ratings order. But suppose you want to sort in decreasing order or in order of ratings instead of titles. In such a case, you can use the second form of `sort()`. It takes three arguments. The first two, again, are iterators that indicate the range. The final argument is a pointer to a function (more generally, a function object) to be used instead of `operator<()` for making the comparison. The return value should be convertible to `bool`, with `false` meaning the two arguments are in the wrong order. Here's an example of such a function:

```
bool WorseThan(const Review & r1, const Review & r2)
{
    if (r1.rating < r2.rating)
        return true;
    else
        return false;
}
```

With this function in place, you can use the following statement to sort the `books` vector of `Review` objects in order of increasing rating values:

```
sort(books.begin(), books.end(), WorseThan);
```

Note that the `WorseThan()` function does a less complete job than `operator<()` of ordering `Review` objects. If two objects have the same title member, the `operator<()` function sorts by using the rating member. But if two objects have the same rating member, `WorseThan()` treats them as equivalent. The first kind of ordering is called *total ordering*, and the second kind is called *strict weak ordering*. With total ordering, if both $a < b$ and $b < a$ are false, then a and b must be identical. With strict weak ordering, that's not so. They might be identical, or they might just have one aspect that is the same, such as the `rating` member in the `WorseThan()` example. So instead of saying the two objects are identical, the best you can say for strict weak ordering is that they are *equivalent*.

Listing 16.9 illustrates the use of these STL functions.

Listing 16.9 **vect3.cpp**

```
// vect3.cpp -- using STL functions
#include <iostream>
#include <string>
```

```
#include <vector>
#include <algorithm>

struct Review {
    std::string title;
    int rating;
};

bool operator<(const Review & r1, const Review & r2);
bool worseThan(const Review & r1, const Review & r2);
bool FillReview(Review & rr);
void ShowReview(const Review & rr);
int main()
{
    using namespace std;

    vector<Review> books;
    Review temp;
    while (FillReview(temp))
        books.push_back(temp);
    if (books.size() > 0)
    {
        cout << "Thank you. You entered the following "
             << books.size() << " ratings:\n"
              << "Rating\tBook\n";
        for_each(books.begin(), books.end(), ShowReview);

        sort(books.begin(), books.end());
        cout << "Sorted by title:\nRating\tBook\n";
        for_each(books.begin(), books.end(), ShowReview);

        sort(books.begin(), books.end(), worseThan);
        cout << "Sorted by rating:\nRating\tBook\n";
        for_each(books.begin(), books.end(), ShowReview);

        random_shuffle(books.begin(), books.end());
        cout << "After shuffling:\nRating\tBook\n";
        for_each(books.begin(), books.end(), ShowReview);
    }
    else
        cout << "No entries. ";
    cout << "Bye.\n";
    return 0;
}

bool operator<(const Review & r1, const Review & r2)
{
```

```
    if (r1.title < r2.title)
        return true;
    else if (r1.title == r2.title && r1.rating < r2.rating)
        return true;
    else
        return false;
}

bool worseThan(const Review & r1, const Review & r2)
{
    if (r1.rating < r2.rating)
        return true;
    else
        return false;
}

bool FillReview(Review & rr)
{
    std::cout << "Enter book title (quit to quit): ";
    std::getline(std::cin,rr.title);
    if (rr.title == "quit")
        return false;
    std::cout << "Enter book rating: ";
    std::cin >> rr.rating;
    if (!std::cin)
        return false;
    // get rid of rest of input line
    while (std::cin.get() != '\n')
        continue;
    return true;
}

void ShowReview(const Review & rr)
{
    std::cout << rr.rating << "\t" << rr.title << std::endl;
}
```

Here's a sample run of the program in Listing 16.9:

```
Enter book title (quit to quit): The Cat Who Can Teach You Weight Loss
Enter book rating: 8
Enter book title (quit to quit): The Dogs of Dharma
Enter book rating: 6
Enter book title (quit to quit): The Wimps of Wonk
Enter book rating: 3
Enter book title (quit to quit): Farewell and Delete
Enter book rating: 7
```

```
Enter book title (quit to quit): quit
Thank you. You entered the following 4 ratings:
Rating  Book
8       The Cat Who Can Teach You Weight Loss
6       The Dogs of Dharma
3       The Wimps of Wonk
7       Farewell and Delete
Sorted by title:
Rating  Book
7       Farewell and Delete
8       The Cat Who Can Teach You Weight Loss
6       The Dogs of Dharma
3       The Wimps of Wonk
Sorted by rating:
Rating  Book
3       The Wimps of Wonk
6       The Dogs of Dharma
7       Farewell and Delete
8       The Cat Who Can Teach You Weight Loss
After shuffling:
Rating  Book
7       Farewell and Delete
3       The Wimps of Wonk
6       The Dogs of Dharma
8       The Cat Who Can Teach You Weight Loss
Bye.
```

The Range-Based `for` Loop (C++11)

The range-based `for` loop, mentioned in Chapter 5, "Loops and Relational Expressions," is designed to work with the STL. To review, here's the first example from Chapter 5:

```
double prices[5] = {4.99, 10.99, 6.87, 7.99, 8.49};
for (double x : prices)
    cout << x << std::endl;
```

The contents of the parentheses for the `for` loop declare a variable of the type stored in a container and then the name of the container. Next, the body of the loop uses the named variable to access each container element in turn. Consider, for instance, this statement from Listing 16.9:

```
for_each(books.begin(), books.end(), ShowReview);
```

It can be replaced with the following range-based `for` loop:

```
for (auto x : books) ShowReview(x);
```

The compiler will use the type of `books`, which is `vector<Review>`, to deduce that `x` is type `Review`, and the loop will pass each `Review` object in `books` to `ShowReview()` in turn.

Unlike for_each(), the range-based for can alter the contents of a container. The trick is to specify a reference parameter. For example, suppose we have this function:

```
void InflateReview(Review &r){r.rating++;}
```

You could apply this function to each element in books with the following loop:

```
for (auto & x : books) InflateReview(x);
```

Generic Programming

Now that you have some experience using the STL, let's look at the underlying philosophy. The STL is an example of *generic programming*. Object-oriented programming concentrates on the data aspect of programming, whereas generic programming concentrates on algorithms. The main things the two approaches have in common are abstraction and the creation of reusable code, but the philosophies are quite different.

A goal of generic programming is to write code that is independent of data types. Templates are the C++ tools for creating generic programs. Templates, of course, let you define a function or class in terms of a generic type. The STL goes further by providing a generic representation of algorithms. Templates make this possible, but not without the added element of careful and conscious design. To see how this mixture of templates and design works, let's look at why iterators are needed.

Why Iterators?

Understanding iterators is perhaps the key to understanding the STL. Just as templates make algorithms independent of the type of data stored, iterators make the algorithms independent of the type of container used. Thus, they are an essential component of the STL's generic approach.

To see why iterators are needed, let's look at how you might implement a find function for two different data representations and then see how you could generalize the approach. First, let's consider a function that searches an ordinary array of double for a particular value. You could write the function like this:

```
double * find_ar(double * ar, int n, const double & val)
{
    for (int i = 0; i < n; i++)
        if (ar[i] == val)
            return &ar[i];
    return 0;  // or, in C++11, return nullptr;
}
```

If the function finds the value in the array, it returns the address in the array where the value is found; otherwise, it returns the null pointer. It uses subscript notation to move through the array. You could use a template to generalize to arrays of any type having an == operator. Nonetheless, this algorithm is still tied to one particular data structure—the array.

So let's look at searching another kind of data structure, the linked list. (Chapter 12 uses a linked list to implement a `Queue` class.) The list consists of linked `Node` structures:

```
struct Node
{
    double item;
    Node * p_next;
};
```

Suppose you have a pointer that points to the first node in the list. The `p_next` pointer in each node points to the next node, and the `p_next` pointer for the last node in the list is set to `0`. You could write a `find_ll()` function this way:

```
Node* find_ll(Node * head, const double & val)
{
    Node * start;
    for (start = head; start!= 0; start = start->p_next)
        if (start->item == val)
            return start;
    return 0;
}
```

Again, you could use a template to generalize this to lists of any data type supporting the `==` operator. Nonetheless, this algorithm is still tied to one particular data structure—the linked list.

If you consider details of implementation, the two `find` functions use different algorithms: One uses array indexing to move through a list of items, and the other resets `start` to `start->p_next`. But broadly, the two algorithms are the same: Compare the value with each value in the container in sequence until you find a match.

The goal of generic programming in this case would be to have a single `find` function that would work with arrays or linked lists or any other container type. That is, not only should the function be independent of the data type stored in the container, it should be independent of the data structure of the container itself. Templates provide a generic representation for the data type stored in a container. What's needed is a generic representation of the process of moving through the values in a container. The iterator is that generalized representation.

What properties should an iterator have in order to implement a `find` function? Here's a short list:

- You should be able to dereference an iterator in order to access the value to which it refers. That is, if `p` is an iterator, `*p` should be defined.
- You should be able to assign one iterator to another. That is, if `p` and `q` are iterators, the expression `p = q` should be defined.
- You should be able to compare one iterator to another for equality. That is, if `p` and `q` are iterators, the expressions `p == q` and `p != q` should be defined.

- You should be able to move an iterator through all the elements of a container. This can be satisfied by defining ++p and p++ for an iterator p.

There are more things an iterator could do, but nothing more it need do—at least, not for the purposes of a find function. Actually, the STL defines several levels of iterators of increasing capabilities, and we'll return to that matter later. Note, by the way, that an ordinary pointer meets the requirements of an iterator. Hence, you can rewrite the find_arr() function like this:

```
typedef double * iterator;
iterator find_ar(iterator ar, int n, const double & val)
{
    for (int i = 0; i < n; i++, ar++)
        if (*ar == val)
            return ar;
    return 0;
}
```

Then you can alter the function parameter list so that it takes a pointer to the beginning of the array and a pointer to one past-the-end of the array as arguments to indicate a range. (Listing 7.8 in Chapter 7, "Functions: C++'s Programming Modules," does something similar.) And the function can return the end pointer as a sign the value was not found. The following version of find_ar() makes these changes:

```
typedef double * iterator;
iterator find_ar(iterator begin, iterator end, const double & val)
{
    iterator ar;
    for (ar = begin; ar != end;  ar++)
        if (*ar == val)
            return ar;
    return end;   // indicates val not found
}
```

For the find_ll() function, you can define an iterator class that defines the * and ++ operators:

```
struct Node
{
    double item;
    Node * p_next;
};

class iterator
{
    Node * pt;
public:
    iterator() : pt(0) {}
    iterator (Node * pn) : pt(pn) {}
```

```
    double operator*() { return pt->item;}
    iterator& operator++()     // for ++it
    {
        pt = pt->p_next;
        return *this;
    }
    iterator operator++(int)  // for it++
    {
        iterator tmp = *this;
        pt = pt->p_next;
        return tmp;
    }
// ... operator==(), operator!=(), etc.
};
```

(To distinguish between the prefix and postfix versions of the ++ operator, C++ adopted the convention of letting `operator++()` be the prefix version and `operator++(int)` be the suffix version; the argument is never used and hence needn't be given a name.)

The main point here is not how, in detail, to define the `iterator` class, but that with such a class, the second `find` function can be written like this:

```
iterator find_ll(iterator head, const double & val)
{
    iterator start;
    for (start = head; start!= 0; ++start)
        if (*start == val)
            return start;
    return 0;
}
```

This is very nearly the same as `find_ar()`. The point of difference is in how the two functions determine whether they've reached the end of the values being searched. The `find_ar()` function uses an iterator to one-past-the-end, whereas `find_ll()` uses a null value stored in the final node. Remove that difference, and you can make the two functions identical. For example, you could require that the linked list have one additional element after the last official element. That is, you could have both the array and the linked list have a past-the-end element, and you could end the search when the iterator reaches the past-the-end position. Then `find_ar()` and `find_ll()` would have the same way of detecting the end of data and become identical algorithms. Note that requiring a past-the-end element moves from making requirements on iterators to making requirements on the container class.

The STL follows the approach just outlined. First, each container class (`vector`, `list`, `deque`, and so on) defines an iterator type appropriate to the class. For one class, the iterator might be a pointer; for another, it might be an object. Whatever the implementation, the iterator will provide the needed operations, such as * and ++. (Some classes may need

more operations than others.) Next, each container class will have a past-the-end marker, which is the value assigned to an iterator when it has been incremented one past the last value in the container. Each container class will have `begin()` and `end()` methods that return iterators to the first element in a container and to the past-the-end position. And each container class will have the `++` operation take an iterator from the first element to past-the-end, visiting every container element en route.

To use a container class, you don't need to know how its iterators are implemented nor how past-the-end is implemented. It's enough to know that it does have iterators, that `begin()` returns an iterator to the first element, and that `end()` returns an iterator to past-the-end. For example, suppose you want to print the values in a `vector<double>` object. In that case, you can use this:

```
vector<double>::iterator pr;
for (pr = scores.begin(); pr != scores.end(); pr++)
    cout << *pr << endl;
```

Here the following line identifies `pr` as the iterator type defined for the `vector<double>` class:

```
vector<double>::iterator pr;
```

If you used the `list<double>` class template instead to store scores, you could use this code:

```
list<double>::iterator pr;
for (pr = scores.begin(); pr != scores.end(); pr++)
    cout << *pr << endl;
```

The only change is in the type declared for `pr`. Thus, by having each class define appropriate iterators and designing the classes in a uniform fashion, the STL lets you write the same code for containers that have quite dissimilar internal representations.

With C++ automatic type deduction, you can simplify further and use the following code with either the vector or the list:

```
for (auto pr = scores.begin(); pr != scores.end(); pr++)
    cout << *pr << endl;
```

Actually, as a matter of style, it's better to avoid using the iterators directly; instead, if possible, you should use an STL function, such as `for_each()`, that takes care of the details for you. Alternatively, use the C++11 range-based `for` loop:

```
for (auto x : scores) cout << x << endl;
```

So to summarize the STL approach, you start with an algorithm for processing a container. You express it in as general terms as possible, making it independent of data type and container type. To make the general algorithm work with specific cases, you define iterators that meet the needs of the algorithm and place requirements on the container design. That is, basic iterator properties and container properties stem from requirements placed on the algorithm.

Kinds of Iterators

Different algorithms have different requirements for iterators. For example, a find algorithm needs the ++ operator to be defined so the iterator can step through the entire container. It needs read access to data but not write access. (It just looks at data and doesn't change it.) The usual sorting algorithm, on the other hand, requires random access so that it can swap two non-adjacent elements. If `iter` is an iterator, you can get random access by defining the + operator so that you can use expressions such as `iter + 10`. Also a sort algorithm needs to be able to both read and write data.

