

8.2 THE STANDARD `string` CLASS

I try to catch every sentence, every word you and I say, and quickly lock all these sentences and words away in my literary storehouse because they might come in handy.

ANTON CHEKHOV, *The Seagull*

In Section 8.1, we introduced C strings. These C strings were simply arrays of characters terminated with the null character `'\0'`. In order to manipulate these C strings, you needed to worry about all the details of handling arrays. For example, when you want to add characters to a C string and there is not enough room in the array, you must create another array to hold this longer string of characters. In short, C strings require the programmer to keep track of all the low-level details of how the C strings are stored in memory. This is a lot of extra work and a source of programmer errors. The ANSI/ISO standard for C++ specified that C++ must also have a class `string` that allows the programmer to treat strings as a basic data type without needing to worry about implementation details. In this section we introduce you to this `string` type.

Introduction to the Standard Class `string`

The class `string` is defined in the library whose name is also `<string>`, and the definitions are placed in the `std` namespace. So, in order to use the class `string`, your code must contain the following (or something more or less equivalent):

```
#include <string>
using namespace std;
```

+ operator does
concatenation

The class `string` allows you to treat string values and string expressions very much like values of a simple type. You can use the `=` operator to assign a value to a string variable, and you can use the `+` sign to concatenate two strings. For example, suppose `s1`, `s2`, and `s3` are objects of type `string` and both `s1` and `s2` have string values. Then `s3` can be set equal to the concatenation of the string value in `s1` followed by the string value in `s2` as follows:

```
s3 = s1 + s2;
```

There is no danger of `s3` being too small for its new string value. If the sum of the lengths of `s1` and `s2` exceeds the capacity of `s3`, then more space is automatically allocated for `s3`.

As we noted earlier in this chapter, quoted strings are really C strings and so they are not literally of type `string`. However, C++ provides automatic type casting of quoted strings to values of type `string`. So, you can use quoted strings as if they were literal values of type `string`, and we (and most others) will often refer to quoted strings as if they were values of type `string`. For example,

```
s3 = "Hello Mom!";
```

sets the value of the string variable `s3` to a string object with the same characters as in the C string `"Hello Mom!"`.

The class `string` has a default constructor that initializes a `string` object to the empty string. The class `string` also has a second constructor that takes one argument that is a standard C string and so can be a quoted string. This second constructor initializes the `string` object to a value that represents the same string as its C-string argument. For example,

```
string phrase;  
string noun("ants");
```

The first line declares the `string` variable `phrase` and initializes it to the empty string. The second line declares `noun` to be of type `string` and initializes it to a `string` value equivalent to the C string `"ants"`. Most programmers when talking loosely would say that “`noun` is initialized to `"ants"`,” but there really is a type conversion here. The quoted string `"ants"` is a C string, not a value of type `string`. The variable `noun` receives a `string` value that has the same characters as `"ants"` in the same order as `"ants"`, but the `string` value is not terminated with the null character `'\0'`. In fact, in theory at least, you do not know or care whether the `string` value of `noun` is even stored in an array, as opposed to some other data structure.

There is an alternate notation for declaring a `string` variable and invoking a constructor. The following two lines are exactly equivalent:

```
string noun("ants");  
string noun = "ants";
```

These basic details about the class `string` are illustrated in Display 8.4. Note that, as illustrated there, you can output `string` values using the operator `<<`.

Consider the following line from Display 8.4:

```
phrase = "I love " + adjective + " " + noun + "!";
```

C++ must do a lot of work to allow you to concatenate strings in this simple and natural fashion. The `string` constant `"I love"` is not an object of type `string`. A `string` constant like `"I love"` is stored as a C string (in other words, as a null-terminated array of characters). When C++ sees `"I love"` as an argument to `+`, it finds the definition (or overloading) of `+` that applies to a value such as `"I love"`. There are overloads of the `+` operator that have a C string on the left and a `string` on the right, as well as the reverse of this positioning. There is even a version that has a C string on both sides of the `+` and produces a `string` object as the value returned. Of course, there is also the overloading you expect, with the type `string` for both operands.

C++ did not really need to provide all those overloading cases for `+`. If these overloads were not provided, C++ would look for a constructor that could perform a type conversion to convert the C string `"I love"` to a value for which `+` did apply. In this case, the constructor with the one C-string parameter would perform just such a conversion. However, the extra overloads are presumably more efficient.

The class `string` is often thought of as a modern replacement for C strings. However, in C++ you cannot easily avoid also using C strings when you program with the class `string`.

Converting
C-string constants
to the type
`string`

DISPLAY 8.4 Program Using the Class string

```
1  //Demonstrates the standard class string.
2  #include <iostream>
3  #include <string>
4  using namespace std;

5  int main( )
6  {
7      string phrase;
8      string adjective("fried"), noun("ants");
9      string wish = "Bon appetit!";

10     phrase = "I love " + adjective + " " + noun + "!";
11     cout << phrase << endl
12          << wish << endl;
13     return 0;
14 }
```

Initialized to the empty string

Two ways of initializing a string variable

Sample Dialogue

```
I love fried ants!
Bon appetit!
```

The Class string

The class `string` can be used to represent values that are strings of characters. The class `string` provides more versatile string representation than the C strings discussed in Section 8.1.

The class `string` is defined in the library that is also named `<string>`, and its definition is placed in the `std` namespace. So, programs that use the class `string` should contain the following (or something more or less equivalent):

```
#include <string>
using namespace std;
```

The class `string` has a default constructor that initializes the `string` object to the empty string and a constructor that takes a C string as an argument and initializes the `string` object to a value that represents the string given as the argument. For example:

```
string s1, s2("Hello");
```

I/O with the Class `string`

You can use the insertion operator `<<` and `cout` to output `string` objects just as you do for data of other types. This is illustrated in Display 8.4. Input with the class `string` is a bit more subtle.

The extraction operator `>>` and `cin` work the same for `string` objects as for other data, but remember that the extraction operator ignores initial whitespace and stops reading when it encounters more whitespace. This is as true for strings as it is for other data. For example, consider the following code:

```
string s1, s2;
cin >> s1;
cin >> s2;
```

If the user types in

```
May the hair on your toes grow long and curly!
```

then `s1` will receive the value "May" with any leading (or trailing) whitespace deleted. The variable `s2` receives the string "the". Using the extraction operator `>>` and `cin`, you can only read in words; you cannot read in a line or other string that contains a blank. Sometimes this is exactly what you want, but sometimes it is not at all what you want.

If you want your program to read an entire line of input into a variable of type `string`, you can use the function `getline`. The syntax for using `getline` with `string` objects is a bit different from what we described for C strings in Section 8.1. You do not use `cin.getline`; instead, you make `cin` the first argument to `getline`.² (Thus, this version of `getline` is not a member function.)

```
string line;
cout << "Enter a line of input:\n";
getline(cin, line);
cout << line << "END OF OUTPUT\n";
```

When embedded in a complete program, this code produces a dialogue like the following:

```
Enter some input:
Do bedo to you!
Do bedo to you!END OF OUTPUT
```

If there were leading or trailing blanks on the line, then they too would be part of the `string` value read by `getline`. This version of `getline` is in the

²This is a bit ironic, since the class `string` was designed using more modern object-oriented techniques, and the notation it uses for `getline` is the old fashioned, less object-oriented notation. This is an accident of history. This `getline` function was defined after the `iostream` library was already in use, so the designers had little choice but to make this `getline` a stand-alone function.

library <string>. You can use a stream object connected to a text file in place of `cin` to do input from a file using `getline`.

You cannot use `cin` and `>>` to read in a blank character. If you want to read one character at a time, you can use `cin.get`, which we discussed in Chapter 6. The function `cin.get` reads values of type *char*, not of type *string*, but it can be helpful when handling *string* input. Display 8.5 contains a program that illustrates both `getline` and `cin.get` used for *string* input. The significance of the function `newLine` is explained in the Pitfall subsection entitled *Mixing cin >> variable and getline*

DISPLAY 8.5 Program Using the Class `string` (part 1 of 2)

```

1  //Demonstrates getline and cin.get.
2  #include <iostream>
3  #include <string>

4  void newLine( );

5  int main( )
6  {
7      using namespace std;
8
9      string firstName, lastName, recordName;
10     string motto = "Your records are our records.";

11     cout << "Enter your first and last name:\n";
12     cin >> firstName>>lastName;
13     newLine( );

14     recordName = lastName + ", " + firstName;
15     cout << "Your name in our records is: ";
16     cout << recordName<<endl;

17     cout << "Our motto is\n"
18         << motto <<endl;
19     cout << "Please suggest a better (one-line) motto:\n";
20     getline(cin, motto);
21     cout << "Our new motto will be:\n";
22     cout << motto <<endl;

23     return 0;
24 }
25
26 //Uses iostream:
27 void newLine( )
28 {
29     using namespace std;
30
```

(continued)

DISPLAY 8.5 Program Using the Class string (*part 2 of 2*)

```
31     char nextChar;  
32     do  
33     {  
34         cin.get(nextChar);  
35     } while (nextChar != '\n');  
36 }
```

Sample Dialogue

```
Enter your first and last name:  
B'Elanna Torres  
Your name in our records is: Torres, B'Elanna  
Our motto is  
Your records are our records.  
Please suggest a better (one-line) motto:  
Our records go where no records dared to go before.  
Our new motto will be:  
Our records go where no records dared to go before.
```

I/O with string Objects

You can use the insertion operator << with cout to output string objects. You can input a string with the extraction operator >> and cin. When using >> for input, the code reads in a string delimited with whitespace. You can use the function getline to input an entire line of text into a string object.

EXAMPLES

```
string greeting("Hello"), response, nextWord;  
cout << greeting << endl;  
getline(cin, response);  
cin >> nextWord;
```

SELF-TEST EXERCISES

15. Consider the following code (and assume that it is embedded in a complete and correct program and then run):

```
string s1, s2;
```

```
cout << "Enter a line of input:\n";
cin >> s1 >> s2;
cout << s1 << "*" << s2 << "<END OF OUTPUT";
```

If the dialogue begins as follows, what will be the next line of output?

```
Enter a line of input:
A string is a joy forever!
```

16. Consider the following code (and assume that it is embedded in a complete and correct program and then run):

```
string s;
cout << "Enter a line of input:\n";
getline(cin, s);
cout << s << "<END OF OUTPUT";
```

If the dialogue begins as follows, what will be the next line of output?

```
Enter a line of input:
A string is a joy forever!
```

■ PROGRAMMING TIP More Versions of `getline`

So far, we have described the following way of using `getline`:

```
string line;
cout << "Enter a line of input:\n";
getline(cin, line);
```

This version stops reading when it encounters the end-of-line marker `'\n'`. There is a version that allows you to specify a different character to use as a stopping signal. For example, the following will stop when the first question mark is encountered:

```
string line;
cout << "Enter some input:\n";
getline(cin, line, '?');
```

It makes sense to use `getline` as if it were a *void* function, but it actually returns a reference to its first argument, which is `cin` in the code above. Thus, the following will read a line of text into `s1` and a string of nonwhitespace characters into `s2`:

```
string s1, s2;
getline(cin, s1) >> s2;
```

The invocation `getline (cin,s1)` returns a reference to `cin`, so that after the invocation of `getline`, the next thing to happen is equivalent to

```
cin >> s2;
```

This kind of use of `getline` seems to have been designed for use in a C++ quiz show rather than to meet any actual programming need, but it can come in handy sometimes. ■

PITFALL Mixing `cin >> variable;` and `getline`

Take care in mixing input using `cin >> variable;` with input using `getline`. For example, consider the following code:

```
int n;
string line;
cin >> n;
getline(cin, line);
```



VideoNote

Example using `cin` and `getline` with the `string` class

getline for Objects of the Class string

The `getline` function for string objects has two versions:

```
istream& getline(istream& ins, string& strVar,
                char delimiter);
```

and

```
istream& getline(istream& ins, string& strVar);
```

The first version of this function reads characters from the `istream` object given as the first argument (always `cin` in this chapter), inserting the characters into the string variable `strVar` until an instance of the delimiter character is encountered. The delimiter character is removed from the input and discarded. The second version uses `'\n'` for the default value of `delimiter`; otherwise, it works the same.

These `getline` functions return their first argument (always `cin` in this chapter), but they are usually used as if they were *void* functions.

When this code reads the following input, you might expect the value of `n` to be set to 42 and the value of `line` to be set to a string value representing "Hello hitchhiker.":

```
42
Hello hitchhiker.
```

However, while `n` is indeed set to the value of 42, `line` is set equal to the empty string. What happened?

