Object-Oriented Programming Using C++

Fourth Edition

Joyce Farrell
In this chapter, you will:

- Learn about the limitations of traditional error-handling methods
- Throw exceptions
- Use `try` blocks
- Catch exceptions
- Use multiple `throw` statements and multiple `catch` blocks
- Use the default exception handler
- Understand exception classes in the Standard Runtime Library
- Derive your own exceptions from the `exception` class
- Use exception specifications
- Learn about unwinding the stack
- Handle memory-allocation exceptions
- Learn when to use exception handling
HANDLING EXCEPTIONS

Most beginning programmers assume that their programs will work as expected. Experienced programmers, on the other hand, know that things often go awry. If you issue a command to read a file from a disk, the file might not exist. If you want to write to a disk, the disk might be full or unformatted. If the program asks for user input, users might enter invalid data. Such errors that occur during execution of object-oriented programs are called exceptions—so-named because, presumably, they are not usual occurrences. The object-oriented techniques to manage such errors comprise the group of methods known as exception handling. You learn these techniques in this chapter.

NOTE

Sometimes, computer programs generate errors from which the programmer cannot possibly recover. For example, a power failure might interrupt production of your paycheck. Exception handling does not deal with these kinds of errors; its concern is predictable errors.

UNDERSTANDING THE LIMITATIONS OF TRADITIONAL ERROR HANDLING METHODS

From the time the first computer programs were written, programmers have had to deal with error conditions; errors occurred during the execution of programs long before object-oriented methods emerged. Probably the most used error-handling solution has been to make a decision before working with a potentially error-causing value. For example, many older C++ programs contain logic similar to the following:

```c++
perform a first task
if there was an error, perform some error processing
else perform a second task
    if there was an error perform some error processing
    else perform a third task
...and so on until the job is complete
```

This approach works, but is inferior for several reasons:

» The error handling is interspersed with the logic of the program, making the program harder to read and maintain. The main point of the usual program execution (perform the first task, perform the second task, and so on until the job is complete) is hidden between error-handling statements.

» If, as in this example, each subsequent task is performed only if the previous task was successful, the number of nested decisions can become large, also making the program harder to read, and increasing the chance for mistakes.

» The program is less efficient than it could be because decisions are constantly being made to test for situations that rarely occur.

» Further complications make this approach unwieldy. The pseudocode contains several decisions “if there was an error” without indicating how the error is detected. One possibility is that each of the tasks listed in the pseudocode is a function call, and that if the function produces an error, then the function returns an error code. You already know...
that a function can return only a single value, so if each of the functions returns a code, then each function must be written so it returns nothing else, which might be awkward, inconvenient, or undesirable.

**NOTE** This section describes the disadvantages of traditional error handling. Later in this chapter, you will learn about exception handling and the advantages it provides over traditional error handling. At the end of the chapter, however, you will learn that exception handling also has disadvantages.

To avoid the problems inherent in returning error codes from functions, some older C++ programmers simply exited the program when an error occurred. For example, the function in Figure 12-1 asks a user to enter a positive number and assumes it is in error when the user does not. This function uses the shaded `exit()` function to force the program to end when the user’s entered value is less than 0.

```cpp
int getUserNumber()
{
    int userEntry;
    cout << "Enter a positive number ";
    cin >> userEntry;
    if (userEntry < 0)
        exit(1);
    return userEntry;
}
```

*Figure 12-1 The `getUserNumber()` function that uses the `exit()` function*

In the shaded statement in the program segment in Figure 12-1, if `userEntry` is a negative value, the program that calls the function is terminated and control returns to the operating system. The `exit()` function requires an integer argument. It is traditional to use a 0 argument to indicate a program exited normally, and a non-zero argument to indicate an error.

**NOTE** As an alternative to 1 and 0, many compilers allow you to use the defined constants `EXIT_FAILURE` and `EXIT_SUCCESS` as arguments to the `exit()` function. They are defined as 1 and 0, respectively.

One problem with the approach in the function in Figure 12-1 is that only one course of action is possible when a user enters a negative number (the program ends), making the function less reusable in other programs. For example, if you write a new program in which you want the program to continue and the user to reenter the value when it is negative, then the `getUserNumber()` function is not recyclable. Similarly, if you write a third program in which you want to force the number to 0 when a user enters a negative number, the function is not usable there either.

A general rule of modular programming that provides the most flexibility is that a function should be able to determine an error situation, but not necessarily take action on it. A better
HANDLING EXCEPTIONS

alternative to the function version in Figure 12-1 is to let the function discover the error, and then to notify the calling function of the error so the calling function can decide what to do about it. This approach provides flexibility by letting the calling function decide whether to carry on or terminate when the invalid data entry occurs. This approach promotes function reusability—the getUserNumber() function can be used no matter what actions are needed after invalid data is entered.

Fortunately, object-oriented programming languages provide you with techniques that circumvent the problems of traditional error handling. The name for this group of error-handling techniques is exception handling; the actions you take with exceptions involve trying, throwing, and catching them. You try a function; if it throws an exception, you catch and handle it.

TWO TRUTHS AND A LIE: UNDERSTANDING THE LIMITATIONS OF TRADITIONAL ERROR HANDLING METHODS

1. Error-handling has emerged as a problem since the introduction of object-oriented techniques.
2. When error-handling is interspersed with the logic of the program, the program is harder to read and maintain.
3. A general rule of modular programming that provides the most flexibility is that a function should be able to determine an error situation, but not necessarily take action on it.

The false statement is #1. From the time the first computer programs were written, programmers have had to deal with error conditions; errors occurred during the execution of programs long before object-oriented methods emerged.

THROWING EXCEPTIONS

A function can contain code to check for errors, and then send a message when it detects an error. In object-oriented terminology, an exception is an object that contains information that is passed from the place where a problem occurs to another place that will handle the problem. In C++, an exception object can be of any data type, including a scalar or class type, and a variety of exception objects of different types can be sent from the same function, regardless of the function’s return type. In addition, true to object-oriented style, you can create your own exception types that inherit from a built-in base exception class.

Keep in mind that the general principle underlying good object-oriented error handling is that any called function should check for errors, but should not be required to handle an error if one is found. You find this form of error handling convenient in real life. For example, suppose you call your hair stylist to make a 1 o’clock appointment on Thursday, and suppose the stylist has no openings then. Rather than have the stylist make a decision that, instead, you will come in on Thursday at 3 o’clock, you want to be notified of the error situations. Sometimes you will choose to handle the exception by rescheduling for Thursday at 3 o’clock, but other times you will choose a different time, different day, or perhaps a different stylist. Maximum flexibility is achieved when the calling function is responsible for handling the error detected by the called function.

Copyright 2009 Cengage Learning, Inc. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.
When an object-oriented program detects an error within a function, the function should send an error object to the calling function, or **throw an exception**. A throw resembles a return statement in a function, except that the execution of the program does not continue from where the function was called. It also resembles an `exit()` call, except that control does not have to be returned to the operating system. Instead, when you throw an exception, control continues at a new program location chosen by the programmer where the program can **catch**, or receive, the exception that was thrown.

You throw an exception by using the keyword `throw` followed by any C++ object, including an object that is a built-in scalar type, such as an `int` or a `double`; a nonscalar type, such as a `string` or numeric array; or even a programmer-defined object such as an `Employee` or a `Student`. As you will see later in this chapter, frequently, you will want to throw objects that you derive from the built-in `exception` class.

For example, if you write a `getUserNumber()` function in which a negative `userEntry` represents an error condition, then you can throw an error message exception that is a string, as shown in Figure 12-2.

```cpp
int getUserNumber()
{
    int userEntry;
    const string MSG = "Invalid entry";
    cout << "Enter a positive value ";
    cin >> userEntry;
    if(userEntry < 0)
        throw(MSG);
    return userEntry;
}
```

**Figure 12-2** Function that throws an exception

In the `getUserNumber()` function in Figure 12-2, if `userEntry` is negative, then a string is thrown (in the shaded statement) and the execution of the function is finished. Therefore, only valid (that is, non-negative) entries cause the function to continue to execute all the way through the `return` statement. The string that is thrown is not “returned” from the `getUserNumber()` function; the function has an `int` return type and so can return only an `int`. The `getUserNumber()` function concludes in one of two ways: either the error message string is thrown or the `userEntry` integer is returned.

A function can contain any number of `throw` statements. Assume, for example, that you need one error message if a value is negative, but a different error message if a value is greater than 9. The function in Figure 12-3 throws two different error messages, based on a `userEntry`.

---

Copyright 2009 Cengage Learning, Inc. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.
HANDLING EXCEPTIONS

```cpp
int getPositiveSingleDigit()
{
    int userEntry;
    const int LOW = 0;
    const int HIGH = 9;
    const string MSG1 = "Value is negative";
    const string MSG2 = "Value is too high";
    cout << "Enter a single-digit positive value  ";
    cin >> userEntry;
    if (userEntry < LOW)
        throw(MSG1);
    if (userEntry > HIGH)
        throw(MSG2);
    return userEntry ;
}
```

Figure 12-3 Function that throws one of two exceptions

The shaded statements in Figure 12-3 each throw a string, one if the value entered is too low and a different one if it is too high. Not only can a function make multiple throws, but it also can make throws of different types. Assume you need a function that throws an error message if userEntry is negative, but throws the actual value entered if the user enters a number greater than 9. The getPositiveSingleDigit() function in Figure 12-4 accomplishes this goal. One shaded throw statement throws a string; the other throws an integer.

```cpp
int getPositiveSingleDigit()
{
    int userEntry;
    const int LOW = 0;
    const int HIGH = 9;
    const string ERROR_MSG = "Negative number";
    cout << "Enter a single-digit positive value  ";
    cin >> userEntry;
    if (userEntry < LOW)
        throw(ERROR_MSG);
    if (userEntry > HIGH)
        throw(userEntry);
    return userEntry;
}
```

Figure 12-4 Function that throws one of two exceptions of different types

When you use the version of the getPositiveSingleDigit() function shown in Figure 12-4, if the user enters a value between 0 and 9 inclusive, the actual value is returned to the calling function when the function ends. If the user enters a negative number, the string message
“Negative number” is thrown. If the user entry is greater than 9, the actual number that the user entered is thrown. This does not mean that the multi-digit number is returned to the calling function; only values between 0 and 9 are returned to the calling function, which presumably continues using the value. Instead, any value that is 10 or greater is thrown, or sent, to a different location where it can be caught.

**NOTE** At this point, throwing an exception does not appear to have any advantages over simply exiting a function when an error occurs. When you learn to catch exceptions later in this chapter, you will be able to appreciate more fully the advantages of object-oriented exception handling.

**TWO TRUTHS AND A LIE: THROWING EXCEPTIONS**

1. An exception is an object that contains information that is passed from the place where a problem occurs to another place that will handle the problem.
2. In C++, an exception object can be of any data type, including a scalar or class type.
3. In C++, a function can only throw an object that has the same type as the function’s return type.

The false statement is #3. Any type can be thrown from a function regardless of the return type.

**USING TRY BLOCKS**

When a function might cause an exception, and therefore includes a statement that throws an exception, the call to the potentially offending function should be placed within a try block. A try block consists of one or more statements that the program attempts to execute, but which might result in thrown exceptions. You want to place a function call in a try block when the function might throw an exception, because placing it in a try block allows you to handle, or “take care of,” the problem that caused the exception.