The STL defines five kinds of iterators and describes its algorithms in terms of which kinds of iterators it needs. The five kinds are the input iterator, output iterator, forward iterator, bidirectional iterator, and random access iterator. For example, the `find()` prototype looks like this:

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value);
```

This tells you that this algorithm requires an input iterator. Similarly, the following prototype tells you that the sort algorithm requires a random access iterator:

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

All five kinds of iterators can be dereferenced (that is, the * operator is defined for them) and can be compared for equality (using the == operator, possibly overloaded) and inequality (using the != operator, possibly overloaded). If two iterators test as equal, then dereferencing one should produce the same value as dereferencing the second. That is, if

```
iter1 == iter2
```

is true, then the following is also true:

```
*iter1 == *iter2
```

Of course, these properties hold true for built-in operators and pointers, so these requirements are guides for what you must do when overloading these operators for an iterator class. Now let's look at other iterator properties.

Input Iterators

The term *input* is used from the viewpoint of a program. That is, information going from the container to the program is considered input, just as information from a keyboard to the program is considered input. So an *input iterator* is one that a program can use to read values from a container. In particular, dereferencing an input iterator must allow a program to read a value from a container, but it needn't allow a program to alter that value. So algorithms that require an input iterator are algorithms that don't change values held in a container.

An input iterator has to allow you to access all the values in a container. It does so by supporting the ++ operator, both in prefix and suffix form. If you set an input operator to the first element in a container and increment it until it reaches past-the-end, it will point

to every container item once en route. Incidentally, there is no guarantee that traversing a container a second time with an input iterator will move through the values in the same order. Also after an input iterator has been incremented, there is no guarantee that its prior value can still be dereferenced. Any algorithm based on an input iterator, then, should be a single-pass algorithm that doesn't rely on iterator values from a previous pass or on earlier iterator values from the same pass.

Note that an input iterator is a one-way iterator; it can increment, but it can't back up.

Output Iterators

In STL usage, the term *output* indicates that the iterator is used for transferring information from a program to a container. (Thus the output for the program is input for the container.) An output iterator is similar to an input iterator, except that dereferencing is guaranteed to allow a program to alter a container value but not to read it. If the ability to write without reading seems strange, keep in mind that this property also applies to output sent to your display; `cout` can modify the stream of characters sent to the display, but it can't read what's onscreen. The STL is general enough that its containers can represent output devices, so you can run into the same situation with containers. Also if an algorithm modifies the contents of a container (for example, by generating new values to be stored) without reading the contents, there's no reason to require that it use an iterator that can read the contents.

In short, you can use an input iterator for single-pass, read-only algorithms and an output operator for single-pass, write-only algorithms.

Forward Iterators

Like input and output iterators, forward iterators use only the ++ operators for navigating through a container. So a forward iterator can only go forward through a container one element at a time. However, unlike input and output iterators, it necessarily goes through a sequence of values in the same order each time you use it. Also after you increment a forward iterator, you can still dereference the prior iterator value, if you've saved it, and get the same value. These properties make multiple-pass algorithms possible.

A forward iterator can allow you to both read and modify data, or it can allow you just to read it:

```
int * pirw;       // read-write iterator
const int * pir;  // read-only iterator
```

Bidirectional Iterators

Suppose you have an algorithm that needs to be able to traverse a container in both directions. For example, a reverse function could swap the first and last elements, increment the pointer to the first element, decrement the pointer to a second element, and repeat the process. A bidirectional iterator has all the features of a forward iterator and adds support for the two decrement operators (prefix and postfix).

Random Access Iterators

Some algorithms, such as standard sort and binary search, require the ability to jump directly to an arbitrary element of a container. This is termed *random access*, and it requires a random access iterator. This type of iterator has all the features of a bidirectional iterator, plus it adds operations (such as pointer addition) that support random access and relational operators for ordering the elements. Table 16.3 lists the operations a random access iterator has beyond those of a bidirectional iterator. In this table, `X` represents a random iterator type, `T` represents the type pointed to, `a` and `b` are iterator values, `n` is an integer, and `r` is a random iterator variable or reference.

Table 16.3 **Random Access Iterator Operations**

Expression	Description
`a + n`	Points to the `n`th element after the one `a` points to
`n + a`	Same as `a + n`
`a - n`	Points to the `n`th element before the one `a` points to
`r += n`	Equivalent to `r = r + n`
`r -= n`	Equivalent to `r = r - n`
`a[n]`	Equivalent to `*(a + n)`
`b - a`	The value of `n` such that `b == a + n`
`a < b`	True if `b - a > 0`
`a > b`	True if `b < a`
`a >= b`	True if `!(a < b)`
`a <= b`	True if `!(a > b)`

Expressions such as `a + n` are valid only if both `a` and `a + n` lie within the range of the container (including past-the-end).

Iterator Hierarchy

You have probably noticed that the iterator kinds form a hierarchy. A forward iterator has all the capabilities of an input iterator and of an output iterator, plus its own capabilities. A bidirectional iterator has all the capabilities of a forward iterator, plus its own capabilities. And a random access iterator has all the capabilities of a forward iterator, plus its own capabilities. Table 16.4 summarizes the main iterator capabilities. In it, `i` is an iterator, and `n` is an integer.

Table 16.4 **Iterator Capabilities**

Iterator Capability	Input	Output	Forward	Bidirectional	Random Access
Dereferencing read	Yes	No	Yes	Yes	Yes
Dereferencing write	No	Yes	Yes	Yes	Yes

Table 16.4 **Iterator Capabilities**

Iterator Capability	Input	Output	Forward	Bidirectional	Random Access
Fixed and repeatable order	No	No	Yes	Yes	Yes
`++i i++`	Yes	Yes	Yes	Yes	Yes
`--i i--`	No	No	No	Yes	Yes
`i[n]`	No	No	No	No	Yes
`i + n`	No	No	No	No	Yes
`i - n`	No	No	No	No	Yes
`i += n`	No	No	No	No	Yes
`i -=n`	No	No	No	No	Yes

An algorithm written in terms of a particular kind of iterator can use that kind of iterator or any other iterator that has the required capabilities. So a container with, say, a random access iterator can use an algorithm written for an input iterator.

Why all these different kinds of iterators? The idea is to write an algorithm using the iterator with the fewest requirements possible, allowing it to be used with the largest range of containers. Thus, the `find()` function, by using a lowly input iterator, can be used with any container that contains readable values. The `sort()` function, however, by requiring a random access iterator, can be used just with containers that support that kind of iterator.

Note that the various iterator kinds are not defined types; rather, they are conceptual characterizations. As mentioned earlier, each container class defines a class scope `typedef` name called `iterator`. So the `vector<int>` class has iterators of type `vector<int>::iterator`. But the documentation for this class would tell you that vector iterators are random access iterators. That, in turn, allows you to use algorithms based on any iterator type because a random access iterator has all the iterator capabilities. Similarly, a `list<int>` class has iterators of type `list<int>::iterator`. The STL implements a doubly linked list, so it uses a bidirectional iterator. Thus, it can't use algorithms based on random access iterators, but it can use algorithms based on less demanding iterators.

Concepts, Refinements, and Models

The STL has several features, such as kinds of iterators, that aren't expressible in the C++ language. That is, although you can design, say, a class that has the properties of a forward iterator, you can't have the compiler restrict an algorithm to using only that class. The reason is that the forward iterator is a set of requirements, not a type. The requirements could be satisfied by an iterator class you've designed, but they could also be satisfied by an ordinary pointer. An STL algorithm works with any iterator implementation that meets its requirements. STL literature uses the word *concept* to describe a set of requirements. Thus, there is an input iterator concept, a forward iterator concept, and so on. By the way, if you do need iterators for, say, a container class you're designing, you can look to the STL, which include iterator templates for the standard varieties.

Concepts can have an inheritance-like relationship. For example, a bidirectional iterator inherits the capabilities of a forward iterator. However, you can't apply the C++ inheritance mechanism to iterators. For example, you might implement a forward iterator as a class and a bidirectional iterator as a regular pointer. So in terms of the C++ language, this particular bidirectional iterator, being a built-in type, couldn't be derived from a class. Conceptually, however, it does inherit. Some STL literature uses the term *refinement* to indicate this conceptual inheritance. Thus, a bidirectional iterator is a refinement of the forward iterator concept.

A particular implementation of a concept is termed a *model*. Thus, an ordinary pointer-to-`int` is a model of the concept random access iterator. It's also a model of a forward iterator, for it satisfies all the requirements of that concept.

The Pointer As Iterator

Iterators are generalizations of pointers, and a pointer satisfies all the iterator requirements. Iterators form the interface for STL algorithms, and pointers are iterators, so STL algorithms can use pointers to operate on non-STL containers that are based on pointers. For example, you can use STL algorithms with arrays. Suppose `Receipts` is an array of `double` values, and you would like to sort in ascending order:

```
const int SIZE = 100;
double Receipts[SIZE];
```

The STL `sort()` function, recall, takes as arguments an iterator pointing to the first element in a container and an iterator pointing to past-the-end. Well, `&Receipts[0]` (or just `Receipts`) is the address of the first element, and `&Receipts[SIZE]` (or just `Receipts + SIZE`) is the address of the element following the last element in the array. Thus, the following function call sorts the array:

```
sort(Receipts, Receipts + SIZE);
```

C++ guarantees that the expression `Receipts + n` is defined as long as the result lies in the array or one past-the-end. Thus, C++ supports the "one-past-the-end" concept for pointers into an array, and this makes it possible to apply STL algorithms to ordinary arrays. Thus, the fact that pointers are iterators and that algorithms are iterator based makes it possible to apply STL algorithms to ordinary arrays. Similarly, you can apply STL algorithms to data forms of your own design, provided that you supply suitable iterators (which may be pointers or objects) and past-the-end indicators.

`copy()`, `ostream_iterator`, and `istream_iterator`

The STL provides some predefined iterators. To see why, let's establish some background. There is an algorithm called `copy()` for copying data from one container to another. This algorithm is expressed in terms of iterators, so it can copy from one kind of container to another or even from or to an array, because you can use pointers into an array as iterators. For example, the following copies an array into a vector:

```
int casts[10] = {6, 7, 2, 9 ,4 , 11, 8, 7, 10, 5};
vector<int> dice[10];
copy(casts, casts + 10, dice.begin());   // copy array to vector
```

The first two iterator arguments to `copy()` represent a range to be copied, and the final iterator argument represents the location to which the first item is copied. The first two arguments must be input iterators (or better), and the final argument must be an output iterator (or better). The `copy()` function overwrites existing data in the destination container, and the container has to be large enough to hold the copied elements. So you can't use `copy()` to place data in an empty vector—at least not without resorting to a trick that is revealed later in this chapter.

Now suppose you want to copy information to the display. You could use `copy()` if there was an iterator representing the output stream. The STL provides such an iterator with the `ostream_iterator` template. Using STL terminology, this template is a *model* of the output iterator concept. It is also an example of an *adapter*—a class or function that converts some other interface to an interface used by the STL. You can create an iterator of this kind by including the `iterator` (formerly `iterator.h`) header file and making a declaration:

```
#include <iterator>
...
ostream_iterator<int, char> out_iter(cout, " ");
```

The `out_iter` iterator now becomes an interface that allows you to use `cout` to display information. The first template argument (`int`, in this case) indicates the data type being sent to the output stream. The second template argument (`char`, in this case) indicates the character type used by the output stream. (Another possible value would be `wchar_t`.) The first constructor argument (`cout`, in this case) identifies the output stream being used. It could also be a stream used for file output. The final character string argument is a separator to be displayed after each item sent to the output stream.

You could use the iterator like this:

```
*out_iter++ = 15;   // works like cout << 15 << " ";
```

For a regular pointer, this would mean assigning the value `15` to the pointed-to location and then incrementing the pointer. For this `ostream_iterator`, however, the statement means send `15` and then a string consisting of a space to the output stream managed by `cout`. Then it should get ready for the next output operation. You could use the iterator with `copy()` as follows:

```
copy(dice.begin(), dice.end(), out_iter);    // copy vector to output stream
```

This would mean to copy the entire range of the `dice` container to the output stream—that is, to display the contents of the container.

Or you could skip creating a named iterator and construct an anonymous iterator instead. That is, you could use the adapter like this:

```
copy(dice.begin(), dice.end(), ostream_iterator<int, char>(cout, " ") );
```

Similarly, the `iterator` header file defines an `istream_iterator` template for adapting `istream` input to the iterator interface. It is a model of the input iterator concept. You could use two `istream_iterator` objects to define an input range for `copy()`:

```
copy(istream_iterator<int, char>(cin),
     istream_iterator<int, char>(), dice.begin());
```

Like `ostream_iterator`, `istream_iterator` uses two template arguments. The first indicates the data type to be read, and the second indicates the character type used by the input stream. Using a constructor argument of `cin` means to use the input stream managed by `cin`. Omitting the constructor argument indicates input failure, so the previous code means to read from the input stream until end-of-file, type mismatch, or some other input failure.

Other Useful Iterators

The `iterator` header file provides some other special-purpose predefined iterator types in addition to `ostream_iterator` and `istream_iterator`. They are `reverse_iterator`, `back_insert_iterator`, `front_insert_iterator`, and `insert_iterator`.

Let's start with seeing what a reverse iterator does. In essence, incrementing a reverse iterator causes it to decrement. Why not just decrement a regular iterator? The main reason is to simplify using existing functions. Suppose you want to display the contents of the `dice` container. As you just saw, you can use `copy()` and `ostream_iterator` to copy the contents to the output stream:

```
ostream_iterator<int, char> out_iter(cout, " ");
copy(dice.begin(), dice.end(), out_iter);  // display in forward order
```

Now suppose you want to print the contents in reverse order. (Perhaps you are performing time-reversal studies.) There are several approaches that don't work, but rather than wallow in them, let's go to one that does. The `vector` class has a member function called `rbegin()` that returns a reverse iterator pointing to past-the-end and a member `rend()` that returns a reverse iterator pointing to the first element. Because incrementing a reverse iterator makes it decrement, you can use the following statement to display the contents backward:

```
copy(dice.rbegin(), dice.rend(), out_iter); // display in reverse order
```

You don't even have to declare a reverse iterator.

> **Note**
>
> Both `rbegin()` and `end()` return the same value (past-the-end), but as a different type (`reverse_iterator` versus `iterator`). Similarly, both `rend()` and `begin()` return the same value (an iterator to the first element), but as a different type.

Reverse pointers have to make a special compensation. Suppose `rp` is a reverse pointer initialized to `dice.rbegin()`. What should `*rp` be? Because `rbegin()` returns past-the-end, you shouldn't try to dereference that address. Similarly, if `rend()` is really the location of the first element, `copy()` stops one location earlier because the end of the range is not

in a range. Reverse pointers solve both problems by decrementing first and then dereferencing. That is, `*rp` dereferences the iterator value immediately preceding the current value of `*rp`. If `rp` points to position six, `*rp` is the value of position five, and so on. Listing 16.10 illustrates using `copy()`, an `ostream` iterator, and a reverse iterator.

Listing 16.10 **copyit.cpp**

```
// copyit.cpp -- copy() and iterators
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
    using namespace std;

    int casts[10] = {6, 7, 2, 9 ,4 , 11, 8, 7, 10, 5};
    vector<int> dice(10);
    // copy from array to vector
    copy(casts, casts + 10, dice.begin());
    cout << "Let the dice be cast!\n";
    // create an ostream iterator
    ostream_iterator<int, char> out_iter(cout, " ");
    // copy from vector to output
    copy(dice.begin(), dice.end(), out_iter);
    cout << endl;
    cout <<"Implicit use of reverse iterator.\n";
    copy(dice.rbegin(), dice.rend(), out_iter);
    cout << endl;
    cout <<"Explicit use of reverse iterator.\n";
    vector<int>::reverse_iterator ri;
    for (ri = dice.rbegin(); ri != dice.rend(); ++ri)
        cout << *ri << ' ';
    cout << endl;

    return 0;
}
```

Here is the output of the program in Listing 16.10:

```
Let the dice be cast!
6 7 2 9 4 11 8 7 10 5
Implicit use of reverse iterator.
5 10 7 8 11 4 9 2 7 6
Explicit use of reverse iterator.
5 10 7 8 11 4 9 2 7 6
```

If you have the choice of explicitly declaring iterators or using STL functions to handle the matter internally, for example, by passing an `rbegin()` return value to a function, you should take the latter course. It's one less thing to do and one less opportunity to experience human fallibility.

The other three iterators (`back_insert_iterator`, `front_insert_iterator`, and `insert_iterator`) also increase the generality of the STL algorithms. Many STL functions are like `copy()` in that they send their results to a location indicated by an output iterator. Recall that the following copies values to the location beginning at `dice.begin()`:

```
copy(casts, casts + 10, dice.begin());
```

These values overwrite the prior contents in `dice`, and the function assumes that `dice` has enough room to hold the values. That is, `copy()` does not automatically adjust the size of the destination to fit the information sent to it. Listing 16.10 takes care of that situation by declaring `dice` to have 10 elements, but suppose you don't know in advance how big `dice` should be. Or suppose you want to add elements to `dice` rather than overwrite existing ones.

The three insert iterators solve these problems by converting the copying process to an insertion process. Insertion adds new elements without overwriting existing data, and it uses automatic memory allocation to ensure that the new information fits. A `back_insert_iterator` inserts items at the end of the container, and a `front_insert_iterator` inserts items at the front. Finally, the `insert_iterator` inserts items in front of the location specified as an argument to the `insert_iterator` constructor. All three of these iterators are models of the output container concept.

There are restrictions. A `back_insert_iterator` can be used only with container types that allow rapid insertion at the end. (*Rapid* refers to a constant time algorithm; the section "Container Concepts," later in this chapter, discusses the constant time concept further.) The `vector` class qualifies. A `front_insert_iterator` can be used only with container types that allow constant time insertion at the beginning. Here the `vector` class doesn't qualify, but the `queue` class does. The `insert_iterator` doesn't have these restrictions. Thus, you can use it to insert material at the front of a vector. However, a `front_insert_iterator` does so faster for the container types that support it.

Tip

You can use an `insert_iterator` to convert an algorithm that copies data into one that inserts data.

These iterators take the container type as a template argument and the actual container identifier as a constructor argument. That is, to create a `back_insert_iterator` for a `vector<int>` container called `dice`, you use this:

```
back_insert_iterator<vector<int> > back_iter(dice);
```

The reason you have to declare the container type is that the iterator has to make use of the appropriate container method. The code for the `back_insert_iterator`

constructor will assume that a `push_back()` method exists for the type passed to it. The `copy()` function, being a standalone function, doesn't have the access rights to resize a container. But the declaration just shown allows `back_iter` to use the `vector<int>::push_back()` method, which does have access rights.

Declaring a `front_insert_iterator` has the same form. An `insert_iterator` declaration has an additional constructor argument to identify the insertion location:

```
insert_iterator<vector<int> > insert_iter(dice, dice.begin() );
```

Listing 16.11 illustrates using two of these iterators. Also it uses `for_each()` instead of an `ostream` iterator for output.

Listing 16.11 **inserts.cpp**

```
// inserts.cpp -- copy() and insert iterators
#include <iostream>
#include <string>
#include <iterator>
#include <vector>
#include <algorithm>

void output(const std::string & s) {std::cout << s << " ";}

int main()
{
    using namespace std;
    string s1[4] = {"fine", "fish", "fashion", "fate"};
    string s2[2] = {"busy", "bats"};
    string s3[2] = {"silly", "singers"};
    vector<string> words(4);
    copy(s1, s1 + 4, words.begin());
    for_each(words.begin(), words.end(), output);
    cout << endl;
// construct anonymous back_insert_iterator object
    copy(s2, s2 + 2, back_insert_iterator<vector<string> >(words));
    for_each(words.begin(), words.end(), output);
    cout << endl;

// construct anonymous insert_iterator object
    copy(s3, s3 + 2, insert_iterator<vector<string> >(words,
                                          words.begin()));
    for_each(words.begin(), words.end(), output);
    cout << endl;
    return 0;
}
```

Here is the output of the program in Listing 16.11:

```
fine fish fashion fate
fine fish fashion fate busy bats
silly singers fine fish fashion fate busy bats
```

The first `copy()` copies the four strings from `s1` into `words`. This works in part because `words` is declared to hold four strings, which equals the number of strings being copied. Then the `back_insert_iterator` inserts the strings from `s2` just in front of the end of the `words` array, expanding the size of `words` to six elements. Finally, the `insert_iterator` inserts the two strings from `s3` just in front of the first element of `words`, expanding the size of `words` to eight elements. If the program attempted to copy `s2` and `s3` into `words` by using `words.end()` and `words.begin()` as iterators, there would be no room in `words` for the new data, and the program would probably abort because of memory violations.

If you're feeling overwhelmed by all the iterator varieties, keep in mind that using them will make them familiar. Also keep in mind that these predefined iterators expand the generality of the STL algorithms. Thus, not only can `copy()` copy information from one container to another, it can copy information from a container to the output stream and from the input stream to a container. And you can also use `copy()` to insert material into another container. So you wind up with a single function doing the work of many. And because `copy()` is just one of several STL functions that use an output iterator, these predefined iterators multiply the capabilities of those functions, too.

Kinds of Containers

The STL has both container concepts and container types. The concepts are general categories with names such as container, sequence container, and associative container. The container types are templates you can use to create specific container objects. The original 11 container types are `deque`, `list`, `queue`, `priority_queue`, `stack`, `vector`, `map`, `multimap`, `set`, `multiset`, and `bitset`. (This chapter doesn't discuss `bitset`, which is a container for dealing with data at the bit level.) C++11 adds `forward_list`, `unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset`, and it moves `bitset` from the container category into its own separate category. Because the concepts categorize the types, let's start with them.

Container Concepts

No type corresponds to the basic container concept, but the concept describes elements common to all the container classes. It's sort of a conceptual abstract base class—conceptual because the container classes don't actually use the inheritance mechanism. Or to put it another way, the container concept lays down a set of requirements that all STL container classes must satisfy.

A *container* is an object that stores other objects, which are all of a single type. The stored objects may be objects in the OOP sense, or they may be values of built-in types. Data stored in a container is *owned* by the container. That means when a container

expires, so does the data stored in the container. (However, if the data are pointers, the pointed-to data does not necessarily expire.)

You can't store just any kind of object in a container. In particular, the type has to be *copy constructable* and *assignable*. Basic types satisfy these requirements, as do class types—unless the class definition makes one or both of the copy constructor and the assignment operator private or protected. (C++11 refines the concepts, adding terms such as *CopyInsertable* and *MoveInsertable*, but we'll take a more simplified, if less precise, overview.)

The basic container doesn't guarantee that its elements are stored in any particular order or that the order doesn't change, but refinements to the concept may add such guarantees. All containers provide certain features and operations. Table 16.5 summarizes several of these common features. In the table, `X` represents a container type (such as `vector`), `T` represents the type of object stored in the container, `a` and `b` represent values of type `X`, `r` is a value of type `X&`, and `u` represents an identifier of type `X` (that is, if `X` represents `vector<int>`, then `u` is a `vector<int>` object).

Table 16.5 **Some Basic Container Properties**

Expression	Return Type	Description	Complexity
`X::iterator`	Iterator type pointing to `T`	Any iterator category satisfying forward iterator requirements	Compile time
`X::value_type`	`T`	The type for `T`	Compile time
`X u;`		Creates an empty container called `u`	Constant
`X();`		Creates an empty anonymous container	Constant
`X u(a);`		Copy constructor post condition: `u == a`	Linear
`X u = a;`		Same effect as `X u(a);`	Linear
`r = a;`	`X&`	Copy assignment post condition: `r == a`	Linear
`(&a)->~X();`	`void`	Applies destructor to every element of a container	Linear
`a.begin()`	iterator	Returns an iterator referring to the first element of the container	Constant
`a.end()`	iterator	Returns an iterator that is a past-the-end value	Constant
`a.size()`	unsigned integral type	Returns a number of elements equal to `a.end() - a.begin()`	Constant
`a.swap(b)`	`void`	Swaps contents of `a` and `b`	Constant

Table 16.5 **Some Basic Container Properties**

Expression	Return Type	Description	Complexity
`a == b`	convertible to `bool`	Returns `true` if `a` and `b` have the same size and each element in `a` is equivalent to (`==` is `true`) the corresponding element in `b`	Linear
`a != b`	convertible to `bool`	Returns `!(a == b)`	Linear

The Complexity column in Table 16.5 describes the time needed to perform an operation. This table lists three possibilities, which, from fastest to slowest, are as follows:

- Compile time
- Constant time
- Linear time

If the complexity is compile time, the action is performed during compilation and uses no execution time. A constant complexity means the operation takes place during runtime but doesn't depend on the number of elements in an object. A linear complexity means the time is proportional to the number of elements. Thus, if `a` and `b` are containers, `a == b` has linear complexity because the `==` operation may have to be applied to each element of the container. Actually, that is a worst-case scenario. If two containers have different sizes, no individual comparisons need to be made.

Constant-Time and Linear-Time Complexity

Imagine a long, narrow box filled with large packages arranged in a line, and suppose the box is open at just one end. Suppose your task is to unload the package at the open end. This is a constant time task. Whether there are 10 packages or 1,000 packages behind the one at the end makes no difference.

Now suppose your task is to fetch the package at the closed end of the box. This is a linear time task. If there are 10 packages altogether, you have to unload 10 packages to get the one at the closed end. If there are 100 packages, you have to unload 100 packages at the end. Assuming that you are a tireless worker who can move only 1 package at a time, this task will take 10 times longer than the first one.

Now suppose your task is to fetch an arbitrary package. It might happen that the package you are supposed to get is the first one at hand. However, on the average, the number of packages you have to move is still proportional to the number of packages in the container, so the task still has linear-time complexity.

Replacing the long, narrow box with a similar box having open sides would change the task to constant-time complexity because then you could move directly to the desired package and remove it without moving the others.

> The idea of time complexity describes the effect of container size on execution time but ignores other factors. If a superhero can unload packages from a box with one open end 1,000 times faster than you can, the task as executed by her still has linear-time complexity. In this case, the super hero's linear time performance with a closed box (open end) would be faster than your constant time performance with an open box, as long as the boxes didn't have too many packages.

Complexity requirements are characteristic of the STL. Although the details of an implementation may be hidden, the performance specifications should be public so that you know the computing cost of doing a particular operation.

C++11 Additions to Container Requirements

Table 16.6 shows some additions C++11 has made to the general container requirements. The table uses the notation `rv` to denote a non-constant rvalue of type `X` (for example, the return value of a function). Also the requirement in Table 16.6 that `X::iterator` satisfy the requirements for a forward iterator is a change from the former requirement that it just not be an output iterator.

Table 16.6 **Some Added Basic Container Requirements (C++11)**

Expression	Return Type	Description	Complexity
`X u(rv);`		Move constructor post condition: `u` has the value that `rv` had prior to construction.	Linear
`X u = rv;`		Same effect as `X u(rv);`.	
`a = rv;`	`X&`	Move assignment post condition: `a` has the value that rv had prior to assignment.	Linear
`a.cbegin()`	`const_iterator`	Returns a `const` iterator referring to the first element of the container.	Constant
`a.cend()`	`const_iterator`	Returns a `const` iterator that is a past-the-end value.	Constant

The difference between copy construction and copy assignment on the one hand and move construction and move assignment on the other hand is that a copy operation leaves the original unchanged, whereas a move operation can alter the original, perhaps transferring ownership without doing any copying When the source object is temporary, move operations can provide more efficient code than does regular copying. Chapter 18 discusses move semantics further.

Sequences

You can refine the basic container concept by adding requirements. The *sequence* is an important refinement because several of the STL container types—`deque`, `forward_list` (C++11), `list`, `queue`, `priority_queue`, `stack`, and `vector`—are sequences. (Recall that a queue allows elements to be added at the rear end and removed from the front. A double-ended queue, represented by `deque`, allows addition and removal at both ends.) The requirement that the iterator be at least a forward iterator guarantees that the elements are arranged in a definite order that doesn't change from one cycle of iteration to the next. The `array` class also is classified as a sequence container, although it doesn't satisfy all the requirements.

The sequence also requires that its elements be arranged in strict linear order. That is, there is a first element, there is a last element, and each element but the first and last has exactly one element immediately ahead of it and one element immediately after it. An array and a linked list are examples of sequences, whereas a branching structure (in which each node points to two daughter nodes) is not.

Because elements in sequence have a definite order, operations such as inserting values at a particular location and erasing a particular range become possible. Table 16.7 lists these and other operations required of a sequence. The table uses the same notation as Table 16.5, with the addition of `t` representing a value of type `T`—that is, the type of value stored in the container, of `n`, an integer, and of `p`, `q`, `i`, and `j`, representing iterators.

Table 16.7 **Sequence Requirements**

Expression	Return Type	Description
`X a(n,t);`		Declares a sequence `a` of `n` copies of value `t`
`X(n, t)`		Creates an anonymous sequence of `n` copies of value `t`
`X a(i, j)`		Declares a sequence `a` initialized to the contents of range [`i`, `j`)
`X(i, j)`		Creates an anonymous sequence initialized to the contents of range [`i`, `j`)
`a.insert(p,t)`	iterator	Inserts a copy of `t` before `p`
`a.insert(p,n,t)`	`void`	Inserts `n` copies of `t` before `p`
`a.insert(p,i,j)`	`void`	Inserts copies of elements in the range [`i`, `j`) before `p`
`a.erase(p)`	iterator	Erases the element pointed to by `p`
`a.erase(p,q)`	iterator	Erases the elements in the range [`p`, `q`)
`a.clear()`	`void`	Is the same as `erase(begin(), end())`

Because the `deque`, `list`, `queue`, `priority_queue`, `stack`, and `vector` template classes are all models of the sequence concept, they all support the operators in Table 16.7. In

addition, there are operations that are available to some of these six models. When allowed, they have constant-time complexity. Table 16.8 lists these additional operations.

Table 16.8 **Optional Sequence Requirements**

Expression	Return Type	Meaning	Container
a.front()	T&	*a.begin()	vector, list, deque
a.back()	T&	*--a.end()	vector, list, deque
a.push_front(t)	void	a.insert(a.begin(), t)	list, deque
a.push_back(t)	void	a.insert(a.end(), t)	vector, list, deque
a.pop_front(t)	void	a.erase(a.begin())	list, deque
a.pop_back(t)	void	a.erase(--a.end())	vector, list, deque
a[n]	T&	*(a.begin() + n)	vector, deque
a.at(n)	T&	*(a.begin() + n)	vector, deque

Table 16.8 merits a comment or two. First, notice that `a[n]` and `a.at(n)` both return a reference to the `n`th element (numbering from 0) in a container. The difference between the two is that `a.at(n)` does bounds checking and throws an `out_of_range` exception if `n` is outside the valid range for the container. Next, you might wonder why, say, `push_front()` is defined for `list` and `deque` and not for `vector`. Suppose you want to insert a new value at the front of a vector of 100 elements. To make room, you have to move element 99 to position 100, and then you have to move element 98 to position 99, and so on. This is an operation with linear-time complexity because moving 100 elements would take 100 times as long as moving a single element. But the operations in Table 16.8 are supposed to be implemented only if they can be performed with constant-time complexity. The design for lists and double-ended queues, however, allows an element to be added to the front without moving the other elements to new locations, so they can implement `push_front()` with constant-time complexity. Figure 16.4 illustrates `push_front()` and `push_back()`.

Let's take a closer look at the six sequence container types.

vector

You've already seen several examples using the `vector` template, which is declared in the `vector` header file. In brief, `vector` is a class representation of an array. The class provides automatic memory management that allows the size of a `vector` object to vary dynamically, growing and shrinking as elements are added or removed. It provides random access to elements. Elements can be added to or removed from the end in constant time, but insertion and removal from the beginning and the middle are linear-time operations.

In addition to being a sequence, a `vector` container is also a model of the *reversible container* concept. This adds two more class methods: `rbegin()` returns an iterator to the first element of the reversed sequence, and `rend()` returns a past-the-end iterator for the

reversed sequence. So if `dice` is a `vector<int>` container and `Show(int)` is a function that displays an integer, the following code displays the contents of `dice` first in forward order and then in reverse order:

```
for_each(dice.begin(), dice.end(), Show);    // display in order
cout << endl;
for_each(dice.rbegin(), dice.rend(), Show);  // display in reversed order
cout << endl;
```

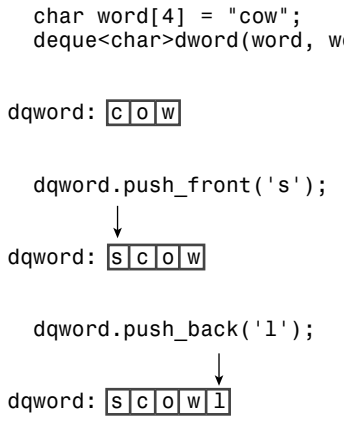


Figure 16.4 `push_front()` and `push_back()`.

The iterator returned by the two methods is of a class scope type `reverse_iterator`. Recall that incrementing such an iterator causes it to move through a reversible container in reverse order.

The `vector` template class is the simplest of the sequence types and is considered the type that should be used by default unless the program requirements are better satisfied by the particular virtues of the other types.

deque

The `deque` template class (declared in the `deque` header file) represents a double-ended queue, a type often called a *deque* (pronounced "deck"), for short. As implemented in the STL, it's a lot like a `vector` container, supporting random access. The main difference is that inserting and removing items from the beginning of a `deque` object are constant-time operations instead of being linear-time operations the way they are for `vector`. So if most operations take place at the beginning and ends of a sequence, you should consider using a `deque` data structure.

The goal of constant-time insertion and removal at both ends of a `deque` makes the design of a `deque` object more complex than that of a `vector` object. Thus, although both offer random access to elements and linear-time insertion and removal from the middle of a sequence, the `vector` container should allow faster execution of these operations.

list

The list template class (declared in the list header file) represents a doubly linked list. Each element, other than the first and last, is linked to the item before it and the item following it, implying that a list can be traversed in both directions. The crucial difference between list and vector is that list provides for constant-time insertion and removal of elements at any location in the list. (Recall that the vector template provides linear-time insertion and removal except at the end, where it provides constant-time insertion and removal.) Thus, vector emphasizes rapid access via random access, whereas list emphasizes rapid insertion and removal of elements.

Like vector, list is a reversible container. Unlike vector, list does not support array notation and random access. Unlike a vector iterator, a list iterator remains pointing to the same element even after items are inserted into or removed from a container. For example, suppose you have an iterator pointing to the fifth element of a vector container. Then suppose you insert an element at the beginning of the container. All the other elements have to be moved to make room, so after the insertion, the fifth element now contains the value that used to be in the fourth element. Thus, the iterator points to the same location but to different data. Inserting a new element into a list, however, doesn't move the existing elements; it just alters the link information. An iterator pointing to a certain item still points to the same item, but it may be linked to different items than before.

The list template class has some list-oriented member functions in addition to those that come with sequences and reversible containers. Table 16.9 lists many of them. (For a complete list of STL methods and functions, see Appendix G.) The Alloc template parameter is one you normally don't have to worry about because it has a default value.

Table 16.9 **Some list Member Functions**

Function	Description
`void merge(list<T, Alloc>& x)`	Merges list x with the invoking list. Both lists must be sorted. The resulting sorted list is in the invoking list, and x is left empty. This function has linear-time complexity.
`void remove(const T & val)`	Removes all instances of val from the list. This function has linear-time complexity.
`void sort()`	Sorts the list by using the < operator; the complexity is N log N for N elements.
`void splice(iterator pos, list<T, Alloc> x)`	Inserts the contents of list x in front of position pos, and x is left empty. This function has constant-time complexity.
`void unique()`	Collapses each consecutive group of equal elements to a single element. This function has linear-time complexity.

Listing 16.12 illustrates these methods, along with the `insert()` method, which comes with all STL classes that model sequences.

Listing 16.12 **`list.cpp`**

```
// list.cpp -- using a list
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>

void outint(int n) {std::cout << n << " ";}

int main()
{
    using namespace std;
    list<int> one(5, 2); // list of 5 2s
    int stuff[5] = {1,2,4,8, 6};
    list<int> two;
    two.insert(two.begin(),stuff, stuff + 5 );
    int more[6] = {6, 4, 2, 4, 6, 5};
    list<int> three(two);
    three.insert(three.end(), more, more + 6);

    cout << "List one: ";
    for_each(one.begin(),one.end(), outint);
    cout << endl << "List two: ";
    for_each(two.begin(), two.end(), outint);
    cout << endl << "List three: ";
    for_each(three.begin(), three.end(), outint);
    three.remove(2);
    cout << endl << "List three minus 2s: ";
    for_each(three.begin(), three.end(), outint);
    three.splice(three.begin(), one);
    cout << endl << "List three after splice: ";
    for_each(three.begin(), three.end(), outint);
    cout << endl << "List one: ";
    for_each(one.begin(), one.end(), outint);
    three.unique();
    cout << endl << "List three after unique: ";
    for_each(three.begin(), three.end(), outint);
    three.sort();
    three.unique();
    cout << endl << "List three after sort & unique: ";
    for_each(three.begin(), three.end(), outint);
    two.sort();
    three.merge(two);
```

```
    cout << endl << "Sorted two merged into three: ";
    for_each(three.begin(), three.end(), outint);
    cout << endl;

    return 0;
}
```

Here is the output of the program in Listing 16.12:

```
List one: 2 2 2 2 2
List two: 1 2 4 8 6
List three: 1 2 4 8 6 6 4 2 4 6 5
List three minus 2s: 1 4 8 6 6 4 4 6 5
List three after splice: 2 2 2 2 2 1 4 8 6 6 4 4 6 5
List one:
List three after unique: 2 1 4 8 6 4 6 5
List three after sort & unique: 1 2 4 5 6 8
Sorted two merged into three: 1 1 2 2 4 4 5 6 6 8 8
```

Program Notes

The program in Listing 16.12 uses the `for_each()` algorithm and an `outint()` function to display the lists. With C++11, you could use the range-based `for` loop instead:

```
for (auto x : three) cout << x << " ";
```

The main difference between `insert()` and `splice()` is that `insert()` inserts a copy of the original range into the destination, whereas `splice()` moves the original range into the destination. Thus, after the contents of `one` are spliced to `three`, `one` is left empty. (The `splice()` method has additional prototypes for moving single elements and a range of elements.) The `splice()` method leaves iterators valid. That is, if you set a particular iterator to point to an element in `one`, that iterator still points to the same element after `splice()` relocates it in `three`.

Notice that `unique()` only reduces adjacent equal values to a single value. After the program executes `three.unique()`, `three` still contains two fours and two sixes that weren't adjacent. But applying `sort()` and then `unique()` does limit each value to a single appearance.

There is a nonmember `sort()` function (Listing 16.9), but it requires random access iterators. Because the trade-off for rapid insertion is to give up random access, you can't use the nonmember `sort()` function with a list. Therefore, the class includes a member version that works within the restrictions of the class.

The `list` Toolbox

The `list` methods form a handy toolbox. Suppose, for example, that you have two mailing lists to organize. You could sort each list, merge them, and then use `unique()` to remove multiple entries.

The `sort()`, `merge()`, and `unique()` methods also each have a version that accepts an additional argument to specify an alternative function to be used for comparing elements. Similarly, the `remove()` method has a version with an additional argument that specifies a function used to determine whether an element is removed. These arguments are examples of predicate functions, a topic to which we'll return later.

`forward_list` (C++11)

C++11 adds `forward_list` as a container class. This class implements a singly linked list. In this kind of list, each item is linked just to the next item, but not to the preceding item. Therefore, the class requires just a forward iterator, not a bidirectional one. Thus, unlike `vector` and `list`, `forward_list` isn't a reversible container. Compared to `list`, `forward_list` is simpler, more compact, but with fewer features.

`queue`

The `queue` template class (declared in the `queue`—formerly `queue.h`—header file) is an adapter class. Recall that the `ostream_iterator` template is an adapter that allows an output stream to use the iterator interface. Similarly, the `queue` template allows an underlying class (`deque`, by default) to exhibit the typical queue interface.

The `queue` template is more restrictive than `deque`. Not only does it not permit random access to elements of a queue, the `queue` class doesn't even allow you to iterate through a queue. Instead, it limits you to the basic operations that define a queue. You can add an element to the rear of a queue, remove an element from the front of a queue, view the values of the front and rear elements, check the number of elements, and test to see if a queue is empty. Table 16.10 lists these operations.

Note that `pop()` is a data removal method, not a data retrieval method. If you want to use a value from a queue, you first use `front()` to retrieve the value and then use `pop()` to remove it from the queue.

Table 16.10 **`queue` Operations**

Method	Description
`bool empty() const`	Returns `true` if the queue is empty and `false` otherwise.
`size_type size() const`	Returns the number of elements in the queue.
`T& front()`	Returns a reference to the element at the front of the queue.
`T& back()`	Returns a reference to the element at the back of the queue.
`void push(const T& x)`	Inserts `x` at the back of the queue.
`void pop()`	Removes the element at the front of the queue.

`priority_queue`

The `priority_queue` template class (declared in the `queue` header file) is another adapter class. It supports the same operations as `queue`. The main difference between the two is that with `priority_queue`, the largest item gets moved to the front of the queue. (Life is

not always fair, and neither are queues.) An internal difference is that the default underlying class is vector. You can alter the comparison used to determine what gets to the head of the queue by providing an optional constructor argument:

```
priority_queue<int> pq1;                // default version
priority_queue<int> pq2(greater<int>);  // use greater<int> to order
```

The greater<>() function is a predefined function object, and it is discussed later in this chapter.

stack

Like queue, stack (declared in the stack—formerly stack.h—header file) is an adapter class. It gives an underlying class (vector, by default) the typical stack interface.

The stack template is more restrictive than vector. Not only does it not permit random access to elements of a stack, the stack class doesn't even allow you to iterate through a stack. Instead, it limits you to the basic operations that define a stack. You can push a value onto the top of a stack, pop an element from the top of a stack, view the value at the top of a stack, check the number of elements, and test whether the stack is empty. Table 16.11 lists these operations.

Table 16.11 **stack Operations**

Method	Description
bool empty() const	Returns true if the stack is empty and false otherwise.
size_type size() const	Returns the number of elements in the stack.
T& top()	Returns a reference to the element at the top of the stack.
void push(const T& x)	Inserts x at the top of the stack.
void pop()	Removes the element at the top of the stack.

Much as with queue, if you want to use a value from a stack, you first use top() to retrieve the value, and then you use pop() to remove it from the stack.

array (C++11)

The array template class, introduced in Chapter 4 and defined in the array header file, is not an STL container because it has a fixed size. Thus, operations that would resize a container, such as push_back() and insert(), are not defined for array. But those member functions that do make sense, such as operator[]() and at(), are provided. And you can use many standard STL algorithms, such as copy() and for_each(), with array objects.

Associative Containers

An *associative container* is another refinement of the container concept. An associative container associates a value with a key and uses the key to find the value. For example, the values could be structures representing employee information, such as name, address, office number, home and work phones, health plan, and so on, and the key could be a

unique employee number. To fetch the employee information, a program would use the key to locate the employee structure. Recall that for a container `X`, in general, the expression `X::value_type` indicates the type of value stored in the container. For an associative container, the expression `X::key_type` indicates the type used for the key.

The strength of an associative container is that it provides rapid access to its elements. Like a sequence, an associative container allows you to insert new elements; however, you can't specify a particular location for the inserted elements. The reason is that an associative container usually has a particular algorithm for determining where to place data so that it can retrieve information quickly.

Associative containers typically are implemented using some form of tree. A *tree* is a data structure in which a root node is linked to one or two other nodes, each of which is linked to one or two nodes, thus forming a branching structure. The node aspect makes it relatively simple to add or remove a new data item, much as with a linked list. But compared to a list, a tree offers much faster search times.

The STL provides four associative containers: `set`, `multiset`, `map`, and `multimap`. The first two types are defined in the `set` header file (formerly separately in `set.h` and `multiset.h`), and the second two types are defined in the `map` header file (formerly separately in `map.h` and `multimap.h`).

The simplest of the bunch is `set`; the value type is the same as the key type, and the keys are unique, meaning there is no more than one instance of a key in a set. Indeed, for `set`, the value is the key. The `multiset` type is like the `set` type except that it can have more than one value with the same key. For example, if the key and value type are `int`, a `multiset` object could hold, say `1`, `2`, `2`, `2`, `3`, `5`, `7`, and `7`.

For the `map` type, the value type is different from the key type, and the keys are unique, with only one value per key. The `multimap` type is similar to `map`, except one key can be associated with multiple values.

There's too much information about these types to cover in this chapter (but Appendix G does list the methods), so let's just look at a simple example that uses `set` and a simple example that uses `multimap`.

A `set` Example

The STL `set` models several concepts. It is an associative set, it is reversible, it is sorted, and the keys are unique, so it can hold no more than one of any given value. Like `vector` and `list`, `set` uses a template parameter to provide the type stored:

```
set<string> A;  // a set of string objects
```

An optional second template argument can be used to indicate a comparison function or object to be used to order the key. By default, the `less<>` template (discussed later) is used. Older C++ implementations may not provide a default value and thus require an explicit template parameter:

```
set<string, less<string> > A;  // older implementation
```

Consider the following code:

```
const int N = 6;
string s1[N] = {"buffoon", "thinkers", "for", "heavy", "can", "for"};
set<string> A(s1, s1 + N); // initialize set A using a range from array
ostream_iterator<string, char> out(cout, " ");
copy(A.begin(), A.end(), out);
```

Like other containers, `set` has a constructor (refer to Table 16.6) that takes a range of iterators as arguments. This provides a simple way to initialize a set to the contents of an array. Remember that the last element of a range is one past-the-end, and `s1 + N` points to one position past-the-end of array `s1`. The output for this code fragment illustrates that keys are unique (the string `"for"` appears twice in the array but once in the set) and that the set is sorted:

```
buffoon can for heavy thinkers
```

Mathematics defines some standard operations for sets. For example, the union of two sets is a set that combines the contents of the two sets. If a particular value is common to both sets, it appears just once in the union because of the unique key feature. The intersection of two sets is a set that consists of the elements that are common to both sets. The difference between two sets is the first set minus the elements common to both sets.

The STL provides algorithms that support these operations. They are general functions rather than methods, so they aren't restricted to `set` objects. However, all `set` objects automatically satisfy the precondition for using these algorithms—namely, that the container be sorted. The `set_union()` function takes five iterators as arguments. The first two define a range in one set, the second two define a range in a second set, and the final iterator is an output iterator that identifies a location to which to copy the resultant set. For example, to display the union of sets `A` and `B`, you can use this:

```
set_union(A.begin(), A.end(), B.begin(), B.end(),
          ostream_iterator<string, char> out(cout, " "));
```

Suppose you want to place the result into a set `C` instead of displaying it. In this case, you would want the last argument to be an iterator into `C`. The obvious choice is `C.begin()`, but that doesn't work for two reasons. The first reason is that associative sets treat keys as constant values, so the iterator returned by `C.begin()` is a constant iterator and can't be used as an output iterator. The second reason not to use `C.begin()` directly is that `set_union()`, like `copy()`, overwrites existing data in a container and requires the container to have sufficient space to hold the new information. `C`, being empty, does not satisfy that requirement. But the `insert_iterator` template discussed earlier solves both problems. Earlier you saw that it converts copying to insertion. Also it models the output iterator concept, so you can use it to write to a container. So you can construct an anonymous `insert_iterator` to copy information to `C`. The constructor, recall, takes the name of the container and an iterator as arguments:

```
set_union(A.begin(), A.end(), B.begin(), B.end(),
          insert_iterator<set<string> >(C, C.begin()));
```

The `set_intersection()` and `set_difference()` functions find the set intersection and set difference of two sets, and they have the same interface as `set_union()`.

Two useful `set` methods are `lower_bound()` and `upper_bound()`. The `lower_bound()` method takes a key-type value as its argument and returns an iterator that points to the first member of the set that is not less than the key argument. Similarly, the `upper_bound()` method takes a key as its argument and returns an iterator that points to the first member of the set that is greater than the key argument. For example, if you had a set of strings, you could use these methods to identify a range encompassing all strings from `"b"` up to `"f"` in the set.

Because sorting determines where additions to a set go, the class has insertion methods that just specify the material to be inserted, without specifying a position. If `A` and `B` are sets of strings, for example, you can use this:

```
string s("tennis");
A.insert(s);                    // insert a value
B.insert(A.begin(), A.end());   // insert a range
```

Listing 16.13 illustrates these uses of sets.

Listing 16.13 **setops.cpp**

```
// setops.cpp -- some set operations
#include <iostream>
#include <string>
#include <set>
#include <algorithm>
#include <iterator>

int main()
{
    using namespace std;
    const int N = 6;
    string s1[N] = {"buffoon", "thinkers", "for", "heavy", "can", "for"};
    string s2[N] = {"metal", "any", "food", "elegant", "deliver","for"};

    set<string> A(s1, s1 + N);
    set<string> B(s2, s2 + N);

    ostream_iterator<string, char> out(cout, " ");
    cout << "Set A: ";
    copy(A.begin(), A.end(), out);
    cout << endl;
    cout << "Set B: ";
    copy(B.begin(), B.end(), out);
    cout << endl;
```