Using `cin >> n` skips leading whitespace on the input, but leaves the rest of the line, in this case just `'\n'`, for the next input. A statement like

```
cin >> n;
```


always leaves something on the line for a following `getline` to read (even if it is just the `'\n'`). In this case, the `getline` sees the `'\n'` and stops reading, so `getline` reads an empty string. If you find your program appearing to mysteriously ignore input data, see if you have mixed these two kinds of input. You may need to use either the `newline` function from Display 8.5 or the function `ignore` from the library `iostream`. For example,

```
cin.ignore(1000, '\n');
```

With these arguments, a call to the `ignore` member function will read and discard the entire rest of the line up to and including the `'\n'` (or until it discards 1000 characters if it does not find the end of the line after 1000 characters).

There can be other baffling problems with programs that use `cin` with both `>>` and `getline`. Moreover, these problems can come and go as you move from one C++ compiler to another. When all else fails, or if you want to be certain of portability, you can resort to character-by-character input using `cin.get`.

These problems can occur with any of the versions of `getline` that we discuss in this chapter. ■

String Processing with the Class `string`

The class `string` allows you to perform the same operations that you can perform with the C strings we discussed in Section 8.1 and more. You can access the characters in a `string` object in the same way that you access array elements, so `string` objects have all the advantages of arrays of characters plus a number of advantages that arrays do not have, such as automatically increasing their capacity. If `lastName` is the name of a `string` object, then `lastName[i]` gives access to the *i*th character in the string represented by `lastName`. This use of array square brackets is illustrated in Display 8.6.

Display 8.6 also illustrates the member function `length`. Every `string` object has a member function named `length` that takes no arguments and returns the length of the string represented by the `string` object. Thus, not only can a `string` object be used like an array but the `length` member function makes it behave like a partially filled array that automatically keeps track of how many positions are occupied.

When used with an object of the class `string`, the array square brackets do not check for illegal indexes. If you use an illegal index (that is, an index that is greater than or equal to the length of the string in the object), then the results are unpredictable but are bound to be bad. You may just get strange behavior without any error message that tells you that the problem is an illegal index value.

There is a member function named `at` that does check for illegal index values. This member function behaves basically the same as the square brackets, except for two points: You use function notation with `at`, so instead of

DISPLAY 8.6 A string Object Can Behave Like an Array

```

1  //Demonstrates using a string object as if it were an array.
2  #include <iostream>
3  #include <string>
4  using namespace std;
5  int main( )
6  {
7      string firstName, lastName;
8
9      cout << "Enter your first and last name:\n";
10     cin >> firstName>>lastName;
11
12     cout << "Your last name is spelled:\n";
13     int i;
14     for (i = 0; i < lastName.length( ); i++)
15     {
16         cout << lastName[i] << " ";
17         lastName[i] = '-';
18     }
19     cout << endl;
20     for (i = 0; i < lastName.length( ); i++)
21         cout << lastName[i] << " "; //Places a "-" under each letter.
22     cout << endl;
23     cout << "Good day " << firstName << endl;
24     return 0;
25 }
```

Sample Dialogue

```

Enter your first and last name:
John Crichton
Your last name is spelled:
C r i c h t o n
- - - - -
Good day John
```

`a[i]`, you use `a.at(i)`; and the `at` member function checks to see if `i` evaluates to an illegal index. If the value of `i` in `a.at(i)` is an illegal index, then you should get a run-time error message telling you what is wrong. In the following two example code fragments, the attempted access is out of range, yet the first of these probably will not produce an error message, although it will be accessing a nonexistent indexed variable:

```

string str("Mary");
cout << str[6] << endl;
```

The second example, however, will cause the program to terminate abnormally, so you at least know that something is wrong:

```
string str("Mary");
cout << str.at(6) << endl;
```

But be warned that some systems give very poor error messages when `str.at(i)` has an illegal index `i`.

You can change a single character in the string by assigning a *char* value to the indexed variable, such as `str[i]`. This may also be done with the member function `at`. For example, to change the third character in the string object `str` to 'X', you can use either of the following code fragments:

```
str.at(2) = 'X';
```

or

```
str[2] = 'X';
```

As in an ordinary array of characters, character positions for objects of type `string` are indexed starting with 0, so the third character in a string is in index position 2.

Display 8.7 gives a partial list of the member functions of the class `string`. In many ways, objects of the class `string` are better behaved than the C strings we introduced in Section 8.1. In particular, the `==` operator on objects of the `string` class returns a result that corresponds to our intuitive notion of strings being equal—namely, it returns *true* if the two strings contain the same characters in the same order, and returns *false* otherwise. Similarly, the comparison operators `<`, `>`, `<=`, `>=` compare string objects using lexicographic ordering. (Lexicographic ordering is alphabetic ordering using the order of symbols given in the ASCII character set in Appendix 3. If the strings consist of all letters and are both either all uppercase or all lowercase letters, then for this case lexicographic ordering is the same as everyday alphabetical ordering.)

DISPLAY 8.7 Member Functions of the Standard Class `string` (part 1 of 2)

Example	Remarks
Constructors	
<code>string str;</code>	Default constructor creates empty string object <code>str</code> .
<code>string str("sample");</code>	Creates a string object with data "sample".
<code>string str(aString);</code>	Creates a string object <code>str</code> that is a copy of <code>aString</code> ; <code>aString</code> is an object of the class <code>string</code> .

(continued)

DISPLAY 8.7 Member Functions of the Standard Class `string` (part 2 of 2)**Accessors**

<code>str[i]</code>	Returns read/write reference to character in <code>str</code> at index <code>i</code> . Does not check for illegal index.
<code>str.at(i)</code>	Returns read/write reference to character in <code>str</code> at index <code>i</code> . Same as <code>str[i]</code> , but this version checks for illegal index.
<code>str.substr(position, length)</code>	Returns the substring of the calling object starting at <code>position</code> and having <code>length</code> characters.
<code>str.length()</code>	Returns the length of <code>str</code> .

Assignment/Modifiers

<code>str1 = str2;</code>	Initializes <code>str1</code> to <code>str2</code> 's data
<code>str1 += str2;</code>	Character data of <code>str2</code> is concatenated to the end of <code>str1</code> .
<code>str.empty()</code>	Returns true if <code>str</code> is an empty string; false otherwise.
<code>str1 + str2</code>	Returns a string that has <code>str2</code> 's data concatenated to the end of <code>str1</code> 's data
<code>str.insert(pos, str2);</code>	Inserts <code>str2</code> into <code>str</code> beginning at position <code>pos</code> .
<code>str.erase(pos, length);</code>	Removes substring of size <code>length</code> , starting at position <code>pos</code> .

Comparison

<code>str1 == str2</code> <code>str1 != str2</code>	Compare for equality or inequality; returns a Boolean value.
<code>str1 < str2</code> <code>str1 > str2</code>	Four comparisons. All are lexicographical comparisons.
<code>str1 <= str2</code> <code>str1 >= str2</code>	

Finds

<code>str.find(str1)</code>	Returns index of the first occurrence of <code>str1</code> in <code>str</code> . If <code>str1</code> is not found, then the special value <code>string::npos</code> is returned.
<code>str.find(str1, pos)</code>	Returns index of the first occurrence of string <code>str1</code> in <code>str</code> ; the search starts at position <code>pos</code> .
<code>str.find_first_of(str1, pos)</code>	Returns the index of the first instance in <code>str</code> of any character in <code>str1</code> , starting the search at position <code>pos</code> .
<code>str.find_first_not_of (str1, pos)</code>	Returns the index of the first instance in <code>str</code> of any character not in <code>str1</code> , starting the search at position <code>pos</code> .

The first three lines of `removePunct` declare variables for use in the function. The `for` loop runs through the characters of the parameters one at a time and tries to find them in the `punct` string. To do this, a string that is the substring of `s`, of length 1 at each character position, is extracted. The position of this substring in the `punct` string is determined using the `find` member function. If this one-character string is not in the `punct` string, then the one-character string is concatenated to the `noPunct` string that is to be returned.

= and == Are Different for strings and C Strings

The operators `=`, `==`, `!=`, `<`, `>`, `<=`, `>=`, when used with the standard C++ type `string`, produce results that correspond to our intuitive notion of how strings compare. They do not misbehave as they do with the C strings, as we discussed in Section 8.1

SELF-TEST EXERCISES

17. Consider the following code:

```
string s1, s2("Hello");
cout << "Enter a line of input:\n";
cin >> s1;
if (s1 == s2)
    cout << "Equal\n";
else
    cout << "Not equal\n";
```

If the dialogue begins as follows, what will be the next line of output?

```
Enter a line of input:
Hello friend!
```

18. What is the output produced by the following code?

```
string s1, s2("Hello");
s1 = s2;
s2[0] = 'J';
cout << s1 << " " << s2;
```

Converting Between string Objects and C Strings

You have already seen that C++ will perform an automatic type conversion to allow you to store a C string in a variable of type `string`. For example, the following will work fine:

```
char aCString[] = "This is my C string.";
string stringVariable;
stringVariable = aCString;
```

However, the following will produce a compiler error message:

```
aCString = stringVariable; //ILLEGAL
```

The following is also illegal:

```
strcpy(aCString, stringVariable); //ILLEGAL
```

`strcpy` cannot take a `string` object as its second argument, and there is no automatic conversion of `string` objects to C strings, which is the problem we cannot seem to get away from.

To obtain the C string corresponding to a `string` object, you must perform an explicit conversion. This can be done with the `string` member function `c_str()`. The correct version of the copying we have been trying to do is the following:

```
strcpy(aCString, stringVariable.c_str( )); //Legal;
```

Note that you need to use the `strcpy` function to do the copying. The member function `c_str()` returns the C string corresponding to the `string` calling object. As we noted earlier in this chapter, the assignment operator does not work with C strings. So, just in case you thought the following might work, we should point out that it too is illegal.

```
aCString = stringVariable.c_str( ); //ILLEGAL
```

Converting Between Strings and Numbers

Prior to C++11 it was a bit complicated to convert between strings and numbers, but in C++11 it is simply a matter of calling a function. Use `stof`, `stod`, `stoi`, or `stol` to convert a string to a `float`, `double`, `int`, or `long`, respectively. Use `to_string` to convert a numeric type to a string. These functions are illustrated in the following example:

```
int i;
double d;
string s;
i = stoi("35"); // Converts the string "35" to an integer 35
d = stod("2.5"); // Converts the string "2.5" to the double 2.5
s = to_string(d*2); // Converts the double 5.0 to a string
                    "5.0000"
cout << i << " " << d << " " << s << endl;
```

The output is 35 2.5 5.0000

8.3 VECTORS

"Well, I'll eat it," said Alice, "and if it makes me grow larger, I can reach the key; and if it makes me grow smaller, I can creep under the door; so either way I'll get into the garden...."

LEWIS CARROLL, *Alice's Adventures in Wonderland*

Vectors can be thought of as arrays that can grow (and shrink) in length while your program is running. In C++, once your program creates an array, it cannot change the length of the array. Vectors serve the same purpose as arrays except that they can change length while the program is running. Vectors are part of a standard C++ library known as the STL (Standard Template Library), which we cover in more detail in Chapter 18.

You need not read the previous sections of this chapter before covering this section.

Vector Basics

Like an array, a vector has a base type, and like an array, a vector stores a collection of values of its base type. However, the syntax for a vector type and a vector variable declaration are different from the syntax for arrays.

You declare a variable *v* for a vector with base type *int* as follows:

```
vector<int> v;
```

Declaring a vector variable

The notation `vector<Base_Type>` is a **template class**, which means you can plug in any type for *Base_Type* and that will produce a class for vectors with that base type. You can think of this as specifying the base type for a vector in the same sense as you specify a base type for an array. You can use any type, including class types, as the base type for a vector. The notation `vector<int>` is a class name, and so the previous declaration of *v* as a vector of type `vector<int>` includes a call to the default constructor for the class `vector<int>`, which creates a vector object that is empty (has no elements).

Vector elements are indexed starting with 0, the same as arrays. The array square brackets notation can be used to read or change these elements, just as with an array. For example, the following changes the value of the *i*th element of the vector *v* and then outputs that changed value. (*i* is an *int* variable.)

```
v[i] = 42;
cout << "The answer is " << v[i];
```

There is, however, a restriction on this use of the square brackets notation with vectors that is unlike the same notation used with arrays. You can use `v[i]` to change the value of the *i*th element. However, you cannot initialize the *i*th element using `v[i]`; you can only change an element that has already been given some value. To add an element to an index position of a vector for the first time, you would normally use the member function `push_back`.