A try block includes the following components:

- the keyword try
- an opening curly brace
- the statements that are tried
- a closing curly brace

For example, Figure 12-5 shows a main() function that calls the getUserNumber() function from Figure 12-2—the one that throws a string message if the user’s value is negative. In this case, the function call is contained in the only statement within the shaded try block, but any number of statements could be placed within the block. When an if, for, or while block contains just one statement, curly braces are not required around the contained statement. With a try block, however, you must include the curly braces, even if only one statement is tried.
HANDLING EXCEPTIONS

In Figure 12-4, the call to the `getUserNumber()` function occurs within the shaded `try` block. When the function executes, if the `userEntry` is valid, then no exception is thrown, and `main()` executes to the end, using the valid value returned by the `getUserNumber()` function.

The `main()` function in Figure 12-5 is incomplete, however. If the `getUserNumber()` function throws an exception, you want to be able to handle the exception; you handle a thrown exception by catching it.

```cpp
int main()
{
    int value;
    try
    {
        value = getUserNumber();
    }
    cout << "Data entry value is " << value << endl;
    // rest of the program goes here
}
```

Figure 12-5 Incomplete `main()` function containing a `try` block

In Figure 12-4, the call to the `getUserNumber()` function occurs within the shaded `try` block. When the function executes, if the `userEntry` is valid, then no exception is thrown, and `main()` executes to the end, using the valid value returned by the `getUserNumber()` function.

The `main()` function in Figure 12-5 is incomplete, however. If the `getUserNumber()` function throws an exception, you want to be able to handle the exception; you handle a thrown exception by catching it.

```cpp
int main()
{
    int value;
    try
    {
        value = getUserNumber();
    }
    cout << "Data entry value is " << value << endl;
    // rest of the program goes here
}
```

Figure 12-5 Incomplete `main()` function containing a `try` block

**NOTE**
Programmers refer to the scenario where no errors occur and hence no exceptions are thrown as the "sunny day" case.

**NOTE**
If a function throws an exception, but the function call has not been placed in a `try` block, then the program terminates.

**TWO TRUTHS AND A LIE: USING TRY BLOCKS**

1. A `try` block consists of one or more statements that are guaranteed to throw exceptions when executed.
2. You want to place a function call in a `try` block when the function might throw an exception, because placing it in a `try` block allows you to handle the problem that caused the exception.
3. When a `try` block contains just one statement, you still must enclose the statement in curly braces.

**T** **T** **F**

The false statement is #1. A try block consists of one or more statements that the program attempts to execute, but which might result in thrown exceptions.

**CATCHING EXCEPTIONS**

To handle a thrown object, you include one or more `catch` blocks in your program immediately following a `try` block. A `catch` block contains statements that execute when an exception is thrown and includes the following components:

- the keyword `catch`
- a single parameter in parentheses
- an opening curly brace
- one or more statements that describe the exception action to be taken
- a closing curly brace
For example, Figure 12-6 shows the \texttt{getUserNumber()} function from Figure 12-2, along with a \texttt{main()} function that uses it. The \texttt{main()} function calls \texttt{getUserNumber()} in a \texttt{try} block. The \texttt{getUserNumber()} function throws a string error message if the user enters a negative number. If the user enters a valid (non-negative) number during the execution of \texttt{getUserNumber()}, control returns to the \texttt{main()} function where the entered value is displayed, then the \texttt{catch} block is bypassed and the “End of program” message displays. Figure 12-7 shows the execution of the program when the user enters a non-negative value and Figure 12-8 shows the execution when a negative value is entered.

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

int getUserNumber()
{
    int userEntry;
    const string MSG = "Invalid entry";
    cout << "Enter a positive value ";
    cin >> userEntry;
    if (userEntry < 0)
        throw(MSG);
    return userEntry;
}

int main()
{
    int returnedValue;
    try
    {
        returnedValue = getUserNumber();
        cout << "Data entry value is " << 
            returnedValue << endl;
    }
    catch(const string message)
    {
        cout << "There was an error!" << endl;
        cout << message << endl;
    }
    cout << "End of program" << endl;
    return 0;
}
```

Figure 12-6 A \texttt{main()} function that tries \texttt{getUserNumber()} and catches the thrown exception
HANDLING EXCEPTIONS

If the user enters a negative number during the execution of `getUserNumber()` in the program shown in Figure 12-6, then the function is abandoned, the rest of the `try` block in the `main()` function (the statement that displays the entered number) is bypassed, and the shaded `catch` block executes. In this case, the `catch` block executes, catching the thrown string message. As shown in the output in Figure 12-8, the `catch` block displays two strings: the first (“There was an error!”) is coded within the `catch` block, and the second (locally called `message`) was thrown by the `getUserNumber()` function. After executing the `catch` block, the `main()` function continues and displays “End of program”.

If you want a `catch` block to execute, it must `catch` the correct type of argument thrown from a `try` block. If an argument is thrown and no `catch` block has a matching parameter type, then the program terminates. However, a `catch` block is not required to display, or to use in any way, what is thrown. For example, the `catch` block in the program in Figure 12-6 is not required to display the `message` parameter. Instead, the `catch` block could display only its own message, assign a default value to `returnedValue`, or contain any number of valid C++ statements, including those that call other functions. The `catch` block could even contain no statements at all.

USING MULTIPLE THROW STATEMENTS AND MULTIPLE CATCH BLOCKS

Often, several types of exceptions can occur within a function. You can write a function to throw any number of exceptions, and you can provide multiple `catch` blocks to react appropriately to each type of exception that is thrown.

For example, the `getPositiveSingleDigit()` function in Figure 12-4 throws a string message when the user enters a negative number, but throws the out-of-range value when the user enters...
a number that is greater than 9. Figure 12-9 shows a program that uses the function in a try block and uses multiple catch blocks. In the program in Figure 12-9, if the string is thrown from the getPositiveSingleDigit() function, it is caught by the first catch block, if the integer is thrown, it is caught by the second catch block, and if the function ends normally and the userEntry is returned, it is assigned to the value variable in the main() function.

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

int getPositiveSingleDigit()
{
    int userEntry;
    const int LOW = 0;
    const int HIGH = 9;
    const string ERROR_MSG = "Negative number";
    cout << "Enter a single-digit positive value ";
    cin >> userEntry;
    if (userEntry < LOW)
        throw(ERROR_MSG);
    if (userEntry > HIGH)
        throw(userEntry);
    return userEntry;
}

int main()
{
    int value;
    try
    {
        value = getPositiveSingleDigit();
        cout << "The entered value was good" << endl;
    }
    catch(string msg)
    {
        cout << "A message was thrown " << endl;
        cout << msg << endl;
    }
    catch(int badValue)
    {
        const int REDUCTION = 10;
        value = badValue % REDUCTION;
        cout << "A value was thrown " << endl;
        cout << "The number you entered, " << badValue << ", is too large." << endl;
        cout << "So it is being reduced to " << value << endl;
    }
    cout << "The value at the end of the program is " << value << endl;
    return 0;
}
```

Figure 12-9 A program that tries a function and catches two types of exceptions

Copyright 2009 Cengage Learning, Inc. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.
When you run the program in Figure 12-9, if no exception is thrown, the program bypasses both `catch` blocks and prints the valid value, as shown in Figure 12-10. If the user enters a negative number, as shown in Figure 12-11, then a string is thrown. In this case, the first `catch` block executes, and the second `catch` block is bypassed. If the user enters a number greater than or equal to 100, as in Figure 12-12, then an integer is thrown. In this case, the first `catch` block is bypassed, and the second `catch` block executes. In this `catch` block, the remainder that results when the number is divided by 10 is used as the value; the value of `badValue % 10` always produces a number between 0 and 9 inclusive. In Figure 12-12, whether the `getPositiveSingleDigit()` function ends normally (with the `return` statement) or throws an exception, the program ends with the `cout` statement that follows the last `catch` block.

The `getPositiveSingleDigit()` function in Figure 12-6 remains flexible so that a wide variety of applications can use it. A different calling `main()` function might want to force negative values to their positive absolute value, or reprompt the user when the entered value is too high, or force all invalid entries to 9. By creating the `getPositiveSingleDigit()` function to throw exceptions, all client programs can choose how to best handle various scenarios.

**DETERMINING THE ORDER OF `catch` BLOCKS**

When you include multiple `catch` blocks in a program, the first `catch` block that can accept a thrown object is the one that will execute. When you create a function that throws several
types, such as an integer and an Employee, it doesn’t matter which catch block you place first. If an Employee object is thrown, the appropriate catch block executes whether it is written before or after the integer catch block. Similarly, if you develop an Inventory class, and a function throws both an Employee and an Inventory object, it does not matter which catch block is placed first.

However, if you need to throw both a base class object and an object that is a member of its derived class from the same function, and you want to carry out different operations when they are caught, then you must code the catch for the derived object first. For example, if you create an Employee class and a child PartTimeEmployee class, and either might be thrown from a function, then you must code the catch block for the PartTimeEmployee object first. If you code the Employee catch block first, the derived PartTimeEmployee object that is thrown will incorrectly match the first catch block because, as a child of Employee, a PartTimeEmployee “is an” Employee. If the Employee catch block precedes the PartTimeEmployee catch block, no object can ever reach the second catch block.

**T T F**

**TWO TRUTHS AND A LIE: USING MULTIPLE throw STATEMENTS AND MULTIPLE catch BLOCKS**

1. You can write a function to throw any number of exceptions, and you can provide multiple catch blocks to react appropriately to each type of exception that is thrown.
2. If no exception is thrown in a program, only the default catch block executes.
3. When you include multiple catch blocks in a program, the first catch block that is able to accept a thrown object is the one that will execute.

The false statement is #2. If no exception is thrown in a program, no catch blocks execute.

**USING THE DEFAULT EXCEPTION HANDLER**

When any object is thrown with a throw statement, then a subsequent catch block has a usable match if one of the following conditions is true:

- The type of the thrown object and the type of the catch parameter are identical (for example, int and int).
- The type of the thrown object and the type of the catch parameter are the same, except the catch parameter has the const qualifier, a reference qualifier, or both (for example, int can be caught by const int, int&, or const int&).
- The catch parameter type is a parent class of the thrown argument. For example, if PartTimeEmployee derives from Employee, a catch block with an Employee parameter can catch both.

If you throw an exception and no catch block exists with an acceptable parameter type, then the program terminates. To avoid termination, you can code a default exception handler or default catch block that catches any type of object not previously caught. You create a default
exception handler by creating a catch block with an ellipsis (…) as its parameter. If you use a default catch block, it must be the last catch block listed after a try. The default catch block will catch any type of thrown object that has not been caught by an earlier catch block.

**Note**: Besides avoiding unexpected program termination, you might want to code a default catch block when you need to handle several exception types the same way. For example, if a function might throw an int, double, string, or Employee and you want to handle the int exception one way but all the other types in a single alternate way, then you need only two catch blocks—one for the int and the other one that is a default catch block.

**Two Truths and a Lie: Using the Default Exception Handler**

1. A default catch block catches any type of object not previously caught.
2. You create a default exception handler by creating a catch block followed by empty parentheses.
3. If you use a default catch block, it must be the last catch block listed after a try.

