24 Exception Handling: A Deeper Look

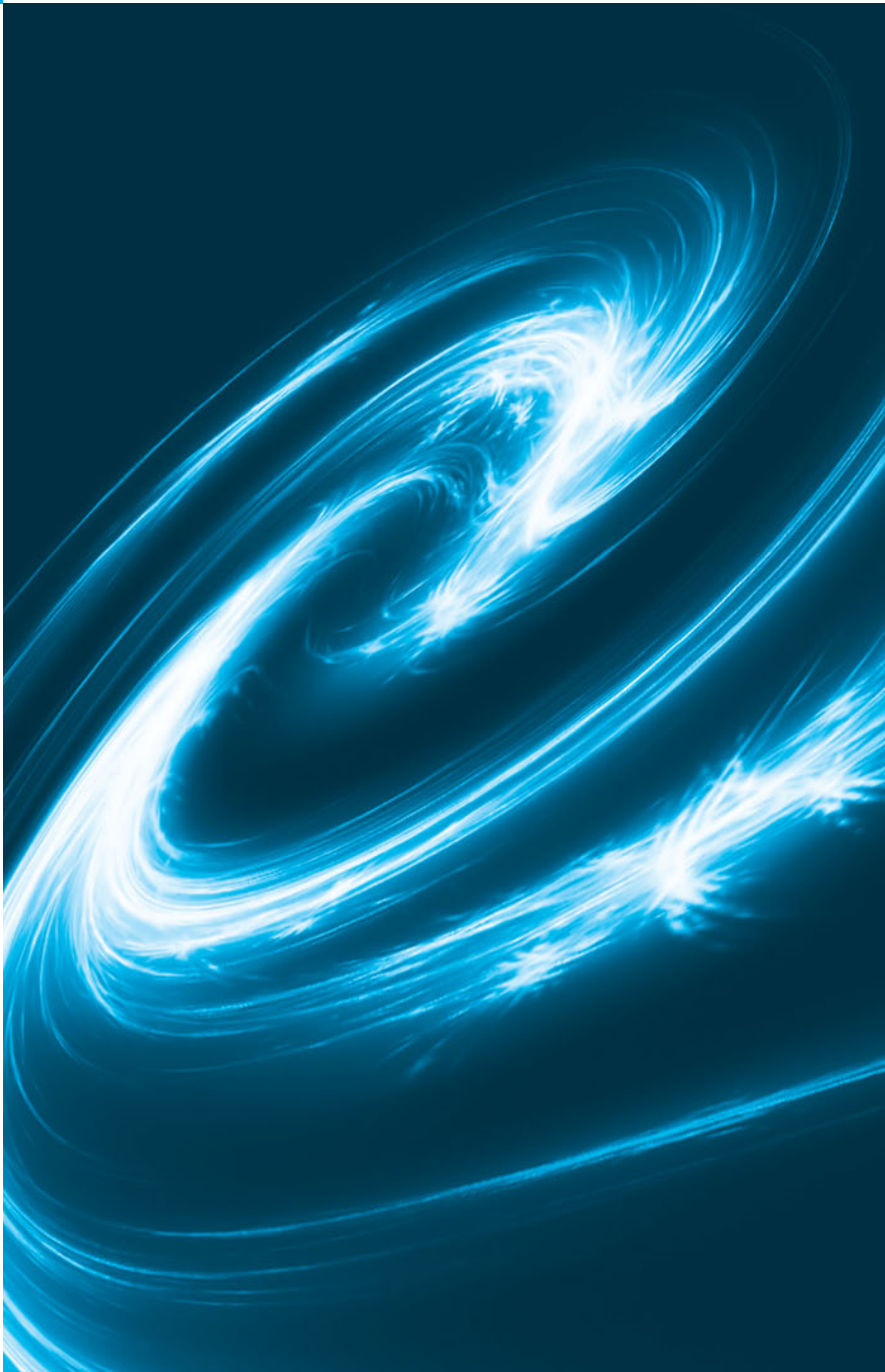
It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.
—Franklin Delano Roosevelt

If they're running and they don't look where they're going I have to come out from somewhere and catch them.
—Jerome David Salinger

Objectives

In this chapter you'll learn:

- To use `try`, `catch` and `throw` to detect, handle and indicate exceptions, respectively.
- To process uncaught and unexpected exceptions.
- To declare new exception classes.
- How stack unwinding enables exceptions not caught in one scope to be caught in another.
- To handle `new` failures.
- To use `unique_ptr` to prevent memory leaks.
- To understand the standard exception hierarchy.



Outline

24.1 Introduction

As you know, an **exception** is an indication of a problem that occurs during a program's execution. **Exception handling** enables you to create applications that can resolve (or handle) exceptions. In many cases, handling an exception allows a program to continue executing as if no problem had been encountered. The features presented in this chapter enable you to write **robust** and **fault-tolerant programs** that can deal with problems continue executing or terminate gracefully.

We begin with a review of exception-handling concepts via an example that demonstrates handling an exception that occurs when a function attempts to divide by zero. We show how to handle exceptions that occur in a constructor or destructor and exceptions that occur if operator `new` fails to allocate memory for an object. We introduce several C++ Standard Library exception handling classes.

Software Engineering Observation 24.1

Exception handling provides a standard mechanism for processing errors. This is especially important when working on a project with a large team of programmers.

Software Engineering Observation 24.2

Incorporate your exception-handling strategy into your system from inception. Including effective exception handling after a system has been implemented can be difficult.

24.2 Example: Handling an Attempt to Divide by Zero

Let's consider a simple example of exception handling (Figs. 24.1–24.2). We show how to deal with a common arithmetic problem—division by zero. In C++, *division by zero* using integer arithmetic typically causes a program to terminate prematurely. In floating-point arithmetic, some C++ implementations allow division by zero, in which case a result of positive or negative infinity is displayed as `INF` or `-INF`, respectively.

In this example, we define a function named `quotient` that receives two integers input by the user and divides its first `int` parameter by its second `int` parameter. Before performing the division, the function casts the first `int` parameter's value to type `double`. Then, the second `int` parameter's value is (implicitly) promoted to type `double` for the calculation. So function `quotient` actually performs the division using two `double` values and returns a `double` result.

Although division by zero is often allowed in floating-point arithmetic, for the purpose of this example we treat any attempt to divide by zero as an error. Thus, function `quotient` tests its second parameter to ensure that it isn't zero before allowing the division to proceed. If the second parameter is zero, the function throws an exception to indicate to the caller that a problem occurred. The caller (`main` in this example) can then process the exception and allow the user to type two new values before calling function `quotient` again. In this way, the program can continue executing even after an improper value is entered, thus making the program more robust.

The example consists of two files. `DivideByZeroException.h` (Fig. 24.1) defines an *exception class* that represents the type of the problem that might occur in the example, and `fig24_02.cpp` (Fig. 24.2) defines the `quotient` function and the `main` function that calls it. Function `main` contains the code that demonstrates exception handling.