You add elements to a vector in order of positions, first at position 0, then position 1, then 2, and so forth. The member function `push_back` adds an element in the next available position. For example, the following gives initial values to elements 0, 1, and 2 of the vector `sample`:

```
vector<double> sample;
sample.push_back(0.0);
sample.push_back(1.1);
sample.push_back(2.2);
```

In C++11 we can initialize a vector the same way we initialize an array:

```
vector<double> sample = {0.0, 1.1, 2.2};
```

The number of elements in a vector is called the **size** of the vector. The member function `size` can be used to determine how many elements are in a vector. For example, after the previously shown code is executed, `sample.size()` returns 3. You can write out all the elements currently in the vector `sample` as follows:

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

The function `size` returns a value of type *unsigned int*, not a value of type *int*. (The type *unsigned int* allows only nonnegative integer values.) This returned value should be automatically converted to type *int* when it needs to be of type *int*, but some compilers may warn you that you are using an *unsigned int* where an *int* is required. If you want to be very safe, you can always apply a type cast to convert the returned *unsigned int* to an *int* or, in cases like this *for* loop, use a loop control variable of type *unsigned int* as follows:

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

Equivalently, we could use the ranged *for* loop:

```
for (auto i : sample)
    cout << i << endl;
```

A simple demonstration illustrating some basic vector techniques is given in Display 8.9.

There is a vector constructor that takes one integer argument and will initialize the number of positions given as the argument. For example, if you declare `v` as follows:

```
vector<int> v(10);
```

then the first ten elements are initialized to 0, and `v.size()` would return 10. You can then set the value of the *i*th element using `v[i]` for values of *i* equal to 0 through 9. In particular, the following could immediately follow the declaration:

```
for (unsigned int i = 0; i < 10; i++)
    v[i] = i;
```


DISPLAY 8.9 Using a Vector

```

1  #include <iostream>
2  #include <vector>
3  using namespace std;

4  int main( )
5  {
6      vector<int> v;
7      cout << "Enter a list of positive numbers.\n"
8           << "Place a negative number at the end.\n";

9      int next;
10     cin >> next;
11     while (next > 0)
12     {
13         v.push_back(next);
14         cout << next << " added. ";
15         cout << "v.size( ) = " << v.size( ) << endl;
16         cin >> next;
17     }

18     cout << "You entered:\n";
19     for (unsigned int i = 0; i < v.size( ); i++)
20         cout << v[i] << " ";
21     cout << endl;

22     return 0;
23 }
```

Sample Dialogue

```

Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 -1
2 added. v.size( ) = 1
4 added. v.size( ) = 2
6 added. v.size( ) = 3
8 added. v.size( ) = 4
You entered:
2 4 6 8
```

To set the *i*th element, for *i* greater than or equal to 10, you would use `push_back`.

When you use the constructor with an integer argument, vectors of numbers are initialized to the zero of the number type. If the vector base type is a class type, the default constructor is used for initialization.

The vector definition is given in the library `vector`, which places it in the `std` namespace. Thus, a file that uses vectors would include the following (or something similar):

```
#include <vector>
using namespace std;
```

PITFALL Using Square Brackets Beyond the Vector Size

If `v` is a vector and `i` is greater than or equal to `v.size()`, then the element `v[i]` does not yet exist and needs to be created by using `push_back` to add elements up to and including position `i`. If you try to set `v[i]` for `i` greater than or equal to `v.size()`, as in

```
v[i] = n;
```

then you may or may not get an error message, but your program will undoubtedly misbehave at some point. ■

Vectors

Vectors are used very much like arrays are used, but a vector does not have a fixed size. If it needs more capacity to store another element, its capacity is automatically increased. Vectors are defined in the library `<vector>`, which places them in the `std` namespace. Thus, a file that uses vectors would include the following (or something similar):

```
#include <vector>
using namespace std;
```

The vector class for a given *Base_Type* is written `vector<Base_Type>`. Two sample vector declarations are

```
vector<int> v; //default constructor
               //producing an empty vector.
vector<AClass> record(20); //vector constructor
                           //for AClass to initialize
                           //20 elements.
```

Elements are added to a vector using the member function `push_back`, as illustrated below:

```
v.push_back(42);
```

Once an element position has received its first element, either with `push_back` or with a constructor initialization, that element position can then be accessed using square bracket notation, just like an array element.

■ PROGRAMMING TIP Vector Assignment Is Well Behaved

The assignment operator with vectors does an element-by-element assignment to the vector on the left-hand side of the assignment operator (increasing capacity if needed and resetting the size of the vector on the left-hand side of the assignment operator). Thus, provided the assignment operator on the base type makes an independent copy of the element of the base type, then the assignment operator on the vector will make an independent copy.

Note that for the assignment operator to produce a totally independent copy of the vector on the right-hand side of the assignment operator requires that the assignment operator on the base type make completely independent copies. The assignment operator on a vector is only as good (or bad) as the assignment operator on its base type. (Details on overloading the assignment operator for classes that need it are given in Chapter 11.) ■

Efficiency Issues

At any point in time a vector has a **capacity**, which is the number of elements for which it currently has memory allocated. The member function `capacity()` can be used to find out the capacity of a vector. Do not confuse the capacity of a vector with the size of a vector. The *size* is the number of elements in a vector, while the *capacity* is the number of elements for which there is memory allocated. Typically, the capacity is larger than the size, and the capacity is always greater than or equal to the size.

Whenever a vector runs out of capacity and needs room for an additional member, the capacity is automatically increased. The exact amount of the increase is implementation-dependent but always allows for more capacity than is immediately needed. A commonly used implementation scheme is for the capacity to double whenever it needs to increase. Since increasing capacity is a complex task, this approach of reallocating capacity in large chunks is more efficient than allocating numerous small chunks.

Size and Capacity

The **size** of a vector is the number of elements in the vector. The **capacity** of a vector is the number of elements for which it currently has memory allocated. For a vector `v`, the size and capacity can be recovered with the member functions `v.size()` and `v.capacity()`.

You can completely ignore the capacity of a vector and that will have no effect on what your program does. However, if efficiency is an issue, you might want to manage capacity yourself and not simply accept the default behavior of doubling capacity whenever more is needed. You can use the member function `reserve` to explicitly increase the capacity of a vector. For example,

```
v.reserve(32);
```

sets the capacity to at least 32 elements, and

```
v.reserve(v.size( ) + 10);
```

sets the capacity to at least 10 more than the number of elements currently in the vector. Note that you can rely on `v.reserve` to increase the capacity of a vector, but it does not necessarily decrease the capacity of a vector if the argument is smaller than the current capacity.

You can change the size of a vector using the member function `resize`. For example, the following resizes a vector to 24 elements:

```
v.resize(24);
```

If the previous size was less than 24, then the new elements are initialized as we described for the constructor with an integer argument. If the previous size was greater than 24, then all but the first 24 elements are lost. The capacity is automatically increased if need be. Using `resize` and `reserve`, you can shrink the size and capacity of a vector when there is no longer any need for some elements or some capacity.

SELF-TEST EXERCISES

19. Is the following program legal? If so, what is the output?

```
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
    vector<int> v(10);
    int i;

    for (i = 0; i < v.size( ); i++)
        v[i] = i;

    vector<int> copy;
    copy = v;
    v[0] = 42;

    for (i = 0; i < copy.size( ); i++)
        cout << copy[i] << " ";
    cout << endl;

    return 0;
}
```

20. What is the difference between the size and the capacity of a vector?

CHAPTER SUMMARY

- A C-string variable is the same thing as an array of characters, but it is used in a slightly different way. A string variable uses the null character `'\0'` to mark the end of the string stored in the array.
- C-string variables usually must be treated like arrays, rather than simple variables of the kind we used for numbers and single characters. In particular, you cannot assign a C-string value to a C-string variable using the equal sign, `=`, and you cannot compare the values in two C-string variables using the `==` operator. Instead, you must use special C-string functions to perform these tasks.
- The ANSI/ISO standard `<string>` library provides a fully featured class called `string` that can be used to represent strings of characters.
- Objects of the class `string` are better behaved than C strings. In particular, the assignment and equal operators, `=` and `==`, have their intuitive meaning when used with objects of the class `string`.
- Vectors can be thought of as arrays that can grow (and shrink) in length while your program is running.

Answers to Self-Test Exercises

1. The following two are equivalent to each other (but not equivalent to any others):

```
char stringVar[10] = "Hello";
char stringVar[10] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

The following two are equivalent to each other (but not equivalent to any others):

```
char stringVar[6] = "Hello";
char stringVar[] = "Hello";
```

The following is not equivalent to any of the others:

```
char stringVar[10] = {'H', 'e', 'l', 'l', 'o'};
```

2. "DoBeDo to you"
3. The declaration means that `stringVar` has room for only six characters (including the null character `'\0'`). The function `strcat` does not check that there is room to add more characters to `stringVar`, so `strcat` will write all the characters in the string "and Good-bye." into memory, even though that requires more memory than has been assigned to `stringVar`. This means memory that should not be changed will be changed. The net effect is unpredictable, but bad.

- If the value to be inserted is greater than the current node's value:
 - then, if the current node's right pointer is `null`, set the current node's right pointer to point to the new node
 - If the current node's right pointer is not `null`, set the current node's pointer to the address of the right node and repeat from step 2.

Write a method in your base `BinaryTree` class named `printInOrder` which is recursive and operates as follows: for each `Node`, you should first follow the left pointer if it is not null, then print the current `Node`'s value and then follow the right pointer if it is not null.

Write a driver program which constructs a `BinaryTree` and a `BinarySearchTree` and insert the same values into both. Then call the `printInOrder` method on both trees. The `BinarySearchTree` object should print out in sorted order.



Exception Handling

16

16.1 EXCEPTION-HANDLING BASICS 929

A Toy Example of Exception Handling 929

Defining Your Own Exception Classes 938

Multiple Throws and Catches 938

Pitfall: Catch the More Specific Exception First 942

Programming Tip: Exception Classes
Can Be Trivial 943

Throwing an Exception in a Function 943

Exception Specification 945

Pitfall: Exception Specification in Derived
Classes 947

16.2 PROGRAMMING TECHNIQUES FOR EXCEPTION HANDLING 948

When to Throw an Exception 948

Pitfall: Uncaught Exceptions 950


Pitfall: Nested *try-catch* Blocks 950

Pitfall: Overuse of Exceptions 950

Exception Class Hierarchies 951

Testing for Available Memory 951

Rethrowing an Exception 952



It's the exception that proves the rule.

COMMON MAXIM (possibly a corruption of something like: *It's the exception that tests the rule.*)

INTRODUCTION

One way to write a program is to first assume that nothing unusual or incorrect will happen. For example, if the program takes an entry off a list, you might assume that the list is not empty. Once you have the program working for the core situation where things always go as planned, you can then add code to take care of the exceptional cases. In C++, there is a way to reflect this approach in your code. Basically, you write your code as if nothing very unusual happens. After that, you use the C++ exception-handling facilities to add code for those unusual cases. Exception handling is commonly used to handle error situations, but perhaps a better way to view exceptions is as a way to handle “exceptional situations.” After all, if your code correctly handles an “error,” then it no longer is an error.

Perhaps the most important use of exceptions is to deal with functions that have some special case that is handled differently depending on how the function is used. Perhaps the function will be used in many programs, some of which will handle the special case in one way and some of which will handle it in some other way. For example, if there is a division by zero in the function, then it may turn out that for some invocations of the function, the program should end, but for other invocations of the function something else should happen. You will see that such a function can be defined to throw an exception if the special case occurs, and that exception will allow the special case to be handled outside of the function. That way, the special case can be handled differently for different invocations of the function.

In C++, exception handling proceeds as follows: Either some library software or your code provides a mechanism that signals when something unusual happens. This is called *throwing an exception*. At another place in your program, you place the code that deals with the exceptional case. This is called *handling the exception*. This method of programming makes for cleaner code. Of course, we still need to explain the details of how you do this in C++.

PREREQUISITES

With the exception of one subsection that can be skipped, Section 16.1 uses material only from Chapters 2 to 6 and 10 to 11. The Pitfall subsection of Section 16.1 entitled “Exception Specification in Derived Classes” uses material from Chapter 15. This Pitfall subsection can be skipped without loss of continuity.

With the exception of one subsection that can be skipped, Section 16.2 uses material only from Chapters 2 to 8 and 10 to 12 and Section 15.1 of Chapter 15 in addition to Section 16.1. The subsection of Section 16.2 entitled “Testing for Available Memory” uses material from Chapter 15. This subsection can be skipped without loss of continuity.

16.1 EXCEPTION-HANDLING BASICS

Well, the program works for most cases. I didn't know it had to work for that case.

COMPUTER SCIENCE STUDENT, APPEALING A GRADE

Exception handling is meant to be used sparingly and in situations that are more involved than what is reasonable to include in a simple introductory example. So, we will teach you the exception-handling details of C++ by means of simple examples that would not normally use exception handling. This makes a lot of sense for learning about exception handling, but do not forget that these first examples are toy examples, and in practice, you would not use exception handling for anything that simple.

A Toy Example of Exception Handling

For this example, suppose that milk is such an important food in our culture that people almost never run out of it, but still we would like our programs to accommodate the very unlikely situation of running out of milk. The basic code, which assumes we do not run out of milk, might be as follows:

```
cout << "Enter number of donuts:\n";
cin >> donuts;
cout << "Enter number of glasses of milk:\n";
cin >> milk;
dpg = donuts/static_cast<double>(milk);
cout << donuts << " donuts.\n"
    << milk << " glasses of milk.\n"
    << "You have " << dpg
    << " donuts for each glass of milk.\n";
```

If there is no milk, then this code will include a division by zero, which is an error. To take care of the special situation in which we run out of milk, we can add a test for this unusual situation. The complete program with this added test for the special situation is shown in Display 16.1. The program in Display 16.1 does not use exception handling. Now, let's see how this program can be rewritten using the C++ exception-handling facilities.

DISPLAY 16.1 Handling a Special Case Without Exception Handling

```
1  include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int donuts, milk;
7      double dpg;
8      cout << "Enter number of donuts:\n";
9      cin >> donuts;
10     cout << "Enter number of glasses of milk:\n";
11     cin >> milk;
12
13     if (milk <= 0)
14     {
15         cout << donuts << " donuts, and No Milk!\n"
16             << "Go buy some milk.\n";
17     }
18     else
19     {
20         dpg = donuts/static_cast<double>(milk);
21         cout << donuts << " donuts.\n"
22             << milk << " glasses of milk.\n"
23             << "You have " << dpg
24             << " donuts for each glass of milk.\n";
25     }
26     cout << "End of program.\n";
27     return 0;
28 }
```

Sample Dialogue

```
Enter number of donuts:
12
Enter number of glasses of milk:
0
12 donuts, and No Milk!
Go buy some milk.
End of program.
```

In Display 16.2, we have rewritten the program from Display 16.1 using an exception. This is only a toy example, and you would probably not use an exception in this case. However, it does give us a simple example. Although the program as a whole is not simpler, at least the part between the words *try* and *catch* is cleaner, and this hints at the advantage of using exceptions. Look

DISPLAY 16.2 Same Thing Using Exception Handling (*part 1 of 2*)

```

1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int donuts, milk;
7      double dpg;
8
9      try
10     {
11         cout << "Enter number of donuts:\n";
12         cin >> donuts;
13         cout << "Enter number of glasses of milk:\n";
14         cin >> milk;
15
16         if (milk <= 0)
17             throw donuts;
18
19         dpg = donuts/static_cast<double>(milk);
20         cout << donuts << " donuts.\n"
21              << milk << " glasses of milk.\n"
22              << "You have " << dpg
23              << " donuts for each glass of milk.\n";
24     }
25     catch(int e)
26     {
27         cout << e << " donuts, and No Milk!\n"
28              << "Go buy some milk.\n";
29     }
30
31     cout << "End of program.\n";
32     return 0;
33 }

```

Sample Dialogue 1

```

Enter number of donuts:
12
Enter number of glasses of milk:
6
12 donuts.
6 glasses of milk.
You have 2 donuts for each glass of milk.

```

(continued)

DISPLAY 16.2 Same Thing Using Exception Handling (*part 2 of 2*)*Sample Dialogue 2*

```
Enter number of donuts:  
12  
Enter number of glasses of milk:  
0  
12 donuts, and No Milk!  
Go buy some milk.  
End of program.
```

at the code between the words *try* and *catch*. That code is basically the same as the code in Display 16.1, but rather than the big *if-else* statement (shown in color in Display 16.1) this new program has the following smaller *if* statement (plus some simple nonbranching statements):

```
if (milk <= 0)  
    throw donuts;
```

This *if* statement says that if there is no milk, then do something exceptional. That something exceptional is given after the word *catch*. The idea is that the normal situation is handled by the code following the word *try*, and that the code following the word *catch* is used only in exceptional circumstances. We have thus separated the normal case from the exceptional case. In this toy example, this separation does not really buy us too much, but in other situations it will prove to be very helpful. Let's look at the details.

The basic way of handling exceptions in C++ consists of the *try-throw-catch* threesome. A **try block** has the syntax

```
try  
{  
    Some_Code  
}
```

This *try* block contains the code for the basic algorithm that tells the computer what to do when everything goes smoothly. It is called a *try* block because you are not 100 percent sure that all will go smoothly, but you want to "give it a try."

Now if something *does* go wrong, you want to throw an exception, which is a way of indicating that something went wrong. The basic outline, when we add a *throw*, is as follows:

```
try  
{
```

```

    Code_To_Try
    Possibly_Throw_An_Exception
    More_Code
}

```

The following is an example of a *try* block with a *throw* statement included (copied from Display 16.2):

```

try
{
    cout << "Enter number of donuts:\n";
    cin >> donuts;
    cout << "Enter number of glasses of milk:\n";
    cin >> milk;
    if (milk <= 0)
        throw donuts;
    dpq = donuts/static_cast<double>(milk);
    cout << donuts << " donuts.\n"
         << milk << " glasses of milk.\n"
         << "You have " << dpq
         << " donuts for each glass of milk.\n";
}

```

The following statement **throws** the *int* value donuts:

```
throw donuts;
```

The value thrown, in this case donuts, is sometimes called an **exception**, and the execution of a *throw* statement is called **throwing an exception**. You can throw a value of any type. In this case, an *int* value is thrown.

***throw* Statement**

SYNTAX

```
throw Expression_for_Value_to_Be_Thrown;
```

When the *throw* statement is executed, the execution of the enclosing *try* block is stopped. If the *try* block is followed by a suitable *catch* block, then flow of control is transferred to the *catch* block. A *throw* statement is almost always embedded in a branching statement, such as an *if* statement. The value thrown can be of any type.

EXAMPLE

```

if (milk <= 0)
    throw donuts;

```

As the name suggests, when something is “thrown,” something goes from one place to another place. In C++, what goes from one place to another is the flow of control (as well as the value thrown). When an exception is thrown, the code in the surrounding *try* block stops executing and another portion of code, known as a *catch block*, begins execution. This executing of the *catch* block is called catching the exception or handling the exception. When an exception is thrown, it should ultimately be handled by (caught by) some *catch* block. In Display 16.2, the appropriate *catch* block immediately follows the *try* block. We repeat the *catch* block here:

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
        << "Go buy some milk.\n";
}
```

This *catch* block looks very much like a function definition that has a parameter of a type *int*. It is not a function definition, but in some ways, a *catch* block is like a function. It is a separate piece of code that is executed when your program encounters (and executes) the following (within the preceding *try* block):

```
throw Some_int;
```

So, this *throw* statement is similar to a function call, but instead of calling a function, it calls the *catch* block and says to execute the code in the *catch* block. A *catch* block is often referred to as an **exception handler**, which is a term that suggests that a *catch* block has a function-like nature.

What is that identifier *e* in the following line from a *catch* block?

```
catch(int e)
```

That identifier *e* looks like a parameter and acts very much like a parameter. So, we will call this *e* the **catch-block parameter**. (But remember, this does not mean that the *catch* block is a function.) The *catch*-block parameter does two things:

1. The *catch*-block parameter is preceded by a type name that specifies what kind of thrown value the *catch* block can catch.
2. The *catch*-block parameter gives you a name for the thrown value that is caught, so you can write code in the *catch* block that does things with the thrown value that is caught.

We will discuss these two functions of the *catch*-block parameter in reverse order. In this subsection, we will discuss using the *catch*-block parameter as a name for the value that was thrown and is caught. In the subsection entitled “Multiple Throws and Catches,” later in this chapter, we will discuss which *catch* block (which exception handler) will process a value that is thrown. Our current example has only one *catch* block. A common

name for a *catch*-block parameter is *e*, but you can use any legal identifier in place of *e*.

Let's see how the *catch* block in Display 16.2 works. When a value is thrown, execution of the code in the *try* block ends and control passes to the *catch* block (or blocks) that are placed right after the *try* block. The *catch* block from Display 16.2 is reproduced here:

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
        << "Go buy some milk.\n";
}
```

When a value is thrown, the thrown value must be of type *int* in order for this particular *catch* block to apply. In Display 16.2, the value thrown is given by the variable *donuts*, and since *donuts* is of type *int*, this *catch* block can catch the value thrown.

Suppose the value of *donuts* is 12 and the value of *milk* is 0, as in the second sample dialogue in Display 16.2. Since the value of *milk* is not positive, the *throw* statement within the *if* statement is executed. In that case, the value of the variable *donuts* is thrown. When the *catch* block in Display 16.2 catches the value of *donuts*, the value of *donuts* is plugged in for the *catch*-block parameter *e* and the code in the *catch* block is executed, producing the following output:

```
12 donuts, and No Milk!
Go buy some milk.
```

If the value of *donuts* is positive, the *throw* statement is not executed. In this case, the entire *try* block is executed. After the last statement in the *try* block is executed, the statement after the *catch* block is executed. Note that if no exception is thrown, then the *catch* block is ignored.

This makes it sound like a *try-throw-catch* setup is equivalent to an *if-else* statement. It almost is equivalent, except for the value thrown. A *try-throw-catch* setup is similar to an *if-else* statement *with the added ability to send a message to one of the branches*. This does not sound much different from an *if-else* statement, but it turns out to be a big difference in practice.

To summarize in a more formal tone, a *try* block contains some code that we are assuming includes a *throw* statement. The *throw* statement is normally executed only in exceptional circumstances, but when it is executed, it throws a value of some type. When an exception (a value like *donuts* in Display 16.2) is thrown, that is the end of the *try* block. All the rest of the code in the *try* block is ignored and control passes to a suitable *catch* block. A *catch* block applies only to an immediately preceding *try* block. If the exception is thrown, then that exception object is plugged in for the *catch*-block parameter, and the statements in the *catch* block are executed. For example, if you look at the dialogues in Display 16.2, you will see that as soon

catch-Block Parameter

The *catch*-block parameter is an identifier in the heading of a *catch* block that serves as a placeholder for an exception (a value) that might be thrown. When a (suitable) value is thrown in the preceding *try* block, that value is plugged in for the *catch*-block parameter. You can use any legal (nonreserved word) identifier for a *catch*-block parameter.

EXAMPLE

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
        << "Go buy some milk.\n";
}
```

e is the *catch*-block parameter.

as the user enters a nonpositive number, the *try* block stops and the *catch* block is executed. For now, we will assume that every *try* block is followed by an appropriate *catch* block. We will later discuss what happens when there is no appropriate *catch* block.

Next, we summarize what happens when no exception is thrown in a *try* block. If no exception (no value) is thrown in the *try* block, then after the *try* block is completed, program execution continues with the code after the *catch* block. In other words, if no exception is thrown, then the *catch* block is ignored. Most of the time when the program is executed, the *throw* statement will not be executed, and so in most cases, the code in the *try* block will run to completion and the code in the *catch* block will be ignored completely.

try-throw-catch

This is the basic mechanism for throwing and catching exceptions. The *throw* **statement** throws the exception (a value). The *catch* **block** catches the exception (the value). When an exception is thrown, the *try* block ends and then the code in the *catch* block is executed. After the *catch* block is completed, the code after the *catch* block(s) is executed (provided the *catch* block has not ended the program or performed some other special action).

If no exception is thrown in the *try* block, then after the *try* block is completed, program execution continues with the code after the *catch* block(s). (In other words, if no exception is thrown, then the *catch* block(s) are ignored.)

SYNTAX

```

try
{
    Some_Statements
    < Either some code with a throw statement or a
        function invocation that might throw an
        exception>
    Some_More_Statements
}
catch(Type_Name e)
{
    < Code to be performed if a value of the
        catch-block parameter type is thrown in the
        try block>
}

```

EXAMPLE

See Display 16.2.