The false statement is #2. You create a default exception handler by creating a catch block with an ellipsis (…) as its parameter.

**Understanding Exception Classes in the Standard Runtime Library**

The C++ Standard library contains a class that it uses for the exceptions its functions throw. The class is called exception and is defined in the <exception> header file. Several classes already have been derived from the built-in exception class. Figure 12-13 shows the relationship of these classes to the exception class. Two of the first-generation descendents of exception are logic_error and runtime_error; each of these has several descendents of its own.

**Figure 12-13** Standard Library exception class hierarchy
The category of exceptions known as *logic error* s are errors in preconditions of some operation, such as passing an invalid value to a function. The category known as *runtime error* s include problems that occur during program execution. For example, an *overflow error* occurs when the result of an arithmetic operation is larger than the largest number that can be stored on a computer. You will learn more about the other exception types as you continue to study C++, but in brief:

- A *bad_alloc* exception occurs when you attempt to allocate additional memory and an insufficient amount exists.
- The *dynamic_cast* operator converts pointers from parent to child types. A *bad_cast* exception occurs when a *dynamic_cast* to a reference type fails.
- A *bad_exception* is generated if you rethrow an exception from within an unexpected handler. You will learn about rethrowing exceptions later in this chapter.
- The *typeid* operator returns information about an object’s data type. A *bad_typeid* exception is generated if you attempt to apply the *typeid* operator to a null expression.

You want to understand what these built-in exception types do in a general way so that when you extend the types to create your own exception classes you can choose the most appropriate type from which to inherit. The reason you want to extend from the appropriate type is so that your exceptions can be caught with others like themselves. For example, if you create an exception class that extends *logic error*, then when your exception type is thrown, it will be caught in the same place that other *logic error* s are caught.

**AN EXAMPLE OF AN AUTOMATICALLY THROWN logic_error**

The group of exceptions that are *logic error* exceptions include problems that occur because of errors in program logic. For example, the *string* class in the Standard Library contains a number of useful functions for working with *strings*. One of them, the *substr()* function, throws an *out of range* exception (a child of *logic error*) when it receives an invalid parameter. The function extracts part of a *string* by using two parameters, a starting point and a length, and returns a substring from the original. For example, if a *string* named *alpha* contains “abcdefg” then the value of *alpha.substr(2, 3)* is “cde” because three characters are extracted starting with “c” which is in position 2 (“a” is in position 0). However, if the starting position parameter indicates a position beyond the length of the *string* object being used, then *substr()* throws a built-in exception.

Figure 12-14 shows a program in which the objective is to insert a dash as the fourth position in a telephone number that has no dashes. The user is prompted to enter a phone number. Then two substrings are extracted—the first three characters and the last four characters, so that a dash can be placed between them. Figure 12-15 shows a typical execution of the program in which all goes well. Figure 12-16 shows what happens when a user enters a phone number.
number for which the second call to `substr()` fails because there are only two positions in the entered string and the second call to `substr()` uses a starting position of 3. Although you do not see a `throw` statement in the program in Figure 12-14, you can tell an `out_of_range` exception is thrown from `substr()` because the output shows it has been caught.

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

int main()
{
    string phoneNumber;
    cout << "Enter your 7-digit phone number without spaces ";
    cin >> phoneNumber;
    try
    {
        string exchange = phoneNumber.substr(0, 3);
        string number = phoneNumber.substr(3, 7);
        cout << "Formatted number is: " << exchange << "-" << number << endl;
    }
    catch(out_of_range exc)
    {
        cout << "Exception caught" << endl;
        cout << "Phone number is invalid" << endl;
    }
    return 0;
}
```

Figure 12-14 Program that extracts phone number

Figure 12-15 Typical execution of program in Figure 12-14

Figure 12-16 Exceptional execution of program in Figure 12-14
The program in Figure 12-14 could have been written without placing the `substr()` calls in a `try` block. Then, if a user enters a too-short phone number, the program would end abruptly and issue a runtime error message. By using the `substr()` function within a `try` block and handling the thrown exception, the program can continue elegantly. The programmer could simply display a message as in Figure 12-14, or the programmer could substitute a default phone number, prompt the user for a new phone number, or take whatever action was most appropriate for the application at hand.

**USING THE `what()` FUNCTION**

When you use any descendent of the built-in exception class, you inherit a function named `what()` that returns an exception object’s error message. Figure 12-17 shows the `catch` block from Figure 12-14 rewritten to display the `what()` value of the `exc` parameter. Figure 12-18 shows an execution of the program with a too-short user entry. The message displayed is the built-in one that comes supplied with every `out_of_range` exception object. If this message is suitable for your program, there is no need to write your own messages as in the original version of the program. Instead, you can use the built-in `what()` value.

```cpp
catch(out_of_range exc)
{
    cout << exc.what() << endl;
}
```

*Figure 12-17* Rewritten `catch` block from Figure 12-14 that uses the `what()` function

**Figure 12-18** Execution of the program in Figure 12-14 using the substitution `catch` block in Figure 12-17

**EXPLICITLY THROWING A BUILT-IN exception**

Although some classes in the C++ Standard library use the hierarchy of exceptions that descend from the `exception` class, you also can use any of the built-in `exception` types whenever you want to throw an exception in one of your programs. Just like when you declare objects of any data type, to declare an `exception` object, you use the class name and an identifier. If you create an `exception` base class object, you can supply the constructor with an argument, or you can use the default constructor. If you create an object that is an instance of any of exception’s children, then the constructor requires a `string` argument that is used as an informational message about the exception.
HANDLING EXCEPTIONS

For example, Figure 12-19 contains a function named `getNonZeroValue()` that is similar to others you have seen in this chapter. It accepts a prompt as a parameter so that it can be used in many types of applications where the prompt differs. In the shaded statement, the function declares an object of type `runtime_error` (which is a child of `exception`). The object’s name is `error` and an informational string is passed to the `runtime_error`’s constructor. Within the function, the passed-in prompt is displayed for the user, a user entry is accepted, and, if the entry is 0, the `runtime_error` is thrown. If the user’s value is nonzero, nothing is thrown, and the entered value is returned to the calling function.

```cpp
int getNonZeroValue(string prompt)
{
    runtime_error error("User's entry was zero");
    int usersEntry;
    cout << prompt;
    cin >> usersEntry;
    if (usersEntry == 0)
        throw(error);
    return usersEntry;
}
```

Figure 12-19 Function that declares and throws a `runtime_error`

**NOTE**: The `runtime_error` class was selected to use as a basis for the exception in the `getNonZeroValue()` function because user entry errors are errors that occur at runtime. The program would work as well if you used another exception type.

**NOTE**: When you use the `runtime_error` class, you must use the statements `#include<exception>` and `using namespace std;` in your program.

**NOTE**: Choosing appropriate `what()` messages is a fine art. It takes skill to foresee all the uses a client might have for your function and to make the `what()` message appropriate and useful for each one.

Figure 12-20 shows a program that uses the `getNonZeroValue()` function in Figure 12-19. This program allows a user to enter a time in minutes it took to complete a marathon. The `getNonZeroValue()` function call is placed in a `try` block (see the first shaded statement), because if the data entry function returns a 0, and the problem is not handled, the program will exit prematurely. When the miles per hour variable is calculated, the time is divided into the number of miles in the race. If the time is 0, an error will occur because division by zero is an undefined operation. If the user enters a 0 and the exception is thrown from the function, then the `catch` block executes and displays two messages: “Error!” and the `what()` reason that is available within the `catch` block as part of the `runtime_exception` object named `e`. (See the second shaded statement in the figure.)
Figures 12-21 and 12-22 show two executions of the program in Figure 12-20. In one figure, the user has entered a valid value and the program ends normally. In the other figure, the user has entered 0, the exception is caught, and the returned value of \texttt{what()} associated with the exception is displayed.

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

int main()
{
    const double MINS_IN_HR = 60.0;
    const int MILES_IN_RACE = 26;
    double hours;
    int timeInMinutes;
    double mph = 0;

    try
    {
        timeInMinutes = getNonZeroValue("Enter time in minutes to complete the race ");
        hours = timeInMinutes / MINS_IN_HR;
        mph = MILES_IN_RACE / hours;
    }
    catch(runtime_error e)
    {
        cout << "Error!" << endl;
        cout << e.what() << endl;
    }

    cout << "You ran at a rate of " << mph << " miles per hour" << endl;
    return 0;
}
```

**Figure 12-20** Program that tries and catches a \texttt{runtime\_error} exception

**NOTE** You are not required to place the function call to \texttt{getNonZeroValue()} in a \texttt{try} block. If you do not, but the function throws an exception, it will be thrown to the operating system and you will receive an error message similar to "The application has requested the Runtime to terminate it in an unusual way."

Figures 12-21 and 12-22 show two executions of the program in Figure 12-20. In one figure, the user has entered a valid value and the program ends normally. In the other figure, the user has entered 0, the exception is caught, and the returned value of \texttt{what()} associated with the exception is displayed.
HANDLING EXCEPTIONS

C++ programs can throw exceptions of any data type. This facility is unusual among programming languages; other languages such as Java and C# require that exception objects belong to a predefined exception class or a class you derive from it. Although using the exception class as a basis for inheritance for your own exceptions is not required in C++, several advantages exist to doing so. These advantages include the following:

» That technique is similar to other programming languages, making your programs easier for object-oriented programmers to understand.

» Exception-handling using the built-in classes as a basis provides a more uniform way to handle errors. This is beneficial when many programmers combine multiple modules to create a large project.

» If all exceptions derive from exception or one of its subclasses, then when you want to handle several program exceptions in the same way, you include a catch block with an exception parameter. This technique is superior to using the default parameter (the one that uses the ellipsis) because the exception object can have an identifier.

» When an exception is of type exception or one of its subclasses, you can pass a string to the constructor that can be used as an informative message returned from the what() function. Alternatively, you can override the parent class’s what() function with your own version and programmers who use your exception class will understand its purpose.
Suppose your organization has use for a data entry function that checks to assure user's entry falls within a specified range. When the user's data is out of range, you could throw a runtime_error. However, when you throw a runtime_error, although you can extract the message returned by the what() function, you do not have a way to determine what the user's invalid entry was. Client programs might want to use the value to display, as part of a calculation, or as an argument to another function. Therefore, you might want to create a custom exception class that can access the user's out-of-range entry.

Figure 12-23 shows such a class. The class name is RangeException, and it inherits from runtime_error. The class contains a private data member that holds a user's entered value. The two public members are a constructor, which requires an integer argument, and a function that returns an integer value.

```cpp
class RangeException : public runtime_error
{
    private:
        int user;
    public:
        RangeException(int);
        int getUser();
};
RangeException::RangeException(int user) :
    runtime_error("Value is out of range")
{
    this->user = user;
}
int RangeException::getUser()
{
    return user;
}
```

Figure 12-23 The RangeException class

The RangeException class constructor accepts an integer and assigns it to its private integer data member. It also passes an error message to the runtime_error class constructor. The getUser() function in the class returns the user-entered value.

