CHAPTER 16

Exceptions, Templates, and the Standard Template Library (STL)

TOPICS

16.1 Exceptions
16.2 Function Templates
16.3 Focus on Software Engineering: Where to Start When Defining Templates
16.4 Class Templates
16.5 Introduction to the Standard Template Library (STL)

16.1 Exceptions

CONCEPT: Exceptions are used to signal errors or unexpected events that occur while a program is running.

Error testing is usually a straightforward process involving `if` statements or other control mechanisms. For example, the following code segment will trap a division-by-zero error before it occurs:

```
if (denominator == 0)
     cout << "ERROR: Cannot divide by zero.\n";
else
     quotient = numerator / denominator;
```

But what if similar code is part of a function that returns the quotient, as in the following example?

```
// An unreliable division function
double divide(int numerator, int denominator)
{
     if (denominator == 0)
     {
          cout << "ERROR: Cannot divide by zero.\n";
          return 0;
     }
          else
               return static_cast<double>(numerator) / denominator;
}
```

Functions commonly signal error conditions by returning a predetermined value. Apparently, the function in this example returns 0 when division by zero has been attempted. This is unreliable, however, because 0 is a valid result of a division operation. Even though the function displays an error message, the part of the program that calls the function will not know when an error has occurred. Problems like these require sophisticated error handling techniques.

Throwing an Exception

One way of handling complex error conditions is with *exceptions*. An exception is a value or an object that signals an error. When the error occurs, an exception is "thrown." For example, the following code shows the `divide` function, modified to throw an exception when division by zero has been attempted.

VideoNote
Throwing an Exception

```
double divide(int numerator, int denominator)
{
    if (denominator == 0)
        throw "ERROR: Cannot divide by zero.\n";
    else
        return static_cast<double>(numerator) / denominator;
}
```

The following statement causes the exception to be thrown.

```
throw "ERROR: Cannot divide by zero.\n";
```

The `throw` key word is followed by an argument, which can be any value. As you will see, the value of the argument is used to determine the nature of the error. The function above simply throws a string containing an error message.

The line containing a `throw` statement is known as the *throw point*. When a `throw` statement is executed, control is passed to another part of the program known as an *exception handler*. When an exception is thrown by a function, the function aborts.

Handling an Exception

To handle an exception, a program must have a *try/catch* construct. The general format of the try/catch construct is:

```
try
{
    // code here calls functions or object member
    // functions that might throw an exception.
}
catch(ExceptionParameter)
{
    // code here handles the exception
}
// Repeat as many catch blocks as needed.
```

VideoNote
Handling an Exception

The first part of the construct is the *try block*. This starts with the key word `try` and is followed by a block of code executing any statements that might directly or indirectly cause an exception to be thrown. The try block is immediately followed by one or more *catch blocks*, which are the exception handlers. A catch block starts with the key word `catch`, followed by a set of parentheses containing the definition of an exception parameter. For example, here is a try/catch construct that can be used with the `divide` function:

```
try
{
    quotient = divide(num1, num2);
    cout << "The quotient is " << quotient << endl;
}
catch (string exceptionString)
{
    cout << exceptionString;
}
```

Because the `divide` function throws an exception whose value is a string, there must be an exception handler that catches a string. The catch block shown catches the error message in the `exceptionString` parameter and then displays it with `cout`. Now let's look at an entire program to see how `throw`, `try`, and `catch` work together. In the first sample run of Program 16-1, valid data are given. This shows how the program should run with no errors. In the second sample running, a denominator of 0 is given. This shows the result of the exception being thrown.

Program 16-1

```
// This program demonstrates an exception being thrown and caught.
#include <iostream>
#include <string>
using namespace std;

// Function prototype
double divide(int, int);

int main()
{
    int num1, num2;  // To hold two numbers
    double quotient; // To hold the quotient of the numbers

    // Get two numbers.
    cout << "Enter two numbers: ";
    cin >> num1 >> num2;

    // Divide num1 by num2 and catch any
    // potential exceptions.
    try
    {
        quotient = divide(num1, num2);
        cout << "The quotient is " << quotient << endl;
    }
    catch (string exceptionString)
    {
        cout << exceptionString;
    }

```

(program continues)

Program 16-1 *(continued)*

```
    cout << "End of the program.\n";
    return 0;
}

//*******************************************
// The divide function divides the numerator *
// by the denominator. If the denominator is *
// zero, the function throws an exception.   *
//*******************************************

double divide(int numerator, int denominator)
{
    if (denominator == 0)
    {
        string exceptionString = "ERROR: Cannot divide by zero.\n";
        throw exceptionString;
    }

    return static_cast<double>(numerator) / denominator;
}
```

Program Output with Example Input Shown in Bold

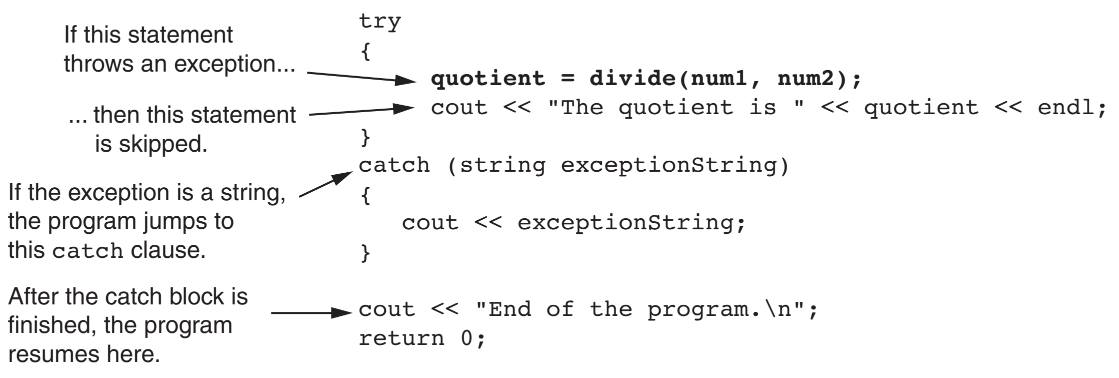
```
Enter two numbers: 12 2 [Enter]
The quotient is 6
End of the program.
```

Program Output with Different Example Input Shown in Bold

```
Enter two numbers: 12 0 [Enter]
ERROR: Cannot divide by zero.
End of the program.
```

As you can see from the second output screen, the exception caused the program to jump out of the `divide` function and into the catch block. After the catch block has finished, the program resumes with the first statement after the try/catch construct. This is illustrated in Figure 16-1.

Figure 16-1

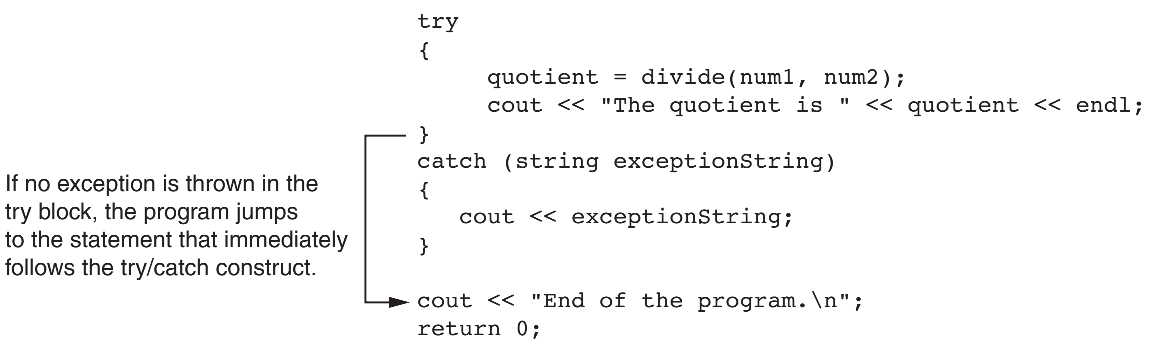

If this statement throws an exception...
... then this statement is skipped.
If the exception is a string, the program jumps to this catch clause.
After the catch block is finished, the program resumes here.

```
try
{
   quotient = divide(num1, num2);
   cout << "The quotient is " << quotient << endl;
}
catch (string exceptionString)
{
   cout << exceptionString;
}

cout << "End of the program.\n";
return 0;
```

In the first output screen the user entered nonnegative values. No exception was thrown in the try block, so the program skipped the catch block and jumped to the statement immediately following the try/catch construct, which is in line 30. This is illustrated in Figure 16-2.

Figure 16-2



If no exception is thrown in the try block, the program jumps to the statement that immediately follows the try/catch construct.

```
try
{
     quotient = divide(num1, num2);
     cout << "The quotient is " << quotient << endl;
}
catch (string exceptionString)
{
   cout << exceptionString;
}

cout << "End of the program.\n";
return 0;
```

What if an Exception Is Not Caught?

There are two possible ways for a thrown exception to go uncaught. The first possibility is for the try/catch construct to contain no catch blocks with an exception parameter of the right data type. The second possibility is for the exception to be thrown from outside a try block. In either case, the exception will cause the entire program to abort execution.

Object-Oriented Exception Handling with Classes

Now that you have an idea of how the exception mechanism in C++ works, we will examine an object-oriented approach to exception handling. Recall the `Rectangle` class that was introduced in Chapter 13. That class had the mutator functions `setWidth` and `setLength` for setting the rectangle's width and length. If a negative value was passed to either of these functions, the class displayed an error message and aborted the program. The following code shows an improved version of the `Rectangle` class. This version throws an exception when a negative value is passed to `setWidth` or `setLength`. (These files are stored in the Student Source Code Folder `Chapter 16\Rectangle Version 1`.)

Contents of `Rectangle.h` (Version 1)

```
// Specification file for the Rectangle class
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
    private:
        double width;     // The rectangle's width
        double length;    // The rectangle's length
    public:
        // Exception class
        class NegativeSize
            { };          // Empty class declaration

15          // Default constructor
16          Rectangle()
17              { width = 0.0; length = 0.0; }
18
19          // Mutator functions, defined in Rectangle.cpp
20          void setWidth(double);
21          void setLength(double);
22
23          // Accessor functions
24          double getWidth() const
25              { return width; }
26
27          double getLength() const
28              { return length; }
29
30          double getArea() const
31              { return width * length; }
32  };
33  #endif
```

Notice the empty class declaration that appears in the public section, in lines 12 and 13. The `NegativeSize` class has no members. The only important part of the class is its name, which will be used in the exception-handling code. Now look at the `Rectangle.cpp` file, where the `setWidth` and `setLength` member functions are defined.