```
    cout << "Union of A and B:\n";
    set_union(A.begin(), A.end(), B.begin(), B.end(), out);
    cout << endl;

    cout << "Intersection of A and B:\n";
    set_intersection(A.begin(), A.end(), B.begin(), B.end(), out);
    cout << endl;

    cout << "Difference of A and B:\n";
    set_difference(A.begin(), A.end(), B.begin(), B.end(), out);
    cout << endl;

    set<string> C;
    cout << "Set C:\n";
    set_union(A.begin(), A.end(), B.begin(), B.end(),
        insert_iterator<set<string> >(C, C.begin()));
    copy(C.begin(), C.end(), out);
    cout << endl;

    string s3("grungy");
    C.insert(s3);
    cout << "Set C after insertion:\n";
    copy(C.begin(), C.end(),out);
    cout << endl;

    cout << "Showing a range:\n";
    copy(C.lower_bound("ghost"),C.upper_bound("spook"), out);
    cout << endl;

    return 0;
}
```

Here is the output of the program in Listing 16.13:

```
Set A: buffoon can for heavy thinkers
Set B: any deliver elegant food for metal
Union of A and B:
any buffoon can deliver elegant food for heavy metal thinkers
Intersection of A and B:
for
Difference of A and B:
buffoon can heavy thinkers
Set C:
any buffoon can deliver elegant food for heavy metal thinkers
Set C after insertion:
any buffoon can deliver elegant food for grungy heavy metal thinkers
Showing a range:
grungy heavy metal
```

Like most of the examples in this chapter, the code in Listing 16.13 takes the lazy route for handling the `std` namespace:

```
using namespace std;
```

It does so in order to simplify the presentation. The examples use so many elements of the `std` namespace that using directives or the scope-resolution operators would tend to make the code look a bit fussy:

```
std::set<std::string> B(s2, s2 + N);
std::ostream_iterator<std::string, char> out(std::cout, " ");
std::cout << "Set A: ";
std::copy(A.begin(), A.end(), out);
```

A `multimap` Example

Like `set`, `multimap` is a reversible, sorted, associative container. However, with `multimap`, the key type is different from the value type, and a `multimap` object can have more than one value associated with a particular key.

The basic `multimap` declaration specifies the key type and the type of value, stored as template arguments. For example, the following declaration creates a `multimap` object that uses `int` as the key type and `string` as the type of value stored:

```
multimap<int,string> codes;
```

An optional third template argument can be used to indicate a comparison function or an object to be used to order the key. By default, the `less<>` template (discussed later) is used with the key type as its parameter. Older C++ implementations may require this template parameter explicitly.

To keep information together, the actual value type combines the key type and the data type into a single pair. To do this, the STL uses a `pair<class T, class U>` template class for storing two kinds of values in a single object. If `keytype` is the key type and `datatype` is the type of the stored data, the value type is `pair<const keytype, datatype>`. For example, the value type for the `codes` object declared earlier is `pair<const int, string>`.

Suppose that you want to store city names, using the area code as a key. This happens to fit the `codes` declaration, which uses an `int` for a key and a `string` as a data type. One approach is to create a pair and then insert it into the `multimap` object:

```
pair<const int, string> item(213, "Los Angeles");
codes.insert(item);
```

Or you can create an anonymous `pair` object and insert it in a single statement:

```
codes.insert(pair<const int, string> (213, "Los Angeles"));
```

Because items are sorted by key, there's no need to identify an insertion location.

Given a pair object, you can access the two components by using the first and second members:

```
pair<const int, string> item(213, "Los Angeles");
cout << item.first << ' ' << item.second << endl;
```

What about getting information about a multimap object? The count() member function takes a key as its argument and returns the number of items that have that key. The lower_bound() and upper_bound() member functions take a key and work as they do for set. Also the equal_range() member function takes a key as its argument and returns iterators representing the range matching that key. In order to return two values, the method packages them into a pair object, this time with both template arguments being the iterator type. For example, the following would print a list of cities in the codes object with area code 718:

```
pair<multimap<KeyType, string>::iterator,
     multimap<KeyType, string>::iterator> range
                          = codes.equal_range(718);
cout << "Cities with area code 718:\n";
std::multimap<KeyType, std::string>::iterator it;
for (it = range.first; it != range.second; ++it)
    cout <<  (*it).second    << endl;
```

Declarations like those just listed helped motivate the C++11 automatic type deduction feature, which allows you to simplify the code as follows:

```
auto range = codes.equal_range(718);
cout << "Cities with area code 718:\n";
for (auto it = range.first; it != range.second; ++it)
    cout <<  (*it).second    << endl;
```

Listing 16.14 demonstrates most of these techniques. It also uses typedef to simplify some of the code writing.

Listing 16.14 **multmap.cpp**

```
// multmap.cpp -- use a multimap
#include <iostream>
#include <string>
#include <map>
#include <algorithm>

typedef int KeyType;
typedef std::pair<const KeyType, std::string> Pair;
typedef std::multimap<KeyType, std::string> MapCode;

int main()
{
```

```
    using namespace std;
    MapCode codes;

    codes.insert(Pair(415, "San Francisco"));
    codes.insert(Pair(510, "Oakland"));
    codes.insert(Pair(718, "Brooklyn"));
    codes.insert(Pair(718, "Staten Island"));
    codes.insert(Pair(415, "San Rafael"));
    codes.insert(Pair(510, "Berkeley"));

    cout << "Number of cities with area code 415: "
         << codes.count(415) << endl;
    cout << "Number of cities with area code 718: "
         << codes.count(718) << endl;
    cout << "Number of cities with area code 510: "
         << codes.count(510) << endl;
    cout << "Area Code   City\n";
    MapCode::iterator it;
    for (it = codes.begin(); it != codes.end(); ++it)
        cout << "    " << (*it).first << "     "
            << (*it).second    << endl;

    pair<MapCode::iterator, MapCode::iterator> range
         = codes.equal_range(718);
    cout << "Cities with area code 718:\n";
    for (it = range.first; it != range.second; ++it)
        cout <<  (*it).second    << endl;

    return 0;
}
```

Here is the output of the program in Listing 16.14:

```
Number of cities with area code 415: 2
Number of cities with area code 718: 2
Number of cities with area code 510: 2
Area Code   City
    415     San Francisco
    415     San Rafael
    510     Oakland
    510     Berkeley
    718     Brooklyn
    718     Staten Island
Cities with area code 718:
Brooklyn
Staten Island
```

Unordered Associative Containers (C++11)

An *unordered associative container* is yet another refinement of the container concept. Like an associative container, an unordered associative container associates a value with a key and uses the key to find the value. The underlying difference is that associative containers are based on tree structures, whereas unordered associative containers are based on another form of data structure called a *hash table*. The intent is to provide containers for which adding and deleting elements is relatively quick and for which there are efficient search algorithms. The four unordered associative containers are called `unordered_set`, `unordered_multiset`, `unordered_map`, and `unordered_multimap`. Appendix G looks a bit further at these additions.

Function Objects (a.k.a. Functors)

Many STL algorithms use *function objects*, also known as *functors*. A *functor* is any object that can be used with `()` in the manner of a function. This includes normal function names, pointers to functions, and class objects for which the `()` operator is overloaded—that is, classes for which the peculiar-looking function `operator()()` is defined. For example, you could define a class like this:

```
class Linear
{
private:
    double slope;
    double y0;
public:
    Linear(double sl_ = 1, double y_ = 0)
        : slope(sl_), y0(y_) {}
    double operator()(double x) {return y0 + slope * x; }
};
```

The overloaded `()` operator then allows you to use `Linear` objects like functions:

```
Linear f1;
Linear f2(2.5, 10.0);
double y1 = f1(12.5);   // right-hand side is f1.operator()(12.5)
double y2 = f2(0.4);
```

Here `y1` is calculated using the expression `0 + 1 * 12.5`, and `y2` is calculated using the expression `10.0 + 2.5 * 0.4`. In the expression `y0 + slope * x`, the values for `y0` and `slope` come from the constructor for the object, and the value of `x` comes from the argument to `operator()()`.

Remember the `for_each` function? It applied a specified function to each member of a range:

```
for_each(books.begin(), books.end(), ShowReview);
```

In general, the third argument could be a functor, not just a regular function. Actually, this raises a question: How do you declare the third argument? You can't declare it as a function pointer because a function pointer specifies the argument type. Because a container can contain just about any type, you don't know in advance what particular argument type should be used. The STL solves that problem by using templates. The `for_each` prototype looks like this:

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
```

The `ShowReview()` prototype is this:

```
void ShowReview(const Review &);
```

This makes the identifier `ShowReview` have the type `void (*)(const Review &)`, so that is the type assigned to the template argument `Function`. With a different function call, the `Function` argument could represent a class type that has an overloaded `()` operator. Ultimately, the `for_each()` code will have an expression using `f()`. In the `ShowReview()` example, `f` is a pointer to a function, and `f()` invokes the function. If the final `for_each()` argument is an object, then `f()` becomes the object that invokes its overloaded `()` operator.

Functor Concepts

Just as the STL defines concepts for containers and iterators, it defines functor concepts:

- A *generator* is a functor that can be called with no arguments.
- A *unary function* is a functor that can be called with one argument.
- A *binary function* is a functor that can be called with two arguments.

For example, the functor supplied to `for_each()` should be a unary function because it is applied to one container element at a time.

Of course, these concepts come with refinements:

- A unary function that returns a `bool` value is a *predicate*.
- A binary function that returns a `bool` value is a *binary predicate*.

Several STL functions require predicate or binary predicate arguments. For example, Listing 16.9 uses a version of `sort()` that takes a binary predicate as its third argument:

```
bool WorseThan(const Review & r1, const Review & r2);
...
sort(books.begin(), books.end(), WorseThan);
```

The `list` template has a `remove_if()` member that takes a predicate as an argument. It applies the predicate to each member in the indicated range, removing those elements for which the predicate returns `true`. For example, the following code would remove all elements greater than 100 from the list `three`:

```
bool tooBig(int n){ return n > 100; }
list<int> scores;
...
scores.remove_if(tooBig);
```

Incidentally, this last example shows where a class functor might be useful. Suppose you want to remove every value greater than 200 from a second list. It would be nice if you could pass the cut-off value to `tooBig()` as a second argument so you could use the function with different values, but a predicate can have but one argument. If, however, you design a `TooBig` class, you can use class members instead of function arguments to convey additional information:

```
template<class T>
class TooBig
{
private:
    T cutoff;
public:
    TooBig(const T & t) : cutoff(t) {}
    bool operator()(const T & v) { return v > cutoff; }
};
```

Here one value (`v`) is passed as a function argument, and the second argument (`cutoff`) is set by the class constructor. Given this definition, you can initialize different `TooBig` objects to different cut-off values to be used in calls to `remove_if()`. Listing 16.15 illustrates the technique.

Listing 16.15 **functor.cpp**

```
// functor.cpp -- using a functor
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>

template<class T>  // functor class defines operator()()
class TooBig
{
private:
    T cutoff;
public:
    TooBig(const T & t) : cutoff(t) {}
    bool operator()(const T & v) { return v > cutoff; }
};

void outint(int n) {std::cout << n << " ";}
```

```
int main()
{
    using std::list;
    using std::cout;
    using std::endl;

    TooBig<int> f100(100); // limit = 100
    int vals[10] = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
    list<int> yadayada(vals, vals + 10); // range constructor
    list<int> etcetera(vals, vals + 10);
 // C++11 can use the following instead
//  list<int> yadayada = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
//  list<int> etcetera {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
    cout << "Original lists:\n";
    for_each(yadayada.begin(), yadayada.end(), outint);
    cout << endl;
    for_each(etcetera.begin(), etcetera.end(), outint);
    cout << endl;
    yadayada.remove_if(f100);               // use a named function object
    etcetera.remove_if(TooBig<int>(200));   // construct a function object
    cout <<"Trimmed lists:\n";
    for_each(yadayada.begin(), yadayada.end(), outint);
    cout << endl;
    for_each(etcetera.begin(), etcetera.end(), outint);
    cout << endl;
    return 0;
}
```

One functor (`f100`) is a declared object, and the second (`TooBig<int>(200)`) is an anonymous object created by a constructor call. Here's the output of the program in Listing 16.15:

```
Original lists:
50 100 90 180 60 210 415 88 188 201
50 100 90 180 60 210 415 88 188 201
Trimmed lists:
50 100 90 60 88
50 100 90 180 60 88 188
```

Suppose that you already have a template function with two arguments:

```
template <class T>
bool tooBig(const T & val, const T & lim)
{
    return val > lim;
}
```

You can use a class to convert it to a one-argument function object:

```
template<class T>
class TooBig2
{
private:
    T cutoff;
public:
    TooBig2(const T & t) : cutoff(t) {}
    bool operator()(const T & v) { return tooBig<T>(v, cutoff); }
};
```

That is, you can use the following:

```
TooBig2<int> tB100(100);
int x;
cin >> x;
if (tB100(x))   // same as if (tooBig(x,100))
    ...
```

So the call `tB100(x)` is the same as `tooBig(x,100)`, but the two-argument function is converted to a one-argument function object, with the second argument being used to construct the function object. In short, the class functor `TooBig2` is a function adapter that adapts a function to meet a different interface.

As noted in the listing, C++11's initializer-list feature simplifies initialization. You can replace

```
int vals[10] = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
list<int> yadayada(vals, vals + 10); // range constructor
list<int> etcetera(vals, vals + 10);
```

with this:

```
list<int> yadayada = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
list<int> etcetera {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
```

Predefined Functors

The STL defines several elementary functors. They perform actions such as adding two values and comparing two values for equality. They are provided to help support STL functions that take functions as arguments. For example, consider the `transform()` function. It has two versions. The first version takes four arguments. The first two arguments are iterators that specify a range in a container. (By now you must be familiar with that approach.) The third is an iterator that specifies where to copy the result. The final is a functor that is applied to each element in the range to produce each new element in the result. For example, consider the following:

```
const int LIM = 5;
double arr1[LIM] = {36, 39, 42, 45, 48};
```

```
vector<double> gr8(arr1, arr1 + LIM);
ostream_iterator<double, char> out(cout, " ");
transform(gr8.begin(), gr8.end(), out, sqrt);
```

This code calculates the square root of each element and sends the resulting values to the output stream. The destination iterator can be in the original range. For example, replacing `out` in this example with `gr8.begin()` would copy the new values over the old values. Clearly, the functor used must be one that works with a single argument.

The second version uses a function that takes two arguments, applying the function to one element from each of two ranges. It takes an additional argument, which comes third in order, identifying the start of the second range. For example, if `m8` were a second `vector<double>` object and if `mean(double, double)` returned the mean of two values, the following would output the average of each pair of values from `gr8` and `m8`:

```
transform(gr8.begin(), gr8.end(), m8.begin(), out, mean);
```

Now suppose you want to add the two arrays. You can't use `+` as an argument because, for type `double`, `+` is a built-in operator, not a function. You could define a function to add two numbers and use it:

```
double add(double x, double y) { return x + y; }
...
transform(gr8.begin(), gr8.end(), m8.begin(), out, add);
```

But then you'd have to define a separate function for each type. It would be better to define a template, except that you don't have to because the STL already has. The `functional` (formerly `function.h`) header defines several template class function objects, including one called `plus<>()`.

Using the `plus<>` class for ordinary addition is possible, if awkward:

```
#include <functional>
...
plus<double> add;  // create a plus<double> object
double y = add(2.2, 3.4); // using plus<double>::operator()()
```

But it makes it easy to provide a function object as an argument:

```
transform(gr8.begin(), gr8.end(), m8.begin(), out, plus<double>() );
```

Here, rather than create a named object, the code uses the `plus<double>` constructor to construct a functor to do the adding. (The parentheses indicate calling the default constructor; what's passed to `transform()` is the constructed function object.)