Defining an Exception Class to Represent the Type of Problem That Might Occur

Figure 24.1 defines class `DivideByZeroException` as a derived class of Standard Library class **`runtime_error`** (defined in header **`<stdexcept>`**). Class `runtime_error`—a derived class of Standard Library class **`exception`** (defined in header **`<exception>`**)—is the C++ standard base class for representing runtime errors. Class `exception` is the standard C++ base class for all exceptions. (Section 24.11 discusses class `exception` and its derived classes in detail.) A typical exception class that derives from the `runtime_error` class defines only a constructor (e.g., lines 12–13) that passes an error-message string to the base-class `runtime_error` constructor. Every exception class that derives directly or indirectly from `exception` contains the `virtual` function **`what`**, which returns an exception object's error message. You're not required to derive a custom exception class, such as `DivideByZeroException`, from the standard exception classes provided by C++. However, doing so allows you to use the `virtual` function `what` to obtain an appropriate error message. We use an object of this `DivideByZeroException` class in Fig. 24.2 to indicate when an attempt is made to divide by zero.

```
 1  // Fig. 24.1: DivideByZeroException.h
 2  // Class DivideByZeroException definition.
 3  #include <stdexcept> // stdexcept header contains runtime_error
 4  using namespace std;
 5
 6  // DivideByZeroException objects should be thrown by functions
 7  // upon detecting division-by-zero exceptions
 8  class DivideByZeroException : public runtime_error
 9  {
10  public:
11     // constructor specifies default error message
12     DivideByZeroException()
13        : runtime_error( "attempted to divide by zero" ) {}
14  }; // end class DivideByZeroException
```

Fig. 24.1 | Class `DivideByZeroException` definition.

Demonstrating Exception Handling

Figure 24.2 uses exception handling to wrap code that might throw a "divide-by-zero" exception and to handle that exception, should one occur. The user enters two integers, which are passed as arguments to function `quotient` (lines 10–18). This function divides its first parameter (numerator) by its second parameter (denominator). Assuming that the user does not specify 0 as the denominator for the division, function quotient returns the division result. If the user inputs 0 for the denominator, quotient throws an exception. In the sample output, the first two lines show a successful calculation, and the next two show a failure due to an attempt to divide by zero. When the exception occurs, the program informs the user of the mistake and prompts the user to input two new integers. After we discuss the code, we'll consider the user inputs and flow of program control that yield these outputs.

```cpp
// Fig. 24.2: fig24_02.cpp
// A simple exception-handling example that checks for
// divide-by-zero exceptions.
#include <iostream>
#include "DivideByZeroException.h" // DivideByZeroException class
using namespace std;

// perform division and throw DivideByZeroException object if
// divide-by-zero exception occurs
double quotient( int numerator, int denominator )
{
   // throw DivideByZeroException if trying to divide by zero
   if ( denominator == 0 )
      throw DivideByZeroException(); // terminate function

   // return division result
   return static_cast< double >( numerator ) / denominator;
} // end function quotient

int main()
{
   int number1; // user-specified numerator
   int number2; // user-specified denominator
   double result; // result of division

   cout << "Enter two integers (end-of-file to end): ";

   // enable user to enter two integers to divide
   while ( cin >> number1 >> number2 )
   {
      // try block contains code that might throw exception
      // and code that will not execute if an exception occurs
      try
      {
         result = quotient( number1, number2 );
         cout << "The quotient is: " << result << endl;
      } // end try
      catch ( DivideByZeroException &divideByZeroException )
      {
         cout << "Exception occurred: "
            << divideByZeroException.what() << endl;
      } // end catch
```

Fig. 24.2 | Exception-handling example that throws exceptions on attempts to divide by zero. (Part 1 of 2.)

```
43
44         cout << "\nEnter two integers (end-of-file to end): ";
45      } // end while
46
47      cout << endl;
48   } // end main
```

```
Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): ^Z
```

Fig. 24.2 | Exception-handling example that throws exceptions on attempts to divide by zero. (Part 2 of 2.)

Enclosing Code in a `try` Block

The program begins by prompting the user to enter two integers. The integers are input in the condition of the `while` loop (line 29). Line 35 passes the values to function `quotient` (lines 10–18), which either divides the integers and returns a result, or **throws an exception** (i.e., indicates that an error occurred) on an attempt to divide by zero. Exception handling is geared to situations in which the function that detects an error is unable to handle it.

A `try` block encloses statements that might cause exceptions and statements that should be skipped if an exception occurs. The `try` block in lines 33–37 encloses the invocation of function `quotient` and the statement that displays the division result. In this example, because the invocation of function `quotient` (line 35) can *throw* an exception, we enclose this function invocation in a `try` block. Enclosing the output statement (line 36) in the `try` block ensures that the output will occur only if function `quotient` returns a result.

Software Engineering Observation 24.3

Exceptions may surface through explicitly mentioned code in a `try` block, through calls to other functions and through deeply nested function calls initiated by code in a `try` block.

Defining a `catch` Handler to Process a `DivideByZeroException`

Exceptions are processed by `catch` handlers. At least one `catch` handler (lines 38–42) must immediately follow each `try` block. The exception parameter is declared as a *reference* to the type of exception the `catch` handler can process (`DivideByZeroException` in this case). When an exception occurs in a `try` block, the `catch` handler that executes is the one whose type matches the type of the exception that occurred (i.e., the type in the `catch` block matches the thrown exception type exactly or is a base class of it). If an exception parameter includes an optional parameter name, the `catch` handler can use that parameter name to interact with the caught exception in the body of the `catch` handler, which is delimited by braces (`{` and `}`). A `catch` handler typically reports the error to the user, logs it to a file, terminates the program gracefully or tries an alternate strategy to accomplish the failed task. In this example, the catch handler simply reports that the user attempted to divide by zero. Then the program prompts the user to enter two new integer values.

Common Programming Error 24.1

It's a syntax error to place code between a try *block and its corresponding* catch *handlers or between its* catch *handlers.*

Common Programming Error 24.2

Each catch *handler can have only a single parameter—specifying a comma-separated list of exception parameters is a syntax error.*

Common Programming Error 24.3

It's a logic error to catch the same type in two different catch *handlers following a single* try *block.*

Termination Model of Exception Handling

If an exception occurs as the result of a statement in a try block, the try block expires (i.e., terminates immediately). Next, the program searches for the first catch handler that can process the type of exception that occurred. The program locates the matching catch by comparing the thrown exception's type to each catch's exception-parameter type until the program finds a match. A match occurs if the types are *identical* or if the thrown exception's type is a *derived class* of the exception-parameter type. When a match occurs, the code contained in the matching catch handler executes. When a catch handler finishes processing by reaching its closing right brace (}), the exception is considered handled and the local variables defined within the catch handler (including the catch parameter) go out of scope. Program control does *not* return to the point at which the exception occurred (known as the **throw point**), because the try block has *expired*. Rather, control resumes with the first statement (line 44) after the last catch handler following the try block. This is known as the **termination model of exception handling**. Some languages use the **resumption model of exception handling**, in which, after an exception is handled, control resumes just after the throw point. As with any other block of code, when a try block terminates, local variables defined in the block go out of scope.

Common Programming Error 24.4

Logic errors can occur if you assume that after an exception is handled, control will return to the first statement after the throw point.

Error-Prevention Tip 24.1

With exception handling, a program can continue executing (rather than terminating) after dealing with a problem. This helps ensure the kind of robust applications that contribute to what's called mission-critical computing or business-critical computing.

If the try block completes its execution successfully (i.e., no exceptions occur in the try block), then the program ignores the catch handlers and program control continues with the first statement after the last catch following that try block.

If an exception that occurs in a try block has no matching catch handler, or if an exception occurs in a statement that is not in a try block, the function that contains the statement terminates immediately, and the program attempts to locate an enclosing `try` block in the calling function. This process is called **stack unwinding** and is discussed in Section 24.6.