Contents of `Rectangle.cpp` (Version 1)

```
 1  // Implementation file for the Rectangle class.
 2  #include "Rectangle.h"
 3
 4  //***********************************************************
 5  // setWidth sets the value of the member variable width. *
 6  //***********************************************************
 7
 8  void Rectangle::setWidth(double w)
 9  {
10      if (w >= 0)
11          width = w;
12      else
13          throw NegativeSize();
14  }
15
16  //*************************************************************
17  // setLength sets the value of the member variable length. *
18  //*************************************************************
19
20  void Rectangle::setLength(double len)
21  {
22      if (len >= 0)
23          length = len;
24      else
25          throw NegativeSize();
26  }
```

In the `setWidth` function, the parameter `w` is tested by the `if` statement in line 10. If `w` is greater than or equal to 0, its value is assigned to the `width` member variable. If `w` holds a negative number, however, the statement in line 13 is executed:

```
throw NegativeSize();
```

The `throw` statement's argument, `NegativeSize()`, causes an instance of the `NegativeSize` class to be created and thrown as an exception.

The same series of events takes place in the `setLength` function. If the value in the `len` parameter is greater than or equal to 0, its value is assigned to the `length` member variable. If `len` holds a negative number, an instance of the `NegativeSize` class is thrown as an exception in line 25.

This way of reporting errors is much more graceful than simply aborting the program. Any code that uses the `Rectangle` class must simply have a catch block to handle the `NegativeSize` exceptions that the `Rectangle` class might throw. Program 16-2 shows an example. (This file is stored in the Student Source Code Folder `Chapter 16\ Rectangle Version 1`.)

Program 16-2

```
// This program demonstrates Rectangle class exceptions.
#include <iostream>
#include "Rectangle.h"
using namespace std;

int main()
{
    double width;
    double length;

    // Create a Rectangle object.
    Rectangle myRectangle;

    // Get the width and length.
    cout << "Enter the rectangle's width: ";
    cin >> width;
    cout << "Enter the rectangle's length: ";
    cin >> length;

    // Store these values in the Rectangle object.
    try
    {
        myRectangle.setWidth(width);
        myRectangle.setLength(length);
        cout << "The area of the rectangle is "
             << myRectangle.getArea() << endl;
    }
    catch (Rectangle::NegativeSize)
    {
        cout << "Error: A negative value was entered.\n";
    }
```

(program continues)

Program 16-2 *(continued)*

```
    cout << "End of the program.\n";

    return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter the rectangle's width: 10 [Enter]
Enter the rectangle's length: 20 [Enter]
The area of the rectangle is 200
End of the program.
```

Program Output with Different Example Input Shown in Bold

```
Enter the rectangle's width: 5 [Enter]
Enter the rectangle's length: −5 [Enter]
Error: A negative value was entered.
End of the program.
```

The `catch` statement in line 28 catches the `NegativeSize` exception when it is thrown by any of the statements in the try block. Inside the `catch` statement's parentheses is the name of the `NegativeSize` class. Because the `NegativeSize` class is declared inside the `Rectangle` class, we have to fully qualify the class name with the scope resolution operator.

Notice that we did not define a parameter of the `NegativeSize` class in the `catch` statement. In this case the `catch` statement only needs to specify the type of exception it handles.

Multiple Exceptions

The programs we have studied so far test only for a single type of error and throw only a single type of exception. In many cases a program will need to test for several different types of errors and signal which one has occurred. C++ allows you to throw and catch multiple exceptions. The only requirement is that each different exception be of a different type. You then code a separate catch block for each type of exception that may be thrown in the try block.

For example, suppose we wish to expand the `Rectangle` class so it throws one type of exception when a negative value is specified for the `width`, and another type of exception when a negative value is specified for the `length`. First, we declare two different exception classes, such as:

```
// Exception class for a negative width
class NegativeWidth
    { };

// Exception class for a negative length
class NegativeLength
    { };
```

An instance of `NegativeWidth` will be thrown when a negative value is specified for the `width`, and an instance of `NegativeLength` will be thrown when a negative value is specified for the `length`. The code for the modified `Rectangle` class is shown here. (These files are stored in the Student Source Code Folder `Chapter 16\Rectangle Version 2`.)

Contents of `Rectangle.h` (Version 2)

```
// Specification file for the Rectangle class
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
    private:
        double width;      // The rectangle's width
        double length;     // The rectangle's length
    public:
        // Exception class for a negative width
        class NegativeWidth
            { };

        // Exception class for a negative length
        class NegativeLength
            { };

        // Default constructor
        Rectangle()
            { width = 0.0; length = 0.0; }

        // Mutator functions, defined in Rectangle.cpp
        void setWidth(double);
        void setLength(double);

        // Accessor functions
        double getWidth() const
            { return width; }

        double getLength() const
            { return length; }

        double getArea() const
            { return width * length; }
};
#endif
```

Contents of `Rectangle.cpp` (Version 2)

```
// Implementation file for the Rectangle class.
#include "Rectangle.h"

//***********************************************************
// setWidth sets the value of the member variable width.   *
//***********************************************************

void Rectangle::setWidth(double w)
{
    if (w >= 0)
        width = w;
12        else
13             throw NegativeWidth();
14   }
15
16   //*********************************************************
17   // setLength sets the value of the member variable length. *
18   //*********************************************************
19
20   void Rectangle::setLength(double len)
21   {
22        if (len >= 0)
23             length = len;
24        else
25             throw NegativeLength();
26   }
```

Notice that in the definition of the `setWidth` function (in `Rectangle.cpp`) that an instance of the `NegativeWidth` class is thrown in line 13. In the definition of the `setLength` function an instance of the `NegativeLength` class is thrown in line 25. Program 16-3 demonstrates this class. (This file is stored in the Student Source Code Folder `Chapter 16\Rectangle Version 2`.)

Program 16-3

```
// This program demonstrates Rectangle class exceptions.
#include <iostream>
#include "Rectangle.h"
using namespace std;

int main()
{
     double width;
     double length;

     // Create a Rectangle object.
     Rectangle myRectangle;

     // Get the width and length.
     cout << "Enter the rectangle's width: ";
     cin >> width;
     cout << "Enter the rectangle's length: ";
     cin >> length;

     // Store these values in the Rectangle object.
     try
     {
          myRectangle.setWidth(width);
          myRectangle.setLength(length);
          cout << "The area of the rectangle is "
               << myRectangle.getArea() << endl;
     }
     catch (Rectangle::NegativeWidth)
29      {
30          cout << "Error: A negative value was given "
31               << "for the rectangle's width.\n";
32      }
33      catch (Rectangle::NegativeLength)
34      {
35          cout << "Error: A negative value was given "
36               << "for the rectangle's length.\n";
37      }
38
39      cout << "End of the program.\n";
40      return 0;
41  }
```

Program Output with Example Input Shown in Bold

```
Enter the rectangle's width: 10 [Enter]
Enter the rectangle's length: 20 [Enter]
The area of the rectangle is 200
End of the program.
```

Program Output with Different Example Input Shown in Bold

```
Enter the rectangle's width: –5 [Enter]
Enter the rectangle's length: 5 [Enter]
Error: A negative value was given for the rectangle's width.
End of the program.
```

Program Output with Different Example Input Shown in Bold

```
Enter the rectangle's width: 5 [Enter]
Enter the rectangle's length: –5 [Enter]
Error: A negative value was given for the rectangle's length.
End of the program.
```

The try block, in lines 21 through 27, contains code that can throw two different types of exceptions. The statement in line 23 can potentially throw a `NegativeWidth` exception, and the statement in line 24 can potentially throw a `NegativeLength` exception. To handle each of these types of exception, there are two `catch` statements. The statement in line 28 catches `NegativeWidth` exceptions, and the statement in line 33 catches `NegativeLength` exceptions.

When an exception is thrown by code in the try block, C++ searches the try/catch construct for a `catch` statement that can handle the exception. If the construct contains a `catch` statement that is compatible with the exception, control of the program is passed to the catch block.

Using Exception Handlers to Recover from Errors

Program 16-3 demonstrates how a try/catch construct can have several `catch` statements in order to handle different types of exceptions. However, the program does not use the exception handlers to recover from any of the errors. When the user enters a negative value for either the width or the length, this program still halts. Program 16-4 shows a better example of effective exception handling. It attempts to recover from the exceptions and get valid data from the user. (This file is stored in the Student Source Code Folder `Chapter 16\Rectangle Version 2`.)

Program 16-4

```
// This program handles the Rectangle class exceptions.
#include <iostream>
#include "Rectangle.h"
using namespace std;

int main()
{
    double width;           // Rectangle's width
    double length;          // Rectangle's length
    bool tryAgain = true;   // Flag to reread input

    // Create a Rectangle object.
    Rectangle myRectangle;

    // Get the rectangle's width.
    cout << "Enter the rectangle's width: ";
    cin >> width;

    // Store the width in the myRectangle object.
    while (tryAgain)
    {
        try
        {
            myRectangle.setWidth(width);
            // If no exception was thrown, then the
            // next statement will execute.
            tryAgain = false;
        }
        catch (Rectangle::NegativeWidth)
        {
            cout << "Please enter a nonnegative value: ";
            cin >> width;
        }
    }

    // Get the rectangle's length.
    cout << "Enter the rectangle's length: ";
    cin >> length;

    // Store the length in the myRectangle object.
    tryAgain = true;
    while (tryAgain)
    {
        try
        {
            myRectangle.setLength(length);
            // If no exception was thrown, then the
            // next statement will execute.
            tryAgain = false;
        }
        catch (Rectangle::NegativeLength)
52            {
53                cout << "Please enter a nonnegative value: ";
54                cin >> length;
55            }
56        }
57
58        // Display the area of the rectangle.
59        cout << "The rectangle's area is "
60             << myRectangle.getArea() << endl;
61        return 0;
62    }
```

Program Output with Example Input Shown in Bold

```
Enter the rectangle's width: −1 [Enter]
Please enter a nonnegative value: 10 [Enter]
Enter the rectangle's length: −5 [Enter]
Please enter a nonnegative value: 50 [Enter]
The rectangle's area is 500
```

Let's look at how this program recovers from a `NegativeWidth` exception. In line 10 a `bool` flag variable, `tryAgain`, is defined and initialized with the value `true`. This variable will indicate whether we need to get a value from the user again. Lines 16 and 17 prompt the user to enter the rectangle's width. Then the program enters the `while` loop in lines 20 through 34. The loop repeats as long as `tryAgain` is `true`. Inside the loop, the `Rectangle` class's `setWidth` member function is called in line 24. This statement is in a try block. If a `NegativeWidth` exception is thrown, the program will jump to the `catch` statement in line 29. In the `catch` block that follows, the user is asked to enter a nonnegative number. The program then jumps out of the try/catch construct. Because `tryAgain` is still `true`, the loop will repeat.