The STL provides functor equivalents for all the built-in arithmetic, relational, and logical operators. Table 16.12 shows the names for these functor equivalents. They can be used with the C++ built-in types or with any user-defined type that overloads the corresponding operator.

Table 16.12 **Operators and Functor Equivalents**

Operator	Functor Equivalent
+	plus
-	minus
*	multiplies
/	divides
%	modulus
-	negate
==	equal_to
!=	not_equal_to
>	greater
<	less
>=	greater_equal
<=	less_equal
&&	logical_and
||	logical_or
!	logical_not

Caution

Older C++ implementations use the functor name `times` instead of `multiplies`.

Adaptable Functors and Function Adapters

The predefined functors in Table 16.12 are all *adaptable*. Actually, the STL has five related concepts: adaptable generators, adaptable unary functions, adaptable binary functions, adaptable predicates, and adaptable binary predicates.

What makes a functor adaptable is that it carries `typedef` members identifying its argument types and return type. The members are called `result_type`, `first_argument_type`, and `second_argument_type`, and they represent what they sound like. For example, the return type of a `plus<int>` object is identified as `plus<int>::result_type`, and this would be a `typedef` for `int`.

The significance of a functor being adaptable is that it can then be used by function adapter objects, which assume the existence of these `typedef` members. For example, a function with an argument that is an adaptable functor can use the `result_type` member to declare a variable that matches the function's return type.

Indeed, the STL provides function adapter classes that use these facilities. For example, suppose you want to multiply each element of the vector `gr8` by 2.5. That calls for using the `transform()` version with a unary function argument, like the example shown earlier:

```
transform(gr8.begin(), gr8.end(), out, sqrt);
```

The `multiplies()` functor can do the multiplication, but it's a binary function. So you need a function adapter that converts a functor that has two arguments to one that has one argument. The earlier `TooBig2` example shows one way, but the STL has automated the process with the `binder1st` and `binder2nd` classes, which convert adaptable binary functions to adaptable unary functions.

Let's look at `binder1st`. Suppose you have an adaptable binary function object `f2()`. You can create a `binder1st` object that binds a particular value, called `val`, to be used as the first argument to `f2()`:

```
binder1st(f2, val) f1;
```

Then, invoking `f1(x)` with its single argument returns the same value as invoking `f2()` with `val` as its first argument and `f1()`'s argument as its second argument. That is, `f1(x)` is equivalent to `f2(val, x)`, except that it is a unary function instead of a binary function. The `f2()` function has been adapted. Again, this is possible only if `f2()` is an adaptable function.

This might seem a bit awkward. However, the STL provides the `bind1st()` function to simplify using the `binder1st` class. You give it the function name and value used to construct a `binder1st` object, and it returns an object of that type. For example, you can convert the binary function `multiplies()` to a unary function that multiplies its argument by 2.5. Just do this:

```
bind1st(multiplies<double>(), 2.5)
```

Thus, the solution to multiplying every element in `gr8` by 2.5 and displaying the results is this:

```
transform(gr8.begin(), gr8.end(), out,
        bind1st(multiplies<double>(), 2.5));
```

The `binder2nd` class is similar, except that it assigns the constant to the second argument instead of to the first. It has a helper function called `bind2nd` that works analogously to `bind1st`.

Listing 16.16 incorporates some of the recent examples into a short program.

Listing 16.16 **funadap.cpp**

```
// funadap.cpp -- using function adapters
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
#include <functional>

void Show(double);
const int LIM = 6;
int main()
{
    using namespace std;
    double arr1[LIM] = {28, 29, 30, 35, 38, 59};
    double arr2[LIM] = {63, 65, 69, 75, 80, 99};
    vector<double> gr8(arr1, arr1 + LIM);
    vector<double> m8(arr2, arr2 + LIM);
    cout.setf(ios_base::fixed);
    cout.precision(1);
    cout << "gr8:\t";
    for_each(gr8.begin(), gr8.end(), Show);
    cout << endl;
    cout << "m8: \t";
    for_each(m8.begin(), m8.end(), Show);
    cout << endl;

    vector<double> sum(LIM);
    transform(gr8.begin(), gr8.end(), m8.begin(), sum.begin(),
              plus<double>());
    cout << "sum:\t";
    for_each(sum.begin(), sum.end(), Show);
    cout << endl;

    vector<double> prod(LIM);
    transform(gr8.begin(), gr8.end(), prod.begin(),
              bind1st(multiplies<double>(), 2.5));
    cout << "prod:\t";
    for_each(prod.begin(), prod.end(), Show);
    cout << endl;

    return 0;
}
```

```
void Show(double v)
{
    std::cout.width(6);
    std::cout << v << ' ';
}
```

Here is the output of the program in Listing 16.16:

```
gr8:      28.0   29.0   30.0   35.0   38.0   59.0
m8:       63.0   65.0   69.0   75.0   80.0   99.0
sum:      91.0   94.0   99.0  110.0  118.0  158.0
prod:     70.0   72.5   75.0   87.5   95.0  147.5
```

C++11 provides an alternative to function pointers and functors. It's called a *lambda expression*, another topic discussed in Chapter 18.

Algorithms

The STL contains many nonmember functions for working with containers. You've seen a few of them already: `sort()`, `copy()`, `find()`, `for_each()`, `random_shuffle()`, `set_union()`, `set_intersection()`, `set_difference()`, and `transform()`. You've probably noticed that they feature the same overall design, using iterators to identify data ranges to be processed and to identify where results are to go. Some also take a function object argument to be used as part of the data processing.

There are two main generic components to the algorithm function designs. First, they use templates to provide generic types. Second, they use iterators to provide a generic representation for accessing data in a container. Thus, the `copy()` function can work with a container that holds type `double` values in an array, with a container that holds `string` values in a linked list, or with a container that stores user-defined objects in a tree structure, such as is used by `set`. Because pointers are a special case of iterators, STL functions such as `copy()` can be used with ordinary arrays.

The uniform container design allows meaningful relationships between containers of different kinds. For example, you can use `copy()` to copy values from an ordinary array to a `vector` object, from a `vector` object to a `list` object, and from a `list` object to a `set` object. You can use `==` to compare different kinds of containers—for example, `deque` and `vector`. This is possible because the overloaded `==` operator for containers uses iterators to compare contents, so a `deque` object and a `vector` object test as equal if they have the same content in the same order.

Algorithm Groups

The STL divides the algorithm library into four groups:

- Nonmodifying sequence operations
- Mutating sequence operations

- Sorting and related operations
- Generalized numeric operations

The first three groups are described in the `algorithm` (formerly `algo.h`) header file, and the fourth group, being specifically oriented toward numeric data, gets its own header file, called `numeric`. (Formerly, they, too, were in `algo.h`.)

Nonmodifying sequence operations operate on each element in a range. These operations leave a container unchanged. For example, `find()` and `for_each()` belong to this category.

Mutating sequence operations also operate on each element in a range. As the name suggests, however, they can mutate, or change, the contents of a container. The change could be in values or in the order in which the values are stored. The `transform()`, `random_shuffle()`, and `copy()` functions fall into this category.

Sorting and related operations include several sorting functions (including `sort()`) and a variety of other functions, including the set operations.

The numeric operations include functions to sum the contents of a range, calculate the inner product of two containers, calculate partial sums, and calculate adjacent differences. Typically, these are operations that are characteristic of arrays, so `vector` is the container most likely to be used with them.

Appendix G provides a complete summary of these functions.

General Properties of Algorithms

As you've seen again and again in this chapter, STL functions work with iterators and iterator ranges. The function prototype indicates the assumptions made about the iterators. For example, the `copy()` function has this prototype:

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

Because the identifiers `InputIterator` and `OutputIterator` are template parameters, they just as easily could have been `T` and `U`. However, the STL documentation uses the template parameter names to indicate the concept that the parameter models. So this declaration tells you that the range parameters must be input iterators or better and that the iterator indicating where the result goes must be an output parameter or better.

One way of classifying algorithms is on the basis of where the result of the algorithm is placed. Some algorithms do their work in place, and others create copies. For example, when the `sort()` function is finished, the result occupies the same location that the original data did. So `sort()` is an *in-place algorithm*. The `copy()` function, however, sends the result of its work to another location, so it is a *copying algorithm*. The `transform()` function can do both. Like `copy()`, it uses an output iterator to indicate where the results go. Unlike `copy()`, `transform()` allows the output iterator to point to a location in the input range, so it can copy the transformed values over the original values.

Some algorithms come in two versions: an in-place version and a copying version. The STL convention is to append `_copy` to the name of the copying version. The latter version takes an additional output iterator parameter to specify the location to which to copy the outcome. For example, there is a `replace()` function that has this prototype:

```
template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value);
```

It replaces each instance of `old_value` with `new_value`. This occurs in place. Because this algorithm both reads from and writes to container elements, the iterator type has to be `ForwardIterator` or better. The copying version has this prototype:

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
             OutputIterator result,
             const T& old_value, const T& new_value);
```

This time the resulting data is copied to a new location, given by `result`, so the read-only input iterator is sufficient for specifying the range.

Note that `replace_copy()` has an `OutputIterator` return type. The convention for copying algorithms is that they return an iterator pointing to the location one past the last value copied.

Another common variation is that some functions have a version that performs an action conditionally, depending on the result of applying a function to a container element. These versions typically append `_if` to the function name. For example, `replace_if()` replaces an old value with a new value if applying a function to the old value returns the value `true`. Here's the prototype:

```
template<class ForwardIterator, class Predicate class T>
void replace_if(ForwardIterator first, ForwardIterator last,
             Predicate pred, const T& new_value);
```

(Recall that a predicate is a unary function that returns a `bool` value.) There's also a version called `replace_copy_if()`. You can probably figure out what it does and what its prototype is like.

As with `InputIterator`, `Predicate` is a template parameter name and could just as easily be called `T` or `U`. However, the STL chooses to use `Predicate` to remind the user that the actual argument should be a model of the `Predicate` concept. Similarly, the STL uses terms such as `Generator` and `BinaryPredicate` to identify arguments that should model other function object concepts. Keep in mind that although the documentation can remind you what the iterator or functor requirements are, these names are not something the compiler can check. If you use the wrong sort of iterator, you can expect to see a long list of error messages as the compiler tries to instantiate the template.

The STL and the `string` Class

The `string` class, although not part of the STL, is designed with the STL in mind. For example, it has `begin()`, `end()`, `rbegin()`, and `rend()` members. Thus, it can use the STL interface. Listing 16.17 uses the STL to show all the permutations you can form from the letters in a word. A *permutation* is a rearrangement of the order of the elements in a container. The `next_permutation()` algorithm transforms the contents of a range to the next permutation; in the case of a string, the permutations are arranged in ascending alphabetical order. The algorithm returns `true` if it succeeds and `false` if the range already is in the final sequence. To get all the permutations of a range, you should start with the elements in the earliest possible order, and the program uses the STL `sort()` algorithm for that purpose.

Listing 16.17 **strgstl.cpp**

```
// strgstl.cpp -- applying the STL to a string
#include <iostream>
#include <string>
#include <algorithm>

int main()
{
    using namespace std;
    string letters;
    cout << "Enter the letter grouping (quit to quit): ";
    while (cin >> letters && letters != "quit")
    {
        cout << "Permutations of " << letters << endl;
        sort(letters.begin(), letters.end());
        cout << letters << endl;
        while (next_permutation(letters.begin(), letters.end()))
            cout << letters << endl;
        cout << "Enter next sequence (quit to quit): ";
    }
    cout << "Done.\n";
    return 0;
}
```

Here's a sample run of the program in Listing 16.17:

```
Enter the letter grouping (quit to quit): awl
Permutations of awl
alw
awl
law
lwa
wal
```

```
wla
Enter next sequence (quit to quit): all
Permutations of all
all
lal
lla
Enter next sequence (quit to quit): quit
Done.
```

Note that the `next_permutation()` algorithm automatically provides only unique permutations, which is why the output shows more permutations for the word *awl* than for the word *all*, which has duplicate letters.

Functions Versus Container Methods

Sometimes you have a choice between using an STL method and an STL function. Usually, the method is the better choice. First, it should be better optimized for a particular container. Second, being a member function, it can use a template class's memory management facilities and resize a container when needed.

Suppose, for example, that you have a list of numbers and you want to remove all instances of a certain value, say `4`, from the list. If `la` is a `list<int>` object, you can use the list `remove()` method:

```
la.remove(4);  // remove all 4s from the list
```

After this method call, all elements with the value `4` are removed from the list, and the list is automatically resized.

There is also an STL algorithm called `remove()` (see Appendix G). Instead of being invoked by an object, it takes range arguments. So if `lb` is a `list<int>` object, a call to the function could look like this:

```
remove(lb.begin(), lb.end(), 4);
```

However, because this `remove()` is not a member, it can't adjust the size of the list. Instead, it makes sure all the nonremoved items are at the beginning of the list, and it returns an iterator to the new past-the-end value.You can then use this iterator to fix the list size. For example, you can use the list `erase()` method to remove a range that describes the part of the list that is no longer needed. Listing 16.18 shows how this process works.

Listing 16.18 **listrmv.cpp**

```
// listrmv.cpp -- applying the STL to a string
#include <iostream>
#include <list>
#include <algorithm>
```

```
void Show(int);
const int LIM = 10;
int main()
{
    using namespace std;
    int ar[LIM] = {4, 5, 4, 2, 2, 3, 4, 8, 1, 4};
    list<int> la(ar, ar + LIM);
    list<int> lb(la);
    cout << "Original list contents:\n\t";
    for_each(la.begin(), la.end(), Show);
    cout << endl;
    la.remove(4);
    cout << "After using the remove() method:\n";
    cout << "la:\t";
    for_each(la.begin(), la.end(), Show);
    cout << endl;
    list<int>::iterator last;
    last = remove(lb.begin(), lb.end(), 4);
    cout << "After using the remove() function:\n";
    cout << "lb:\t";
    for_each(lb.begin(), lb.end(), Show);
    cout << endl;
    lb.erase(last, lb.end());
    cout << "After using the erase() method:\n";
    cout << "lb:\t";
    for_each(lb.begin(), lb.end(), Show);
    cout << endl;
    return 0;
}

void Show(int v)
{
    std::cout << v << ' ';
}
```

Here's the output of the program in Listing 16.18:

```
Original list contents:
    4 5 4 2 2 3 4 8 1 4
After using the remove() method:
la: 5 2 2 3 8 1
After using the remove() function:
lb: 5 2 2 3 8 1 4 8 1 4
After using the erase() method:
lb: 5 2 2 3 8 1
```

As you can see, the `remove()` method reduces the list `la` from 10 elements to 6 elements. However, list `lb` still contains 10 elements after the `remove()` function is applied to it. The last 4 elements are disposable because each is either the value `4` or a duplicate of a value moved farther to the front of the list.

Although the methods are usually better suited, the nonmethod functions are more general. As you've seen, you can use them on arrays and `string` objects as well as STL containers, and you can use them with mixed container types—for example, to save data from a vector container to a list or a set.

Using the STL

The STL is a library whose parts are designed to work together. The STL components are tools, but they are also building blocks to create other tools. Let's illustrate this with an example. Suppose you want to write a program that lets the user enter words. At the end, you'd like a record of the words as they were entered, an alphabetical list of the words used (capitalization differences ignored), and a record of how many times each word was entered. To keep things simple, let's assume that the input contains no numbers or punctuation.

Entering and saving the list of words is simple enough. Following the example of Listings 16.8 and 16.9, you can create a `vector<string>` object and use `push_back()` to add input words to the vector:

```
vector<string> words;
string input;
while (cin >> input && input != "quit")
    words.push_back(input);
```

What about getting the alphabetic word list? You can use `sort()` followed by `unique()`, but that approach overwrites the original data because `sort()` is an in-place algorithm. There is an easier way that avoids this problem. You can create a `set<string>` object and copy (using an insert iterator) the words from the vector to the set. A set automatically sorts its contents, which means you don't have to call `sort()`, and a set allows only one copy of a key, so that takes the place of calling `unique()`. Wait! The specification called for ignoring the case differences. One way to handle that is to use `transform()` instead of `copy()` to copy data from the vector to the set. For the transformation function, you can use one that converts a string to lowercase:

```
set<string> wordset;
transform(words.begin(), words.end(),
    insert_iterator<set<string> > (wordset, wordset.begin()), ToLower);
```

The `ToLower()` function is easy to write. You just use `transform()` to apply the `tolower()` function to each element in the string, using the string both as source and destination. Remember, `string` objects, too, can use the STL functions. Passing and returning the string as a reference means the algorithm works on the original string without having to make copies. Here's the code for `ToLower()`:

```
string & ToLower(string & st)
{
    transform(st.begin(), st.end(), st.begin(), tolower);
    return st;
}
```

One possible problem is that the `tolower()` function is defined as `int tolower(int)`, and some compilers want the function to match the element type, which is `char`. One solution is to replace `tolower` with `toLower` and to provide the following definition:

```
char toLower(char ch) { return tolower(ch); }
```

To get the number of times each word appears in the input, you can use the `count()` function. It takes a range and a value as arguments, and it returns the number of times the value appears in the range. You can use the `vector` object to provide the range and the `set` object to provide the list of words to count. That is, for each word in the set, you can count how many times it appears in the vector. To keep the resulting count associated with the correct word, you can store the word and the count as a `pair<const string, int>` object in a `map` object. The word will be the key (just one copy), and the count will be the value. This can be done in a single loop:

```
map<string, int> wordmap;
set<string>::iterator  si;
for (si = wordset.begin(); si != wordset.end(); si++)
    wordmap.insert(pair<string, int>(*si, count(words.begin(),
    words.end(), *si)));
```

The `map` class has an interesting feature: You can use array notation with keys that serve as indexes to access the stored values. For example, `wordmap["the"]` would represent the value associated with the key `"the"`, which in this case is the number of occurrences of the string `"the"`. Because the `wordset` container holds all the keys used by `wordmap`, you can use the following code as an alternative and more attractive way of storing results:

```
for (si = wordset.begin(); si != wordset.end(); si++)
    wordmap[*si] = count(words.begin(), words.end(), *si);
```

Because `si` points to a string in the `wordset` container, `*si` is a string and can serve as a key for `wordmap`. This code places both keys and values into the `wordmap` map.

Similarly, you can use the array notation to report results:

```
for (si = wordset.begin(); si != wordset.end(); si++)
    cout << *si << ": " << wordmap[*si] << endl;
```

If a key is invalid, the corresponding value is `0`.

Listing 16.19 puts these ideas together and includes code to display the contents of the three containers (a vector with the input, a set with a word list, and a map with a word count).

Listing 16.19 **usealgo.cpp**

```
//usealgo.cpp -- using several STL elements
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <map>
#include <iterator>
#include <algorithm>
#include <cctype>
using namespace std;

char toLower(char ch) { return tolower(ch); }
string & ToLower(string & st);
void display(const string & s);

int main()
{
    vector<string> words;
    cout << "Enter words (enter quit to quit):\n";
    string input;
    while (cin >> input && input != "quit")
        words.push_back(input);

    cout << "You entered the following words:\n";
    for_each(words.begin(), words.end(), display);
    cout << endl;

    // place words in set, converting to lowercase
    set<string> wordset;
    transform(words.begin(), words.end(),
        insert_iterator<set<string> > (wordset, wordset.begin()),
        ToLower);
    cout << "\nAlphabetic list of words:\n";
    for_each(wordset.begin(), wordset.end(), display);
    cout << endl;

    // place word and frequency in map
    map<string, int> wordmap;
    set<string>::iterator si;
    for (si = wordset.begin(); si != wordset.end(); si++)
        wordmap[*si] = count(words.begin(), words.end(), *si);

    // display map contents
    cout << "\nWord frequency:\n";
```

```
    for (si = wordset.begin(); si != wordset.end(); si++)
        cout << *si << ": " << wordmap[*si] << endl;

    return 0;
}

string & ToLower(string & st)
{
    transform(st.begin(), st.end(), st.begin(), toLower);
    return st;
}

void display(const string & s)
{
    cout << s << " ";
}
```

Here is a sample run of the program in Listing 16.19:

```
Enter words (enter quit to quit):
The dog saw the cat and thought the cat fat
The cat thought the cat perfect
quit
You entered the following words:
The dog saw the cat and thought the cat fat The cat thought the cat perfect

Alphabetic list of words:
and cat dog fat perfect saw the thought

Word frequency:
and: 1
cat: 4
dog: 1
fat: 1
perfect: 1
saw: 1
the: 5
thought: 2
```

The moral here is that your attitude when using the STL should be to avoid writing as much code as possible. STL's generic and flexible design should save you lots of work. Also the STL designers are algorithm people who are very much concerned with efficiency. So the algorithms are well chosen and inline.

Other Libraries

C++ provides some other class libraries that are more specialized than the examples covered so far in this chapter. For instance, the `complex` header file provides a `complex` class template for complex numbers, with specializations for `float`, `long`, and `long double`. The class provides standard complex number operations, along with standard functions that can be used with complex numbers. The C++11 random header file extends random number functionality.

Chapter 14 introduced another example, the `valarray` template class, supported by the `valarray` header file. This class template is designed to represent numeric arrays and provides support for a variety of numeric array operations, such as adding the contents of one array to another, applying math functions to each element of an array and applying linear algebra operations to arrays.

`vector`, `valarray`, and `array`

Perhaps you are wondering why C++ has three array templates: `vector`, `valarray`, and `array`. These classes were developed by different groups for different purposes. The `vector` template class is part of a system of container classes and algorithms. The `vector` class supports container-oriented activities, such as sorting, insertion, rearrangement, searching, transferring data to other containers, and other manipulations. The `valarray` class template, on the other hand, is oriented toward numeric computation, and it is not part of the STL. It doesn't have `push_back()` and `insert()` methods, for example, but it does provide a simple, intuitive interface for many mathematical operations. Finally, `array` is designed as a substitute for the built-in array type, combining the compactness and efficiency of that type with a better, safer interface. Being of fixed size, `array` doesn't support `push_back()` and `insert()`, but it does offer several other STL methods. These include `begin()`, `end()`, `rbegin()`, and `rend()`, making it easy to apply STL algorithms to `array` objects.

Suppose, for example, that you have these declarations:

```
vector<double> ved1(10), ved2(10), ved3(10);
array<double, 10> vod1, vod2, vod3;
valarray<double> vad1(10), vad2(10), vad3(10);
```

Furthermore, assume that `ved1`, `ved2`, `vod1`, `vod2`, `vad1`, and `vad2` all acquire suitable values. Suppose you want to assign the sum of the first elements of two arrays to the first element of a third array, and so on. With the `vector` class, you would do this:

```
transform(ved1.begin(), ved1.end(), ved2.begin(), ved3.begin(),
          plus<double>());
```

You can do the same with the `array` class:

```
transform(vod1.begin(), vod1.end(), vod2.begin(), vod3.begin(),
          plus<double>());
```

However, the valarray class overloads all the arithmetic operators to work with valarray objects, so you would use this:

```
vad3 = vad1 + vad2;    // + overloaded
```

Similarly, the following would result in each element of vad3 being the product of the corresponding elements in vad1 and vad2:

```
vad3 = vad1 * vad2;    // * overloaded
```

Suppose you want to replace every value in an array with that value multiplied by 2.5. The STL approach is this:

```
transform(ved3.begin(), ved3.end(), ved3.begin(),
          bind1st(multiplies<double>(), 2.5));
```

The valarray class overloads multiplying a valarray object by a single value, and it also overloads the various computed assignment operators, so you could use either of the following:

```
vad3 = 2.5 * vad3;     // * overloaded
vad3 *= 2.5;           // *= overloaded
```

Suppose you want to take the natural logarithm of every element of one array and store the result in the corresponding element of a second array. The STL approach is this:

```
transform(ved1.begin(), ved1.end(), ved3.begin(),
          log);
```

The valarray class overloads the usual math functions to take a valarray argument and to return a valarray object, so you can use this:

```
vad3 = log(vad1);     // log() overloaded
```

Or you could use the apply() method, which also works for non-overloaded functions:

```
vad3 = vad1.apply(log);
```

The apply() method doesn't alter the invoking object; instead, it returns a new object that contains the resulting values.

The simplicity of the valarray interface is even more apparent when you do a multistep calculation:

```
vad3 = 10.0* ((vad1 + vad2) / 2.0 + vad1 * cos(vad2));
```

The vector-STL version is left as an exercise for the motivated reader.

The valarray class also provides a sum() method that sums the contents of a valarray object, a size() method that returns the number of elements, a max() method that returns the largest value in an object, and a min() method that returns the smallest value.

As you can see, valarray has a clear notational advantage over vector for mathematical operations, but it is also much less versatile. The valarray class does have a resize()

method, but there's no automatic resizing of the sort you get when you use the `vector` `push_back()` method. There are no methods for inserting values, searching, sorting, and the like. In short, the `valarray` class is more limited than the `vector` class, but its narrower focus allows a much simpler interface.

Does the simpler interface that `valarray` provides translate to better performance? In most cases, no. The simple notation is typically implemented with the same sort of loops you would use with ordinary arrays. However, some hardware designs allow vector operations in which the values in an array are loaded simultaneous into an array of registers and then processed simultaneously. In principle, `valarray` operations could be implemented to take advantage of such designs.

Can you use the STL with `valarray` objects? Answering this question provides a quick review of some STL principles. Suppose you have a `valarray<double>` object that has 10 elements:

```
valarray<double> vad(10);
```

After the array has been filled with numbers, can you, say, use the STL sort function on it? The `valarray` class doesn't have `begin()` and `end()` methods, so you can't use them as the range arguments:

```
sort(vad.begin(), vad.end());  // NO, no begin(), end()
```

Also `vad` is an object, not a pointer, so you can't imitate ordinary array usage and provide `vad` and `vad + 10`, as the following code attempts to do:

```
sort(vad, vad + 10);  // NO, vad an object, not an address
```

You can use the address operator:

```
sort(&vad[0], &vad[10]);  // maybe?
```

But the behavior of using a subscript one past the end is undefined for `valarray`. This doesn't necessarily mean using `&vad[10]` won't work. (In fact, it did work for all six compilers used to test this code.) But it does mean that it might not work. For the code to fail, you probably would need a very unlikely circumstance, such as the array being butted against the end of the block of memory set aside for the heap. But, if a $385-million mission depended on your code, you might not want to risk that failure.

C++11 remedies the situation by providing `begin()` and `end()` template functions that take a `valarray` object as an argument. So you would use `begin(vad)` instead of `vad.begin()`. These functions return values that are compatible with STL range requirements:

```
sort(begin(vad), end(vad)); // C++11 fix!
```

Listing 16.20 illustrates some of the relative strengths of the `vector` and `valarray` classes. It uses `push_back()` and the automatic sizing feature of `vector` to collect data. Then after sorting the numbers, the program copies them from the `vector` object to a `valarray` object of the same size and does a few math operations.

Listing 16.20 **valvect.cpp**

```
// valvect.cpp -- comparing vector and valarray
#include <iostream>
#include <valarray>
#include <vector>
#include <algorithm>
int main()
{
    using namespace std;
    vector<double> data;
    double temp;

    cout << "Enter numbers (<=0 to quit):\n";
    while (cin >> temp && temp > 0)
        data.push_back(temp);
    sort(data.begin(), data.end());
    int size = data.size();
    valarray<double> numbers(size);
    int i;
    for (i = 0; i < size; i++)
        numbers[i] = data[i];
    valarray<double> sq_rts(size);
    sq_rts = sqrt(numbers);
    valarray<double> results(size);
    results = numbers + 2.0 * sq_rts;
    cout.setf(ios_base::fixed);
    cout.precision(4);
    for (i = 0; i < size; i++)
    {
        cout.width(8);
        cout << numbers[i] << ": ";
        cout.width(8);
        cout << results[i] << endl;
    }
    cout << "done\n";
    return 0;
}
```

Here is a sample run of the program in Listing 16.20:

```
Enter numbers (<=0 to quit):
3.3 1.8 5.2 10 14.4 21.6 26.9 0
  1.8000:   4.4833
  3.3000:   6.9332
  5.2000:   9.7607
 10.0000:  16.3246
```

```
 14.4000:  21.9895
 21.6000:  30.8952
 26.9000:  37.2730
done
```

The `valarray` class has many features besides the ones discussed so far. For example, if `numbers` is a `valarray<double>` object, the following statement creates an array of `bool` values, with `vbool[i]` set to the value of `numbers[i] > 9`—that is, to `true` or `false`:

```
valarray<bool> vbool = numbers > 9;
```

There are extended versions of subscripting. Let's look at one—the `slice` class. A `slice` class object can be used as an array index, in which case it represents, in general, not just one value but a subset of values. A `slice` object is initialized to three integer values: the *start*, the *number*, and the *stride*. The *start* indicates the index of the first element to be selected, the *number* indicates the number of elements to be selected, and the *stride* represents the spacing between elements. For example, the object constructed by `slice(1,4,3)` means select the four elements whose indexes are 1, 4, 7, and 10. That is, start with the start element, add the stride to get the next element, and so on until four elements are selected. If, say, `varint` is a `vararray<int>` object, then the following statement would set elements 1, 4, 7, and 10 to the value `10`:

```
varint[slice(1,4,3)] = 10;  // set selected elements to 10
```

This special subscripting facility allows you to use a one-dimensional `valarray` object to represent two-dimensional data. For example, suppose you want to represent an array with 4 rows and 3 columns. You can store the information in a 12-element `valarray` object. Then a `slice(0,3,1)` object used as a subscript would represent elements 0, 1, and 2—that is, the first row. Similarly, a `slice(0,4,3)` subscript would represent elements 0, 3, 6, and 9—that is, the first column. Listing 16.21 illustrates some features of `slice`.

Listing 16.21 **vslice.cpp**

```
// vslice.cpp -- using valarray slices
#include <iostream>
#include <valarray>
#include <cstdlib>

const int SIZE = 12;
typedef std::valarray<int> vint;     // simplify declarations
void show(const vint & v, int cols);
int main()
{
    using std::slice;                // from <valarray>
    using std::cout;
    vint valint(SIZE);               // think of as 4 rows of 3

    int i;
```

```
    for (i = 0; i < SIZE; ++i)
        valint[i] = std::rand() % 10;
    cout << "Original array:\n";
    show(valint, 3);                  // show in 3 columns
    vint vcol(valint[slice(1,4,3)]); // extract 2nd column
    cout << "Second column:\n";
    show(vcol, 1);                    // show in 1 column
    vint vrow(valint[slice(3,3,1)]); // extract 2nd row
    cout << "Second row:\n";
    show(vrow, 3);
    valint[slice(2,4,3)]  = 10;       // assign to 2nd column
    cout << "Set last column to 10:\n";
    show(valint, 3);
    cout << "Set first column to sum of next two:\n";
    // + not defined for slices, so convert to valarray<int>
    valint[slice(0,4,3)]  = vint(valint[slice(1,4,3)])
                            + vint(valint[slice(2,4,3)]);
    show(valint, 3);
    return 0;
}

void show(const vint & v, int cols)
{
    using std::cout;
    using std::endl;

    int lim = v.size();
    for (int i = 0; i < lim; ++i)
    {
        cout.width(3);
        cout << v[i];
        if (i % cols == cols - 1)
            cout << endl;
        else
            cout << ' ';
    }
    if (lim % cols != 0)
        cout << endl;
}
```

The + operator is defined for valarray objects, such as valint, and it's defined for a single int element, such as valint[1]. But as the code in Listing 16.21 notes, the + operator isn't defined for slice-subscripted valarray units, such as valint[slice(1,4,3)]. Therefore, the program constructs full objects from the slices to enable addition:

```
vint(valint[slice(1,4,3)])    // calls a slice-based constructor
```

The `valarray` class provides constructors just for this purpose.

Here is a sample run of the program in Listing 16.21:

```
Original array:
  0   3   3
  2   9   0
  8   2   6
  6   9   1
Second column:
  3
  9
  2
  9
Second row:
  2   9   0
Set last column to 10:
  0   3  10
  2   9  10
  8   2  10
  6   9  10
Set first column to sum of next two:
 13   3  10
 19   9  10
 12   2  10
 19   9  10
```

Because values are set using `rand()`, different implementations of `rand()` will result in different values.

There's more, including the `gslice` class to represent multidimensional slices, but this should be enough to give you a sense of what `valarray` is about.

The `initializer_list` Template (C++11)

The `initializer_list` template is another C++11 addition to the C++ library. You can use the initializer-list syntax to initialize an STL container to a list of values:

```
std::vector<double> payments {45.99, 39.23, 19.95, 89.01};
```

This would create a container for four elements and initialize the elements to the four values in the list. What makes this possible is that the container classes now have constructors taking an `initializer_list<T>` argument. A `vector<double>` object, for example, has a constructor that accepts an `initializer_list<double>` argument, and the previous declaration is the same as this:

```
std::vector<double> payments({45.99, 39.23, 19.95, 89.01});
```

Here, the list is written explicitly as a constructor argument.

Normally, as part of the C++11 universal initialization syntax, you can invoke a class constructor using {} instead of () notation:

```
shared_ptr<double> pd {new double}; // ok to use {} instead of ()
```

But this would create problems if there also is an initializer_list constructor:

```
std::vector<int> vi{10};    // ??
```

Which constructor does this invoke?