Flow of Program Control When the User Enters a Nonzero Denominator

Consider the flow of control when the user inputs the numerator `100` and the denominator `7`. In line 13, function `quotient` determines that the `denominator` does not equal zero, so line 17 performs the division and returns the result (`14.2857`) to line 35 as a `double`. Program control then continues sequentially from line 35, so line 36 displays the division result—line 37 ends the `try` block. Because the `try` block completed successfully and did not throw an exception, the program does not execute the statements contained in the `catch` handler (lines 38–42), and control continues to line 44 (the first line of code after the `catch` handler), which prompts the user to enter two more integers.

Flow of Program Control When the User Enters a Denominator of Zero

Now consider the case in which the user inputs the numerator `100` and the denominator `0`. In line 13, `quotient` determines that the `denominator` equals zero, which indicates an attempt to divide by zero. Line 14 throws an exception, which we represent as an object of class `DivideByZeroException` (Fig. 24.1).

To throw an exception, line 14 uses keyword **`throw`** followed by an operand that represents the type of exception to throw. Normally, a `throw` statement specifies one operand. (In Section 24.4, we discuss how to use a `throw` statement with no operand.) The operand of a `throw` can be of any type. If the operand is an object, we call it an **exception object**—in this example, the exception object is an object of type `DivideByZeroException`. However, a `throw` operand also can assume other values, such as the value of an expression that does not result in an object of a class (e.g., `throw x > 5`) or the value of an `int` (e.g., `throw 5`). The examples in this chapter focus exclusively on throwing objects of exception classes.

Common Programming Error 24.5

Use caution when `throw`ing the result of a conditional expression (`?:`)—promotion rules could cause the value to be of a type different from the one expected. For example, when throwing an `int` or a `double` from the same conditional expression, the `int` is promoted to a `double`. So, a `catch` handler that catches an `int` would never execute based on such a conditional expression.

As part of throwing an exception, the `throw` operand is created and used to initialize the parameter in the `catch` handler, which we discuss momentarily. The `throw` statement in line 14 creates a `DivideByZeroException` object. When line 14 throws the exception, function `quotient` exits immediately. So, line 14 throws the exception *before* function `quotient` can perform the division in line 17. This is a central characteristic of exception handling: *A function should throw an exception before the error has an opportunity to occur.*

Because we enclosed the call to `quotient` (line 35) in a `try` block, program control enters the `catch` handler (lines 38–42) that immediately follows the `try` block. This `catch` handler serves as the exception handler for the divide-by-zero exception. In general, when an exception is thrown within a `try` block, the exception is caught by a `catch` handler that specifies the type matching the thrown exception. In this program, the `catch` handler specifies that it catches `DivideByZeroException` objects—this type matches the object type thrown in function `quotient`. Actually, the `catch` handler catches a *reference* to the `DivideByZeroException` object created by function `quotient`'s `throw` statement (line 14), so that the `catch` handler does not make a copy of the exception object.

The `catch`'s body (lines 40–41) prints the error message returned by function `what` of base-class `runtime_error`—i.e., the string that the `DivideByZeroException` constructor (lines 12–13 in Fig. 24.1) passed to the `runtime_error` base-class constructor.

Performance Tip 24.1

Catching an exception object by reference eliminates the overhead of copying the object that represents the thrown exception.

Good Programming Practice 24.1

Associating each type of runtime error with an appropriately named exception object improves program clarity.

24.3 When to Use Exception Handling

Exception handling is designed to process synchronous errors, which occur when a statement executes, such as *out-of-range array subscripts*, *arithmetic overflow* (i.e., a value outside the representable range of values), *division by zero*, *invalid function parameters* and *unsuccessful memory allocation* (due to lack of memory). Exception handling is not designed to process errors associated with asynchronous events (e.g., disk I/O completions, network message arrivals, mouse clicks and keystrokes), which occur in parallel with, and independent of, the program's flow of control.

Software Engineering Observation 24.4

Exception handling provides a single, uniform technique for processing problems. This helps programmers on large projects understand each other's error-processing code.

Software Engineering Observation 24.5

Avoid using exception handling as an alternate form of flow of control. These "additional" exceptions can "get in the way" of genuine error-type exceptions.

Software Engineering Observation 24.6

Exception handling enables predefined software components to communicate problems to application-specific components, which can then process the problems in an application-specific manner.

Exception handling also is useful for processing problems that occur when a program interacts with software elements, such as member functions, constructors, destructors and classes. Such software elements often use exceptions to notify programs when problems occur. This enables you to implement *customized error handling* for each application.

Performance Tip 24.2

When no exceptions occur, exception-handling code incurs little or no performance penalty. Thus, programs that implement exception handling operate more efficiently than do programs that intermix error-handling code with program logic.

Software Engineering Observation 24.7

Functions with common error conditions should return `0` *or* `NULL` *(or other appropriate values, such as* `bool`*s) rather than throw exceptions. A program calling such a function can check the return value to determine success or failure of the function call.*

Complex applications normally consist of predefined software components and application-specific components that use the predefined components. When a predefined component encounters a problem, that component needs a mechanism to communicate the problem to the application-specific component—the *predefined component cannot know in advance how each application processes a problem that occurs.*

24.4 Rethrowing an Exception

It's possible that an exception handler, upon receiving an exception, might decide either that it cannot process that exception or that it can process the exception only partially. In such cases, the exception handler can *defer the exception handling (or perhaps a portion of it) to another exception handler.* In either case, you achieve this by **rethrowing the exception** via the statement

```
throw;
```

Regardless of whether a handler can process an exception, the handler can *rethrow* the exception for further processing outside the handler. The next enclosing `try` block detects the rethrown exception, which a `catch` handler listed after that enclosing `try` block attempts to handle.

Common Programming Error 24.6

Executing an empty `throw` *statement outside a* `catch` *handler calls function* **`terminate`***, which abandons exception processing and terminates the program immediately.*

The program of Fig. 24.3 demonstrates rethrowing an exception. In `main`'s `try` block (lines 29–34), line 32 calls function `throwException` (lines 8–24). The `throwException` function also contains a `try` block (lines 11–15), from which the `throw` statement in line 14 `throw`s an instance of standard-library-class `exception`. Function `throwException`'s `catch` handler (lines 16–21) catches this exception, prints an error message (lines 18–19) and rethrows the exception (line 20). This terminates function `throwException` and returns control to line 32 in the `try...catch` block in `main`. The `try` block terminates (so line 33 does not execute), and the `catch` handler in `main` (lines 35–38) catches this exception and prints an error message (line 37). Since we do not use the exception parameters in the `catch` handlers of this example, we omit the exception parameter names and specify only the type of exception to catch (lines 16 and 35).

```
1  // Fig. 24.3: fig24_03.cpp
2  // Rethrowing an exception.
3  #include <iostream>
4  #include <exception>
5  using namespace std;
```

Fig. 24.3 | Rethrowing an exception. (Part 1 of 2.)

```

// throw, catch and rethrow exception
void throwException()
{
   // throw exception and catch it immediately
   try
   {
      cout << "  Function throwException throws an exception\n";
      throw exception(); // generate exception
   } // end try
   catch ( exception & ) // handle exception
   {
      cout << "  Exception handled in function throwException"
         << "\n  Function throwException rethrows exception";
      throw; // rethrow exception for further processing
   } // end catch

   cout << "This also should not print\n";
} // end function throwException

int main()
{
   // throw exception
   try
   {
      cout << "\nmain invokes function throwException\n";
      throwException();
      cout << "This should not print\n";
   } // end try
   catch ( exception & ) // handle exception
   {
      cout << "\n\nException handled in main\n";
   } // end catch

   cout << "Program control continues after catch in main\n";
} // end main
```

```
main invokes function throwException
  Function throwException throws an exception
  Exception handled in function throwException
  Function throwException rethrows exception

Exception handled in main
Program control continues after catch in main
```

Fig. 24.3 | Rethrowing an exception. (Part 2 of 2.)