SELF-TEST EXERCISES

1. What output is produced by the following code?

```

int waitTime = 46;
try
{
    cout << "Try block entered.\n";
    if (waitTime > 30)
        throw waitTime;
    cout << "Leaving try block.\n";
}
catch(int thrownValue)
{
    cout << "Exception thrown with\n"
        << "waitTime equal to " << thrownValue << endl;
}
cout << "After catch block." << endl;

```

2. What would be the output produced by the code in Self-Test Exercise 1 if we make the following change? Change the line

```

int waitTime = 46;
to
int waitTime = 12;

```

3. In the code given in Self-Test Exercise 1, what is the *throw* statement?
4. What happens when a *throw* statement is executed? This is a general question. Tell what happens in general, not simply what happens in the code in Self-Test Question 1 or some other sample code.
5. In the code given in Self-Test Exercise 1, what is the *try* block?
6. In the code given in Self-Test Exercise 1, what is the *catch* block?
7. In the code given in Self-Test Exercise 1, what is the *catch*-block parameter?

Defining Your Own Exception Classes

A *throw* statement can throw a value of any type. A common thing to do is to define a class whose objects can carry the precise kind of information you want thrown to the *catch* block. An even more important reason for defining a specialized exception class is so that you can have a different type to identify each possible kind of exceptional situation.

An exception class is just a class. What makes it an exception class is how it's used. Still, it pays to take some care in choosing an exception class's name and other details. Display 16.3 contains an example of a program with a programmer-defined exception class. This is just a toy program to illustrate some C++ details about exception handling. It uses much too much machinery for such a simple task, but it is an otherwise uncluttered example of some C++ details.

Notice the *throw* statement, reproduced in what follows:

```
throw NoMilk(donuts);
```

The part `NoMilk(donuts)` is an invocation of a constructor for the class `NoMilk`. The constructor takes one *int* argument (in this case `donuts`) and creates an object of the class `NoMilk`. That object is then "thrown."

Multiple Throws and Catches

A *try* block can potentially throw any number of exception values, and they can be of differing types. In any one execution of the *try* block, only one exception will be thrown (since a thrown exception ends the execution of the *try* block), but different types of exception values can be thrown on different occasions when the *try* block is executed. Each *catch* block can only catch values of one type, but you can catch exception values of differing types by placing more than one *catch* block after a *try* block. For example, the program in Display 16.4 has two *catch* blocks after its *try* block.

Note that there is no parameter in the *catch* block for `DivideByZero`. If you do not need a parameter, you can simply list the type with no parameter.

DISPLAY 16.3 Defining Your Own Exception Class

```

1  #include <iostream>
2  using namespace std;

3  class NoMilk
4  {
5  public:
6      NoMilk();
7      NoMilk(int howMany);
8      int getDonuts();
9  private:
10     int count;
11 };

12 int main()
13 {
14     int donuts, milk;
15     double dpg;
16     try
17     {
18         cout << "Enter number of donuts:\n";
19         cin >> donuts;
20         cout << "Enter number of glasses of milk:\n";
21         cin >> milk;
22         if (milk <= 0)
23             throw NoMilk(donuts);
24         dpg = donuts/static_cast<double>(milk);
25         cout << donuts << " donuts.\n"
26              << milk << " glasses of milk.\n"
27              << "You have " << dpg
28              << " donuts for each glass of milk.\n";
29     }
30     catch(NoMilk e)
31     {
32         cout << e.getDonuts() << " donuts, and No Milk!\n"
33              << "Go buy some milk.\n";
34     }
35     cout << "End of program.";
36     return 0;
37 }

38
39 NoMilk::NoMilk()
40 {}
41 NoMilk::NoMilk(int howMany) : count(howMany)
42 {}
43
44 int NoMilk::getDonuts()
45 {
46     return count;
47 }

```

This is just a toy example to learn C++ syntax. Do not take it as an example of good typical use of exception handling.

The sample dialogues are the same as in Display 16.2.

DISPLAY 16.4 Catching Multiple Exceptions (part 1 of 2)

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class NegativeNumber
6  {
7  public:
8      NegativeNumber();
9      NegativeNumber(string takeMeToYourCatchBlock);
10     string getMessage();
11 private:
12     string message;
13 };
14
15 class DivideByZero
16 {};
17
18 int main()
19 {
20     int jemHadar, klingons;
21     double portion;
22
23     try
24     {
25         cout << "Enter number of JemHadar warriors:\n";
26         cin >> jemHadar;
27         if (jemHadar < 0)
28             throw NegativeNumber("JemHadar");
29
30         cout << "How many Klingon warriors do you have?\n";
31         cin >> klingons;
32         if (klingons < 0)
33             throw NegativeNumber("Klingons");
34         if (klingons != 0)
35             portion = jemHadar/static_cast<double>(klingons);
36         else
37             throw DivideByZero();
38         cout << "Each Klingon must fight "
39              << portion << " JemHadar.\n";
40     }
41     catch(NegativeNumber e)
42     {
43         cout << "Cannot have a negative number of "
44              << e.getMessage() << endl;
45     }

```

Although not done here, exception classes can have their own interface and implementation files and can be put in a namespace. This is another toy example.

(continued)

DISPLAY 16.4 Catching Multiple Exceptions (*part 2 of 2*)

```
46     catch (DivideByZero)
47     {
48         cout << "Send for help.\n";
49     }
50
51     cout << "End of program.\n";
52     return 0;
53 }
54
55
56 NegativeNumber::NegativeNumber()
57 {}
58
59 NegativeNumber::NegativeNumber(string takeMeToYourCatchBlock)
60     : message(takeMeToYourCatchBlock)
61 {}
62
63 string NegativeNumber::getMessage()
64 {
65     return message;
66 }
```

Sample Dialogue 1

```
Enter number of JemHadar warriors:
1000
How many Klingon warriors do you have?
500
Each Klingon must fight 2.0 JemHadar.
End of program
```

Sample Dialogue 2

```
Enter number of JemHadar warriors:
-10
Cannot have a negative number of JemHadar
End of program.
```

Sample Dialogue 3

```
Enter number of JemHadar warriors:
1000
How many Klingon warriors do you have?
0
Send for help.
End of program.
```

This case is discussed a bit more in the Programming Tip section entitled “Exception Classes Can Be Trivial.”

PITFALL Catch the More Specific Exception First

When catching multiple exceptions, the order of the *catch* blocks can be important. When an exception value is thrown in a *try* block, the following *catch* blocks are tried in order, and the first one that matches the type of the exception thrown is the one that is executed.

For example, the following is a special kind of *catch* block that will catch a thrown value of any type:

```
catch(...)  
{  
    <Place whatever you want in here>  
}
```

The three dots do not stand for something omitted. You actually type in those three dots in your program. This makes a good default *catch* block to place after all other *catch* blocks. For example, we could add it to the *catch* blocks in Display 16.4 as follows:

```
catch(NegativeNumber e)  
{  
    cout << "Cannot have a negative number of "  
        << e.getMessage() << endl;  
}  
catch(DivideByZero)  
{  
    cout<< "Send for help.\n";  
}  
catch(...)  
{  
    cout << "Unexplained exception.\n";  
}
```

However, it only makes sense to place this default *catch* block at the end of a list of *catch* blocks. For example, suppose we instead used:

```
catch(NegativeNumber e)  
{  
    cout << "Cannot have a negative number of "  
        << e.getMessage() << endl;  
}  
catch(...)  
{  
    cout << "Unexplained exception.\n";  
}  
catch(DivideByZero)
```

```
{
    cout << "Send for help.\n";
}
```

With this second ordering, an exception (a thrown value) of type `NegativeNumber` will be caught by the `NegativeNumber` *catch* block, as it should be. However, if a value of type `DivideByZero` were thrown, it would be caught by the block that starts *catch*(...). So, the `DivideByZero` *catch* block could never be reached. Fortunately, most compilers tell you if you make this sort of mistake. ■

■ PROGRAMMING TIP Exception Classes Can Be Trivial

Here we reproduce the definition of the exception class `DivideByZero` from Display 16.4:

```
class DivideByZero
{
};
```

This exception class has no member variables and no member functions (other than the default constructor). It has nothing but its name, but that is useful enough. Throwing an object of the class `DivideByZero` can activate the appropriate *catch* block, as it does in Display 16.4.

When using a trivial exception class, you normally do not have anything you can do with the exception (the thrown value) once it gets to the *catch* block. The exception is just being used to get you to the *catch* block. Thus, you can omit the *catch*-block parameter. (You can omit the *catch*-block parameter anytime you do not need it, whether the exception type is trivial or not.) ■

Throwing an Exception in a Function

Sometimes it makes sense to delay handling an exception. For example, you might have a function with code that throws an exception if there is an attempt to divide by zero, but you may not want to catch the exception in that function. Perhaps some programs that use that function should simply end if the exception is thrown, and other programs that use the function should do something else. So you would not know what to do with the exception if you caught it inside the function. In these cases, it makes sense to not catch the exception in the function definition, but instead to have any program (or other code) that uses the function place the function invocation in a *try* block and catch the exception in a *catch* block that follows that *try* block.

Look at the program in Display 16.5. It has a *try* block, but there is no *throw* statement visible in the *try* block. The statement that does the throwing in that program is

```
if (bottom == 0)
    throw DivideByZero();
```

DISPLAY 16.5 Throwing an Exception Inside a Function *(part 1 of 2)*

```
1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4
5  class DivideByZero
6  {};
7
8  double safeDivide(int top, int bottom) throw (DivideByZero);
9
10 int main()
11 {
12     int numerator;
13     int denominator;
14     double quotient;
15     cout << "Enter numerator:\n";
16     cin >> numerator;
17     cout << "Enter denominator:\n";
18     cin >> denominator;
19
20     try
21     {
22         quotient = safeDivide(numerator, denominator);
23     }
24     catch(DivideByZero)
25     {
26         cout << "Error: Division by zero!\n"
27              << "Program aborting.\n";
28         exit(0);
29     }
30
31     cout << numerator << "/" << denominator
32          << " = " << quotient << endl;
33
34     cout << "End of program.\n";
35     return 0;
36 }
37
38
39 double safeDivide(int top, int bottom) throw (DivideByZero)
40 {
41     if (bottom == 0)
42         throw DivideByZero();
43
44     return top/static_cast<double>(bottom);
45 }
```

(continued)

DISPLAY 16.5 Throwing an Exception Inside a Function (*part 2 of 2*)

Sample Dialogue 1

```
Enter numerator:  
5  
Enter denominator:  
10  
5/10 = 0.5  
End of Program.
```

Sample Dialogue 2

```
Enter numerator:  
5  
Enter denominator:  
0  
Error: Division by zero!  
Program aborting.
```

This statement is not visible in the *try* block. However, it is in the *try* block in terms of program execution, because it is in the definition of the function `safeDivide` and there is an invocation of `safeDivide` in the *try* block.

Exception Specification

If a function does not catch an exception, it should at least warn programmers that any invocation of the function might possibly throw an exception. If there are exceptions that might be thrown, but not caught, in the function definition, then those exception types should be listed in an **exception specification**, which is illustrated by the following function declaration from Display 16.5:

```
double safeDivide(int top, int bottom) throw (DivideByZero);
```

As illustrated in Display 16.5, the exception specification should appear in both the function declaration and the function definition. If a function has more than one function declaration, then all the function declarations must have identical exception specifications. The exception specification for a function is also sometimes called the **throw list**.

If there is more than one possible exception that can be thrown in the function definition, then the exception types are separated by commas, as illustrated here:

```
void someFunction( ) throw (DivideByZero, OtherException);
```

All exception types listed in the exception specification are treated normally. When we say the exception is treated normally, we mean it is treated as we have described before this subsection. In particular, you can place the function invocation in a *try* block followed by a *catch* block to catch that type of exception, and if the function throws the exception (and does not catch it inside the function), then the *catch* block following the *try* block will catch the exception. If there is no exception specification (no throw list) at all (not even an empty one), then it is the same as if all possible exception types were listed in the exception specification; that is, any exception that is thrown is treated normally.

What happens when an exception is thrown in a function but is not listed in the exception specification (and not caught inside the function)? In that case, the program ends. In particular, notice that if an exception is thrown in a function but is not listed in the exception specification (and not caught inside the function), then it will not be caught by any *catch* block, but instead your program will end. Remember, if there is no specification list at all, not even an empty one, then it is the same as if all exceptions were listed in the specification list, and so throwing an exception will not end the program in the way described in this paragraph.

Keep in mind that the exception specification is for exceptions that “get outside” the function. If they do not get outside the function, they do not belong in the exception specification. If they get outside the function, they belong in the exception specification no matter where they originate. If an exception is thrown in a *try* block that is inside a function definition and is caught in a *catch* block inside the function definition, then its type need not be listed in the exception specification. If a function definition includes an invocation of another function and that other function can throw an exception that is not caught, then the type of the exception should be placed in the exception specification.

To say that a function should not throw any exceptions that are not caught inside the function, you use an empty exception specification like so:

```
void someFunction( ) throw ( );
```

By way of summary:

```
void someFunction( ) throw (DivideByZero, OtherException);
//Exceptions of type DivideByZero or OtherException are
//treated normally. All other exceptions end the program
//if not caught in the function body.
```

```
void someFunction( ) throw ( );
//Empty exception list; all exceptions end the
//program if thrown but not caught in the function body.
```

```
void someFunction( );
//All exceptions of all types treated normally.
```

Keep in mind that an object of a derived class¹ is also an object of its base class. So, if D is a derived class of class B and B is in the exception specification, then a thrown object of class D will be treated normally, since it is an object of class B and B is in the exception specification. However, no automatic type conversions are done. If *double* is in the exception specification, that does not account for throwing an *int* value. You would need to include both *int* and *double* in the exception specification.

One final warning: Not all compilers treat the exception specification as they are supposed to. Some compilers essentially treat the exception specification as a comment, and so with those compilers, the exception specification has no effect on your code. This is another reason to place all exceptions that might be thrown by your functions in the exception specification. This way all compilers will treat your exceptions the same way. Of course, you could get the same compiler consistency by not having any exception specification at all, but then your program would not be as well documented and you would not get the extra error checking provided by compilers that do use the exception specification. With a compiler that does process the exception specification, your program will terminate as soon as it throws an exception that you did not anticipate. (Note that this is a run-time behavior, but which run-time behavior you get depends on your compiler.)

Warning!

PITFALL Exception Specification in Derived Classes

When you redefine or override a function definition in a derived class, it should have the same exception specification as it had in the base class, or it should have an exception specification whose exceptions are a subset of those in the base class exception specification. Put another way, when you redefine or override a function definition, you cannot add any exceptions to the exception specification (but you can delete some exceptions if you want). This makes sense, since an object of the derived class can be used anywhere an object of the base class can be used, and so a redefined or overwritten function must fit any code written for an object of the base class. ■

SELF-TEST EXERCISES

8. What is the output produced by the following program?

```
#include <iostream>
using namespace std;
void sampleFunction(double test) throw (int);
```

¹ If you have not yet learned about derived classes, you can safely ignore the remarks about them.

```

int main()
{
    try
    {
        cout << "Trying.\n";
        sampleFunction(98.6);
        cout << "Trying after call.\n";
    }
    catch(int)
    {
        cout << "Catching.\n";
    }
    cout << "End of program.\n";
    return 0;
}

void sampleFunction(double test) throw (int)
{
    cout << "Starting sampleFunction.\n";
    if (test < 100)
        throw 42;
}

```

9. What is the output produced by the program in Self-Test Exercise 8 if the following change were made to the program? Change

```
sampleFunction(98.6);
```

in the *try* block to

```
sampleFunction(212);
```

16.2 PROGRAMMING TECHNIQUES FOR EXCEPTION HANDLING

Only use this in exceptional circumstances.

WARREN PEACE, *The Lieutenant's Tools*

So far, we have shown you lots of code that explains how exception handling works in C++, but we have not yet shown even one example of a program that makes good and realistic use of exception handling. However, now that you know the mechanics of exception handling, this section can go on to explain exception-handling techniques.

When to Throw an Exception

We have given some very simple code in order to illustrate the basic concepts of exception handling. However, our examples were unrealistically simple. A more complicated but better guideline is to separate throwing an exception

and catching the exception into separate functions. In most cases, you should include any *throw* statement within a function definition, list the exception in the exception specification for that function, and place the *catch* clause in a *different function*. Thus, the preferred use of the *try-throw-catch* triad is as illustrated here:

```
void functionA() throw (MyException)
{
    .
    .
    .
    throw MyException(<Maybe an argument>);
    .
    .
    .
}
```

Then, in *some other function* (perhaps even some other function in some other file), you have

```
void functionB()
{
    .
    .
    .
    try
    {
        .
        .
        .
        functionA();
        .
        .
        .
    }
    catch(MyException e)
    {
        <Handle exception>
    }
    .
    .
    .
}
```

Moreover, even this kind of use of a *throw* statement should be reserved for cases in which it is unavoidable. If you can easily handle a problem in some other way, do not throw an exception. Reserve *throw* statements for situations in which the way the exceptional condition is handled depends on how and where the function is used. If the way that the exceptional condition is handled depends on how and where the function is invoked, then the best

When to Throw an Exception

For the most part, *throw* statements should be used within functions and listed in an exception specification for the function. Moreover, they should be reserved for situations in which the way the exceptional condition is handled depends on how and where the function is used. If the way that the exceptional condition is handled depends on how and where the function is invoked, then the best thing to do is to let the programmer who invokes the function handle the exception. In all other situations, it is almost always preferable to avoid throwing an exception.

thing to do is to let the programmer who invokes the function handle the exception. In all other situations, it is almost always preferable to avoid throwing exceptions.

PITFALL Uncaught Exceptions

Every exception that is thrown by your code should be caught someplace in your code. If an exception is thrown but not caught anywhere, your program will end. ■

PITFALL Nested *try-catch* Blocks

You can place a *try* block and following *catch* blocks inside a larger *try* block or inside a larger *catch* block. In rare cases, this may be useful, but if you are tempted to do this, you should suspect that there is a nicer way to organize your program. It is almost always better to place the inner *try-catch* blocks inside a function definition and place an invocation of the function in the outer *try* or *catch* block (or maybe just eliminate one or more *try* blocks completely).

If you place a *try* block and following *catch* blocks inside a larger *try* block, and an exception is thrown in the inner *try* block but not caught in the inner *try-catch* blocks, then the exception is thrown to the outer *try* block for processing and might be caught there. ■

PITFALL Overuse of Exceptions

Exceptions allow you to write programs whose flow of control is so involved that it is almost impossible to understand the program. Moreover, this is not hard to do. Throwing an exception allows you to transfer flow of control from

anyplace in your program to almost anyplace else in your program. In the early days of programming, this sort of unrestricted flow of control was allowed via a construct known as a *goto*. Programming experts now agree that such unrestricted flow of control is very poor programming style. Exceptions allow you to revert to these bad old days of unrestricted flow of control. Exceptions should be used sparingly and only in certain ways. A good rule is the following: If you are tempted to include a *throw* statement, then think about how you might write your program or class definition without this *throw* statement. If you think of an alternative that produces reasonable code, then you probably do not want to include the *throw* statement. ■

Exception Class Hierarchies

It can be very useful to define a hierarchy of exception classes. For example, you might have an `ArithmeticError` exception class and then define an exception class `DivideByZeroError` that is a derived class of `ArithmeticError`. Since a `DivideByZeroError` is an `ArithmeticError`, every *catch* block for an `ArithmeticError` will catch a `DivideByZeroError`. If you list `ArithmeticError` in an exception specification, then you have, in effect, also added `DivideByZeroError` to the exception specification, whether or not you list `DivideByZeroError` by name in the exception specification.



VideoNote
The STL Exception Class

Testing for Available Memory

In Chapter 13, we created new dynamic variables with code such as the following:

```
struct Node
{
    int data;
    Node *link;
};
typedef Node* NodePtr;
. . .
NodePtr pointer = new Node;
```

This works fine as long as there is sufficient memory available to create the new node. But, what happens if there is not sufficient memory? If there is not sufficient memory to create the node, then a `bad_alloc` exception is thrown. The type `bad_alloc` is part of the C++ language. You do not need to define it.

Since *new* will throw a `bad_alloc` exception when there is not enough memory to create the node, you can check for running out of memory as follows:

```
try
{
    NodePtr pointer = new Node;
}
```

```
    catch (badAlloc)
    {
        cout << "Ran out of memory!";
    }
```

Of course, you can do other things besides simply giving a warning message, but the details of what you do will depend on your particular programming task.

Rethrowing an Exception

It is legal to throw an exception within a *catch* block. In rare cases, you may want to catch an exception and then, depending on the details, decide to throw the same or a different exception for handling farther up the chain of exception-handling blocks.

SELF-TEST EXERCISES

10. What happens when an exception is never caught?
11. Can you nest a *try* block inside another *try* block?

CHAPTER SUMMARY

- Exception handling allows you to design and code the normal case for your program separately from the code that handles exceptional situations.
- An exception can be thrown in a *try* block. Alternatively, an exception can be thrown in a function definition that does not include a *try* block (or does not include a *catch* block to catch that type of exception). In this case, an invocation of the function can be placed in a *try* block.
- An exception is caught in a *catch* block.
- A *try* block may be followed by more than one *catch* block. In this case, always list the *catch* block for a more specific exception class before the *catch* block for a more general exception class.
- Do not overuse exceptions.

Answers to Self-Test Exercises

1. Try block entered.
Exception thrown with
waitTime equal to 46
After catch block.

2. Try block entered.
Leaving try block.
After catch block.

3. `throw waitTime;`

Note that the following is an *if* statement, not a *throw* statement, even though it contains a *throw* statement:

```
if (waitTime > 30)
    throw waitTime;
```

4. When a *throw* statement is executed, that is the end of the enclosing *try* block. No other statements in the *try* block are executed, and control passes to the following *catch* block(s). When we say control passes to the following *catch* block, we mean that the value thrown is plugged in for the *catch*-block parameter (if any), and the code in the *catch* block is executed.

5. `try`

```
{
    cout << "Try block entered.";
    if (waitTime > 30)
        throw (waitTime);
    cout << "Leaving try block.";
}
```

6. `catch(int thrownValue)`

```
{
    cout << "Exception thrown with\n"
         << "waitTime equal to" << thrownValue << endl;
}
```

7. `thrownValue` is the *catch*-block parameter.

8. Trying.
Starting `sampleFunction`.
Catching.
End of program.

9. Trying.
Starting `sampleFunction`.
Trying after call.
End of program.

10. If an exception is not caught anywhere, then your program ends.

11. Yes, you can have a *try* block and corresponding *catch* blocks inside another larger *try* block. However, it would probably be better to place the inner *try* and *catch* blocks in a function definition and place an invocation of the function in the larger *try* block.

PRACTICE PROGRAMS

Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.



VideoNote
Solution to Practice
Program 16.1

1. A function that returns a special error code is often better implemented by throwing an exception instead. This way, the error code cannot be ignored or mistaken for valid data. The following class maintains an account balance.

```
class Account
{
private:
    double balance;
public:
    Account()
    {
        balance = 0;
    }
    Account(double initialDeposit)
    {
        balance = initialDeposit;
    }
    double getBalance()
    {
        return balance;
    }
    // returns new balance or -1 if error
    double deposit(double amount)
    {
        if (amount > 0)
            balance += amount;
        else
            return -1; // Code indicating error
        return balance;
    }
    // returns new balance or -1 if invalid amount
    double withdraw(double amount)
    {
        if ((amount > balance) || (amount < 0))
            return -1;
        else
            balance -= amount;
        return balance;
    }
};
```

Rewrite the class so that it throws appropriate exceptions instead of returning `-1` as an error code. Write test code that attempts to withdraw and deposit invalid amounts and catches the exceptions that are thrown.

2. The Standard Template Library includes a class named `exception` that is the parent class for any exception thrown by an STL function. Therefore, any exception can be caught by this class. The following code sets up a *try-catch* block for STL exceptions:

```
#include <iostream>
#include <string>
#include <exception>
using namespace std;

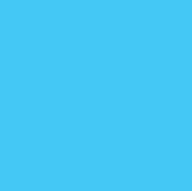
int main()
{
    string s = "hello";
    try
    {
        cout << "No exception thrown." << endl;
    }
    catch (exception& e)
    {
        cout << "Exception caught: " <<
            e.what() << endl;
    }
    return 0;
}
```

Modify the code so that an exception is thrown in the try block. You could try accessing an invalid index in a string using the `at` member function.

PROGRAMMING PROJECTS

Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit www.myprogramminglab.com to complete many of these Programming Projects online and get instant feedback.

1. Write a function to convert a hexadecimal number given as a `String` to an integer value. If the value to be converted is not a valid hexadecimal number, throw an `InvalidNumberException` before attempting the conversion. You will need to develop both the function and the exception class. Write a driver program to test valid and invalid input to your function.



*All men are mortal.
Aristotle is a man.
Therefore, Aristotle is mortal.
All X's are Y.
Z is an X.
Therefore, Z is Y.
All cats are mischievous.
Garfield is a cat.
Therefore, Garfield is mischievous.*

A SHORT LESSON ON SYLLOGISMS

INTRODUCTION

This chapter discusses C++ templates. Templates allow you to define functions and classes that have parameters for type names. This will allow you to design functions that can be used with arguments of different types and to define classes that are much more general than those you have seen before this chapter.

PREREQUISITES

Section 17.1 uses material from Chapters 2 through 5 and Sections 7.1, 7.2, and 7.3 of Chapter 7. It does not use any material on classes. Section 17.2 uses material from Chapters 2 through 7 and 10 through 12.

17.1 TEMPLATES FOR ALGORITHM ABSTRACTION

Many of our previous C++ function definitions have an underlying algorithm that is much more general than the algorithm we gave in the function definition. For example, consider the function `swapValues`, which we first discussed in Chapter 5. For reference, we now repeat the function definition:

```
void swapValues(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

Notice that the function `swapValues` applies only to variables of type `int`. Yet the algorithm given in the function body could just as well be used to swap the values in two variables of type `char`. If we want to also use the function

swapValues with variables of type *char*, we can overload the function name by adding the following definition:

```
void swapValues(char& variable1, char& variable2)
{
    char temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

But there is something inefficient and unsatisfying about these two definitions of the swapValues function: They are almost identical. The only difference is that one definition uses the type *int* in three places and the other uses the type *char* in the same three places. Proceeding in this way, if we wanted to have the function swapValues apply to pairs of variables of type *double*, we would have to write a third almost identical function definition. If we wanted to apply swapValues to still more types, the number of almost identical function definitions would be even larger. This would require a good deal of typing and would clutter up our code with lots of definitions that look identical. We should be able to say that the following function definition applies to variables of any type:

```
void swapValues(Type_Of_The_Variables& variable1,
               Type_Of_The_Variables& variable2)
{
    Type_Of_The_Variables temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

As we will see, something like this is possible. We can define one function that applies to all types of variables, although the syntax is a bit different from what we have shown above. That syntax is described in the next subsection.

Templates for Functions

Display 17.1 shows a C++ template for the function swapValues. This function template allows you to swap the values of any two variables, of any type, as long as the two variables have the same type. The definition and the function declaration begin with the line

```
template<class T>
```

This is often called the **template prefix**, and it tells the compiler that the definition or function declaration that follows is a **template** and that T is a

type parameter. In this context, the word *class* actually means *type*.¹ As we will see, the type parameter *T* can be replaced by any type, whether the type is a class or not. Within the body of the function definition, the type parameter *T* is used just like any other type.

The function template definition is, in effect, a large collection of function definitions. For the function template for `swapValues` shown in Display 17.1, there is, in effect, one function definition for each possible type name. Each of these definitions is obtained by replacing the type parameter *T* with a type name. For example, the function definition that follows is obtained by replacing *T* with the type name *double*:

```
void swapValues(double& variable1, double& variable2)
{
    double temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

A template
overloads the
function name

Another definition for `swapValues` is obtained by replacing the type parameter *T* in the function template with the type name *int*. Yet another definition is obtained by replacing the type parameter *T* with *char*. The one function template shown in Display 17.1 overloads the function name `swapValues` so that there is a slightly different function definition for every possible type.

The compiler will not literally produce definitions for every possible type for the function name `swapValues`, but it will behave exactly as if it had produced all those function definitions. A separate definition will be produced for each different type for which you use the template, but not for any types you do not use. Only one definition is generated for a single type regardless of the number of times you use the template for that type. Notice that the function `swapValues` is called twice in Display 17.1: One time the arguments are of type *int* and the other time the arguments are of type *char*.

Consider the following function call from Display 17.1:

```
swapValues(integer1, integer2);
```

When the C++ compiler gets to this function call, it notices the types of the arguments—in this case *int*—and then it uses the template to produce a function definition with the type parameter *T* replaced with the type name *int*. Similarly, when the compiler sees the function call

```
swapValues(symbol1, symbol2);
```

¹ In fact, the ANSI standard provides that you may use the keyword *typename* instead of *class* in the template prefix. Although we agree that using *typename* makes more sense than using *class*, the use of *class* is a firmly established tradition, and so we use *class* for the sake of consistency with most other programmers and authors.

DISPLAY17.1 A Function Template

```

1  //Program to demonstrate a function template.
2  #include <iostream>
3  using namespace std;

4  //Interchanges the values of variable1 and variable2.
5  template<class T>
6  void swapValues(T& variable1, T& variable2)
7  {
8      T temp;
9
10     temp = variable1;
11     variable1 = variable2;
12     variable2 = temp;
13 }

14 int main( )
15 {
16     int integer1 = 1, integer2 = 2;
17     cout << "Original integer values are "
18         << integer1 << " " << integer2 << endl;
19     swapValues(integer1, integer2);
20     cout << "Swapped integer values are "
21         << integer1 << " " << integer2 << endl;

22     char symbol1 = 'A', symbol2 = 'B';
23     cout << "Original character values are "
24         << symbol1 << " " << symbol2 << endl;
25     swapValues(symbol1, symbol2);
26     cout << "Swapped character values are "
27         << symbol1 << " " << symbol2 << endl;

28     return 0;
29 }

```

Output

```

Original integer values are 1 2
Swapped integer values are 2 1
Original character values are A B
Swapped character values are B A

```

it notices the types of the arguments—in this case *char*—and then it uses the template to produce a function definition with the type parameter *T* replaced with the type name *char*.

Notice that you need not do anything special when you call a function that is defined with a function template; you call it just as you would any

Calling a
function
template

other function. The compiler does all the work of producing the function definition from the function template.

Notice that in Display 17.1 we placed the function template definition before the `main` part of the program, and we used no template function declaration. A function template may have a function declaration, just like an ordinary function. You may (or may not) be able to place the function declaration and definition for a function template in the same locations that you place function declarations and definitions for ordinary functions. However, some compilers do not support template function declarations and do not support separate compilation of template functions. When these are supported, the details can be messy and can vary from one compiler to another. Your safest strategy is to not use template function declarations and to be sure the function template definition appears in the same file in which it is used and appears before the function template is used.

We said that a function template definition should appear in the same file as the file that uses the template function (that is, the same file as the file that has an invocation of the template function). However, the function template definition can appear via a `#include` directive. You can give the function template definition in one file and then `#include` that file in a file that uses the template function. That is the cleanest and safest general strategy. However, even that may not work on some compilers. If it does not work, consult a local expert.

Although we will not be using template function declarations in our code, we will describe them and give examples of them for the benefit of readers whose compilers support the use of these function declarations.

In the function template in Display 17.1, we used the letter `T` as the parameter for the type. This is traditional but is not required by the C++ language. The type parameter can be any identifier (other than a keyword). `T` is a good name for the type parameter, but sometimes other names may work better. For example, the function template for `swapValues` given in Display 17.1 is equivalent to the following:

```
template<class VariableType>
void swapValues(VariableType& variable1,
               VariableType& variable2)
{
    VariableType temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

More than one
type parameter

It is possible to have function templates that have more than one type parameter. For example, a function template with two type parameters named `T1` and `T2` would begin as follows:

```
template<class T1, class T2>
```


However, most function templates require only one type parameter. You cannot have unused template parameters; that is, each template parameter must be used in your template function.

PITFALL Compiler Complications

C++ does not allow you to separate interface (header) and implementation files for template definitions in the usual way, so you need to include your template definition with your code that uses it. As usual, at least the function declaration must precede any use of the template function. This is because the header can't correctly match the implementation when the type is unknown.

Your safest strategy is not to use template function declarations and to be sure the function template definition appears in the same file in which it is used and appears before the function template is called. However, the function template definition can appear via a `#include` directive. You can give the function template definition in one file and then `#include` that file in a file that uses the template function.

Another common technique is to put your definition and implementation, all in the header file. If you use this technique, then you would only have a header (.h) file and no implementation (.cpp) file. Sometimes the .hpp file extension is used when all of the code is in the header file. Finally, an alternate approach is to include the implementation (.cpp) file for your template class instead of the header file (.h).

Some C++ compilers have additional special requirements for using templates. If you have trouble compiling your templates, check your manuals or check with a local expert. You may need to set special options or rearrange the way you order the template definitions and the other items in your files. ■



VideoNote

Issues Compiling Programs
with Templates

Function Template

The function definition and the function declaration for a function template are each prefaced with the following:

```
template<class Type_Parameter>
```

The function declaration (if used) and definition are the same as any ordinary function declaration and definition, except that the *Type_Parameter* can be used in place of a type.

For example, the following is a function declaration for a function template:

```
template<class T>  
void showStuff(int stuff1, T stuff2, T stuff3);
```

(continued)

The definition for this function template might be as follows:

```
template<class T>
void showStuff(int stuff1, T stuff2, T stuff3)
{
    cout << stuff1 << endl
         << stuff2 << endl
         << stuff3 << endl;
}
```

The function template given in this example is equivalent to having one function declaration and one function definition for each possible type name. The type name is substituted for the type parameter (which is *T* in the example above). For instance, consider the following function call:

```
showStuff(2, 3.3, 4.4);
```

When this function call is executed, the compiler uses the function definition obtained by replacing *T* with the type name *double*. A separate definition will be produced for each different type for which you use the template but not for any types you do not use. Only one definition is generated for a specific type regardless of the number of times you use the template.

SELF-TEST EXERCISES

1. Write a function template named `maximum`. The function takes two values of the same type as its arguments and returns the larger of the two arguments (or either value if they are equal). Give both the function declaration and the function definition for the template. You will use the operator `<` in your definition. Therefore, this function template will apply only to types for which `<` is defined. Write a comment for the function declaration that explains this restriction.
2. We have used three kinds of absolute value function: `abs`, `labs`, and `fabs`. These functions differ only in the type of their argument. It might be better to have a function template for the absolute value function. Give a function template for an absolute value function called `absolute`. The template will apply only to types for which `<` is defined, for which the unary negation operator is defined, and for which the constant `0` can be used in a comparison with a value of that type. Thus, the function `absolute` can be called with any of the number types, such as *int*, *long*, and *double*. Give both the function declaration and the function definition for the template.
3. Define or characterize the template facility for C++.

17.2 TEMPLATES FOR DATA ABSTRACTION

Equal wealth and equal opportunities of culture . . . have simply made us all members of one class.

EDWARD BELLAMY, *Looking Backward: 2000–1887*

As you saw in the previous section, function definitions can be made more general by using templates. In this section, you will see that templates can also make class definitions more general.

Syntax for Class Templates

The syntax for class templates is basically the same as that for function templates. The following is placed before the template definition:

```
template<class T>
```

The type parameter `T` is used in the class definition just like any other type. As with function templates, the type parameter `T` represents a type that can be any type at all; the type parameter does not have to be replaced with a class type. As with function templates, you may use any (nonkeyword) identifier instead of `T`.

Type parameter

For example, the following is a class template. An object of this class contains a pair of values of type `T`; if `T` is `int`, the object values are pairs of integers, if `T` is `char`, the object values are pairs of characters, and so on.

```
//Class for a pair of values of type T:
template<class T>
class Pair
{
public:
    Pair();

    Pair(T firstValue, T secondValue);

    void setElement(int position, T value);
    //Precondition: position is 1 or 2.
    //Postcondition:
    //The position indicated has been set to value.

    T getElement(int position) const;
    //Precondition: position is 1 or 2.
    //Returns the value in the position indicated.
private:
    T first;
    T second;
};
```

Once the class template is defined, you can declare objects of this class. The declaration must specify what type is to be filled in for `T`. For example, the

Declaring objects

following code declares the object `score` so it can record a pair of integers and declares the object `seats` so it can record a pair of characters:

```
Pair<int> score;
Pair<char> seats;
```

The objects are then used just like any other objects. For example, the following sets the score to be 3 for the first team and 0 for the second team:

```
score.setElement(1, 3);
score.setElement(2, 0);
```

Defining member functions

The member functions for a class template are defined the same way as member functions for ordinary classes. The only difference is that the member function definitions are themselves templates. For example, the following are appropriate definitions for the member function `setElement` and for the constructor with two arguments:

```
//Uses iostream and cstdlib:
template<class T>
void Pair<T>::setElement(int position, T value)
{
    if (position == 1)
        first = value;
    else if (position == 2)
        second = value;
    else
    {
        cout << "Error: Illegal pair position.\n";
        exit(1);
    }
}

template<class T>
Pair<T>::Pair(T firstValue, T secondValue)
    : first(firstValue), second(secondValue)
{
    //Body intentionally empty.
}
```

Notice that the class name before the scope resolution operator is `Pair<T>`, not simply `Pair`.