Figure 12-24 contains a function that prompts the user for a value that falls within the range dictated by the two parameters. If the user's entry is out of range, the function constructs a RangeException object using the user's entered value as an argument. The RangeException is then thrown. If the user's entered value is within range, no RangeException object is created or thrown, and the user's valid value is returned from the function.
Figure 12-25 shows an application that uses the `getValueInRange()` function. (See shaded statement.) The application declares low and high range limits that are passed to the function in a try block. If the function is successful, the in-range result is displayed at the end of the program. However, if the function throws a `RangeException`, the catch block executes. Within it, the `what()` message is displayed and then the out-of-range value is extracted from the `RangeException` object. The value is used in one of two statements—to indicate either how far too high or too low the user’s value was. By endowing the `RangeException` class with the capability to store and retrieve the entered value from the function, the programmer has increased the usefulness of the exception over a “plain” `runtime_error`. Figure 12-26 shows three program executions in which the entered value is in range, too high, and too low.

```cpp
#include<iostream>
#include<exception>
using namespace std;
int main()
{
    const int LOW = 3;
    const int HIGH = 7;
    int score;
    int badUserValue;
    int difference;
    try
    {
        score = getValueInRange(LOW, HIGH);
    }

    Figure 12-25 An application that uses the `getValueInRange()` function (continued)
```cpp
catch(RangeException e)
{
    cout << e.what() << endl;
    badUserValue = e.getUser();
    if(badUserValue < LOW)
    {
        difference = LOW - badUserValue;
        cout << "The entry was " << difference << " too low" << endl;
    }
    else
    {
        difference = badUserValue - HIGH;
        cout << "The entry was " << difference << " too high" << endl;
    }
    score = 0;
}
cout << "Score is " << score << endl;
return 0;
}
```

Figure 12-25 (continued)

Figure 12-26 Three executions of the program in Figure 12-25
HANDLING EXCEPTIONS

The `RangeException` class used in the function called by the program in Figure 12-25 could have been written to return a message about the entered value instead of returning the actual value. However, it is more flexible for the client if the function returns the value because the client can choose to handle the value in the way best suited to its needs. Because the returned value is an `int`, the client has the option of displaying it, using it in a decision, performing arithmetic with it, and so on. Well-designed exceptions and functions provide the maximum flexibility for client programs.

**NOTE** When creating a class to hold an exception, make sure that the instantiation of your exception class does not result in the same problem as the original error did. For example, if the original error was caused by insufficient memory, it's probably a poor idea to have the exception class constructor allocate more memory.

OVERRIDING THE EXCEPTION CLASS `what()` FUNCTION
When you create your own exception class derived from a built-in exception, you can pass an error message to the parent class’s constructor; the message will become the one returned by the exception’s `what()` function. As an alternative, you can override the parent class’s `what()` function with your own version. It makes sense to use the `what()` function for your exception’s error message instead of creating some other function with a name like `displayErrorMessage()` for the following reasons:

» Other programmers will immediately understand the purpose of and know how to use `what()` with exception objects.

» If a client program catches a group of exceptions, such as all exceptions, or all `runtime_exception`s, then one catch block can handle all of them, but use `what()` with the generic parameter to display the message appropriate to the caught subtype.

The built-in `what()` function is a virtual function and returns a constant character pointer. To override the exception class `what()` method so that its behaviors are consistent with the parent version, you create the following definition within your derived class:

```cpp
virtual const char* what() ;
```

**NOTE** The `what()` function is a virtual function. In Chapter 10 you learned that a virtual function is one that is included in a child class just once even if it is inherited from multiple parents. Using virtual inheritance from the `exception` base class prevents ambiguity problems when an exception is derived from two child classes of `exception` that both have the `exception` base class in common.

**NOTE** When you override a function, the name and parameter list must match, but the return type is not required to match. However, if you want a child class function to always be used in the same way as the parent version, it makes sense to use the same return type.

For example, Figure 12-27 contains a rewritten `RangeException` function that contains its own version of the `what()` function. The changes from Figure 12-23 are shaded. An empty string is passed to the `runtime_error` constructor because a string argument is required by the constructor. Because the message in the `what()` method of the revised `RangeException`
class is the same as the one used in Figure 12-23, when this class is used with the program in Figure 12-25, the output is identical to that shown in Figure 12-26.

```cpp
class RangeException : public runtime_error
{
    private:
        int user;
    public:
        RangeException(int);
        int getUser();
        virtual const char* what();
};
RangeException::RangeException(int user) : runtime_error("")
{
    this->user = user;
}
int RangeException::getUser()
{
    return user;
}
const char* RangeException::what()
{
    return "Value is out of range";
}
```

Figure 12-27 RangeException class with what() method that overrides parent’s version

One advantage of overriding the what() function in your derived exception class is that you can customize the returned message based on current conditions. For example, in the RangeException class what() function, you could use an if statement to return different messages based on the current value of the user field.

**TWO TRUTHS AND A LIE: DERIVING YOUR OWN EXCEPTIONS FROM THE exception CLASS**

1. You must inherit from the exception class when you create your own exceptions in C++.
2. When you extend an exception class, you can override an exception class’s what() function with your own version.
3. The built-in what() function is a virtual function that returns a constant character pointer.

The false statement is #1. Although using the exception class as a basis for inheritance for your own exceptions is not...
HANDLING EXCEPTIONS

USING EXCEPTION SPECIFICATIONS

In C++, any function might throw any type of object. You might not realize how many different types of objects a function throws if you fail to carefully examine the code. You can explicitly indicate the exceptions that a function can possibly throw by writing an exception specification, which is a declaration of a function's possible throw types. An exception specification is also called a throw list. Creating an exception specification provides documentation for later users of the function by indicating what types of errors might possibly be thrown. The user then can plan appropriate catch blocks.

You write an exception specification in both a function's prototype and in a function's header immediately after the list of function parameters. Simply write the keyword throw followed by a list of argument types in parentheses. For example, for a dataEntry() function that takes no parameters and returns an integer—and that might throw a char, a double, or an Employee object—you could code the function header as follows:

```cpp
int dataEntry() throw(char, double, Employee)
```

If you write an exception specification with empty parentheses following throw, you declare that the function will not throw any exceptions. For example, the following function will not throw any exceptions:

```cpp
int dataEntry() throw()
```

Remember that function headers and prototypes that do not include an exception specification list can throw anything (or might throw nothing). You have seen many such functions throughout this chapter. In other words, if you do not specify the exceptions, then any type of exception might be thrown. Once you do specify the exceptions for a function (by including an exception specification list), then only those types listed should be thrown. In some compilers, if you use a throw specification clause with a function, and then throw a type that is not listed, an error will occur and the program will stop prematurely.

EXCEPTION SPECIFICATIONS IN ANSI C++

The American National Standards Institute (ANSI) is an organization that oversees the development of standards for products, services, processes, systems, and personnel. ANSI has developed a C++ Standard that covers exception specifications. In ANSI C++, if a function throws an error whose type was not listed in its exception specification, then it will produce a run-time error called an unexpected exception, and abort the program.

If you include an exception specification list with a function, and code a throw type that is not listed, the program will compile and execute if the unlisted type is never actually thrown. For example, if a function specification list does not include type double, but the function is written to throw a double when a user enters a negative number, and if no user ever enters a negative number, the function still will run correctly. However, if a user does enter a negative value and the double is thrown, then the program will end because double was not included in the specification list.
HOW VISUAL C++ DEPARTS FROM THE ANSI STANDARD

Visual C++ departs from the ANSI standard in its implementation of exception specifications. In Visual C++, when you include an exception specification, you receive a warning that the exception specification is ignored. Of course, assuming you want to specify exceptions for documentation purposes, you can ignore this warning.

In Visual C++ the following specifications have unique meanings:

- `throw()` means the function does not throw an exception. However, if one is thrown, the compiler does not interpret it as an unexpected exception. You should not, however, use `throw()` with your functions unless they do not throw any exception because the program might not execute correctly due to some code optimizations the compiler might perform based on the assumption that no exceptions will be thrown.

- `throw(...)` means the function can throw an exception.

- `throw(type)` means the function can throw an exception of type `type`. However, in Visual C++, this is interpreted the same as `throw(...)`. Although Visual C++ does not adhere to the exception specification standard, including an exception specification for documentation purposes is still a good idea. However, beware that including an exception specification serves to indicate to others what your function does, but does not force the function to do what you say it will.

TWO TRUTHS AND A LIE: USING EXCEPTION SPECIFICATIONS

1. You write an exception specification in both a function’s prototype and in a function’s header immediately after the list of function parameters.
2. If you write an exception specification with an ellipsis in the parentheses following `throw`, you declare that the function will not throw any exceptions.
3. Function headers and prototypes that do not include an exception specification list can throw anything or might throw nothing.

The false statement is #2. If you write an exception specification with empty parentheses following `throw`, you declare that the function will not throw any exceptions.

UNWINDING THE STACK

In a function you write, you can try a function call and, if the function you call throws an exception, you can catch the exception in your function. However, if your function doesn’t catch the exception, then a function that calls your function can still catch the exception.

A simpler way to say this is that if function A calls function B, function B calls function C, and function C throws an exception, then if B does not catch the exception, function A can.
function catches the exception, then the program terminates. This process is called **unwinding the stack**, because each time you call a function, the address of the place to which the logic should return at the end of the function is stored in a memory location called the stack. Each time you return from a function, the correct return-to destination is retrieved from the stack.

You can picture the stack as a stack of plates, as shown in Figure 12-28. When you stack plates on top of each other, the first one is placed on the bottom, and each new one is subsequently placed on the top of the existing pile. Similarly, when a `main()` function calls `functionA()`, the computer “remembers” where to return at the end of the function by placing the return location (memory address) that resides in `main()` at the bottom of the stack. When `functionA()` calls `functionB()`, the return location within `functionA()` is placed “on top of” the `main()` address in the stack.

![Figure 12-28 Building the stack as functions are called](image-url)

```c
functionB()
{
    // some statements
    return;
}
functionA()
{
    functionB();
    // some statements
    return;
}
main()
{
    functionA();
    // some statements
    return;
}
```

Copyright 2009 Cengage Learning, Inc. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.
With a stack of plates, when you want to dismantle the stack, you must remove the last plate stacked before you can remove the previous one. Similarly, in the program example in Figure 12-28, when functionB() ends, the top address in the stack (the one that returns to functionA()) is retrieved from the stack and the logic continues within functionA(). Figure 12-29 illustrates this process. When functionA() ends, the next address in the stack (the one that returns to main()) is retrieved, and the logic continues with the rest of the main() function. When the main() function ends, there is nothing in the stack, so the program terminates.

```
functionB()
{
    // some statements
    return;
}

functionA()
{
    functionB();
    // some statements
    return;
}

main()
{
    functionA();
    // some statements
    return;
}
```

```
functionB()
{
    // some statements
    return;
}

functionA()
{
    functionB();
    // some statements
    return;
}

main()
{
    functionA();
    // some statements
    return;
}
```

Figure 12-29 Unwinding the stack
Consider a simple `Dog` class like the one shown in Figure 12-30. It holds a `Dog`'s name and age, and supports overloaded insertion and extraction operators. In this example, the overloaded extraction operator that provides data entry uses the shaded `if` statement to instantiate and throw a `RangeException` object (as defined in Figure 12-23) if the `Dog`'s age is higher than the specified limit.