If a nonnegative number is passed to the `setWidth` member function in line 24, no exception will be thrown. In that case, the statement in line 27 will execute, which sets `tryAgain` to `false`. The program then jumps out of the try/catch construct. Because `tryAgain` is now `false`, the loop will not repeat.

The same strategy is used in lines 37 through 56 to get and validate the rectangle's length.

Extracting Data from the Exception Class

Sometimes we might want an exception object to pass data back to the exception handler. For example, suppose we would like the `Rectangle` class not only to signal when a negative value has been given, but also to pass the value back. This can be accomplished by giving the exception class members in which data can be stored.

In our next modification of the `Rectangle` class, the `NegativeWidth` and `NegativeLength` classes have been expanded, each with a member variable and a constructor. Here is the code for the `NegativeWidth` class:

```
class NegativeWidth
{
private:
     double value;
public:
    NegativeWidth(double val)
        { value = val; }

    double getValue() const
        { return value; }
};
```

When we throw this exception, we want to pass the invalid value as an argument to the class's constructor. This is done in the `setWidth` member function with the following statement:

```
throw NegativeWidth(w);
```

This `throw` statement creates an instance of the `NegativeWidth` class and passes a copy of the `w` variable to the constructor. The constructor then stores this number in `NegativeWidth`'s member variable, `value`. The class instance carries this member variable to the catch block that intercepts the exception.

In the catch block, the value is extracted with code such as

```
catch (Rectangle::NegativeWidth e)
{
    cout << "Error: " << e.getValue()
         << " is an invalid value for the"
         << " rectangle's width.\n";
}
```

Notice that the catch block defines a parameter object named `e`. This is necessary because we want to call the class's `getValue` function to retrieve the value that caused the exception.

Here is the code for the `NegativeLength` class:

```
class NegativeLength
{
private:
    double value;
public:
    NegativeLength(double val)
        { value = val; }

    double getValue() const
        { return value; }
};
```

This class also has a member variable named `value` and a constructor that initializes the variable. When we throw this exception, we follow the sane general steps that were just described for the `NegativeWidth` exception. The complete code for the revised `Rectangle` class is shown here. Program 16-5 demonstrates these classes. (These files are stored in the Student Source Code Folder `Chapter 16\Rectangle Version 3`.)

Contents of `Rectangle.h` (Version 3)

```
// Specification file for the Rectangle class
#ifndef RECTANGLE_H
#define RECTANGLE_H

```

```
class Rectangle
{
    private:
        double width;      // The rectangle's width
        double length;     // The rectangle's length
    public:
        // Exception class for a negative width
        class NegativeWidth
        {
        private:
            double value;
        public:
            NegativeWidth(double val)
                { value = val; }

            double getValue() const
                { return value; }
        };

        // Exception class for a negative length
        class NegativeLength
        {
        private:
            double value;
        public:
            NegativeLength(double val)
                { value = val; }

            double getValue() const
                { return value; }
        };

        // Default constructor
        Rectangle()
            { width = 0.0; length = 0.0; }

        // Mutator functions, defined in Rectangle.cpp
        void setWidth(double);
        void setLength(double);

        // Accessor functions
        double getWidth() const
            { return width; }

        double getLength() const
            { return length; }

        double getArea() const
            { return width * length; }
};
#endif
```

Contents of `Rectangle.cpp` (Version 3)

```
// Implementation file for the Rectangle class.
#include "Rectangle.h"

//***********************************************************
// setWidth sets the value of the member variable width.   *
//***********************************************************

void Rectangle::setWidth(double w)
{
    if (w >= 0)
        width = w;
    else
        throw NegativeWidth(w);
}

//***********************************************************
// setLength sets the value of the member variable length. *
//***********************************************************

void Rectangle::setLength(double len)
{
    if (len >= 0)
        length = len;
    else
        throw NegativeLength(len);
}
```

Program 16-5

```
// This program demonstrates Rectangle class exceptions.
#include <iostream>
#include "Rectangle.h"
using namespace std;

int main()
{
    double width;
    double length;

    // Create a Rectangle object.
    Rectangle myRectangle;

    // Get the width and length.
    cout << "Enter the rectangle's width: ";
    cin >> width;
    cout << "Enter the rectangle's length: ";
    cin >> length;

    // Store these values in the Rectangle object.
    try
```

```
        {
            myRectangle.setWidth(width);
            myRectangle.setLength(length);
            cout << "The area of the rectangle is "
                 << myRectangle.getArea() << endl;
        }
        catch (Rectangle::NegativeWidth e)
        {
            cout << "Error: " << e.getValue()
                 << " is an invalid value for the"
                 << " rectangle's width.\n";
        }
        catch (Rectangle::NegativeLength e)
        {
            cout << "Error: " << e.getValue()
                 << " is an invalid value for the"
                 << " rectangle's length.\n";
        }

        cout << "End of the program.\n";
        return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter the rectangle's width: -1 [Enter]
Enter the rectangle's length: 10 [Enter]
Error: -1 is an invalid value for the rectangle's width.
End of the program.
```

Program Output with Different Example Input Shown in Bold

```
Enter the rectangle's width: 5 [Enter]
Enter the rectangle's length: -1 [Enter]
Error: -1 is an invalid value for the rectangle's length.
End of the program.
```

Unwinding the Stack

Once an exception has been thrown, the program cannot jump back to the throw point. The function that executes a `throw` statement will immediately terminate. If that function was called by another function, and the exception is not caught, then the calling function will terminate as well. This process, known as *unwinding the stack*, continues for the entire chain of nested function calls, from the throw point, all the way back to the try block.

If an exception is thrown by the member function of a class object, then the class destructor is called. If statements in the try block or branching from the try block created any other objects, their destructors will be called as well.

Rethrowing an Exception

It is possible for try blocks to be nested. For example, look at this code segment:

```
try
{
     doSomething();
}
catch(exception1)
{
     // code to handle exception 1
}
catch(exception2)
{
     // code to handle exception 2
}
```

In this try block, the function `doSomething` is called. There are two catch blocks, one that handles `exception1` and another that handles `exception2`. If the `doSomething` function also has a try block, then it is nested inside the one shown.

With nested try blocks, it is sometimes necessary for an inner exception handler to pass an exception to an outer exception handler. Sometimes, both an inner and an outer catch block must perform operations when a particular exception is thrown. These situations require that the inner catch block *rethrow* the exception so the outer catch block has a chance to catch it.

A catch block can rethrow an exception with the `throw;` statement. For example, suppose the `doSomething` function (called in the throw block above) calls the `doSomethingElse` function, which potentially can throw `exception1` or `exception3`. Suppose `doSomething` does not want to handle `exception1`. Instead, it wants to rethrow it to the outer block. The following code segment illustrates how this is done:

```
try
{
     doSomethingElse();
}
catch(exception1)
{
     throw; // Rethrow the exception
}
catch(exception3)
{
     // Code to handle exception 3
}
```

When the first catch block catches `exception1`, the `throw;` statement simply throws the exception again. The catch block in the outer try/catch construct will then handle the exception.

Handling the `bad_alloc` Exception

Recall from Chapter 9 that when the `new` operator fails to allocate memory, an exception is thrown. Now that you've seen how to handle exceptions, you can write code that determines whether the `new` operator was successful.

When the new operator fails to allocate memory, C++ throws a bad_alloc exception. The bad_alloc exception type is defined in the new header file, so any program that attempts to catch this exception should have the following directive:

```
#include <new>
```

The bad_alloc exception is in the std namespace, so be sure to have the using namespace std; statement in your code as well.

Here is the general format of a try/catch construct that catches the bad_alloc exception:

```
try
{
     // Code that uses the new operator
}
catch (bad_alloc)
{
     // Code that responds to the error
}
```

Program 16-6 shows an example. The program uses the new operator to allocate a 10,000-element array of doubles. If the new operator fails, an error message is displayed.

Program 16-6

```
// This program demonstrates the bad_alloc exception.
#include <iostream>
#include <new>          // Needed for bad_alloc
using namespace std;

int main()
{
     double *ptr = nullptr; // Pointer to double

     try
     {
          ptr = new double [10000];
     }
     catch (bad_alloc)
     {
          cout << "Insufficient memory.\n";
     }

     return 0;
}
```

Checkpoint

16.1 What is the difference between a try block and a catch block?

16.2 What happens if an exception is thrown, but not caught?

16.3 If multiple exceptions can be thrown, how does the catch block know which exception to catch?

16.4 After the catch block has handled the exception, where does program execution resume?

16.5 How can an exception pass data back to the exception handler?

16.2 Function Templates

CONCEPT: A function template is a "generic" function that can work with any data type. The programmer writes the specifications of the function, but substitutes parameters for data types. When the compiler encounters a call to the function, it generates code to handle the specific data type(s) used in the call.

Introduction

Overloaded functions make programming convenient because only one function name must be remembered for a set of functions that perform similar operations. Each of the functions, however, must still be written individually, even if they perform the same operation. For example, suppose a program uses the following overloaded `square` functions.

```
int square(int number)
{
    return number * number;
}
double square(double number)
{
    return number * number;
}
```

The only differences between these two functions are the data types of their return values and their parameters. In situations like this, it is more convenient to write a *function template* than an overloaded function. Function templates allow you to write a single function definition that works with many different data types, instead of having to write a separate function for each data type used.

A function template is not an actual function, but a "mold" the compiler uses to generate one or more functions. When writing a function template, you do not have to specify actual types for the parameters, return value, or local variables. Instead, you use a *type parameter* to specify a generic data type. When the compiler encounters a call to the function, it examines the data types of its arguments and generates the function code that will work with those data types. (The generated code is known as a *template function*.)

VideoNote
Writing a Function Template

Here is a function template for the `square` function:

```
template <class T>
T square(T number)
{
    return number * number;
}
```

The beginning of a function template is marked by a *template prefix*, which begins with the key word `template`. Next is a set of angled brackets that contains one or more generic data types used in the template. A generic data type starts with the key word `class` followed by a parameter name that stands for the data type. The example just given only uses one, which is named `T`. (If there were more, they would be separated by commas.) After this, the function definition is written as usual, except the type parameters are substituted for the actual data type names. In the example the function header reads

```
T square(T number)
```

`T` is the type parameter, or generic data type. The header defines `square` as a function that returns a value of type `T` and uses a parameter, `number`, which is also of type `T`. As mentioned before, the compiler examines each call to `square` and fills in the appropriate data type for `T`. For example, the following call uses an `int` argument:

```
int y, x = 4;
y = square(x);
```

This code will cause the compiler to generate the function

```
int square(int number)
{
    return number * number;
}
```

while the following statements

```
double y, f = 6.2
y = square(f);
```

will generate the function

```
double square(double number)
{
    return number * number;
}
```

Program 16-7 demonstrates how this function template is used.