The name of a class template may be used as the type for a function parameter. For example, the following is a possible declaration for a function with a parameter for a pair of integers:

```
int addUp(const Pair<int>& thePair);
//Returns the sum of the two integers in thePair.
```

Class Template Syntax

The class definition and the definitions of the member functions are prefaced with the following:

```
template<class Type_Parameter>
```

The class and member function definitions are then the same as for any ordinary class, except that the *Type_Parameter* can be used in place of a type.

For example, the following is the beginning of a class template definition:

```
template<class T>
class Pair
{
public:
    Pair();
    Pair(T firstValue, T secondValue);
    void setElement(int position, T value);
    . . .
```

Member functions and overloaded operators are then defined as function templates. For example, the definition of a function definition for the sample class template above could begin as follows:

```
template<class T>
void Pair<T>::setElement(int position, T value)
{
    . . .
```

Note that we specified the type, in this case *int*, that is to be filled in for the type parameter *T*.

You can even use a class template within a function template. For example, rather than defining the specialized function `addUp` given above, you could instead define a function template as follows so that the function applies to all kinds of numbers:

```
template<class T>
T addUp(const Pair<T>& thePair);
//Precondition: The operator + is defined for values of type T.
//Returns the sum of the two values in thePair.
```

The ASCII Character Set

Only the printable characters are shown. Character number 32 is the blank.

32		56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	\$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(64	@	88	X	112	p
41)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[115	s
44	,	68	D	92	\	116	t
45	-	69	E	93]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g		

Some Library Functions

The following lists are organized according to what the function is used for, rather than what library it is in. The function declaration gives the number and types of arguments as well as the type of the value returned. In most cases, the function declarations give only the type of the parameter and do not give a parameter name. (See the section “Alternate Form for Function Declarations” in Chapter 4 for an explanation of this kind of function declaration.)

Arithmetic Functions

Function Declaration	Description	Header File
<code>int abs(int);</code>	Absolute value	<code>cstdlib</code>
<code>long labs(long);</code>	Absolute value	<code>cstdlib</code>
<code>double fabs(double);</code>	Absolute value	<code>cmath</code>
<code>double sqrt(double);</code>	Square root	<code>cmath</code>
<code>double pow(double, double);</code>	Returns the first argument raised to the power of the second argument.	<code>cmath</code>
<code>double exp(double);</code>	Returns e (base of the natural logarithm) to the power of its argument.	<code>cmath</code>
<code>double log(double);</code>	Natural logarithm (ln)	<code>cmath</code>
<code>double log10(double);</code>	Base 10 logarithm	<code>cmath</code>
<code>double ceil(double);</code>	Returns the smallest integer that is greater than or equal to its argument.	<code>cmath</code>
<code>double floor(double);</code>	Returns the largest integer that is less than or equal to its argument.	<code>cmath</code>

Input and Output Member Functions

Form of a Function Call	Description	Header File
<code>Stream_Var.open (External_File_Name);</code>	Connects the file with the <i>External_File_Name</i> to the stream named by the <i>Stream_Var</i> . The <i>External_File_Name</i> is a string value.	<code>fstream</code>
<code>Stream_Var.fail();</code>	Returns <i>true</i> if the previous operation (such as <code>open</code>) on the stream <i>Stream_Var</i> has failed.	<code>fstream</code> or <code>iostream</code>
<code>Stream_Var.close();</code>	Disconnects the stream <i>Stream_Var</i> from the file it is connected to.	<code>fstream</code>
<code>Stream_Var.bad();</code>	Returns <i>true</i> if the stream <i>Stream_Var</i> is corrupted.	<code>fstream</code> or <code>iostream</code>
<code>Stream_Var.eof();</code>	Returns <i>true</i> if the program has attempted to read beyond the last character in the file connected to the input stream <i>Stream_Var</i> . Otherwise, it returns <i>false</i> .	<code>fstream</code> or <code>iostream</code>
<code>Stream_Var.get (Char_Variable);</code>	Reads one character from the input stream <i>Stream_Var</i> and sets the <i>Char_Variable</i> equal to this character. Does <i>not</i> skip over whitespace.	<code>fstream</code> or <code>iostream</code>
<code>Stream_Var.getline (String_Var, Max_Characters +1);</code>	One line of input from the stream <i>Stream_Var</i> is read, and the resulting string is placed in <i>String_Var</i> . If the line is more than <i>Max_Characters</i> long, only the first <i>Max_Characters</i> are read. The declared size of the <i>String_Var</i> should be <i>Max_Characters</i> +1 or larger.	<code>fstream</code> or <code>iostream</code>
<code>Stream_Var.peek();</code>	Reads one character from the input stream <i>Stream_Var</i> and returns that character. But the character read is <i>not</i> removed from the input stream; the next read will read the same character.	<code>fstream</code> or <code>iostream</code>

Input and Output Member Functions (*continued*)

Form of a Function Call	Description	Header File
<i>Stream_Var</i> .put(<i>Char_Exp</i>);	Writes the value of the <i>Char_Exp</i> to the output stream <i>Stream_Var</i> .	fstream or iostream
<i>Stream_Var</i> .putback(<i>Char_Exp</i>);	Places the value of <i>Char_Exp</i> in the input stream <i>Stream_Var</i> so that that value is the next input value read from the stream. The file connected to the stream is not changed.	fstream or iostream
<i>Stream_Var</i> .precision(<i>Int_Exp</i>);	Specifies the number of digits output after the decimal point for floating-point values sent to the output stream <i>Stream_Var</i> .	fstream or iostream
<i>Stream_Var</i> .width(<i>Int_Exp</i>);	Sets the field width for the next value output to the stream <i>Stream_Var</i> .	fstream or iostream
<i>Stream_Var</i> .setf(<i>Flag</i>);	Sets flags for formatting output to the stream <i>Stream_Var</i> . See Display 6.5 for the list of possible flags.	fstream or iostream
<i>Stream_Var</i> .unsetf(<i>Flag</i>);	Unsets flags for formatting output to the stream <i>Stream_Var</i> . See Display 6.5 for the list of possible flags.	fstream or iostream

Character Functions

For all of these the actual type of the argument is *int*, but for most purposes you can think of the argument type as *char*. If the value returned is a value of type *int*, you must perform an explicit or implicit typecast to obtain a *char*.

Function Declaration	Description	Header File
<i>bool</i> isalnum(<i>char</i>);	Returns <i>true</i> if its argument satisfies either isalpha or isdigit. Otherwise, returns <i>false</i> .	cctype
<i>bool</i> isalpha(<i>char</i>);	Returns <i>true</i> if its argument is an upper- or lowercase letter. It may also return <i>true</i> for other arguments. The details are implementation dependent. Otherwise, returns <i>false</i> .	cctype
<i>bool</i> isdigit(<i>char</i>);	Returns <i>true</i> if its argument is a digit. Otherwise, returns <i>false</i> .	cctype
<i>bool</i> ispunct(<i>char</i>);	Returns <i>true</i> if its argument is a printable character that does not satisfy isalnum and is not whitespace. (These characters are considered punctuation characters.) Otherwise, returns <i>false</i> .	cctype
<i>bool</i> isspace(<i>char</i>);	Returns <i>true</i> if its argument is a whitespace character (such as blank, tab, or new line). Otherwise, returns <i>false</i> .	cctype
<i>bool</i> iscntrl(<i>char</i>);	Returns <i>true</i> if its argument is a control character. Otherwise, returns <i>false</i> .	cctype
<i>bool</i> islower(<i>char</i>);	Returns <i>true</i> if its argument is a lowercase letter. Otherwise, returns <i>false</i> .	cctype
<i>bool</i> isupper(<i>char</i>);	Returns <i>true</i> if its argument is an uppercase letter. Otherwise, returns <i>false</i> .	cctype
<i>int</i> tolower(<i>char</i>);	Returns the lowercase version of its argument. If there is no lowercase version, returns its argument unchanged.	cctype
<i>int</i> toupper(<i>char</i>);	Returns the uppercase version of its argument. If there is no uppercase version, returns its argument unchanged.	cctype

String Functions

Function Declaration	Description	Header File
<code>int atoi(const chara[]);</code>	Converts a string of characters to an integer.	cstdlib
<code>int stoi(const string)</code>	Converts a STL string object to an integer. C++11 and higher.	string
<code>long atol(const chara[]);</code>	Converts a string of characters to a long integer.	cstdlib
<code>long stol(const string)</code>	Converts a STL string object to a long. C++11 and higher.	string
<code>double atof(const char a[]);</code>	Converts a string of characters to a double.	cstdlib ¹
<code>strcat(String_Variable, String_Expression);</code>	Appends the value of the <i>String_Expression</i> to the end of the string in the <i>String_Variable</i> .	cstring
<code>strcmp(String_Exp1, String_Exp2)</code>	Returns <i>true</i> if the values of the two string expressions are different; otherwise, returns <i>false</i> . ²	cstring
<code>strcpy(String_Variable, String_Expression);</code>	Changes the value of the <i>String_Variable</i> to the value of the <i>String_Expression</i> .	cstring
<code>strlen(String_Expression)</code>	Returns the length of the <i>String_Expression</i> .	cstring
<code>strncat(String_Variable, String_Expression, Limit);</code>	Same as <code>strcat</code> except that at most <i>Limit</i> characters are appended.	cstring
<code>strncmp(String_Exp1, String_Exp2, Limit)</code>	Same as <code>strcmp</code> except that at most <i>Limit</i> characters are compared.	cstring
<code>strncpy(String_Variable, String_Expression, Limit);</code>	Same as <code>strcpy</code> except that at most <i>Limit</i> characters are copied.	cstring
<code>strstr(String_Expression, Pattern)</code>	Returns a pointer to the first occurrence of the string <i>Pattern</i> in <i>String_Expression</i> . Returns the NULL pointer if the <i>Pattern</i> is not found.	cstring
<code>strchr(String_Expression, Character)</code>	Returns a pointer to the first occurrence of the <i>Character</i> in <i>String_Expression</i> . Returns the NULL pointer if <i>Character</i> is not found.	cstring
<code>strrchr(String_Expression, Character)</code>	Returns a pointer to the last occurrence of the <i>Character</i> in <i>String_Expression</i> . Returns the NULL pointer if <i>Character</i> is not found.	cstring

¹ Some implementations place it in `cmath`.

² Returns an integer that is less than zero, zero, or greater than zero according to whether *String_Exp1* is less than, equal to, or greater than *String_Exp2*, respectively. The ordering is lexicographic ordering.

Random Number Generator

Function Declaration	Description	Header File
<code>int random(int);</code>	The call <code>random(n)</code> returns a pseudorandom integer greater than or equal to 0 and less than or equal to <code>n-1</code> . (Not available in all implementations. If not available, then you must use <code>rand</code> .)	<code>cstdlib</code>
<code>int rand();</code>	The call <code>rand()</code> returns a pseudorandom integer greater than or equal to 0 and less than or equal to <code>RAND_MAX</code> . <code>RAND_MAX</code> is a predefined integer constant that is defined in <code>cstdlib</code> . The value of <code>RAND_MAX</code> is implementation dependent but will be at least 32767.	<code>cstdlib</code>
<code>void srand(unsigned int);</code> <code>void srandom(unsigned int);</code> (The type <code>unsigned int</code> is an integer type that only allows nonnegative values. You can think of the argument type as <code>int</code> with the restriction that it must be nonnegative.)	Reinitializes the random number generator. The argument is the seed. Calling <code>srand</code> multiple times with the same argument will cause <code>rand</code> or <code>random</code> (whichever you use) to produce the same sequence of pseudorandom numbers. If <code>rand</code> or <code>random</code> is called without any previous call to <code>srand</code> , the sequence of numbers produced is the same as if there had been a call to <code>srand</code> with an argument of 1.	<code>cstdlib</code>

Trigonometric Functions

These functions use radians, not degrees.

Function Declaration	Description	Header File
<code>double acos(double);</code>	Arc cosine	cmath
<code>double asin(double);</code>	Arc sine	cmath
<code>double atan(double);</code>	Arc tangent	cmath
<code>double cos(double);</code>	Cosine	cmath
<code>double cosh(double);</code>	Hyperbolic cosine	cmath
<code>double sin(double);</code>	Sine	cmath
<code>double sinh(double);</code>	Hyperbolic sine	cmath
<code>double tan(double);</code>	Tangent	cmath
<code>double tanh(double);</code>	Hyperbolic tangent	cmath