```cpp
class Dog
{
    friend ostream& operator<<(ostream&, Dog&);  // overloaded insertion operator
    friend istream& operator>>(istream&, Dog&)  // overloaded extraction operator
        throw(RangeException);
private:
    string name;
    int age;
};
ostream& operator<<(ostream& out, Dog& dog)  // overloaded insertion operator
{
    out << dog.name << " Age: " << dog.age << " years old";
    return out;
}
istream& operator>>(istream& in, Dog &dog)  // overloaded extraction operator
    throw(RangeException)
{
    const int HIGH_AGE = 20;
    cout << "Enter dog's name ";
    in >> dog.name;
    cout << "Enter age ";
    in >> dog.age;
    if(dog.age > HIGH_AGE)
    {
        RangeException re(dog.age);
        throw(re);
    }
    return in;
}
```

**Figure 12-30** The `Dog` class

Figure 12-31 shows a `KennelReservation` class that holds information about boarding a `Dog`. Each `KennelReservation` includes a kennel number, a month and day for the reservation, and a `Dog`. The extraction operator implementation includes an exception specification (which is shaded) indicating that the extraction operator might throw a `RangeException` object. If you examine the `operator>>()` function code, you cannot find a `throw` statement. The `throw` is hidden within the `Dog` class overloaded `operator>>()` function that is used in the last shaded statement in the figure. If the `Dog` class input function throws an exception, it will be passed to the `KennelReservation` input function. Since the `KennelReservation` input function does not catch the exception, the exception will be passed on to any function that uses the `KennelReservation` `operator>>()` function.
class KennelReservation
{
    friend ostream& operator<<(ostream&, KennelReservation);
    friend istream& operator>>(istream&, KennelReservation&)
        throw (RangeException);
private:
    int kennelNumber;
    Dog dog;
    int month;
    int day;
};
ostream& operator<<(ostream& out, KennelReservation res)
{
    out << "Reservation for " << res.dog << " for " << res.month << "/" << res.day << " for kennel # " << res.kennelNumber;
    return out;
}
istream& operator>>(istream& in, KennelReservation& res)
    throw(RangeException)
{
    cout << "Enter kennel # ";
    in >> res.kennelNumber;
    cout << "Enter month ";
    in >> res.month;
    cout << "Enter day ";
    in >> res.day;
    in >> res.dog; // call to Dog operator>> ()
    return in;
}

Figure 12-31 The KennelReservation class

Within the KennelReservation class in Figure 12-31, both shaded exception specifications could be deleted if the program is compiled using Visual C++, and the class still would function properly. However, they provide useful documentation. Without the exception specifications, a client who uses the KennelReservation class might be unaware or forget that the Dog class throws an exception. Using the exception specification clause in the KennelReservation input function prototype and header provides a reminder to users that they might want to place the KennelReservation operator>>() call within a try block, and they might want to catch any potentially thrown RangeException object.

Figure 12-32 shows a main() function that instantiates a KennelReservation object, and Figure 12-33 shows a typical execution. When the user enters KennelReservation data, Dog data also is entered. When the user's data causes an exception, the Dog input function throws an exception to the KennelReservation input function, which throws the
exception to main(), which in turn catches it and displays a message. In this main() function, the Dog is allowed to have a high age, but a warning message is issued. Other client programs that use the KennelReservation class might use a default age or require the user to reenter the age.

```cpp
int main()
{
    KennelReservation aRes;
    try
    {
        cin >> aRes;
    }
    catch (RangeException re)
    {
        cout << "Age was entered as " << re.getUser() << ", that is a very high age, but technically possible."
        cout << "Please call the owner and verify the age."
        cout << aRes << endl;
    return 0;
    }
}
```

Figure 12-32 A main() function that instantiates a KennelReservation

Figure 12-33 Typical execution of program in Figure 12-32
RETHROWING EXCEPTIONS

In the KennelReservation example in the last section, a Dog class function threw an exception to a KennelReservation class function. The KennelReservation function did not catch the exception, so the exception automatically was thrown up the stack to the main() function. If main() had not caught the exception, the program would have terminated.

The KennelReservation function was written to not handle the exception, but it could have been written to handle it, and then the exception would have stopped there and not have traveled up the stack to main(). As yet another alternative, the KennelReservation function could have been written to partially handle the exception but still send it on to main(). When a catch block receives an exception, it can handle it completely, partially handle it and pass it on, or just pass it on. Rethrowing an exception is the act of explicitly passing a caught exception along to a higher-level catch block. You rethrow an exception using the following statement:

    throw;

Figure 12-34 shows a rewritten operator>>() function for the KennelReservation class. The only changes from Figure 12-31 are shaded:

> The statement that accepts Dog information has been placed in a try block.
> Instead of allowing any thrown exception to just be passed up the chain, a catch block is inserted to display a message for a new rule about purchasing an insurance policy that the kennel has implemented.
> The throw; statement throws the caught RangeException object up the stack to any function that calls this one.
HANDLING EXCEPTIONS

Figure 12-35 shows an execution of the program in Figure 12-31 into which the revised KennelReservation class operator>>() function has been inserted. You can see from the output that when an exception is thrown from the Dog class, the KennelReservation class handles it by displaying the insurance message, and then main() handles it further by displaying the reminder to check with the owner about the accuracy of the age.

```
istream& operator>>(istream& in, KennelReservation& res)
    throw(RangeException)
    {
        cout << "Enter kennel # ";
        in >> res.kennelNumber;
        cout << "Enter month ";
        in >> res.month;
        cout << "Enter day ";
        in >> res.day;
        try
        {
            in >> res.dog; // call to Dog operator>>()
        }
        catch(RangeException re)
        {
            cout << endl << "Note - as of January first" << endl <<
                 "owners of dogs that are " << re.getUser() <<
                 " years old " << endl <<
                 "are required to purchase " <<
                 "a $20 insurance policy" << endl << endl;
            throw;
        }
        return in;
    }
```

Figure 12-34 A function that handles, then rethrows, an exception

Figure 12-35 shows an execution of the program in Figure 12-31 into which the revised KennelReservation class operator>>() function has been inserted. You can see from the output that when an exception is thrown from the Dog class, the KennelReservation class handles it by displaying the insurance message, and then main() handles it further by displaying the reminder to check with the owner about the accuracy of the age.

```
Enter kennel # 198
Enter month 5
Enter day 10
Enter dog's name Molly
Enter age 27

Note - as of January first
owners of dogs that are 27 years old
are required to purchase a $20 insurance policy

Age was entered as 27.
That is a very high age, but technically possible.
Please call the owner and verify the age.
Reservation for Molly Age: 27 years old for 5/10 for kennel #198
Press any key to continue . . .
```

Figure 12-35 Typical execution of the program in Figure 12-34
Handing Memory Allocation Exceptions

Recall from Chapter 9 that you can use the operator `new` to allocate new memory dynamically while a program is running. For example, a common place to allocate memory is when you want to create an array whose size is not determined until a program executes.

When you allocate memory, enough memory might not be available on your computer. If the `new` operator fails, the program ends abruptly. Because an out-of-memory condition causes a problem in any application, the creators of C++ created an out-of-memory exception handler. The `set_new_handler()` function was created to solve the universal problem of insufficient memory. To use it, you insert `#include<new>` at the top of your program file.

You use the `set_new_handler()` function by creating a function to handle the error, then passing that error-handling function's name (which is a pointer to the function) to the `set_new_handler()` function. The function you create to handle the error cannot return any values; it must be type `void`. You must call `set_new_handler()` within your program with a statement that takes the following form:

```cpp
set_new_handler(nameOfYourErrorHandlingFunction);
```

**NOTE** Notice that when you pass the error-handling function's name to the `set_new_handler()` function, you do not include the parentheses that you normally associate with a function name.

For example, Figure 12-36 shows a shaded `handleMemoryDepletion()` function that simply displays a message and exits the application. The `main()` function that calls it passes the name of the `handleMemoryDepletion()` function to the `set_new_handler()` function (in the shaded statement within `main()`). Then the `main()` function attempts to allocate 30 arrays that each hold 10 million `double`s. The loop control variable is displayed for two reasons: (1) the program takes so long to execute that displaying a count of the number of loops helps to assure you that the program is actually running, and (2) you can see how many loops execute before the computer runs out of memory for the application. Figure 12-37 shows the
output of a typical execution of the program. In the output, you see from the displayed messages that the `handleMemoryDepletion()` function is executed after the memory-allocation loop has executed 21 times.

```cpp
#include<iostream>
#include<new>
using namespace std;

void handleMemoryDepletion()
{
    cout << "Out of memory!" << endl;
    exit(1);
}

int main()
{
    int NUM = 10000000;
    set_new_handler(handleMemoryDepletion);
    const int LIMIT = 30;
    for(int x = 0; x < LIMIT; ++x)
    {
        double *a = new double[NUM];
        cout << "x is " << x << endl;
    }
    return 0;
}
```

Figure 12-36 Program that attempts to allocate a significant amount of memory

Figure 12-37 Output of program in Figure 12-36
WHEN TO USE EXCEPTION HANDLING

You are never required to create exceptions or handle them in your programs. You can always deal with most errors in your programs by making decisions. For example, to prevent division by 0, you can catch the exception that would be thrown by that operation, or you can simply write an if statement such as the following:

```cpp
if(divisor != 0)
    answer = dividend / divisor;
else
    cout << "Attempting to divide by 0!";
```

The example code illustrates a perfectly legal and reasonable method of preventing division by zero, and it represents the most efficient method of handling the error if you think it will be a frequent problem. Because a program that contains this code does not have to instantiate an exception object every time the divisor is 0, the program that uses the if statement saves time and computer memory. (Programmers say this program has little “overhead” or that it is “less expensive.”) On the other hand, if you think dividing by zero will be infrequent—that is, the exception to the rule—then the decision version will execute the if statement many times when it will turn out not to have been needed. In other words, if the divisor is 0 in only one case out of 1000, then the if statement is executed unnecessarily 999 times. In that case, it is more efficient to eliminate the if test and instantiate an exception object when needed.

Of course, it is often impossible to predict how frequently or infrequently some event will occur. As a programmer, you have to use your judgment, or the judgments of the clients who will use your programs.

---

**TWO TRUTHS AND A LIE: HANDLING MEMORY ALLOCATION EXCEPTIONS**

1. The `set_new_handler()` function is used to solve insufficient memory problems.
2. Within the `set_new_handler()` function, you place all the statements that should execute when there is insufficient memory to carry out an operation.
3. The function you pass to `set_new_handler()` must be a `void` function.

The false statement is #2. You use the `set_new_handler()` function by creating a function to handle the error, then passing that error-handling function's name to the `set_new_handler()` function. By creating a function to handle the error, you can ensure that the correct actions are taken when memory allocation fails.

---

**NOTE**

Exception handling is critical in operations that are prone to failure. For example, when you open a file, it might not exit, or when you attempt to connect to a remote server, the connection might be down.

---

**NOTE**

As with any powerful tool, if you are not careful when using exception handling, it is better not to use it at all. When a function throws an exception, the function is no longer a “black box” and no longer follows the rules of structure by having only one entry and exit point. Instead, the function acquires “extra” exit points, which means it is possible to leave files open or objects half-constructed, causing errors that are difficult to find and repair. It is always difficult to draw the line between “using” and “abusing” a powerful tool.
YOU DO IT

creating a typical data entry application

Unless you are an extremely careful typist, you have probably made the mistake of typing non-numeric data when numeric data was required by a program. In the next set of steps you will create a small, typical data entry application so that you can experiment with entering valid and invalid data.

1. First you will create a small, typical data entry program similar to many you have created throughout this book. Then you will execute with valid and invalid data to observe the results. Open a new file in your C++ editor. Type the first statements you will need in this program.

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

2. Type a data entry function that accepts a string prompt, prompts the user, gets an integer, and returns it.