Program 16-7

```
// This program uses a function template.
#include <iostream>
#include <iomanip>
using namespace std;

// Template definition for square function.
template <class T>
T square(T number)
{
    return number * number;
}

int main()
{
    int userInt;       // To hold integer input
    double userDouble; // To hold double input

    cout << setprecision(5);
    cout << "Enter an integer and a floating-point value: ";
    cin >> userInt >> userDouble;
    cout << "Here are their squares: ";
    cout << square(userInt) << " and "
         << square(userDouble) << endl;
   return 0;
}
```

(program output continues)

Program 16-7 *(continued)*

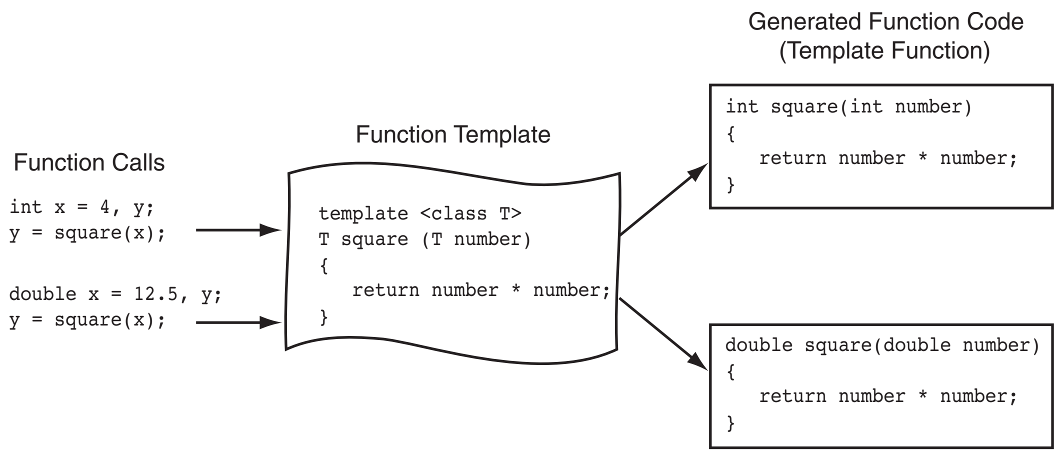
Program Output with Example Input Shown in Bold

```
Enter an integer and a floating-point value: 12 4.2 [Enter]
Here are their squares: 144 and 17.64
```

NOTE: All type parameters defined in a function template must appear at least once in the function parameter list.

Because the compiler encountered two calls to `square` in Program 16-7, each with a different parameter type, it generated the code for two instances of the function: one with an `int` parameter and `int` return type, the other with a `double` parameter and `double` return type. This is illustrated in Figure 16-3.

Figure 16-3



Notice in Program 16-7 that the template appears before all calls to `square`. As with regular functions, the compiler must already know the template's contents when it encounters a call to the template function. Templates, therefore, should be placed near the top of the program or in a header file.

NOTE: A function template is merely the specification of a function and by itself does not cause memory to be used. An actual instance of the function is created in memory when the compiler encounters a call to the template function.

Program 16-8 shows another example of a function template. The function, `swapVars`, uses two references to type `T` as parameters. The function swaps the contents of the variables referenced by the parameters.

Program 16-8

```
// This program demonstrates the swapVars function template.
#include <iostream>
using namespace std;

```

```
template <class T>
void swapVars(T &var1, T &var2)
{
     T temp;

     temp = var1;
     var1 = var2;
     var2 = temp;
}

int main()
{
     char firstChar, secondChar;        // Two chars
     int firstInt, secondInt;           // Two ints
     double firstDouble, secondDouble;  // Two doubles

     // Get and swapVars two chars
     cout << "Enter two characters: ";
     cin >> firstChar >> secondChar;
     swapVars(firstChar, secondChar);
     cout << firstChar << " " << secondChar << endl;

     // Get and swapVars two ints
     cout << "Enter two integers: ";
     cin >> firstInt >> secondInt;
     swapVars(firstInt, secondInt);
     cout << firstInt << " " << secondInt << endl;

     // Get and swapVars two doubles
     cout << "Enter two floating-point numbers: ";
     cin >> firstDouble >> secondDouble;
     swapVars(firstDouble, secondDouble);
     cout << firstDouble << " " << secondDouble << endl;
     return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter two characters: A B [Enter]
B A
Enter two integers: 5 10 [Enter]
10 5
Enter two floating-point numbers: 1.2 9.6 [Enter]
9.6 1.2
```

Using Operators in Function Templates

The `square` template shown earlier uses the `*` operator with the `number` parameter. This works well as long as `number` is of a primitive data type such as `int`, `float`, etc. If a user-defined class object is passed to the `square` function, however, the class must contain code for an overloaded `*` operator. If not, the compiler will generate a function with an error.

Always remember that a class object passed to a function template must support all the operations the function will perform on the object. For instance, if the function performs a comparison on the object (with >, <, ==, or another relational operator), those operators must be overloaded by the class object.

Function Templates with Multiple Types

More than one generic type may be used in a function template. Each type must have its own parameter, as shown in Program 16-9. This program uses a function template named `larger`. This template uses two type parameters: `T1` and `T2`. The sizes of the function parameters, `var1` and `var2`, are compared, and the function returns the number of bytes occupied by the larger of the two. Because the function parameters are specified with different types, the function generated from this template can accept two arguments of different types.

Program 16-9

```
// This program demonstrates a function template
// with two type parameters.
#include <iostream>
using namespace std;

template <class T1, class T2>
int largest(const T1 &var1, T2 &var2)
{
    if (sizeof(var1) > sizeof(var2))
        return sizeof(var1);
    else
        return sizeof(var2);
}

int main()
{
    int i = 0;
    char c = ' ';
    float f = 0.0;
    double d = 0.0;

    cout << "Comparing an int and a double, the largest\n"
         << "of the two is " << largest(i, d) << " bytes.\n";

    cout << "Comparing a char and a float, the largest\n"
         << "of the two is " << largest(c, f) << " bytes.\n";

    return 0;
}
```

Program Output

```
Comparing an int and a double, the largest
of the two is 8 bytes.
Comparing a char and a float, the largest
of the two is 4 bytes.
```

NOTE: Each type parameter declared in the template prefix must be used somewhere in the template definition.

Overloading with Function Templates

Function templates may be overloaded. As with regular functions, function templates are overloaded by having different parameter lists. For example, there are two overloaded versions of the `sum` function in Program 16-10. The first version accepts two arguments, and the second version accepts three.

Program 16-10

```
// This program demonstrates an overloaded function template.
#include <iostream>
using namespace std;

template <class T>
T sum(T val1, T val2)
{
    return val1 + val2;
}

template <class T>
T sum(T val1, T val2, T val3)
{
    return val1 + val2 + val3;
}

int main()
{
    double num1, num2, num3;

    // Get two values and display their sum.
    cout << "Enter two values: ";
    cin >> num1 >> num2;
    cout << "Their sum is " << sum(num1, num2) << endl;

    // Get three values and display their sum.
    cout << "Enter three values: ";
    cin >> num1 >> num2 >> num3;
    cout << "Their sum is " << sum(num1, num2, num3) << endl;
    return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter two values: 12.5 6.9 [Enter]
Their sum is 19.4
Enter three values: 45.76 98.32 10.51 [Enter]
Their sum is 154.59
```

There are other ways to perform overloading with function templates as well. For example, a program might contain a regular (nontemplate) version of a function as well as a template version. As long as each has a different parameter list, they can coexist as overloaded functions.

16.3 Focus on Software Engineering: Where to Start When Defining Templates

Quite often, it is easier to convert an existing function into a template than to write a template from scratch. With this in mind, you should start designing a function template by writing it first as a regular function. For example, the `swapVars` template in Program 16-8 would have been started as something like the following:

```
void swapVars(int &var1, int &var2)
{
     int temp;

     temp = var1;
     var1 = var2;
     var2 = temp;
}
```

Once this function is properly tested and debugged, converting it to a template is a simple process. First, the `template <class T>` header is added, then all the references to `int` that must be changed are replaced with the data type parameter `T`.

Checkpoint

16.6 When does the compiler actually generate code for a function template?

16.7 The following function accepts an `int` argument and returns half of its value as a `double`:

```
double half(int number)
{
     return number / 2.0;
}
```

Write a template that will implement this function to accept an argument of any type.

16.8 What must you be sure of when passing a class object to a function template that uses an operator, such as `*` or `>`?

16.9 What is the best method for writing a function template?

16.4 Class Templates

CONCEPT: Templates may also be used to create generic classes and abstract data types. Class templates allow you to create one general version of a class without having to duplicate code to handle multiple data types.

Recall the `IntArray` class from Chapter 14. By overloading the `[]` operator, this class allows you to implement `int` arrays that perform bounds checking. But suppose you would like to have a version of this class for other data types? Of course, you could design specialized classes such as `LongArray`, `FloatArray`, `DoubleArray`, and so forth. A better solution, however, is to design a single class template that works with any primitive data type. In this section, we will convert the `IntArray` class into a generalized template named `SimpleVector`.

Declaring a class template is very similar to declaring a function template. First, a template prefix, such as `template<class T>`, is placed before the class declaration. As with function templates, `T` (or whatever identifier you choose to use) is a data type parameter. Then, throughout the class declaration, the data type parameter is used where you wish to support any data type. Below is the `SimpleVector` class template declaration.