24.5 Processing Unexpected Exceptions

Function `unexpected` calls the function registered with function **`set_unexpected`** (defined in header `<exception>`). If no function has been registered in this manner, function `terminate` is called by default. Cases in which function `terminate` is called include:

1. the exception mechanism cannot find a matching `catch` for a thrown exception
2. a destructor attempts to `throw` an exception during stack unwinding

3. an attempt is made to rethrow an exception when there's no exception currently being handled
4. a call to function `unexpected` defaults to calling function `terminate`

(Section 15.5.1 of the C++ Standard Document discusses several additional cases.) Function **`set_terminate`** can specify the function to invoke when `terminate` is called. Otherwise, `terminate` calls **`abort`**, *which terminates the program without calling the destructors of any remaining objects of automatic or static storage class.* This could lead to resource leaks when a program terminates prematurely.

Common Programming Error 24.7

Aborting a program component due to an uncaught exception could leave a resource—such as a file stream or an I/O device—in a state in which other programs are unable to acquire the resource. This is known as a resource leak.

Function `set_terminate` and function `set_unexpected` each return a pointer to the last function called by `terminate` and `unexpected`, respectively (0, the first time each is called). This enables you to save the function pointer so it can be restored later. Functions `set_terminate` and `set_unexpected` take as arguments pointers to functions with `void` return types and no arguments.

If the last action of a programmer-defined termination function is not to exit a program, function `abort` will be called to end program execution.

24.6 Stack Unwinding

When an exception is thrown but not caught in a particular scope, the function call stack is "unwound," and an attempt is made to `catch` the exception in the next outer `try...catch` block. Unwinding the function call stack means that the function in which the exception was not caught terminates, all local variables in that function are destroyed and control returns to the statement that originally invoked that function. If a `try` block encloses that statement, an attempt is made to `catch` the exception. If a `try` block does not enclose that statement, stack unwinding occurs again. If no `catch` handler ever catches this exception, function `terminate` is called to terminate the program. The program of Fig. 24.4 demonstrates stack unwinding.

```
// Fig. 24.4: fig24_04.cpp
// Stack unwinding.
#include <iostream>
#include <stdexcept>
using namespace std;

// function3 throws runtime error
void function3() throw ( runtime_error )
{
   cout << "In function 3" << endl;

```

Fig. 24.4 | Stack unwinding. (Part 1 of 2.)

```
12     // no try block, stack unwinding occurs, return control to function2
13     throw runtime_error( "runtime_error in function3" ); // no print
14  } // end function3
15
16  // function2 invokes function3
17  void function2() throw ( runtime_error )
18  {
19     cout << "function3 is called inside function2" << endl;
20     function3(); // stack unwinding occurs, return control to function1
21  } // end function2
22
23  // function1 invokes function2
24  void function1() throw ( runtime_error )
25  {
26     cout << "function2 is called inside function1" << endl;
27     function2(); // stack unwinding occurs, return control to main
28  } // end function1
29
30  // demonstrate stack unwinding
31  int main()
32  {
33     // invoke function1
34     try
35     {
36        cout << "function1 is called inside main" << endl;
37        function1(); // call function1 which throws runtime_error
38     } // end try
39     catch ( runtime_error &error ) // handle runtime error
40     {
41        cout << "Exception occurred: " << error.what() << endl;
42        cout << "Exception handled in main" << endl;
43     } // end catch
44  } // end main
```

```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
Exception occurred: runtime_error in function3
Exception handled in main
```

Fig. 24.4 | Stack unwinding. (Part 2 of 2.)

In main, the try block (lines 34–38) calls function1 (lines 24–28). Next, function1 calls function2 (lines 17–21), which in turn calls function3 (lines 8–14). Line 13 of function3 throws a runtime_error object. However, because no try block encloses the throw statement in line 13, stack unwinding occurs—function3 terminates at line 13, then returns control to the statement in function2 that invoked function3 (i.e., line 20). Because no try block encloses line 20, stack unwinding occurs again—function2 terminates at line 20 and returns control to the statement in function1 that invoked function2 (i.e., line 27). Because no try block encloses line 27, stack unwinding occurs one more time—function1 terminates at line 27 and returns control to the statement in main that invoked `function1` (i.e., line 37). The `try` block of lines 34–38 encloses this statement, so the first matching `catch` handler located after this `try` block (line 39–43) catches and processes the exception. Line 41 uses function `what` to display the exception message. Recall that function `what` is a `virtual` function of class `exception` that can be overridden by a derived class to return an appropriate error message.

24.7 Constructors, Destructors and Exception Handling

First, let's discuss an issue that we've mentioned but not yet resolved satisfactorily: What happens when an error is detected in a *constructor*? For example, how should an object's constructor respond when `new` fails because it was unable to allocate required memory for storing that object's internal representation? Because the constructor cannot return a value to indicate an error, we must choose an alternative means of indicating that the object has not been constructed properly. One scheme is to return the improperly constructed object and hope that anyone using it would make appropriate tests to determine that it's in an inconsistent state. Another scheme is to set some variable outside the constructor. The preferred alternative is to require the constructor to `throw` an exception that contains the error information, thus offering an opportunity for the program to handle the failure.

Before an exception is thrown by a constructor, destructors are called for any member objects built as part of the object being constructed. Destructors are called for every automatic object constructed in a `try` block before an exception is thrown. Stack unwinding is guaranteed to have been completed at the point that an exception handler begins executing. If a destructor invoked as a result of stack unwinding throws an exception, `terminate` is called.

If an object has member objects, and if an exception is thrown before the outer object is fully constructed, then destructors will be executed for the member objects that have been constructed prior to the occurrence of the exception. If an array of objects has been partially constructed when an exception occurs, only the destructors for the constructed objects in the array will be called.

An exception could preclude the operation of code that would normally *release a resource* (such as memory or a file), thus causing a *resource leak*. One technique to resolve this problem is to initialize a local object to acquire the resource. When an exception occurs, the destructor for that object will be invoked and can free the resource.

Error-Prevention Tip 24.2

When an exception is thrown from the constructor for an object that's created in a `new` *expression, the dynamically allocated memory for that object is released.*

24.8 Exceptions and Inheritance

Various exception classes can be derived from a common base class, as we discussed in Section 24.2, when we created class `DivideByZeroException` as a derived class of class `exception`. If a `catch` handler catches a pointer or reference to an exception object of a base-class type, it also can `catch` a pointer or reference to all objects of classes publicly derived from that base class—this allows for polymorphic processing of related errors.