```cpp
int getInt(string prompt)
{
    int integer;
    cout << prompt;
    cin >> integer;
    return integer;
}
```

3. Type a `main()` function that declares a variable for the user's age, sends a prompt to the `getInt()` function, accepts the returned value, and displays it.

```cpp
int main()
{
    int age;
    age = getInt("Enter your age ");
    cout << "Age is " << age << endl;
    return 0;
}
```
4. Save the program as SimpleDataEntry.cpp. Compile and execute the program. First, enter a valid integer when you are prompted. The output looks similar to Figure 12-38.

![Figure 12-38 Typical execution of SimpleDataEntry program](image)

5. Execute the program again, but type non-integer data, such as is shown in Figure 12-39. Depending on your compiler, the results might look different than those shown in Figure 12-39, but in any event, the output value is incorrect. If the age was used in a calculation or decision, for example to determine eligibility for an insurance policy, all the assumptions would be wrong.

![Figure 12-39 Execution of SimpleDataEntry program when user enters non-integer data](image)

MODIFYING THE SimpleDataEntry PROGRAM TO THROW AN EXCEPTION

Next, you will modify the SimpleDataEntry program so that invalid user data can be handled.

1. Open the SimpleDataEntry.cpp program if it is not still open, and immediately save it as DataEntryThrowException.cpp.

2. The cin object’s operator>>() function returns a 0 when it fails, and, as you know, C++ interprets 0 as false. Therefore, you can check the returned value in the cin statement for errors in one of two ways:

   if((cin >> integer) == 0)...

   or

   if(!(cin >> integer))...
HANDLING EXCEPTIONS

Although these two decisions always have the same outcome, the second option is a little clearer because you read it as “if not integer input successful” and there is less of a chance of someone reading your program mistakenly thinking you are comparing the value of integer to 0. You are not comparing integer to 0, you are comparing the return value of the cin operator>>() function to 0. If the cin statement has failed, you can instantiate a runtime_error and throw it.

In the getInt() function, replace the cin statement with the following if statement that, when true, declares a runtime_error object and throws it.

```cpp
if(!(cin >> integer))
{
  runtime_error e("Non-integer data");
  throw e;
}
```

3. In the main() function of the DataEntryThrowException program, surround the call to the getInt() function with a try block as follows:

```cpp
try
{
  age = getInt("Enter your age ");
}
```

4. Follow the try block with a catch block that can catch the runtime_error that might be thrown from getInt():

```cpp
catch(runtime_error e)
{
  cout << e.what() << endl;
  cout << "Age is being set to 0" << endl;
  age = 0;
}
```

5. Save the program, compile it, and run it. If you enter an integer age, the program works just as it did before. However, if you enter non-integer data as shown in Figure 12-40, the error is handled, appropriate messages are displayed, and the program ends elegantly.

![Typical execution of DataEntryThrowException program](C:\Windows\system32\cmd.exe)

**Figure 12-40** Typical execution of DataEntryThrowException program
CREATING A CUSTOM EXCEPTION CLASS

The DataEntryThrowException program handles the thrown exception, but does not provide much information about it. When you display the what() message, you describe what the user did not enter, but you might also want to know what the user did enter. In the next steps you will create and use a class that can hold that information.

1. Open the DataEntryThrowException program if it is not still open. Immediately save it as CustomException.cpp. Near the top of the file, before the definition of getInt(), define a class named NonIntegerException that extends runtime_error. The class contains a string that holds the user-entered data that causes the exception. It also has two public members: a constructor that accepts a string that represents the invalid data entered by a user, and a method that returns that string.

   ```cpp
   class NonIntegerException : public runtime_error {
      private:
         string nonIntegerData;
      public:
         NonIntegerException(string);
         string getNonIntegerData();
   };
   ```

2. Next, add the implementation of the NonIntegerException constructor. It sets the what() function return value to “Non-integer data” and it assigns the string parameter to its single data member.

   ```cpp
   NonIntegerException::NonIntegerException(string s) :
      runtime_error("Non-integer data")
   {
      nonIntegerData = s;
   }
   ```

3. The getNonIntegerData() member function returns the invalid user-entered string.

   ```cpp
   string NonIntegerException::getNonIntegerData()
   {
      return nonIntegerData;
   }
   ```
4. The following changes must be made to the `getInt()` function:
   » Add a `string` to hold the user's invalid data.
   » When the invalid data is entered, the `cin` object no longer works, so must use the `clear()` function to clear it. So you will add a class to `clear()` within the decision that determines whether the entered value is not an integer.
   » Read in the rest of the data from the input buffer and store it in the `string` declared for that purpose.
   » Replace the instantiation of the `runtime_error` with a `NonIntegerException` to which you pass the string of bad data.

The four bold statements in the code below accomplish these tasks:

```cpp
int getInt(string prompt)
{
    int integer;
    string badData;
    cout << prompt;
    if(!(cin >> integer))
    {
        cin.clear();
        cin >> badData;
        NonIntegerException e(badData);
        throw e;
    }
    return integer;
}
```

5. In the program's `main()` method, change the parameter type in the `catch` block to `NonIntegerException`.

6. Within the `catch` block, following the statement that displays the `what()` value, add the following statement that displays the value of the invalid data:

   ```cpp
   cout << "The entered value was " << e.getNonIntegerData();
   ```

7. Save the program, compile, and execute it. If you enter an integer for the age, the `getInt()` method ends normally, nothing is thrown, the `catch` block in `main()` is bypassed, and `age` is simply displayed by the last statement in the program. However, if you enter non-integer data, the execution is similar to that shown in Figure 12-41. An exception object is created, supplied with the improperly-entered data, and thrown to the `main()` function where several actions are taken—the `what()` method value is displayed, the bad data is displayed, and the user is informed that `age` is being set to 0. Then, following the `catch` block, the last output statement of the program displays `age`. 
THROWING AND CATCHING MULTIPLE EXCEPTIONS

In the next set of steps, you will create a new exception type and then write a program in which two exception types are thrown and caught.

1. Open the CustomException.cpp file if it is not still open. Immediately save it as MultipleExceptions.cpp.

2. Following the definition of and implementations for the NonIntegerException class, insert the definition for a new class named OutOfRangeException that returns the out-of-range value for an application. The class descends from runtime_error. It contains a private data member that is an integer, a constructor that accepts the integer as its parameter, and a function that returns the integer.

   ```cpp
   class OutOfRangeException : public runtime_error
   {
   private:
     int value;
   public:
     OutOfRangeException(int);
     int getValue();
   };
   ```

3. Implement the constructor and member function for the class. The constructor passes “Out of range” to runtime_error and assigns its integer parameter to its internal value field. The getValue() function returns value.

   ```cpp
   OutOfRangeException::OutOfRange Exception (int val) :
   
   runtime_error("Out of range")
   {  
     value = val;
   }

   int OutOfRangeException::getValue()
   {  
     return value;
   }
   ```

Figure 12-41 Typical execution of the CustomException program
4. Modify the `getInt()` function as follows and as shown in the shaded portions of Figure 12-42.
   » Change the function name to `getDoubleDigitInt()`.
   » Declare two constants that represent the high and low limits for a double-digit integer.
   » Currently an `if` statement determines whether the user's entry is numeric and takes appropriate action if it is not. Follow that decision with one that determines if the user's numeric entry is too high or too low. If it is either, create an `OutOfRangeException` and throw it.

   ```c++
   int getDoubleDigitInt(string prompt)
   {
       int integer;
       string badData;
       const int HIGH = 99;
       const int LOW = 10;
       cout << prompt;
       if(!(cin >> integer))
       {
           cin.clear();
           cin >> badData;
           NonIntegerException e(badData);
           throw e;
       }
       if(integer < LOW || integer > HIGH)
       {
           OutOfRangeException e(integer);
           throw e;
       }
       return integer;
   }
   ```

   **Figure 12-42** The `getDoubleDigitInt()` function

5. Next modify the existing `main()` program so it becomes one that prompts the user for an age. Assume the application requires the age to be numeric, of course, but also between 10 and 120. If the user's entry is numeric, but not a two-digit number, handle the exception by testing the value. If the value is over 99, but less than or equal to 120 make an “exception to the exception” by using the entered value. Use the shaded code in Figure 12-43 as a guide to the changes to implement in the existing `main()` function.
CHAPTER TWELVE