Contents of `SimpleVector.h`

```
// SimpleVector class template
#ifndef SIMPLEVECTOR_H
#define SIMPLEVECTOR_H
#include <iostream>
#include <new>      // Needed for bad_alloc exception
#include <cstdlib>  // Needed for the exit function
using namespace std;

template <class T>
class SimpleVector
{
private:
    T *aptr;         // To point to the allocated array
    int arraySize;   // Number of elements in the array
    void memError(); // Handles memory allocation errors
    void subError(); // Handles subscripts out of range

public:
    // Default constructor
    SimpleVector()
        { aptr = 0; arraySize = 0;}

    // Constructor declaration
    SimpleVector(int);

    // Copy constructor declaration
    SimpleVector(const SimpleVector &);

    // Destructor declaration
    ~SimpleVector();

    // Accessor to return the array size
    int size() const
        { return arraySize; }

    // Accessor to return a specific element
    T getElementAt(int position);

    // Overloaded [] operator declaration
    T &operator[](const int &);
};

//************************************************************
// Constructor for SimpleVector class. Sets the size of the *
// array and allocates memory for it.                       *
//************************************************************
```

```

template <class T>
SimpleVector<T>::SimpleVector(int s)
{
    arraySize = s;
    // Allocate memory for the array.
    try
    {
        aptr = new T [s];
    }
    catch (bad_alloc)
    {
        memError();
    }

    // Initialize the array.
    for (int count = 0; count < arraySize; count++)
        *(aptr + count) = 0;
}

//*******************************************
// Copy Constructor for SimpleVector class. *
//*******************************************

template <class T>
SimpleVector<T>::SimpleVector(const SimpleVector &obj)
{
    // Copy the array size.
    arraySize = obj.arraySize;

    // Allocate memory for the array.
    aptr = new T [arraySize];
    if (aptr == 0)
        memError();

    // Copy the elements of obj's array.
    for(int count = 0; count < arraySize; count++)
        *(aptr + count) = *(obj.aptr + count);
}

//*************************************
// Destructor for SimpleVector class.  *
//*************************************

template <class T>
SimpleVector<T>::~SimpleVector()
{
    if (arraySize > 0)
        delete [] aptr;
}

//*********************************************************
// memError function. Displays an error message and      *
// terminates the program when memory allocation fails. *
//*********************************************************
```

```

template <class T>
void SimpleVector<T>::memError()
{
    cout << "ERROR:Cannot allocate memory.\n";
    exit(EXIT_FAILURE);
}

//**************************************************************
// subError function. Displays an error message and          *
// terminates the program when a subscript is out of range. *
//**************************************************************

template <class T>
void SimpleVector<T>::subError()
{
    cout << "ERROR: Subscript out of range.\n";
    exit(EXIT_FAILURE);
}

//*********************************************************
// getElementAt function. The argument is a subscript.  *
// This function returns the value stored at the        *
// subcript in the array.                               *
//*********************************************************

template <class T>
T SimpleVector<T>::getElementAt(int sub)
{
    if (sub < 0 || sub >= arraySize)
        subError();
    return aptr[sub];
}

//**********************************************************
// Overloaded [] operator. The argument is a subscript.  *
// This function returns a reference to the element     *
// in the array indexed by the subscript.                *
//**********************************************************

template <class T>
T &SimpleVector<T>::operator[](const int &sub)
{
    if (sub < 0 || sub >= arraySize)
        subError();
    return aptr[sub];
}
#endif
```

NOTE: The `arraySize` member variable is declared as an `int`. This is because it holds the size of the array, which will be an integer value, regardless of the data type of the array. This is also why the size member function returns an `int`.

Defining Objects of the Class Template

Class template objects are defined like objects of ordinary classes, with one small difference: the data type you wish to pass to the type parameter must be specified. Placing the data type name inside angled brackets immediately following the class name does this. For example, the following statements create two `SimpleVector` objects: `intTable` and `doubleTable`.

```
SimpleVector<int> intTable(10);
SimpleVector<double> doubleTable(10);
```

In the definition of `intTable`, the data type `int` will be used in the template everywhere the type parameter `T` appears. This will cause `intTable` to store an array of `int`s. Likewise, the definition of `doubleTable` passes the data type `double` into the parameter `T`, causing it to store an array of `double`s. This is demonstrated in Program 16-11.

Program 16-11

```
// This program demonstrates the SimpleVector template.
#include <iostream>
#include "SimpleVector.h"
using namespace std;

int main()
{
     const int SIZE = 10; // Number of elements
     int count;           // Loop counter

     // Create a SimpleVector of ints.
     SimpleVector<int> intTable(SIZE);

     // Create a SimpleVector of doubles.
     SimpleVector<double> doubleTable(SIZE);

     // Store values in the two SimpleVectors.
     for (count = 0; count < SIZE; count++)
     {
          intTable[count] = (count * 2);
          doubleTable[count] = (count * 2.14);
     }

     // Display the values in the SimpleVectors.
     cout << "These values are in intTable:\n";
     for (count = 0; count < SIZE; count++)
          cout << intTable[count] << " ";
     cout << endl;
     cout << "These values are in doubleTable:\n";
     for (count = 0; count < SIZE; count++)
          cout << doubleTable[count] << " ";
     cout << endl;

     // Use the standard + operator on the elements.
     cout << "\nAdding 5 to each element of intTable"
          << " and doubleTable.\n";
     for (count = 0; count < SIZE; count++)
```

```
      {
          intTable[count] = intTable[count] + 5;
          doubleTable[count] = doubleTable[count] + 5.0;
      }

      // Display the values in the SimpleVectors.
      cout << "These values are in intTable:\n";
      for (count = 0; count < SIZE; count++)
          cout << intTable[count] << " ";
      cout << endl;
      cout << "These values are in doubleTable:\n";
      for (count = 0; count < SIZE; count++)
          cout << doubleTable[count] << " ";
      cout << endl;

      // Use the standard ++ operator on the elements.
      cout << "\nIncrementing each element of intTable and"
           << " doubleTable.\n";
      for (count = 0; count < SIZE; count++)
      {
          intTable[count]++;
          doubleTable[count]++;
      }

      // Display the values in the SimpleVectors.
      cout << "These values are in intTable:\n";
      for (count = 0; count < SIZE; count++)
          cout << intTable[count] << " ";
      cout << endl;
      cout << "These values are in doubleTable:\n";
      for (count = 0; count < SIZE; count++)
          cout << doubleTable[count] << " ";
      cout << endl;

      return 0;
  }
```

Program Output

```
These values are in intTable:
0 2 4 6 8 10 12 14 16 18
These values are in doubleTable:
0 2.14 4.28 6.42 8.56 10.7 12.84 14.98 17.12 19.26

Adding 5 to each element of intTable and doubleTable.
These values are in intTable:
5 7 9 11 13 15 17 19 21 23
These values are in doubleTable:
5 7.14 9.28 11.42 13.56 15.7 17.84 19.98 22.12 24.26

Incrementing each element of intTable and doubleTable.
These values are in intTable:
6 8 10 12 14 16 18 20 22 24
These values are in doubleTable:
6 8.14 10.28 12.42 14.56 16.7 18.84 20.98 23.12 25.26
```

Class Templates and Inheritance

Inheritance can easily be applied to class templates. For example, in the following template, `SearchableVector` is derived from the `SimpleVector` class.

Contents of `SearchableVector.h`

```
#ifndef SEARCHABLEVECTOR_H
#define SEARCHABLEVECTOR_H
#include "SimpleVector.h"

template <class T>
class SearchableVector : public SimpleVector<T>
{
public:
    // Default constructor
    SearchableVector() : SimpleVector<T>()
        { }

    // Constructor
    SearchableVector(int size) : SimpleVector<T>(size)
        { }

    // Copy constructor
    SearchableVector(const SearchableVector &);

    // Accessor to find an item
    int findItem(const T);
};

//*******************************************************
// Copy constructor                                     *
//*******************************************************

template <class T>
SearchableVector<T>::SearchableVector(const SearchableVector &obj) :
                    SimpleVector<T>(obj.size())
{
    for(int count = 0; count < this->size(); count++)
        this->operator[](count) = obj[count];
}

//********************************************************
// findItem function                                     *
// This function searches for item. If item is found     *
// the subscript is returned. Otherwise -1 is returned.  *
//********************************************************

template <class T>
int SearchableVector<T>::findItem(const T item)
{
    for (int count = 0; count <= this->size(); count++)
46      {
47          if (getElementAt(count) == item)
48              return count;
49      }
50      return -1;
51  }
52  #endif
```

This class template defines a searchable version of the `SimpleVector` class. The member function `findItem` accepts an argument and performs a simple linear search to determine whether the argument's value is stored in the array. If the value is found in the array, its subscript is returned. Otherwise, −1 is returned.

Notice that each time the name `SimpleVector` is used in the class template, the type parameter `T` is used with it. For example, here is the first line of the class declaration, in line 6, which names `SimpleVector` as the base class:

```
class SearchableVector : public SimpleVector<T>
```

Also, here are the function headers for the class constructors:

```
SearchableVector() : SimpleVector<T>()
SearchableVector(int size) : SimpleVector<T>(size)
```

Because `SimpleVector` is a class template, the type parameter must be passed to it.

Program 16-12 demonstrates the class by storing values in two `SearchableVector` objects and then searching for a specific value in each.

Program 16-12

```
// This program demonstrates the SearchableVector template.
#include <iostream>
#include "SearchableVector.h"
using namespace std;

int main()
{
    const int SIZE = 10;   // Number of elements
    int count;             // Loop counter
    int result;            // To hold search results

    // Create two SearchableVector objects.
    SearchableVector<int> intTable(SIZE);
    SearchableVector<double> doubleTable(SIZE);

    // Store values in the objects.
    for (count = 0; count < SIZE; count++)
    {
        intTable[count] = (count * 2);
        doubleTable[count] = (count * 2.14);
    }

```

(program continues)

Program 16-12 *(continued)*

```
    // Display the values in the objects.
    cout << "These values are in intTable:\n";
    for (count = 0; count < SIZE; count++)
        cout << intTable[count] << " ";
    cout << endl << endl;
    cout << "These values are in doubleTable:\n";
    for (count = 0; count < SIZE; count++)
        cout << doubleTable[count] << " ";
    cout << endl;

    // Search for the value 6 in intTable.
    cout << "\nSearching for 6 in intTable.\n";
    result = intTable.findItem(6);
    if (result == -1)
        cout << "6 was not found in intTable.\n";
    else
        cout << "6 was found at subscript " << result << endl;

    // Search for the value 12.84 in doubleTable.
    cout << "\nSearching for 12.84 in doubleTable.\n";
    result = doubleTable.findItem(12.84);
    if (result == -1)
        cout << "12.84 was not found in doubleTable.\n";
    else
        cout << "12.84 was found at subscript " << result << endl;
    return 0;
}
```

Program Output

```
These values are in intTable:
0 2 4 6 8 10 12 14 16 18

These values are in doubleTable:
0 2.14 4.28 6.42 8.56 10.7 12.84 14.98 17.12 19.26

Searching for 6 in intTable.
6 was found at subscript 3

Searching for 12.84 in doubleTable.
12.84 was found at subscript 6
```

The `SearchableVector` class demonstrates that a class template may be derived from another class template. In addition, class templates may be derived from ordinary classes, and ordinary classes may be derived from class templates.

Specialized Templates

Suppose you have a template that works for all data types but one. For example, the `SimpleVector` and `SearchableVector` classes work well with numeric, and even character, data. But they will not work with C-strings. Situations like this require the use of *specialized templates*. A specialized template is one that is designed to work with a specific data type.