Error-Prevention Tip 24.3

Using inheritance with exceptions enables an exception handler to `catch` *related errors with concise notation. One approach is to* `catch` *each type of pointer or reference to a derived-class exception object individually, but a more concise approach is to* `catch` *pointers or references to base-class exception objects instead. Also, catching pointers or references to derived-class exception objects individually is error prone, especially if you forget to test explicitly for one or more of the derived-class pointer or reference types.*

24.9 Processing new Failures

The C++ standard specifies that, when operator `new` fails, it `throw`s a **`bad_alloc`** exception (defined in header `<new>`). In this section, we present two examples of `new` failing. The first uses the version of `new` that `throw`s a `bad_alloc` exception when `new` fails. The second uses function **`set_new_handler`** to handle `new` failures. [*Note:* The examples in Figs. 24.5–24.6 allocate large amounts of dynamic memory, which could cause your computer to become sluggish.]

`new` Throwing `bad_alloc` on Failure

Figure 24.5 demonstrates `new` throwing `bad_alloc` on failure to allocate the requested memory. The `for` statement (lines 16–20) inside the `try` block should loop 50 times and, on each pass, allocate an array of 50,000,000 `double` values. If `new` fails and throws a `bad_alloc` exception, the loop terminates, and the program continues in line 22, where the `catch` handler catches and processes the exception. Lines 24–25 print the message `"Exception occurred:"` followed by the message returned from the base-class-`exception` version of function `what` (i.e., an implementation-defined exception-specific message, such as `"Allocation Failure"` in Microsoft Visual C++). The output shows that the program performed only four iterations of the loop before `new` failed and threw the `bad_alloc` exception. Your output might differ based on the physical memory, disk space available for virtual memory on your system and the compiler you're using.

```
 1  // Fig. 24.5: fig24_05.cpp
 2  // Demonstrating standard new throwing bad_alloc when memory
 3  // cannot be allocated.
 4  #include <iostream>
 5  #include <new> // bad_alloc class is defined here
 6  using namespace std;
 7
 8  int main()
 9  {
10     double *ptr[ 50 ];
11
12     // aim each ptr[i] at a big block of memory
13     try
14     {
15        // allocate memory for ptr[ i ]; new throws bad_alloc on failure
16        for ( int i = 0; i < 50; ++i )
17        {
```

Fig. 24.5 | `new` throwing `bad_alloc` when memory cannot be allocated. (Part 1 of 2.)

```
         ptr[ i ] = new double[ 50000000 ]; // may throw exception
         cout << "ptr[" << i << "] points to 50,000,000 new doubles\n";
      } // end for
   } // end try
   catch ( bad_alloc &memoryAllocationException )
   {
      cerr << "Exception occurred: "
         << memoryAllocationException.what() << endl;
   } // end catch
} // end main
```

```
ptr[0] points to 50,000,000 new doubles
ptr[1] points to 50,000,000 new doubles
ptr[2] points to 50,000,000 new doubles
ptr[3] points to 50,000,000 new doubles
Exception occurred: bad allocation
```

Fig. 24.5 | new throwing bad_alloc when memory cannot be allocated. (Part 2 of 2.)

new Returning 0 on Failure

The C++ standard specifies that compilers can use an older version of new that returns 0 upon failure. For this purpose, header <new> defines object **nothrow** (of type nothrow_t), which is used as follows:

```
      double *ptr = new( nothrow ) double[ 50000000 ];
```

The preceding statement uses the version of new that does *not* throw bad_alloc exceptions (i.e., nothrow) to allocate an array of 50,000,000 doubles.

Software Engineering Observation 24.8

To make programs more robust, use the version of new that throws bad_alloc exceptions on failure.

Handling new Failures Using Function set_new_handler

An additional feature for handling new failures is function set_new_handler (prototyped in standard header <new>). This function takes as its argument a pointer to a function that takes no arguments and returns void. This pointer points to the function that will be called if new fails. This provides you with a uniform approach to handling all new failures, regardless of where a failure occurs in the program. Once set_new_handler registers a **new handler** in the program, operator new does not throw bad_alloc on failure; rather, it defers the error handling to the new-handler function.

If new allocates memory successfully, it returns a pointer to that memory. If new fails to allocate memory and set_new_handler did not register a new-handler function, new throws a bad_alloc exception. If new fails to allocate memory and a new-handler function has been registered, the new-handler function is called. The C++ standard specifies that the new-handler function should perform one of the following tasks:

1. Make more memory available by deleting other dynamically allocated memory (or telling the user to close other applications) and return to operator new to attempt to allocate memory again.

2. Throw an exception of type bad_alloc.
3. Call function abort or exit (both found in header <cstdlib>) to terminate the program.

Figure 24.6 demonstrates set_new_handler. Function customNewHandler (lines 9–13) prints an error message (line 11), then calls abort (line 12) to terminate the program. The output shows that the loop iterated four times before new failed and invoked function customNewHandler. Your output might differ based on the physical memory and disk space available for virtual memory on your system and your compiler.

```
 1  // Fig. 24.6: fig24_06.cpp
 2  // Demonstrating set_new_handler.
 3  #include <iostream>
 4  #include <new> // set_new_handler function prototype
 5  #include <cstdlib> // abort function prototype
 6  using namespace std;
 7
 8  // handle memory allocation failure
 9  void customNewHandler()
10  {
11     cerr << "customNewHandler was called";
12     abort();
13  } // end function customNewHandler
14
15  // using set_new_handler to handle failed memory allocation
16  int main()
17  {
18     double *ptr[ 50 ];
19
20     // specify that customNewHandler should be called on
21     // memory allocation failure
22     set_new_handler( customNewHandler );
23
24     // aim each ptr[i] at a big block of memory; customNewHandler will be
25     // called on failed memory allocation
26     for ( int i = 0; i < 50; ++i )
27     {
28        ptr[ i ] = new double[ 50000000 ]; // may throw exception
29        cout << "ptr[" << i << "] points to 50,000,000 new doubles\n";
30     } // end for
31  } // end main
```

```
ptr[0] points to 50,000,000 new doubles
ptr[1] points to 50,000,000 new doubles
ptr[2] points to 50,000,000 new doubles
ptr[3] points to 50,000,000 new doubles
customNewHandler was called
This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

Fig. 24.6 | set_new_handler specifying the function to call when new fails.

24.10 Class unique_ptr and Dynamic Memory Allocation[1]

A common programming practice is to allocate dynamic memory, assign the address of that memory to a pointer, use the pointer to manipulate the memory and deallocate the memory with `delete` when the memory is no longer needed. If an exception occurs after successful memory allocation but *before* the `delete` statement executes, a *memory leak* could occur. The C++ standard provides class template **`unique_ptr`** in header **`<memory>`** to deal with this situation.

An object of class `unique_ptr` maintains a pointer to dynamically allocated memory. When a `unique_ptr` object destructor is called (for example, when a `unique_ptr` object goes out of scope), it performs a `delete` operation on its pointer data member. Class template `unique_ptr` provides overloaded operators * and -> so that a `unique_ptr` object can be used just as a regular pointer variable is. Figure 24.9 demonstrates a `unique_ptr` object that points to a dynamically allocated object of class `Integer` (Figs. 24.7–24.8).

```
// Fig. 24.7: Integer.h
// Integer class definition.

class Integer
{
public:
   Integer( int i = 0 ); // Integer default constructor
   ~Integer(); // Integer destructor
   void setInteger( int i ); // functions to set Integer
   int getInteger() const; // function to return Integer
private:
   int value;
}; // end class Integer
```

Fig. 24.7 | Integer class definition.