```c
int main()
{
    int age;
    const int MAX_AGE = 120;
    try
    {
        age = getDoubleDigitInt("Enter your age ");
    }
    catch (NonIntegerException e)
    {
        cout << e.what() << endl;
        cout << "The entered value was " << e.getNonIntegerData() << endl;
        cout << "Age is being set to 0" << endl;
        age = 0;
    }
    catch (OutOfRangeException e)
    {
        int val = e.getValue();
        cout << e.what() << endl;
        cout << "Entered value is " << e.getValue() << endl;
        if (val <= MAX_AGE)
        {
            cout << "Entered value is being used" << endl;
            age = val;
        }
        else
        {
            cout << "Age is being set to " << MAX_AGE << endl;
            age = MAX_AGE;
        }
    }
    cout << "Age is " << age << endl;
    return 0;
}
```

Figure 12-43 The main() method of the MultipleExceptions application

6. Save the program, compile and execute it. Run it several times with valid and invalid data. Figure 12-44 shows four executions—with a valid two-digit age, a valid three-digit age, and invalid three-digit age, and an invalid non-numeric age. Examine each output screen and make sure you understand what was thrown and caught in each instance.
HANDLING EXCEPTIONS

USING A GENERIC \texttt{catch} BLOCK

In the next steps you will modify the \texttt{MultipleExceptions} program by substituting the \texttt{main()} function with one that contains a single generic \texttt{catch} handler.

1. Open the \texttt{MultipleExceptions.cpp} file if it is not still open. Immediately save it as \texttt{MultipleExceptionsOneCatch.cpp}.

2. Remove the existing \texttt{main()} method and replace it with the following version that tries the \texttt{getDoubleDigitInt()} function and catches any \texttt{runtime\_error} it throws with a single \texttt{catch} block.

\begin{verbatim}
int main()
{
    int age;
    try
    {
    age = getDoubleDigitInt("Enter your age ");
    }
    catch(runtime\_error e)
    {
    cout << e.what() << endl;
    age = 0;
    }
    cout << "Age is " << age << endl;
    return 0;
}
\end{verbatim}

3. Save, compile, and execute the program. Figure 12-45 shows two executions. Whether an entered value is non-numeric, or numeric but out of range, the exception is caught by the
CHAPTER TWELVE

generic catch block. The what() method displays the correct message based on the type of exception thrown. Because you created the NonIntegerDataException and the OutOfRangeException classes to both descend from runtime_error, you can catch both exception types in a block that behaves polymorphically. The handler catches one type, but displays appropriate, specific messages based on the thrown object’s type.

Figure 12-45 Sample executions of the MultipleExceptionsOneCatch program

CHAPTER SUMMARY

» Traditional error-handling techniques often resulted in programs that were difficult to read and maintain and that were inefficient. Object-oriented programming languages provide exception handling techniques that circumvent the problems of traditional error handling. The actions you take with exceptions involve trying, throwing, and catching them.

» An exception is an object that contains information passed from the place where a problem occurs to another place that will handle the problem. In C++, an exception object can be of any data type, including a scalar or class type, and a variety of exception objects of different types can be sent from the same function, regardless of the function’s return type. In addition, you can create your own exception types that inherit from a built-in base exception class. The general principle underlying good object-oriented error handling is that any called function should check for errors, but should not be required to handle an error if one is found.

» A try block consists of one or more statements that the program attempts to execute, but that might result in thrown exceptions.

» A catch block immediately follows a try block (or another catch block) and contains statements that execute when an exception is thrown. If you want a catch block to execute, it must catch the correct type of argument thrown from a try block. If an argument is thrown and no catch block has a matching parameter type, then the program terminates. However, a catch block is not required to display, or to use in any way, what is thrown.

» A function can throw any number of exceptions, and multiple catch blocks can be written to react appropriately to each type of exception that is thrown. When you include multiple catch blocks in a program, the first catch block that can accept a thrown object is the one that will execute.
HANDLING EXCEPTIONS

» If you throw an exception and no catch block exists with an acceptable parameter type, then the program terminates. To avoid termination, you can code a default exception handler or default catch block that catches any type of object not previously caught. You create a default exception handler by creating a catch block with an ellipsis (…) as its argument. If you use a default catch block, it must be the last catch block listed after a try.

» The C++ Standard library contains a class that it uses for the exceptions its functions throw. The class is called exception and is defined in the <exception> header file. Several classes already have been derived from the built-in exception class. The category of exceptions known as logic_errors are errors in preconditions of some operation, such as passing an invalid value to a function. The category known as runtime_errors include problems that occur during program execution. When you use any descendent of the built-in exception class, you inherit a function named what() that returns an exception object’s error message.

» Although using the exception class as a basis for inheritance for your own exceptions is not required in C++, several advantages exist to doing so. One advantage is that technique is similar to other programming languages. It also provides a more uniform way to handle errors. In addition, if all exceptions derive from exception or one of its subclasses, they can be caught by a single handler. Finally, this technique allows you to use the what() function.

» You can explicitly indicate the exceptions that a function can possibly throw by writing an exception specification, which is a declaration of a function’s possible throw types. An exception specification is also called a throw list. Creating an exception specification provides documentation for later users of the function by indicating what types of errors might possibly be thrown. The user then can plan appropriate catch blocks. The American National Standards Institute (ANSI) is an organization that oversees the development of standards for products, services, processes, systems, and personnel. ANSI has developed a C++ Standard that covers exception specifications. Visual C++ departs from the ANSI standard in its implementation of exception specifications. In Visual C++, when you include an exception specification, you receive a warning that the exception specification is ignored.

» If a function does not catch a thrown exception, it is thrown to the next function higher in the call stack. This process is called unwinding the stack, because each time you call a function, the address of the place to which the logic should return at the end of the function is stored in a memory location called the stack.

» Rethrowing an exception is the act of explicitly passing a caught exception along to a higher-level catch block.

» Because an out-of-memory condition causes a problem in any application, the creators of C++ created an out-of-memory exception handler named set_new_handler(). You use the set_new_handler() function by creating a function to handle the error, then passing that error-handling function’s name (which is a pointer to the function) to the set_new_handler() function.
You are never required to create exceptions or handle them in your programs. You can always deal with most errors in your programs by making decisions. It is most efficient to use exceptions for infrequent error conditions.

**KEY TERMS**

Exceptions are errors that occur in object-oriented programs—so-named because they are not usual occurrences. An exception is an object that contains information that is passed from the place where a problem occurs to another place that will handle the problem.

Exception handling is the group of object-oriented techniques to manage exceptions.

The `exit()` function forces a program’s termination.

A message is a passed object that contains information that another object uses to determine what action to take.

To throw an exception is to send an error object to a calling function.

To catch an exception is to receive one that was thrown.

A try block consists of one or more statements that a program attempts to execute, but which might result in thrown exceptions.

A catch block contains statements that execute when an exception is thrown.

A catch handler is a catch block.

An exception handler is a catch block.

A default exception handler or default catch block catches any type of object not previously caught.

The `what()` function returns an exception object’s error message.

An exception specification is a declaration of a function’s possible throw types.

A throw list is an exception specification.

The American National Standards Institute (ANSI) is an organization that oversees the development of standards for products, services, processes, systems, and personnel.

An unexpected exception is the type thrown in ANSI C++ when a function throws an error whose type was not listed in its exception specification.

Unwinding the stack is the process of searching through a chain of called functions in reverse order of their called order, searching for a handler for an exception.

Rethrowing an exception is the act of explicitly passing a caught exception along to a higher-level catch block.

The `set_new_handler()` function handles insufficient memory problems.
REVIEW QUESTIONS

1. Errors that occur during a program’s execution are called _________.
   a. faults  c. exceptions
   b. omissions d. exclusions

2. A general principle of object-oriented error handling is that a function that might cause an exception should _________.
   a. detect an exception but not handle it
   b. handle an exception but not detect it
   c. both detect and handle exceptions
   d. not allow exceptions to occur

3. When a function sends an error message to the calling function, the function is said to ________ an exception.
   a. try
   b. throw
   c. catch
d. create

4. A throw most closely resembles _________.
   a. a return
   b. a catch
   c. an output statement
   d. a function call

5. In C++, you can throw objects that are ________ type.
   a. int and double
   b. any built-in
   c. any class
   d. any

6. Which of the following is true?
   a. A single function can throw two different data types.
   b. A single throw statement can throw two different data types.
   c. A single catch block can catch two different (and named) data types.
   d. All of the above are true.

7. Which of the following is true?
   a. You cannot have a try without a catch.
   b. You cannot have a throw without a catch.
   c. You cannot have a throw without a try.
   d. You cannot have a function without a throw.
8. A program _________ throw an exception from a function named within a try block.
   a. might                        c. cannot
   b. must                        d. must be compiled twice in order to

9. A catch block contains _________ parameter(s).
   a. no                           c. exactly two
   b. exactly one                  d. any number of

10. A catch block _________ .
    a. must not contain a parameter
    b. must use any parameter it receives
    c. can use any parameter it receives
    d. cannot use any parameter it receives

11. If an exception is thrown from a try block and no catch block exists with a matching parameter type, then the program _________ .
    a. terminates
    b. continues without problem
    c. displays a warning and continues
    d. uses an automatically-supplied default catch block

12. Which of the following is true when you try a function that throws three different types of exceptions and you want to ensure all are handled?
    a. The program will not compile if three exception types are thrown from a single function.
    b. You must provide exactly three catch blocks—one to react appropriately to each type of exception that is thrown.
    c. You must provide at least one catch block, but might provide more.
    d. You must provide one catch block with at least three parameters so the catch block can react appropriately to each type of exception that is thrown.

13. When a try block contains a function that might throw an exception, but no exception is thrown, then the program _________ .
    a. terminates
    b. issues a warning, but continues with statements beyond any catch blocks
    c. continues without incident
    d. uses a default catch block
HANDLING EXCEPTIONS

14. When you include two `catch` blocks in a program, and a thrown object can be accepted by either of them, __________.
   a. the program will not compile
   b. the first `catch` block will execute
   c. the last `catch` block will execute
   d. you cannot predict which `catch` block will execute

15. A default exception handler catches __________.
   a. any type of object not previously caught
   b. all objects
   c. the first object thrown from a function
   d. the last object thrown from a function

16. The C++ Standard library contains an `exception` class and already derived subclasses that include __________.
   a. `logic_error` and `runtime_error`
   b. `compiletime_error` and `runtime_error`
   c. `bad_exception` and `good_exception`
   d. All of the above.

17. When you use any descendent of the built-in `exception` class, you inherit a function named __________ that returns an `exception` object's error message.
   a. `message()`     c. `where()`
   b. `exceptionMessage()` d. `what()`

18. The primary reason you use an exception specification is __________.
   a. to allow multiple exceptions to be thrown
   b. as a substitute for `catch` blocks
   c. to avoid having to use the default `catch` block
   d. for documentation

19. Functions that contain no exception specification list __________.
   a. can throw string messages
   b. can throw anything
   c. might throw nothing
   d. all of the above
20. The purpose of the `set_new_handler()` function is to _________.
   a. exit a program with insufficient memory
   b. recognize out-of-memory situations, and call a function you specify
   c. allocate new memory for dynamically created arrays
   d. throw memory-exception messages to predefined functions

EXERCISES

1. Create a `Job` class that holds a `Job ID` number and the cost of the `Job`. Include insertion and extraction operators. Create a `JobException` class that holds a `Job` and an error message. When the user enters `Job` data, if the `Job` fee is below $250, then create a `JobException` object and throw it. Write a `main()` function that declares an array of eight `Job` objects. If a `JobException` object is thrown during the data entry for any `Job`, require the user to enter data for a new `Job`, and replace the invalid `Job`. Save the file as `Job.cpp`.

2. Create a class named `RealEstate` that has data members to hold the price of a house, the number of bedrooms, and the number of baths. Member functions include overloaded insertion and extraction operators. Write a `main()` function that instantiates a `RealEstate` object, allows the user to enter data, and displays the data members entered. The `main()` function should display an appropriate thrown error message if non-numeric or negative values are entered for any of the data members. Save the file as `RealEstate.cpp`.

3. Complete the following tasks:
   a. Create a class named `TelevisionException` that inherits from `runtime_error`. The class constructor should accept a string argument that is passed to the parent as the `what()` message.