```
std::vector<int> vi(10);    // case A: 10 uninitialized elements
std::vector<int> vi({10});  // case B: 1 element set to 10
```

The answer is that if the class does have a constructor accepting an initializer_list argument, then using the {} syntax invokes that particular constructor. So in this example, case B applies.

The initializer_list elements should all be of one type. However, the compiler will do conversions to match the type:

```
std::vector<double> payments {45.99, 39.23, 19, 89};
// same as std::vector<double> payments {45.99, 39.23, 19.0, 89.0};
```

Here, because the vector element type is double, the list is type initializer_list<double>, and 19 and 89 are converted to double.

The usual list restrictions on narrowing apply:

```
std::vector<int> values = {10, 8, 5.5};  // narrowing, compile-time error
```

Here, the element type is int, and the implied conversion of 5.5 to int is not allowed.

It doesn't make sense to provide an initializer_list constructor unless the class is meant to handle lists of varying sizes. For instance, you don't want an initializer_list constructor for a class taking a fixed number of values. For example, the following declaration does not provide an initializer_list constructor for the three data members:

```
class Position
{
private:
    int x;
    int y;
    int z;
public:
    Position(int xx = 0, int yy  = 0, int zz = 0)
             : x(xx), y(yy), z(zz) {}
    // no initializer_list constructor
    ...
};
```

This allows you to use the {} syntax with the `Position(int,int,int)` constructor:

```
Position A = {20, -3};  // uses Position(20,-3,0)
```

Using `initializer_list`

You can use `initializer_list` objects in your code by including the `initializer_list` header file. The template class has `begin()` and `end()` members, and you can use them to access list elements. It also has a `size()` member that returns the number of elements. Listing 16.22 shows a simple example using `initializer_list`. It requires a compiler that supports this C++11 feature.

Listing 16.22 **ilist.cpp**

```
// ilist.cpp -- use initializer_list (C++11 feature)
#include <iostream>
#include <initializer_list>

double sum(std::initializer_list<double> il);
double average(const std::initializer_list<double> & ril);

int main()
{
    using std::cout;

    cout << "List 1: sum = " << sum({2,3,4})
         <<", ave = " << average({2,3,4}) << '\n';
    std::initializer_list<double> dl = {1.1, 2.2, 3.3, 4.4, 5.5};
    cout << "List 2: sum = " << sum(dl)
         <<", ave = " << average(dl) << '\n';
    dl = {16.0, 25.0, 36.0, 40.0, 64.0};
    cout << "List 3: sum = " << sum(dl)
         <<", ave = " << average(dl) << '\n';
    return 0;
}

double sum(std::initializer_list<double> il)
{
    double tot = 0;
    for (auto p = il.begin(); p !=il.end(); p++)
        tot += *p;
    return tot;
}

double average(const std::initializer_list<double> & ril)
{
    double tot = 0;
```

```
    int n = ril.size();
    double ave = 0.0;
    if (n > 0)
    {
        for (auto p = ril.begin(); p !=ril.end(); p++)
            tot += *p;
        ave = tot / n;
    }
    return ave;
}
```

Here's a sample run:

```
List 1: sum = 9, ave = 3
List 2: sum = 16.5, ave = 3.3
List 3: sum = 181, ave = 36.2
```

Program Notes

You can pass an `initializer_list` object by value or by reference, as shown by `sum()` and `average()`. The object itself is small, typically two pointers (one to the beginning and one past end) or a pointer to the beginning and an integer representing the size, so the choice is not a major performance issue. (The STL passes them by value.)

The function argument can be a list literal, like `{2,3,4}`, or it can be a list variable, like `dl`.

The iterator types for `initializer_list` are `const`, so you can't change the values in a list:

```
*dl.begin() = 2011.6;                // not allowed
```

But, as Listing 16.22 shows, you can attach a list variable to a different list:

```
dl = {16.0, 25.0, 36.0, 40.0, 64.0};  // allowed
```

However, the intended use of the `initializer_list` class is to pass a list of values to a constructor or some other function.

Summary

C++ includes a powerful set of libraries that provide solutions to many common programming problems and the tools to simplify many more problems. The `string` class provides a convenient means to handle strings as objects as well as automatic memory management and a host of methods and functions for working with strings. For example, these methods and functions allow you to concatenate strings, insert one string into another, reverse a string, search a string for characters or substrings, and perform input and output operations.

Smart pointer templates such as `auto_ptr` and C++11's `shared_ptr` and `unique_ptr` make it easier to manage memory allocated by `new`. If you use one of these smart pointers instead of a regular pointer to hold the address returned by `new`, you don't have to remember to use the `delete` operator later. When the smart pointer object expires, its destructor calls the `delete` operator automatically.

The STL is a collection of container class templates, iterator class templates, function object templates, and algorithm function templates that feature a unified design based on generic programming principles. The algorithms use templates to make them generic in terms of type of stored object and an iterator interface to make them generic in terms of the type of container. Iterators are generalizations of pointers.

The STL uses the term *concept* to denote a set of requirements. For example, the concept of forward iterators includes the requirements that a forward iterator object can be dereferenced for reading and writing and that it can be incremented. Actual implementations of the concept are said to *model* the concept. For example, the forward iterator concept could be modeled by an ordinary pointer or by an object designed to navigate a linked list. Concepts based on other concepts are termed *refinements*. For example, the bidirectional iterator is a refinement of the forward iterator concept.

Container classes, such as `vector` and `set`, are models of container concepts, such as containers, sequences, and associative containers. The STL defines several container class templates: `vector`, `deque`, `list`, `set`, `multiset`, `map`, `multimap`, and `bitset`. It also defines the adapter class templates `queue`, `priority_queue`, and `stack`; these classes adapt an underlying container class to give it the characteristic interface suggested by the adapter class template name. Thus, `stack`, although based, by default, on `vector`, allows insertion and removal only at the top of the stack. C++11 adds `forward_list`, `unordered_set`, `unordered_multiset`, `unordered_map`, and `unordered_multimap`.

Some algorithms are expressed as container class methods, but the bulk are expressed as general, nonmember functions. This is made possible by using iterators as the interface between containers and algorithms. One advantage to this approach is that there needs to be just one `for_each()` or `copy()` function, and so on, instead of a separate version for each container. A second advantage is that STL algorithms can be used with non-STL containers, such as ordinary arrays, `string` objects, `array` objects, and any classes you design consistent with the STL iterator and container idiom.

Both containers and algorithms are characterized by the type of iterator they provide or need. You should check that a container features an iterator concept that supports the algorithm's needs. For example, the `for_each()` algorithm uses an input iterator, whose minimal requirements are met by all the STL container class types. But `sort()` requires random access iterators, which not all container classes support. A container class may offer a specialized method as an option if it doesn't meet the requirements for a particular algorithm. For example, the `list` class has a `sort()` method that is based on bidirectional iterators, so it can use that method instead of the general function.

The STL also provides function objects, or functors, that are classes for which the `()` operator is overloaded—that is, for which the `operator()()` method is defined.

Objects of such classes can be invoked by using function notation but can carry additional information. Adaptable functors, for example, have typedef statements that identify the argument types and the return value type for the functor. This information can be used by other components, such as function adapters.

By representing common container types and providing a variety of common operations implemented with efficient algorithms, all done in a generic manner, the STL provides an excellent source of reusable code. You may be able to solve a programming problem directly with the STL tools, or you may be able to use them as building blocks to construct the solution you need.

The complex and valarray template classes support numeric operations for complex numbers and arrays.

Chapter Review

1. Consider the following class declaration:

```
class RQ1
{
private:
    char * st;       // points to C-style string
public:
    RQ1() { st = new char [1]; strcpy(st,""); }
    RQ1(const char * s)
    {st = new char [strlen(s) + 1]; strcpy(st, s); }
    RQ1(const RQ1 & rq)
    {st = new char [strlen(rq.st) + 1]; strcpy(st, rq.st); }
    ~RQ1() {delete [] st};
    RQ & operator=(const RQ & rq);
    // more stuff
};
```

   Convert this to a declaration that uses a string object instead. What methods no longer need explicit definitions?

2. Name at least two advantages string objects have over C-style strings in terms of ease-of-use.

3. Write a function that takes a reference to a string object as an argument and that converts the string object to all uppercase.

4. Which of the following are not examples of correct usage (conceptually or syntactically) of auto_ptr? (Assume that the needed header files have been included.)

```
auto_ptr<int> pia(new int[20]);
auto_ptr<string> (new string);
int rigue = 7;
auto_ptr<int>pr(&rigue);
auto_ptr dbl (new double);
```

5. If you could make the mechanical equivalent of a stack that held golf clubs instead of numbers, why would it (conceptually) be a bad golf bag?
6. Why would a `set` container be a poor choice for storing a hole-by-hole record of your golf scores?
7. Because a pointer is an iterator, why didn't the STL designers simply use pointers instead of iterators?
8. Why didn't the STL designers simply define a base iterator class, use inheritance to derive classes for the other iterator types, and express the algorithms in terms of those iterator classes?
9. Give at least three examples of convenience advantages that a `vector` object has over an ordinary array.
10. If Listing 16.9 were implemented with `list` instead of `vector`, what parts of the program would become invalid? Could the invalid part be fixed easily? If so, how?
11. Consider the `TooBig` functor in Listing 16.15. What does the following code do, and what values get assigned to `bo`?

```
bool bo = TooBig<int>(10)(15);
```

Programming Exercises

1. A *palindrome* is a string that is the same backward as it is forward. For example, "tot" and "otto" are rather short palindromes. Write a program that lets a user enter a string and that passes to a `bool` function a reference to the string. The function should return `true` if the string is a palindrome and `false` otherwise. At this point, don't worry about complications such as capitalization, spaces, and punctuation. That is, this simple version should reject "Otto" and "Madam, I'm Adam." Feel free to scan the list of string methods in Appendix F for methods to simplify the task.
2. Do the same problem as given in Programming Exercise 1 but do worry about complications such as capitalization, spaces, and punctuation. That is, "Madam, I'm Adam" should test as a palindrome. For example, the testing function could reduce the string to "madamimadam" and then test whether the reverse is the same. Don't forget the useful `cctype` library. You might find an STL function or two useful although not necessary.
3. Redo Listing 16.3 so that it gets it words from a file. One approach is to use a `vector<string>` object instead of an array of `string`. Then you can use `push_back()` to copy how ever many words are in your data file into the `vector<string>` object and use the `size()` member to determine the length of the word list. Because the program should read one word at a time from the file, you should use the `>>` operator rather than `getline()`. The file itself should contain words separated by spaces, tabs, or new lines.

4. Write a function with an old-style interface that has this prototype:

```
int reduce(long ar[], int n);
```

   The actual arguments should be the name of an array and the number of elements in the array. The function should sort an array, remove duplicate values, and return a value equal to the number of elements in the reduced array. Write the function using STL functions. (If you decide to use the general `unique()` function, note that it returns the end of the resulting range.) Test the function in a short program.

5. Do the same problem as described in Programming Exercise 4, except make it a template function:

```
template <class T>
int reduce(T ar[], int n);
```

   Test the function in a short program, using both a `long` instantiation and a `string` instantiation.

6. Redo the example shown in Listing 12.12, using the STL `queue` template class instead of the `Queue` class described in Chapter 12.

7. A common game is the lottery card. The card has numbered spots of which a certain number are selected at random. Write a `Lotto()` function that takes two arguments. The first should be the number of spots on a lottery card, and the second should be the number of spots selected at random. The function should return a `vector<int>` object that contains, in sorted order, the numbers selected at random. For example, you could use the function as follows:

```
vector<int> winners;
winners = Lotto(51,6);
```

   This would assign to `winners` a vector that contains six numbers selected randomly from the range 1 through 51. Note that simply using `rand()` doesn't quite do the job because it may produce duplicate values. Suggestion: Have the function create a vector that contains all the possible values, use `random_shuffle()`, and then use the beginning of the shuffled vector to obtain the values. Also write a short program that lets you test the function.

8. Mat and Pat want to invite their friends to a party. They ask you to write a program that does the following:

   - Allows Mat to enter a list of his friends' names. The names are stored in a container and then displayed in sorted order.
   - Allows Pat to enter a list of her friends' names. The names are stored in a second container and then displayed in sorted order.
   - Creates a third container that merges the two lists, eliminates duplicates, and displays the contents of this container.

9. Compared to an array, a linked list features easier addition and removal of elements but is slower to sort. This raises a possibility: Perhaps it might be faster to copy a list to an array, sort the array, and copy the sorted result back to the list than to simply use the list algorithm for sorting. (But it also could use more memory.) Test the speed hypothesis with the following approach:

    a. Create a large `vector<int>` object `vi0`, using `rand()` to provide initial values.

    b. Create a second `vector<int>` object `vi` and a `list<int>` object `li` of the same size as the original and initialize them to values in the original vector.

    c. Time how long the program takes to sort `vi` using the STL `sort()` algorithm, then time how long it takes to sort `li` using the `list sort()` method.

    d. Reset `li` to the unsorted contents of `vi0`. Time the combined operation of copying `li` to `vi`, sorting `vi`, and copying the result back to `li`.

    To time these operations, you can use `clock()` from the `ctime` library. As in Listing 5.14, you can use this statement to start the first timing:

    ```
    clock_t start = clock();
    ```

    Then use the following at the end of the operation to get the elapsed time:

    ```
    clock_t end = clock();
    cout << (double)(end - start)/CLOCKS_PER_SEC;
    ```

    This is by no means a definitive test because the results will depend on a variety of factors, including available memory, whether multiprocessing is going on, and the size of the array or list. (One would expect the relative efficiency advantage of the array over the list to increase with the number of elements being sorted.) Also if you have a choice between a default build and a release build, use the release build for the measurement. With today's speedy computers, you probably will need to use as large an array as possible to get meaningful readings. You might try, for example, 100,000 elements, 1,000,000 elements, and 10,000,000 elements.

10. Modify Listing 16.9 (`vect3.cpp`) as follows:

    a. Add a `price` member to the `Review` structure.

    b. Instead of using a `vector` of `Review` objects to hold the input, use a `vector` of `shared_ptr<Review>` objects. Remember that a `shared_ptr` has to be initialized with a pointer returned by `new`.

    c. Follow the input stage with a loop that allows the user the following options for displaying books: in original order, in alphabetical order, in order of increasing ratings, in order of decreasing ratings, in order of increasing price, in order of decreasing price, and quitting.

Here's one possible approach. After getting the initial input, create another vector of shared_ptrs initialized to the original array. Define an operator<() function that compares pointed-to structures and use it to sort the second vector so that the shared_ptrs are in the order of the book names stored in the pointed-to objects. Repeat the process to get vectors of shared_ptrs sorted by rating and by price. Note that rbegin() and rend() save you the trouble of also creating vectors of reversed order.