In the declaration, the actual data type is used instead of a type parameter. For example, the declaration of a specialized version of the `SimpleVector` class might start like this:

```
class SimpleVector<char *>
```

The compiler would know that this version of the `SimpleVector` class is intended for the `char *` data type. Anytime an object is defined of the type `SimpleVector<char *>`, the compiler will use this template to generate the code.

Checkpoint

16.10 Suppose your program uses a class template named `List`, which is defined as

```
template<class T>
class List
{
    // Members are declared here...
};
```

Give an example of how you would use `int` as the data type in the definition of a `List` object. (Assume the class has a default constructor.)

16.11 As the following `Rectangle` class is written, the `width` and `length` members are `double`s. Rewrite the class as a template that will accept any data type for these members.

```
class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setData(double w, double l)
            { width = w; length = l;}
        double getWidth()
            { return width; }
        double getLength()
            { return length; }
        double getArea()
            { return width * length; }
};
```

16.5 Introduction to the Standard Template Library (STL)

CONCEPT: The Standard Template Library contains many templates for useful algorithms and data structures.

NOTE: Section 7.12 of Chapter 7 presents a concise introduction to the Standard Template Library and discusses the `vector` data type. This discussion is continued in Section 8.5 of Chapter 8. If you have not already studied those sections, do so now.

In addition to its runtime library, which you have used throughout this book, C++ also provides a library of templates. The *Standard Template Library* (or STL) contains numerous generic templates for implementing abstract data types (ADTs) and algorithms. In this section you will be introduced to the general types of ADTs and algorithms that may be found in the STL.

Abstract Data Types

The most important data structures in the STL are *containers* and *iterators*. A container is a class that stores data and organizes it in some fashion. An iterator is an object that behaves like a pointer. It is used to access the individual data elements in a container.

There are two types of container classes in the STL: *sequence* and *associative*. A sequence container organizes data in a sequential fashion similar to an array. The three sequence containers currently provided are listed in Table 16-1.

Table 16-1

Container Name	Description
`vector`	An expandable array. Values may be added to or removed from the end or middle of a `vector`.
`deque`	Like a `vector`, but allows values to be added to or removed from the front.
`list`	A doubly linked list of data elements. Values may be inserted to or removed from any position. (You will learn more about linked lists in Chapter 17.)

Performance Differences Between `vector`s, `deque`s, and `list`s

There is a difference in performance between `vector`s, `deque`s, and `list`s. When choosing one of these templates to use in your program, remember the following points:

- A `vector` is capable of quickly adding values to its end. Insertions at other points are not as efficient.
- A `deque` is capable of quickly adding values to its front and its end. `deque`s are not efficient at inserting values at other positions, however.
- A `list` is capable of quickly inserting values anywhere in its sequence. `list`s do not, however, provide random access.

An associative container uses keys to rapidly access elements. (If you've ever used a relational database, you are probably familiar with the concept of keys.) The four associative containers currently supported are shown in Table 16-2.

Table 16-2

Container Name	Description
`set`	Stores a set of keys. No duplicate values are allowed.
`multiset`	Stores a set of keys. Duplicates are allowed.
`map`	Maps a set of keys to data elements. Only one key per data element is allowed. Duplicates are not allowed.
`multimap`	Maps a set of keys to data elements. Many keys per data element are allowed. Duplicates are allowed.

Iterators are generalizations of pointers and are used to access data stored in containers. The types of iterators are shown in Table 16-3.

Table 16-3

Iterator Type	Description
Forward	Can only move forward in a container (uses the ++ operator).
Bidirectional	Can move forward or backward in a container (uses the ++ and – operators).
Random-access	Can move forward and backward, and can jump to a specific data element in a container.
Input	Can be used with an input stream to read data from an input device or a file.
Output	Can be used with an output stream to write data to an output device or a file.

Iterators are associated with containers. The type of container you have determines the type of iterator you use. For example, `vectors` and `deques` require random-access iterators, while `lists`, `sets`, `multisets`, `maps`, and `multimaps` require bidirectional iterators.

Algorithms

The algorithms provided by the STL are implemented as function templates and perform various operations on elements of containers. There are many algorithms in the STL, but Table 16-4 lists a few of them. (The table gives only general descriptions.)

Table 16-4

Algorithm	Description
`binary_search`	Performs a binary search for an object and returns true if the object is found. *Example:* `binary_search(iter1, iter2, value);` In this statement, `iter1` and `iter2` point to elements in a container. (`iter1` points to the first element in the range, and `iter2` points to the last element in the range.) The statement performs a binary search on the range of elements, searching for `value`. The `binary_search` function returns `true` if the element was found and `false` if the element was not found.
`count`	Returns the number of times a value appears in a range. *Example:* `iter3 = count(iter1, iter2, value);` In this statement, `iter1` and `iter2` point to elements in a container. (`iter1` points to the first element in the range, and `iter2` points to the last element in the range.) The statement returns the number of times `value` appears in the range of elements.
`find`	Finds the first object in a container that matches a value and returns an iterator to it. *Example:* `iter3 = find(iter1, iter2, value);` In this statement, `iter1` and `iter2` point to elements in a container. (`iter1` points to the first element in the range, and `iter2` points to the last element in the range.) The statement searches the range, of elements for `value`. If `value` is found, the function returns an iterator to the element containing it.

(table continues)

Table 16-4 *(continued)*

Algorithm	Description
`for_each`	Executes a function for each element in a container. *Example:* `for_each(iter1, iter2, func);` In this statement, `iter1` and `iter2` point to elements in a container. (`iter1` points to the first element in the range, and `iter2` points to the last element in the range.) The third argument, `func`, is the name of a function. The statement calls the function `func` for each element in the range, passing the element as an argument.
`max_element`	Returns an iterator to the largest object in a range. *Example:* `iter3 = max_element(iter1, iter2);` In this statement, `iter1` and `iter2` point to elements in a container. (`iter1` points to the first element in the range, and `iter2` points to the last element in the range.) The statement returns an iterator to the element containing the largest value in the range.
`min_element`	Returns an iterator to the smallest object in a range. *Example:* `iter3 = min_element(iter1, iter2);` In this statement, `iter1` and `iter2` point to elements in a container. (`iter1` points to the first element in the range, and `iter2` points to the last element in the range.) The statement returns an iterator to the element containing the smallest value in the range.
`random_shuffle`	Randomly shuffles the elements of a container. *Example:* `random_shuffle(iter1, iter2);` In this statement, `iter1` and `iter2` point to elements in a container. (`iter1` points to the first element in the range, and `iter2` points to the last element in the range.) The statement randomly reorders the elements in the range.
`sort`	Sorts a range of elements. *Example:* `sort(iter1, iter2);` In this statement, `iter1` and `iter2` point to elements in a container. (`iter1` points to the first element in the range, and `iter2` points to the last element in the range.) The statement sorts the elements in the range in ascending order.

Example Programs Using the STL

Now that you have been introduced to the types of data structures and algorithms offered by the STL, let's look at some simple programs that use them.

Containers

Program 16-13 provides a limited demonstration of the `vector` class template. The member functions of `vector` used in this program are listed in Table 16-5.

Table 16-5

Member Function	Description
`size()`	Returns the number of elements in the `vector`.
`push_back()`	Accepts as an argument a value to be inserted into the `vector`. The argument is inserted after the last element. (Pushed onto the back of the `vector`.)
`pop_back()`	Removes the last element from the `vector`.
`operator[]`	Allows array-like access of existing `vector` elements. (The `vector` must already contain elements for this operator to work. It cannot be used to insert new values into the `vector`.)

The `vector` class template has many more member functions, but these are enough to demonstrate the class.

Program 16-13

```
// This program provides a simple demonstration of the
// vector STL template.
#include <iostream>
#include <vector> // Include the vector header
using namespace std;

int main()
{
     int count; // Loop counter

     // Define a vector object.
     vector<int> vect;

     // Use the size member function to get
     // the number of elements in the vector.
     cout << "vect starts with " << vect.size()
          << " elements.\n";

     // Use push_back to push values into the vector.
     for (count = 0; count < 10; count++)
          vect.push_back(count);

     // Display the size of the vector now.
     cout << "Now vect has " << vect.size()
          << " elements. Here they are:\n";

     // Use the [] operator to display each element.
     for (count = 0; count < vect.size(); count++)
          cout << vect[count] << " ";
     cout << endl;

     // Use the pop_back member function.
     cout << "Popping the values out of vect...\n";
     for (count = 0; count < 10; count++)
          vect.pop_back();

```

(program continues)

Program 16-13 *(continued)*

```
    // Display the size of the vector now.
    cout << "Now vect has " << vect.size() << " elements.\n";
    return 0;
}
```

Program Output

```
vect starts with 0 elements.
Now vect has 10 elements. Here they are:
0 1 2 3 4 5 6 7 8 9
Popping the values out of vect...
Now vect has 0 elements.
```

VideoNote
Storing Objects in a `vector`

Notice the inclusion of the `vector` header file in line 4, which is required for the `vector` container. `vector`s are one of the simplest types of containers in the STL. In following chapters, you will see examples using other types of containers.

Iterators

In Program 16-13, the `vector`'s elements were accessed by the container's member functions. Iterators may also be used to access and manipulate container elements. Program 16-14 demonstrates the use of an iterator with a `vector` object.

Program 16-14

```
// This program provides a simple iterator demonstration.
#include <iostream>
#include <vector>  // Include the vector header
using namespace std;

int main()
{
    int count;  // Loop counter

    // Define a vector object.
    vector<int> vect;

    // Define an iterator object.
    vector<int>::iterator iter;

    // Use push_back to push values into the vector.
    for (count = 0; count < 10; count++)
    vect.push_back(count);

    // Step the iterator through the vector,
    // and use it to display the vector's contents.
    cout << "Here are the values in vect: ";
    for (iter = vect.begin(); iter < vect.end(); iter++)
    {
        cout << *iter << " ";
    }
27
28      // Step the iterator through the vector backwards.
29      cout << "\nand here they are backwards: ";
30      for (iter = vect.end() - 1; iter >= vect.begin(); iter--)
31      {
32          cout << *iter << " ";
33      }
34      return 0;
35  }
```

Program Output

```
Here are the values in vect: 0 1 2 3 4 5 6 7 8 9
and here they are backwards:  9 8 7 6 5 4 3 2 1 0
```

The definition of an iterator is closely related to the definition of the container it is to be used with. For example, Program 16-14 defines a `vector` of `int`s as:

```
vector<int> vect;
```

The iterator that will work with the `vector` is defined as:

```
vector<int>::iterator iter;
```

This definition creates an iterator specifically for a `vector` of `int`s. The compiler automatically chooses the right type (in this case, a random-access iterator).