```
// Fig. 24.8: Integer.cpp
// Member function definitions of class Integer.
#include <iostream>
#include "Integer.h"
using namespace std;

// Integer default constructor
Integer::Integer( int i )
   : value( i )
{
   cout << "Constructor for Integer " << value << endl;
} // end Integer constructor

```

Fig. 24.8 | Member function definitions of class `Integer`. (Part 1 of 2.)

1. Class `unique_ptr` is a part of the new C++ standard that's already implemented in Visual C++ 2010 and GNU C++. This class replaces the deprecated `auto_ptr` class. To compile this program in GNU C++, use the `-std=C++0x` compiler flag.

```cpp
// Integer destructor
Integer::~Integer()
{
   cout << "Destructor for Integer " << value << endl;
} // end Integer destructor

// set Integer value
void Integer::setInteger( int i )
{
   value = i;
} // end function setInteger

// return Integer value
int Integer::getInteger() const
{
   return value;
} // end function getInteger
```

Fig. 24.8 | Member function definitions of class `Integer`. (Part 2 of 2.)

Line 15 of Fig. 24.9 creates `unique_ptr` object `ptrToInteger` and initializes it with a pointer to a dynamically allocated `Integer` object that contains the value 7. Line 18 uses the `unique_ptr` overloaded `->` operator to invoke function `setInteger` on the `Integer` object that `ptrToInteger` manages. Line 21 uses the `unique_ptr` overloaded `*` operator to dereference `ptrToInteger`, then uses the dot (`.`) operator to invoke function `getInteger` on the `Integer` object. Like a regular pointer, a `unique_ptr`'s `->` and `*` overloaded operators can be used to access the object to which the `unique_ptr` points.

```cpp
 1  // Fig. 24.9: fig24_09.cpp
 2  // unique_ptr object manages dynamically allocated memory.
 3  #include <iostream>
 4  #include <memory>
 5  using namespace std;
 6
 7  #include "Integer.h"
 8
 9  // use unique_ptr to manipulate Integer object
10  int main()
11  {
12     cout << "Creating a unique_ptr object that points to an Integer\n";
13
14     // "aim" unique_ptr at Integer object
15     unique_ptr< Integer > ptrToInteger( new Integer( 7 ) );
16
17     cout << "\nUsing the unique_ptr to manipulate the Integer\n";
18     ptrToInteger->setInteger( 99 ); // use unique_ptr to set Integer value
19
20     // use unique_ptr to get Integer value
21     cout << "Integer after setInteger: " << ( *ptrToInteger ).getInteger()
22  } // end main
```

Fig. 24.9 | `unique_ptr` object manages dynamically allocated memory. (Part 1 of 2.)

```
Creating a unique_ptr object that points to an Integer
Constructor for Integer 7