   b. Create a class named `Television` that has data members to hold the model number of a television, the screen size in inches, and the price. Member functions include overloaded insertion and extraction operators. If more than four digits are entered for the model number, if the screen size is smaller than 12 or greater than 70, or if the price is negative or over $5,000, then throw a `TelevisionException` object with an appropriate message.
   c. Write a `main()` function that instantiates a `Television` object, allows the user to enter data, and displays the data members. If an exception is caught, replace all the data member values with zero values. Save the file as `Television.cpp`.
   d. Create a new application that instantiates an array of five television objects and allows the user to enter data for them. If an exception is caught, make the user reenter the data for that `Television`. Save the file as `Television2.cpp`.

4. Create an `Inventory` class with data members for stock number, quantity, and price, and overloaded data entry and output operators. The data entry operator function should throw:
   » An error message, if the stock number is negative or higher than 999
   » The quantity, if it is less than 0
   » The price, if it is over $100.00
HANDLING EXCEPTIONS

Then perform the following tasks:

a. Write a `main()` function that instantiates an array of five `Inventory` objects, and accepts data for each. Display an appropriate error message for each exception situation. When an exception is detected, replace all data fields with zeroes. At the end of the program, display data fields for all five objects. Save the file as `Inventory.cpp`.

b. Write a `main()` function that instantiates an array of five `Inventory` objects and accepts data for each. If an exception is thrown because of an invalid stock number, force the user to reenter the stock number for the `Inventory` object. If the quantity is invalid, do nothing. If the price is in error, then set the price to 99.99. (Add a `setPrice()` function to the `Inventory` class to accomplish this.) At the end of the program, display data fields for all five objects. Save the file as `Inventory2.cpp`.

c. Write a `main()` function that instantiates an array of five `Inventory` objects and accepts data for each. If an exception is thrown because of an invalid stock number, force the stock number to 999; otherwise, do nothing. (Add any needed functions to the `Inventory` class.) At the end of the program, display data fields for all five objects. Save the file as `Inventory3.cpp`.

d. Write a `main()` function that instantiates an array of five `Inventory` objects and accepts data for each. If any exception is thrown, stop accepting data. Display as many objects as were entered correctly. Save the file as `Inventory4.cpp`.

5. Complete the following tasks:

a. Create an `OutOfRangeException` that descends from the built-in `out_of_range` exception class. Include fields that hold the low and high limits of the range and the value that was out of range. Include get functions for each of the values.

b. Create a `Meal` class. Data fields include a `string` entrée name and a `double` price. Include a data entry function that prompts for and accepts values for both data fields, and that throws an `OutOfRangeException` if the price is less than $5.00 or more than $29.99. Include a public function that returns the `Meal` price so that you can use it in a calculation in the `Party` class that you will create in part d of this problem. Also include an overloaded insertion operator to display a `Meal` object's data values.

c. Create an `EntertainmentAct` class. Data fields include a `string` phone number for the contact person for the act and a `double` fee for the entertainment act. Include a data entry function that prompts for and accepts values for both data fields, and that creates and throws an `OutOfRangeException` if the price is less than $50.00 or more than $3,000. Include a public function that returns the `EntertainmentAct` price so that you can use it in a calculation in the `Party` class that you will create in part d of this problem. Also include an overloaded insertion operator to display an `EntertainmentAct` object's data values.

d. Create a `Party` class for a party-planning organization. A `Party` contains a `Meal`, an `EntertainmentAct`, an integer number of guests invited to the party, and a total cost for the party. The `Party` data entry function prompts the user for `Meal`, `EntertainmentAct`, and guest number values. The function throws an `OutOfRangeException` if the number of guests is less than 2 or more than 1000. The function also calculates the `Party` cost, based on the `Meal`'s price times the number of guests, plus the price of the `EntertainmentAct`.
e. Write a `main()` function that instantiates at least five `Party` objects and accepts data for each. When you run the program, provide data that tests that each type of exception is being recognized. The `main()` function should catch the exceptions and display an appropriate error message about each including what the acceptable range was and what the entered value was. If an exception is caught, the data for the entire `Party` object should be entered again. Save the file as `Party.cpp`.

6. Complete the following tasks:
   a. Create a `DateException` that descends from the built-in `runtime_error` exception class. Include fields that hold a month, day, and year and include get functions for each of the values.
   b. Create a `Date` class. Data fields include month, day, and year. Include an exception function that prompts for and accepts values for each of the data fields, and that throws a `DateException` if the month is less than 1 or more than 12 and if the day is out of range for the entered month. (Assume February can have 29 days.) Allow any value to be accepted for the year. Also include an overloaded insertion operator to display a `Date`'s data values.
   c. Write a `main()` function that instantiates at least five `Date` objects and accepts data for each. When you run the program, provide data that tests that each type of exception is being recognized. The `main()` function should catch the exceptions and display an appropriate error message about each including what the entered value was. If an exception is caught, the data for the `Date` object should be set to the default date 1/1/2010. Save the file as `DateExceptions.cpp`.

7. Complete the following tasks:
   a. Create a class named `Teacher` that holds a `Teacher`'s first and last names and the grade level the `Teacher` teaches. Include a constructor function that uses default first and last names, such as “ZZZZZZZZ” and a default grade level, such as 0, when no arguments are provided.
   b. Create a class named `Student` that holds a `Student`'s first and last names. Include a constructor function that uses default first and last names, such as “ZZZZZZZZ,” when no names are provided.
   c. Create a class named `Classroom`. A `Classroom` holds a `Teacher` and an array of 35 `Student` objects.
   d. Create a class named `School`. A `School` contains an array of 100 `Classroom` objects. (This class is being designed to be a very large class, to more quickly demonstrate running out of memory.)
   e. In a `main()` demonstration function, declare a pointer to an array of 500 `School` objects and include code to handle any memory allocation errors that occur. In a loop, declare 500 array objects at a time, continuing to rewrite the `main()` function if necessary until you create enough `School` objects to cause a memory allocation exception. (Hint: You might want to display a count periodically to assure yourself that the `School` objects are being created.) When memory is exhausted, display an appropriate message and terminate the application. Save the file as `School.cpp`.
HANDLING EXCEPTIONS

8. Complete the following tasks:
   a. Create a NonNumericException class that inherits from runtime_error. The class contains a string that holds any non-numeric data entered and uses its parent’s what() function to display an appropriate error message.
   
   b. Create a class that holds customer orders. Include an order number, quantity ordered, price each, and total which will be calculated as quantity times price. Also include overloaded insertion and extraction operators. Overload the extraction operator so that the following occur:
      » If any numeric data is not entered as a number, throw a NonNumericException.
      » If the entered order number is numeric but more than four digits, throw an appropriate message.
      » If the entered order quantity is over 50, throw the quantity.
      » If the entered price each is more than $39.95, throw the price.
      » If the total (quantity times price) is more than $1000.00, throw the Order object.
   
   c. Write a program that creates an array of five orders. Accept data for each one from the user. Catch and handle the errors as follows:
      » If the order number was numeric but more than four digits, display a message and force the user to renter the number.
      » If the order total was more than the $1000 limit, accept the Order but display a message asking the order taker to check the customer’s credit limit.
      » If any other exception occurs, display an appropriate message and set all the order values to zeroes.
      » If no exceptions occur, accept the Order.
      » After five Orders are accepted, display them.

Save the program as OrderExceptions.cpp.

9. Complete the following tasks:
   a. In Chapter 5, you created a game named GuessAWord that is similar to Hangman and in which a player guesses letters to try to replicate a hidden word. At the end of the game, a count is displayed indicating the number of entries that were required for the player to guess the word. Now modify the program to throw and catch an exception when the user enters a guess that is not a letter of the alphabet. Create a NonLetterException class that descends from a built-in exception class that can store and retrieve the non-alphabetic entered character. Modify the GuessAWord program so that when a NonLetterException is thrown and caught, an appropriate message is displayed for the user informing the user that the character entered was not a letter and displaying the entered character. Save the file as GuessAWord2.cpp.
   
   b. Modify the GuessAWord2 game so that when the player enters a non-alphabetic guess, not only is the player notified of the mistake, but also do not have the guess count as part of the final total of number of guesses made. Save the file as GuessAWord3.cpp.
CHAPTER TWELVE

10. Each of the following files in the Chapter12 folder of the data files provided with your book contains syntax and/or logical errors. Determine the problem in each case, and fix the program. Save your solutions by adding “Fixed” to the filename, as in DEBUG12-1Fixed.cpp.
   a. DEBUG12-1.cpp
   b. DEBUG12-2.cpp
   c. DEBUG12-3.cpp
   d. DEBUG12-4.cpp

CASE PROJECT 1

You have been developing a Fraction class for Teacher’s Pet Software. Each Fraction contains a numerator, denominator, a whole number portion, and has access to several functions you have developed, including overloaded operators. Complete these tasks:

a. Create a FractionException class. The class contains a Fraction object and a string message explaining the reason for the exception. Include a constructor that requires values for the Fraction and the message. Also include a function that displays the message.
b. Modify each Fraction class constructor and data entry function so that it throws a FractionException whenever a client attempts to instantiate a Fraction with a zero denominator.
c. Write a main() function that asks the user to enter values for four Fractions. If the user attempts to create a Fraction with a zero denominator, catch the exception, display a message, and force the Fraction to 0 / 1. Display the four Fraction objects. Save the file as FractionException1.cpp.
d. Write a main() function that asks the user to enter values for four Fractions. If the user attempts to create a Fraction with a zero denominator, catch the exception, display a message, and force the user to reenter the Fraction values. Display the four Fraction objects. Save the file as FractionException2.cpp.
e. Write a main() function that asks the user to enter values for four Fractions. If the user attempts to create a Fraction with a zero denominator, catch the exception, display a message, and terminate the program. Save the file as FractionException3.cpp.

CASE PROJECT 2

You have been developing a BankAccount class for Parkville Bank that contains several fields and functions, including overloaded operators. You have also created child classes derived from the BankAccount class: CheckingAccount, SavingsAccount, and CheckingWithInterest. Complete these tasks:

a. Create a BankAccountException class. The class contains a BankAccount object and a string message explaining the reason for the exception. Include a constructor that requires values for the BankAccount and the message. Also include a function that displays the message.
b. Modify each `BankAccount` class constructor and data entry function so that it throws an exception whenever a client attempts to instantiate a `BankAccount` with an account number that is less than 1000 or greater than 9999.

c. Write a `main()` function that asks the user to enter values for four `BankAccount`s. If the user attempts to create a `BankAccount` with an invalid account number, catch the exception, display a message, and force the account number and balance to 0. Display the four `BankAccount` objects. Save the file as `BankException1.cpp`.

d. Write a `main()` function that declares an array of four `CheckingAccount`s. (The `CheckingAccount` constructor calls the `BankAccount` data entry function, so an exception might be thrown.) If the user attempts to create a `CheckingAccount` with an invalid account number, catch the exception, and, before ending the program, display two messages. The first message is the string message contained in the `BankAccountException` object. The second indicates which `CheckingAccount` (1, 2, 3, or 4) caused the early termination. If all four `CheckingAccount`s are valid, then display them. Save the file as `BankException2.cpp`.

e. Alter the `SavingsAccount` constructor to make sure an exception is thrown when the account number is invalid (based on the same rules for a `BankAccount` account number) and also when the interest rate is negative. (Make sure the messages are different.) Write a `main()` function that declares an array of four `SavingsAccount`s. If the user attempts to create a `SavingsAccount` with an invalid account number or a negative interest rate, catch the exception, display the appropriate message, and display which account caused the early termination (1, 2, 3, or 4). If all four `SavingsAccount`s are valid, then display them. Save the file as `BankException3.cpp`.

UP FOR DISCUSSION

1. Have you ever been victimized by a computer error? For example, were you ever incorrectly denied credit, billed for something you did not purchase, or assigned an incorrect grade in a course? How did you resolve the problem? On the Web, find the most outrageous story you can involving a computer error.

2. Think of a computer error you have read about or that has happened to you. (You might want to select an incident from your response to Question #1.) Describe how exception handling could have prevented this error, or how exception handling would not have been useful in this case.

3. Suppose you have learned a lot about programming from your employer. Is it ethical for you to use this knowledge to start your own home-based programming business on the side? Does it matter whether you are in competition for the same clients as your employer? Does it matter whether you use just your programming expertise or whether you also use information about clients’ preferences and needs gathered from your regular job?