The second `for` loop in lines 23 through 26 causes the iterator to step through each element in the `vector`:

```
for (iter = vect.begin(); iter < vect.end(); iter++)
```

The loop's initialization expression uses the container's `begin()` member function, which returns an iterator pointing to the beginning of the `vector`. The statement

```
iter = vect.begin();
```

causes `iter` to point to the first element in the `vector`. The test expression uses the `end()` member function, which returns an iterator pointing to the location just past the end of the container:

```
iter < vect.end();
```

As long as `iter` points to an element prior to the end of the `vector`, this statement will be true.

The loop's update expression uses the `++` operator to increment the iterator. This causes the iterator to point to the next element in the vector.

The body of the loop uses a `cout` statement in line 25 to display the element that the iterator points to:

```
cout << *iter << " ";
```

Like a pointer, iterators may be dereferenced with the `*` operator. The statement above causes the value pointed to by `iter` to be displayed.

Back to the `vector` Template

Table 16-6 lists several more member functions of the `vector` class template. Some of these accept iterators as arguments and/or return an iterator.

Table 16-6

Member Function	Examples and Description
`at(`*`element`*`)`	Returns the value of the element located at *element* in the vector. *Example:* `x = vect.at(5);` This statement assigns the value of element 5 of vect to x.
`back()`	Returns a reference to the last element in the vector. *Example:* `cout << vect.back() << endl;`
`begin()`	Returns an iterator pointing to the vector's first element. *Example:* `iter = vect.begin();`
`capacity()`	Returns the maximum number of elements that may be stored in the vector without additional memory being allocated. (This is not the same value as returned by the `size` member function.) *Example:* `x = vect.capacity();` This statement assigns the capacity of `vect` to `x`.
`clear()`	Clears a vector of all its elements. *Example:* `vect.clear();` This statement removes all the elements from `vect`.

`empty()`	Returns true if the vector is empty. Otherwise, it returns false.
	Example:
	`if (vect.empty())`
	    `cout << "The vector is empty.";`
	This statement displays the message if `vect` is empty.

`end()`	Returns an iterator pointing to the vector's last element. *Example:* `iter = vect.end();`
`erase()`	Causes the vector element pointed to by the iterator `iter` to be removed. *Example:* `vect.erase(iter);`
`erase(`*`iter1, iter2`*`)`	Causes all the vector elements from the iterator `iter1` to the iterator `iter2` to be removed. *Example:* `vect.erase(firstIter, secondIter);`
`front()`	Returns a reference to the vector's first element. *Example:* `cout << vector.front() << endl;`

Table 16-6 *(continued)*

Member Function	Examples and Description
`insert (iter, value)`	Inserts a value into a `vector`. *Example:* `vect.insert(iter, 22);` This statement inserts the value 22 into `vect`. The value is inserted into the element before the one pointed to by `iter`.
`resize(n, value)`	Resizes a vector by *n* new elements. The elements are initialized with *value*. *Example:* `vect.resize(10, 0);` This statement adds ten new elements to `vect` and initializes the new elements with the value 0.
`reverse()`	Reverses the order of the items stored in a vector. *Example:* `vect.reverse();`

Algorithms

There are many algorithms in the STL, implemented as function templates. Program 16-15 demonstrates `random_shuffle`, `sort`, and `binary_search`.

Program 16-15

```
// A simple demonstration of STL algorithms
#include <iostream>
#include <vector>    // Required for the vector type
#include <algorithm> // Required for STL algorithms
using namespace std;

int main()
{
     int count; // Loop counter

     // Define a vector object.
     vector<int> vect;

     // Use push_back to push values into the vector.
     for (count = 0; count < 10; count++)
          vect.push_back(count);

     // Display the vector's elements.
     cout << "The vector has " << vect.size()
           << " elements. Here they are:\n";
     for (count = 0; count < vect.size(); count++)
          cout << vect[count] << " ";
     cout << endl;

     // Randomly shuffle the vector's contents.
     random_shuffle(vect.begin(), vect.end());

```

(program continues)

Program 16-15 *(continued)*

```
28      // Display the vector's elements.
29      cout << "The elements have been shuffled:\n";
30      for (count = 0; count < vect.size(); count++)
31          cout << vect[count] << " ";
32      cout << endl;
33
34      // Now sort the vector's elements.
35      sort(vect.begin(), vect.end());
36
37      // Display the vector's elements again.
38      cout << "The elements have been sorted:\n";
39      for (count = 0; count < vect.size(); count++)
40          cout << vect[count] << " ";
41      cout << endl;
42
43      // Now search for an element with the value 7.
44      if (binary_search(vect.begin(), vect.end(), 7))
45          cout << "The value 7 was found in the vector.\n";
46      else
47          cout << "The value 7 was not found in the vector.\n";
48      return 0;
49  }
```

Program Output

```
The vector has 10 elements. Here they are:
0 1 2 3 4 5 6 7 8 9
The elements have been shuffled:
4 3 0 2 6 7 8 9 5 1
The elements have been sorted:
0 1 2 3 4 5 6 7 8 9
The value 7 was found in the vector.
```

NOTE: The STL algorithms require the `algorithm` header file.

The `random_shuffle` function rearranges the elements of a container. In line 26 of Program 16-15, it is called in the following manner:

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

The function takes two arguments, which together represent a range of elements within a container. The first argument is an iterator to the first element in the range. In this case, `vect.begin()` is used. The second argument is an iterator to the last element in the range. Here we have used `vect.end()`. These arguments tell `random_shuffle` to rearrange all the elements from the beginning to the end of the `vect` container.

The `sort` algorithm also takes iterators to a range of elements. Here is the function call that appears in line 35:

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

All the elements within the range are sorted in ascending order.

The `binary_search` algorithm searches a range of elements for a value. If the value is found, the function returns `true`. Otherwise, it returns `false`. For example, the following function call, which appears in line 44, searches all the elements in `vect` for the value 7.

```
binary_search(vect.begin(), vect.end(), 7)
```

Program 16-16 demonstrates the `count` algorithm.

Program 16-16

```
// This program demonstrates the STL count algorithm.
#include <iostream>
#include <vector>    // Needed to define the vector
#include <algorithm> // Needed for the count algorithm
using namespace std;

int main()
{
    // Define a vector object.
    vector<int> values;

    // Define an iterator for the vector.
    vector<int>::iterator iter;

    // Store some values in the vector.
    values.push_back(1);
    values.push_back(2);
    values.push_back(2);
    values.push_back(3);
    values.push_back(3);
    values.push_back(3);

    // Display the values in the vector.
    cout << "The values in the vector are:\n";
    for (iter = values.begin(); iter < values.end(); iter++)
        cout << *iter << " ";
    cout << endl << endl;

    // Display the count of each number.
    cout << "The number of 1s in the vector is ";
    cout << count(values.begin(), values.end(), 1) << endl;
    cout << "The number of 2s in the vector is ";
    cout << count(values.begin(), values.end(), 2) << endl;
    cout << "The number of 3s in the vector is ";
    cout << count(values.begin(), values.end(), 3) << endl;
    return 0;
}
```

Program Output

```
The values in the vector are:
1 2 2 3 3 3

The number of 1s in the vector is 1
The number of 2s in the vector is 2
The number of 3s in the vector is 3
```

Program 16-17 demonstrates the `max_element` and `min_element` algorithms.

Program 16-17

```
// A demonstration of the max_element and min_element algorithms
#include <iostream>
#include <vector>    // Needed to define the vector
#include <algorithm> // Needed for the algorithms
using namespace std;

int main()
{
    // Define a vector object.
    vector<int> numbers;

    // Define an iterator for the vector.
    vector<int>::iterator iter;

    // Store some numbers in the vector.
    for (int count = 0; count < 10; count++)
        numbers.push_back(count);

    // Display the numbers in the vector.
    cout << "The numbers in the vector are:\n";
    for (iter = numbers.begin(); iter != numbers.end(); iter++)
        cout << *iter << " ";
    cout << endl << endl;

    // Find the largest value in the vector.
    iter = max_element(numbers.begin(), numbers.end());
    cout << "The largest value in the vector is " << *iter << endl;

    // Find the smallest value in the vector.
    iter = min_element(numbers.begin(), numbers.end());
    cout << "The smallest value in the vector is " << *iter << endl;
    return 0;
}
```

Program Output

```
The numbers in the vector are:
0 1 2 3 4 5 6 7 8 9

The largest value in the vector is 9
The smallest value in the vector is 0
```

Program 16-18 demonstrates the `find` algorithm.

Program 16-18

```
// A demonstration of the STL find algorithm.
#include <iostream>
#include <vector>    // Needed to define the vector
#include <algorithm> // Needed for the find algorithm
using namespace std;
```

```

int main()
{
     // Define a vector object.
     vector<int> numbers;

     // Define an iterator for the vector.
     vector<int>::iterator iter;

     // Store some numbers in the vector.
     for (int x = 0; x < 10; x++)
          numbers.push_back(x);

     // Display the numbers in the vector.
     cout << "The numbers in the vector are:\n";
     for (iter = numbers.begin(); iter != numbers.end(); iter++)
          cout << *iter << " ";
     cout << endl << endl;

     // Find the number 7 in the vector.
     iter = find(numbers.begin(), numbers.end(), 7);
     cout << *iter << endl;
     return 0;
}
```

Program Output

```
The numbers in the vector are:
0 1 2 3 4 5 6 7 8 9

7
```

Program 16-19 demonstrates the `for_each` algorithm.

Program 16-19

```
// A demonstration of the for_each algorithm.
#include <iostream>
#include <vector>    // Needed to define the vector
#include <algorithm> // Needed for the for_each algorithm
using namespace std;

// Function prototype
void doubleValue(int &);

int main()
{
     // Define a vector object.
     vector<int> numbers;

     // Define an iterator for the vector.
     vector<int>::iterator iter;

```

(program continues)

Program 16-19 *(continued)*

```
18         // Store some numbers in the vector.
19         for (int x = 0; x < 10; x++)
20             numbers.push_back(x);
21 
22         // Display the numbers in the vector.
23         cout << "The numbers in the vector are:\n";
24         for (iter = numbers.begin(); iter != numbers.end(); iter++)
25              cout << *iter << " ";
26         cout << endl << endl;
27 
28         // Double the values in the vector.
29         for_each(numbers.begin(), numbers.end(), doubleValue);
30 
31         // Display the numbers in the vector again.
32         cout << "Now the numbers in the vector are:\n";
33         for (iter = numbers.begin(); iter != numbers.end(); iter++)
34             cout << *iter << " ";
35         cout << endl;
36         return 0;
37     }
38 
39     //*****************************************************
40     // Function doubleValue. This function accepts an int    *
41     // reference as its argument. The value of the argument *
42     // is doubled.                                           *
43     //*****************************************************
44 
45     void doubleValue(int &val)
46     {
47         val *= 2;
48     }
```

Program Output

```
The numbers in the vector are:
0 1 2 3 4 5 6 7 8 9

Now the numbers in the vector are:
0 2 4 6 8 10 12 14 16 18
```

In line 29, the following statement calls `for_each`:

```
for_each(numbers.begin(), numbers.end(), doubleValue);
```

The first and second arguments specify a range of elements. In this case, the range is the entire vector. The third argument is the name of a function. The `for_each` algorithm calls the function once for each element in the range, passing the element as an argument to the function.