Using the unique_ptr to manipulate the Integer
Integer after setInteger: 99

Destructor for Integer 99
```

Fig. 24.9 | `unique_ptr` object manages dynamically allocated memory. (Part 2 of 2.)

Because `ptrToInteger` is a local automatic variable in `main`, `ptrToInteger` is destroyed when `main` terminates. The `unique_ptr` destructor forces a `delete` of the `Integer` object pointed to by `ptrToInteger`, which in turn calls the `Integer` class destructor. The memory that `Integer` occupies is released, regardless of how control leaves the block (e.g., by a `return` statement or by an exception). Most importantly, using this technique can *prevent memory leaks*. For example, suppose a function returns a pointer aimed at some object. Unfortunately, the function caller that receives this pointer might not `delete` the object, thus resulting in *a memory leak*. However, if the function returns a `unique_ptr` to the object, the object will be deleted automatically when the `unique_ptr` object's destructor gets called.

Only one `unique_ptr` at a time can own a dynamically allocated object and the object cannot be an array. By using its overloaded assignment operator or copy constructor, a `unique_ptr` can transfer ownership of the dynamic memory it manages. The last `unique_ptr` object that maintains the pointer to the dynamic memory will delete the memory. This makes `unique_ptr` an ideal mechanism for returning dynamically allocated memory to client code. When the `unique_ptr` goes out of scope in the client code, the `unique_ptr`'s destructor deletes the dynamic memory.

24.11 Standard Library Exception Hierarchy

Experience has shown that exceptions fall nicely into a number of categories. The C++ Standard Library includes a hierarchy of exception classes, some of which are shown in Fig. 24.10. As we first discussed in Section 24.2, this hierarchy is headed by base-class `exception` (defined in header `<exception>`), which contains `virtual` function `what`, which derived classes can override to issue appropriate error messages.

Immediate derived classes of base-class `exception` include `runtime_error` and **`logic_error`** (both defined in header `<stdexcept>`), each of which has several derived classes. Also derived from `exception` are the exceptions thrown by C++ operators—for example, `bad_alloc` is thrown by `new` (Section 24.9), **`bad_cast`** is thrown by `dynamic_cast` (Chapter 21) and **`bad_typeid`** is thrown by `typeid` (Chapter 21). Including **`bad_exception`** in the `throw` list of a function means that, if an unexpected exception occurs, function `unexpected` can throw `bad_exception` rather than terminating the program's execution (by default) or calling another function specified by `set_unexpected`.

Common Programming Error 24.8

Placing a `catch` handler that catches a base-class object before a `catch` that catches an object of a class derived from that base class is a logic error. The base-class `catch` catches all objects of classes derived from that base class, so the derived-class `catch` will never execute.

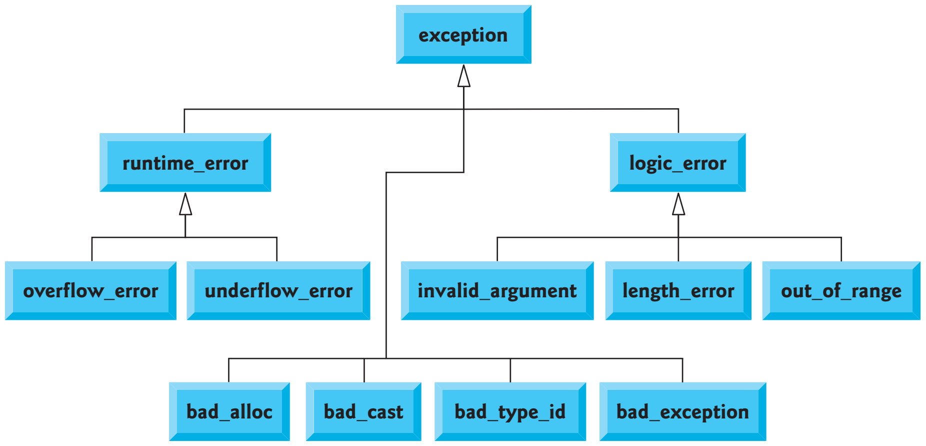


Fig. 24.10 | Some of the Standard Library exception classes.

Class `logic_error` is the base class of several standard exception classes that indicate errors in program logic. For example, class **`invalid_argument`** indicates that an invalid argument was passed to a function. (Proper coding can, of course, prevent invalid arguments from reaching a function.) Class **`length_error`** indicates that a length larger than the maximum size allowed for the object being manipulated was used for that object. Class **`out_of_range`** indicates that a value, such as a subscript into an array, exceeded its allowed range of values.

Class `runtime_error`, which we used briefly in Section 24.6, is the base class of several other standard exception classes that indicate execution-time errors. For example, class **`overflow_error`** describes an **arithmetic overflow error** (i.e., the result of an arithmetic operation is larger than the largest number that can be stored in the computer) and class **`underflow_error`** describes an **arithmetic underflow error** (i.e., the result of an arithmetic operation is smaller than the smallest number that can be stored in the computer).

Common Programming Error 24.9

Exception classes need not be derived from class `exception`, so catching type `exception` is not guaranteed to `catch` all exceptions a program could encounter.

Error-Prevention Tip 24.4

To `catch` all exceptions potentially thrown in a `try` block, use `catch(...)`. One weakness with catching exceptions in this way is that the type of the caught exception is unknown at compile time. Another weakness is that, without a named parameter, there's no way to refer to the exception object inside the exception handler.

Software Engineering Observation 24.9

The standard `exception` hierarchy is a good starting point for creating exceptions. You can build programs that can `throw` standard exceptions, `throw` exceptions derived from the standard exceptions or `throw` your own exceptions not derived from the standard exceptions.

Software Engineering Observation 24.10

Use `catch(...)` to perform recovery that does not depend on the exception type (e.g., releasing common resources). The exception can be rethrown to alert more specific enclosing `catch` handlers.

24.12 Wrap-Up

In this chapter, you learned how to use exception handling to deal with errors in a program. You learned that exception handling enables you to remove error-handling code from the "main line" of the program's execution. We demonstrated exception handling in the context of a divide-by-zero example. We reviewed how to use `try` blocks to enclose code that may throw an exception, and how to use `catch` handlers to deal with exceptions that may arise. You learned how to throw and rethrow exceptions, and how to handle the exceptions that occur in constructors. The chapter continued with discussions of processing `new` failures, dynamic memory allocation with class `unique_ptr` and the standard library exception hierarchy.

Summary

Section 24.1 Introduction

- An exception (p. 877) is an indication of a problem that occurs during a program's execution.
- Exception handling enables you to create programs that can resolve problems that occur at execution time—often allowing programs to continue executing as if no problems had been encountered. More severe problems may require a program to notify the user of the problem before terminating in a controlled manner.

Section 24.2 Example: Handling an Attempt to Divide by Zero

- Class `exception` is the standard base class for exceptions classes (p. 878). It provides virtual function `what` (p. 878) that returns an appropriate error message and can be overridden in derived classes.
- Class `runtime_error` (p. 878), which is defined in header `<stdexcept>` (p. 878), is the C++ standard base class for representing runtime errors.
- C++ uses the termination model (p. 881) of exception handling.
- A `try` block consists of keyword `try` followed by braces (`{}`) that define a block of code in which exceptions might occur. The `try` block encloses statements that might cause exceptions and statements that should not execute if exceptions occur.
- At least one `catch` handler must immediately follow a `try` block. Each `catch` handler specifies an exception parameter that represents the type of exception the `catch` handler can process.
- If an exception parameter includes an optional parameter name, the `catch` handler can use that parameter name to interact with a caught exception object (p. 882).
- The point in the program at which an exception occurs is called the throw point (p. 881).
- If an exception occurs in a `try` block, the `try` block expires and program control transfers to the first `catch` in which the exception parameter's type matches that of the thrown exception.
- When a `try` block terminates, local variables defined in the block go out of scope.
- When a `try` block terminates due to an exception, the program searches for the first `catch` handler that matches the type of exception that occurred. A match occurs if the types are identical

or if the thrown exception's type is a derived class of the exception-parameter type. When a match occurs, the code contained within the matching `catch` handler executes.

- When a `catch` handler finishes processing, the `catch` parameter and local variables defined within the `catch` handler go out of scope. Any remaining `catch` handlers that correspond to the `try` block are ignored, and execution resumes at the first line of code after the `try...catch` sequence.
- If no exceptions occur in a `try` block, the program ignores the `catch` handler(s) for that block. Program execution resumes with the next statement after the `try...catch` sequence.
- If an exception that occurs in a `try` block has no matching `catch` handler, or if an exception occurs in a statement that is not in a `try` block, the function that contains the statement terminates immediately, and the program attempts to locate an enclosing `try` block in the calling function. This process is called stack unwinding (p. 882).
- To throw an exception, use keyword `throw` followed by an operand that represents the type of exception to throw. The operand of a `throw` can be of any type.

Section 24.3 When to Use Exception Handling
- Exception handling is for synchronous errors (p. 883), which occur when a statement executes.
- Exception handling is not designed to process errors associated with asynchronous events (p. 883), which occur in parallel with, and independent of, the program's flow of control.

Section 24.4 Rethrowing an Exception
- The exception handler can defer the exception handling (or perhaps a portion of it) to another exception handler. In either case, the handler achieves this by rethrowing the exception (p. 884).
- Common examples of exceptions are out-of-range array subscripts, arithmetic overflow, division by zero, invalid function parameters and unsuccessful memory allocations.

Section 24.5 Processing Unexpected Exceptions
- Function `unexpected` calls the function registered with function `set_unexpected` (p. 885). If no function has been registered in this manner, function `terminate` (p. 884) is called by default.
- Function `set_terminate` (p. 886) can specify the function to invoke when `terminate` is called. Otherwise, `terminate` calls `abort` (p. 886), which terminates the program without calling the destructors of objects that are declared `static` and `auto`.
- Functions `set_terminate` and `set_unexpected` each return a pointer to the last function called by `terminate` and `unexpected`, respectively (0, the first time each is called). This enables you to save the function pointer so it can be restored later.
- Functions `set_terminate` and `set_unexpected` take as arguments pointers to functions with `void` return types and no arguments.
- If a programmer-defined termination function does not exit a program, function `abort` will be called after the programmer-defined termination function completes execution.

Section 24.6 Stack Unwinding
- Unwinding the function call stack means that the function in which the exception was not caught terminates, all local variables in that function are destroyed and control returns to the statement that originally invoked that function.

Section 24.7 Constructors, Destructors and Exception Handling
- Exceptions thrown by a constructor cause destructors to be called for any objects built as part of the object being constructed before the exception is thrown.
- Each automatic object constructed in a `try` block is destructed before an exception is thrown.

- Stack unwinding completes before an exception handler begins executing.
- If a destructor invoked as a result of stack unwinding throws an exception, `terminate` is called.
- If an object has member objects, and if an exception is thrown before the outer object is fully constructed, then destructors will be executed for the member objects that have been constructed before the exception occurs.
- If an array of objects has been partially constructed when an exception occurs, only the destructors for the constructed array element objects will be called.
- When an exception is thrown from the constructor for an object that is created in a `new` expression, the dynamically allocated memory for that object is released.

Section 24.8 Exceptions and Inheritance

- If a `catch` handler catches a pointer or reference to an exception object of a base-class type, it also can catch a pointer or reference to all objects of classes derived publicly from that base class—this allows for polymorphic processing of related errors.

Section 24.9 Processing **`new`** *Failures*

- The C++ standard document specifies that, when operator `new` fails, it throws a `bad_alloc` exception (p. 889), which is defined in header `<new>`.
- Function `set_new_handler` (p. 889) takes as its argument a pointer to a function that takes no arguments and returns `void`. This pointer points to the function that will be called if `new` fails.
- Once `set_new_handler` registers a `new` handler (p. 890) in the program, operator `new` does not throw `bad_alloc` on failure; rather, it defers the error handling to the `new`-handler function.
- If `new` allocates memory successfully, it returns a pointer to that memory.
- If an exception occurs after successful memory allocation but before the `delete` statement executes, a memory leak could occur.

Section 24.10 Class **`unique_ptr`** *and Dynamic Memory Allocation*

- The C++ Standard Library provides class template `unique_ptr` (p. 892) to deal with memory leaks.
- An object of class `unique_ptr` maintains a pointer to dynamically allocated memory. A `unique_ptr`'s destructor performs a `delete` operation on the `unique_ptr`'s pointer data member.
- Class template `unique_ptr` provides overloaded operators `*` and `->` so that a `unique_ptr` object can be used just as a regular pointer variable is. A `unique_ptr` also transfers ownership of the dynamic memory it manages via its copy constructor and overloaded assignment operator.

Section 24.11 Standard Library Exception Hierarchy

- The C++ Standard Library includes a hierarchy of exception classes. This hierarchy is headed by base-class `exception`.
- Immediate derived classes of base class `exception` include `runtime_error` and `logic_error` (both defined in header `<stdexcept>`), each of which has several derived classes.
- Several operators throw standard exceptions—operator `new` throws `bad_alloc`, operator `dynamic_cast` throws `bad_cast` (p. 894) and operator `typeid` throws `bad_typeid` (p. 894).
- Including `bad_exception` (p. 894) in the throw list of a function means that, if an unexpected exception occurs, function `unexpected` can throw `bad_exception` rather than terminating the program's execution or calling another function specified by `set_unexpected`.

Self-Review Exercises

24.1 List five common examples of exceptions.

24.2 Give several reasons why exception-handling techniques should not be used for conventional program control.

24.3 Why are exceptions appropriate for dealing with errors produced by library functions?

24.4 What's a "resource leak"?

24.5 If no exceptions are thrown in a `try` block, where does control proceed to after the `try` block completes execution?

24.6 What happens if an exception is thrown outside a `try` block?

24.7 Give a key advantage and a key disadvantage of using `catch(...)`.

24.8 What happens if no `catch` handler matches the type of a thrown object?

24.9 What happens if several handlers match the type of the thrown object?

24.10 Why would you specify a base-class type as the type of a `catch` handler, then `throw` objects of derived-class types?

24.11 Suppose a `catch` handler with a precise match to an exception object type is available. Under what circumstances might a different handler be executed for exception objects of that type?

24.12 Must throwing an exception cause program termination?

24.13 What happens when a `catch` handler `throws` an exception?

24.14 What does the statement `throw;` do?

Answers to Self-Review Exercises

24.1 Insufficient memory to satisfy a `new` request, array subscript out of bounds, arithmetic overflow, division by zero, invalid function parameters.

24.2 (a) Exception handling is designed to handle infrequently occurring situations that often result in program termination, so compiler writers are not required to implement exception handling to perform optimally. (b) Flow of control with conventional control structures generally is clearer and more efficient than with exceptions. (c) Problems can occur because the stack is unwound when an exception occurs and resources allocated prior to the exception might not be freed. (d) The "additional" exceptions make it more difficult for you to handle the larger number of exception cases.

24.3 It's unlikely that a library function will perform error processing that will meet the unique needs of all users.

24.4 A program that terminates abruptly could leave a resource in a state in which other programs would not be able to acquire the resource, or the program itself might not be able to reacquire a "leaked" resource.

24.5 The exception handlers (in the `catch` handlers) for that `try` block are skipped, and the program resumes execution after the last `catch` handler.

24.6 An exception thrown outside a `try` block causes a call to `terminate`.

24.7 The form `catch(...)` catches any type of exception thrown in a `try` block. An advantage is that all possible exceptions will be caught. A disadvantage is that the `catch` has no parameter, so it cannot reference information in the thrown object and cannot know the cause of the exception.

24.8 This causes the search for a match to continue in the next enclosing `try` block if there is one. As this process continues, it might eventually be determined that there is no handler in the program that matches the type of the thrown object; in this case, `terminate` is called, which by default calls `abort`. An alternative `terminate` function can be provided as an argument to `set_terminate`.

24.9 The first matching exception handler after the `try` block is executed.

24.10 This is a nice way to `catch` related types of exceptions.

24.11 A base-class handler would catch objects of all derived-class types.

24.12 No, but it does terminate the block in which the exception is thrown.

24.13 The exception will be processed by a `catch` handler (if one exists) associated with the `try` block (if one exists) enclosing the `catch` handler that caused the exception.

24.14 It rethrows the exception if it appears in a `catch` handler; otherwise, function `unexpected` is called.

Exercises

24.15 *(Exceptional Conditions)* List various exceptional conditions that have occurred throughout this text. List as many additional exceptional conditions as you can. For each of these exceptions, describe briefly how a program typically would handle the exception, using the exception-handling techniques discussed in this chapter. Some typical exceptions are division by zero, arithmetic overflow, array subscript out of bounds, exhaustion of the free store, etc.

24.16 *(Catch Parameter)* Under what circumstances would you not provide a parameter name when defining the type of the object that will be caught by a handler?

24.17 *(**`throw`** Statement)* A program contains the statement