The programs in this section give you a brief introduction to using the STL by demonstrating simple operations on a `vector`. In the remaining chapters you will be given specific examples of how to use other STL containers, iterators, and algorithms.

Review Questions and Exercises

Short Answer

1. What is a throw point?
2. What is an exception handler?
3. Explain the difference between a try block and a catch block.
4. What happens if an exception is thrown, but not caught?
5. What is “unwinding the stack”?
6. What happens if an exception is thrown by a class’s member function?
7. How do you prevent a program from halting when the new operator fails to allocate memory?
8. Why is it more convenient to write a function template than a series of overloaded functions?
9. Why must you be careful when writing a function template that uses operators such as `[]` with its parameters?
10. What is a container? What is an iterator?
11. What two types of containers does the STL provide?
12. What STL algorithm randomly shuffles the elements in a container?

Fill-in-the-Blank

13. The line containing a throw statement is known as the __________.
14. The __________ block contains code that directly or indirectly might cause an exception to be thrown.
15. The __________ block handles an exception.
16. When writing function or class templates, you use a(n) __________ to specify a generic data type.
17. The beginning of a template is marked by a(n) __________.
18. When defining objects of class templates, the __________ you wish to pass into the type parameter must be specified.
19. A(n) __________ template works with a specific data type.
20. A(n) __________ container organizes data in a sequential fashion similar to an array.
21. A(n) __________ container uses keys to rapidly access elements.
22. __________ are pointer-like objects used to access data stored in a container.
23. The __________ exception is thrown when the `new` operator fails to allocate the requested amount of memory.

Algorithm Workbench

24. Write a function that searches a numeric array for a specified value. The function should return the subscript of the element containing the value if it is found in the array. If the value is not found, the function should throw an exception.

25. Write a function that dynamically allocates a block of memory and returns a `char` pointer to the block. The function should take an integer argument that is the amount of memory to be allocated. If the `new` operator cannot allocate the memory, the function should return a null pointer.
26. Make the function you wrote in Question 24 a template.
27. Write a template for a function that displays the contents of an array of any type.
28. A program has the following definition statements:

```
vector<int> numbers;
vector<int>::iterator iter;
```

Write code that uses the iterator to display all the values stored in the vector.

29. Write a statement that performs the STL `binary_search` algorithm on the vector defined in Question 28.
30. A program has the following definition:

```
vector<double> numbers;
```

The same program also has the following function:

```
void display(double n)
{
    cout << n << endl;
}
```

Write code that uses the STL `for_each` algorithm to display the elements of the `numbers` vector using the `display` function.

True or False

31. T F There can be only one catch block in a program.
32. T F When an exception is thrown, but not caught, the program ignores the error.
33. T F Data may be passed with an exception by storing it in members of an exception class.
34. T F Once an exception has been thrown, it is not possible for the program to jump back to the throw point.
35. T F All type parameters defined in a function template must appear at least once in the function parameter list.
36. T F The compiler creates an instance of a function template in memory as soon as it encounters the template.
37. T F A class object passed to a function template must overload any operators used on the class object by the template.
38. T F Only one generic type may be used with a template.
39. T F In the function template definition, it is not necessary to use each type parameter declared in the template prefix.
40. T F It is possible to overload two function templates.
41. T F It is possible to overload a function template and an ordinary (nontemplate) function.
42. T F A class template may not be derived from another class template.

43. T F A class template may not be used as a base class.
44. T F Specialized templates work with a specific data type.
45. T F When defining an iterator from the STL, the compiler automatically creates the right kind, depending upon the container it is to be used with.
46. T F STL algorithms are implemented as function templates.

Find the Error

Each of the following declarations or code segments has errors. Locate as many as possible.

47.
```
catch
{
    quotient = divide(num1, num2);
    cout << "The quotient is " << quotient << endl;
}
try (string exceptionString)
{
    cout << exceptionString;
}
```

48.
```
try
{
    quotient = divide(num1, num2);
}
cout << "The quotient is " << quotient << endl;
catch (string exceptionString)
{
    cout << exceptionString;
}
```

49.
```
template <class T>
T square(T number)
{
    return T * T;
}
```

50.
```
template <class T>
int square(int number)
{
    return number * number;
}
```

51.
```
template <class T1, class T2>
T1 sum(T1 x, T1 y)
{
    return x + y;
}
```

52. Assume the following definition appears in a program that uses the `SimpleVector` class template presented in this chapter.

```
int <SimpleVector> array(25);
```

53. Assume the following statement appears in a program that has defined `valueSet` as an object of the `SimpleVector` class presented in this chapter. Assume that `valueSet` is a vector of `int`s, and has 20 elements.

```
cout << valueSet<int>[2] << endl;
```

Programming Challenges

1. **Date Exceptions**

   Modify the `Date` class you wrote for Programming Challenge 1 of Chapter 13. The class should implement the following exception classes:

`InvalidDay`	Throw when an invalid day (< 1 or > 31) is passed to the class.
`InvalidMonth`	Throw when an invalid month (< 1 or > 12) is passed to the class.

   Demonstrate the class in a driver program.

2. **Time Format Exceptions**

   Modify the `MilTime` class you created for Programming Challenge 4 of Chapter 15. The class should implement the following exceptions:

`BadHour`	Throw when an invalid hour (< 0 or > 2359) is passed to the class.
`BadSeconds`	Throw when an invalid number of seconds (< 0 or > 59) is passed to the class.

   Demonstrate the class in a driver program.

3. **Minimum/Maximum Templates**

   Write templates for the two functions `minimum` and `maximum`. The `minimum` function should accept two arguments and return the value of the argument that is the lesser of the two. The `maximum` function should accept two arguments and return the value of the argument that is the greater of the two. Design a simple driver program that demonstrates the templates with various data types.

4. **Absolute Value Template**

   Write a function template that accepts an argument and returns its absolute value. The absolute value of a number is its value with no sign. For example, the absolute value of −5 is 5, and the absolute value of 2 is 2. Test the template in a simple driver program.

5. **Total Template**

   Write a template for a function called `total`. The function should keep a running total of values entered by the user, then return the total. The argument sent into the function should be the number of values the function is to read. Test the template in a simple driver program that sends values of various types as arguments and displays the results.

6. **`IntArray` Class Exception**

   Chapter 14 presented an `IntArray` class that dynamically creates an array of integers and performs bounds checking on the array. If an invalid subscript is used with the class, it displays an error message and aborts the program. Modify the class so it throws an exception instead.

7. **`TestScores` Class**

   Write a class named `TestScores`. The class constructor should accept an array of test scores as its argument. The class should have a member function that returns the average of the test scores. If any test score in the array is negative or greater than 100, the class should throw an exception. Demonstrate the class in a program.

8. **`SimpleVector` Modification**

   Modify the `SimpleVector` class template presented in this chapter to include the member functions `push_back` and `pop_back`. These functions should emulate the STL vector class member functions of the same name. (See Table 16-5.) The `push_back` function should accept an argument and insert its value at the end of the array. The `pop_back` function should accept no argument and remove the last element from the array. Test the class with a driver program.

9. **`SearchableVector` Modification**

   Modify the `SearchableVector` class template presented in this chapter so that it performs a binary search instead of a linear search. Test the template in a driver program.

10. **`SortableVector` Class Template**

    Write a class template named `SortableVector`. The class should be derived from the `SimpleVector` class presented in this chapter. It should have a member function that sorts the array elements in ascending order. (Use the sorting algorithm of your choice.) Test the template in a driver program.

11. **Inheritance Modification**

    Assuming you have completed Programming Challenges 9 and 10, modify the inheritance hierarchy of the `SearchableVector` class template so it is derived from the `SortableVector` class instead of the `SimpleVector` class. Implement a member function named `sortAndSearch`, both a sort and a binary search.

12. **Specialized Templates**

    In this chapter, the section *Specialized Templates* within Section 16.4 describes how to design templates that are specialized for one particular data type. The section introduces a method for specializing a version of the `SimpleVector` class template so it will work with strings. Complete the specialization for both the `SimpleVector` and `SearchableVector` templates. Demonstrate them with a simple driver program.

13. **Rainfall `vector`**

    Modify Programming Challenge 2 in Chapter 7 (Rainfall Statistics) to use an STL vector instead of an array. Refer to the information in Tables 16-5 and 16-6 if you wish to use any of the member functions.

14. **Test Scores `vector`**

    Modify Programming Challenge 2 in Chapter 9 (Test Scores #1) to use an STL vector instead of a dynamically allocated array. Refer to the information in Tables 16-5 and 16-6 if you wish to use any of the member functions.

15. **STL Binary Search**

    Modify programming Challenge 1 in Chapter 8 (Change Account Validation) so it uses a `vector` instead of an array. Also, modify the program so it uses the STL `binary_search` algorithm to locate valid account numbers.

VideoNote
Solving the Exception Project Problem

16. **Exception Project**

    This assignment assumes you have completed Programming Challenge 1 of Chapter 15 (`Employee` and `ProductionWorker` Classes). Modify the `Employee` and `ProductionWorker` classes so they throw exceptions when the following errors occur:

    - The `Employee` class should throw an exception named `InvalidEmployeeNumber` when it receives an employee number that is less than 0 or greater than 9999.
    - The `ProductionWorker` class should throw an exception named `InvalidShift` when it receives an invalid shift.
    - The `ProductionWorker` class should throw an exception named `InvalidPayRate` when it receives a negative number for the hourly pay rate.

    Write a driver program that demonstrates how each of these exception conditions works.

17. **Phone Book Vector**

    This chapter has an accompanying video note that shows how to store an object in a `vector`. After you view that video, write a class named `PhoneBookEntry` that has members for a person's name and phone number. Then write a program that creates at least five `PhoneBookEntry` objects and stores them in a `vector`. After the objects are stored in the `vector`, use a loop to display the contents of each object in the `vector`.