```
throw;
```

Where would you normally expect to find such a statement? What if that statement appeared in a different part of the program?

24.18 *(Exception Handling vs. Other Schemes)* Compare and contrast exception handling with the various other error-processing schemes discussed in the text.

24.19 *(Exception Handling and Program Control)* Why should exceptions *not* be used as an alternate form of program control?

24.20 *(Handling Related Exceptions)* Describe a technique for handling related exceptions.

24.21 *(Throwing Exceptions from a **`catch`**)* Suppose a program `throw`s an exception and the appropriate exception handler begins executing. Now suppose that the exception handler itself `throw`s the same exception. Does this create infinite recursion? Write a program to check your observation.

24.22 *(Catching Derived-Class Exceptions)* Use inheritance to create various derived classes of `runtime_error`. Then show that a `catch` handler specifying the base class can `catch` derived-class exceptions.

24.23 *(Throwing the Result of a Conditional Expression)* Throw the result of a conditional expression that returns either a `double` or an `int`. Provide an `int catch` handler and a `double catch` handler. Show that only the `double catch` handler executes, regardless of whether the `int` or the `double` is returned.

24.24 *(Local Variable Destructors)* Write a program illustrating that all destructors for objects constructed in a block are called before an exception is thrown from that block.

24.25 *(Member Object Destructors)* Write a program illustrating that member object destructors are called for only those member objects that were constructed before an exception occurred.

24.26 *(Catching All Exceptions)* Write a program that demonstrates several exception types being caught with the `catch(...)` exception handler.

24.27 *(Order of Exception Handlers)* Write a program illustrating that the order of exception handlers is important. The first matching handler is the one that executes. Attempt to compile and run your program two different ways to show that two different handlers execute with two different effects.

24.28 *(Constructors Throwing Exceptions)* Write a program that shows a constructor passing information about constructor failure to an exception handler after a `try` block.

24.29 *(Rethrowing Exceptions)* Write a program that illustrates rethrowing an exception.

24.30 *(Uncaught Exceptions)* Write a program that illustrates that a function with its own `try` block does not have to catch every possible error generated within the `try`. Some exceptions can slip through to, and be handled in, outer scopes.

24.31 *(Stack Unwinding)* Write a program that `throw`s an exception from a deeply nested function and still has the `catch` handler following the `try` block enclosing the initial call in `main` catch the exception.