Chapter 16 Exception Handling

C++ How to Program, Late Objects Version, 7/e

OBJECTIVES

In this chapter you'll learn:

- What exceptions are and when to use them.
- 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 scope.
- To handle `new` failures.
- To use `auto_ptr` to prevent memory leaks.
- To understand the standard exception hierarchy.

16.1 Introduction

- In this chapter, we introduce exception handling.
- An exception is an indication of a problem that occurs during a program's execution.
- The name "exception" implies that the problem occurs infrequently—if the "rule" is that a statement normally executes correctly, then the "exception to the rule" is that a problem occurs.
- Exception handling enables you to create applications that can resolve (or handle) exceptions.

16.1 Introduction (cont.)

- In many cases, handling an exception allows a program to continue executing as if no problem had been encountered.
- A more severe problem could prevent a program from continuing normal execution, instead requiring the program to notify the user of the problem before terminating in a controlled manner.
- The features presented in this chapter enable you to write robust and fault-tolerant programs that can deal with problems that may arise and continue executing or terminate gracefully.

Error-Prevention Tip 16.1

Exception handling helps improve a program's fault tolerance.

Software Engineering Observation 16.1

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

16.2 Exception-Handling Overview

- Program logic frequently tests conditions that determine how program execution proceeds.
- Consider the following pseudocode:

```
Perform a task
If the preceding task did not execute correctly
    Perform error processing
Perform next task
If the preceding task did not execute correctly
    Perform error processing
    ...
```

- In this pseudocode, we begin by performing a task. We then test whether that task executed correctly. If not, we perform error processing. Otherwise, we continue with the next task.
- Intermixing program logic with error-handling logic can make the program difficult to read, modify, maintain and debug—especially in large applications.

Performance Tip 16.1

If the potential problems occur infrequently, intermixing program logic and error-handling logic can degrade a program's performance, because the program must (potentially frequently) perform tests to determine whether the task executed correctly and the next task can be performed.

16.2 Exception-Handling Overview (cont.)

- Exception handling enables you to remove error-handling code from the "main line" of the program's execution, which improves program clarity and enhances modifiability.
- You can decide to handle any exceptions you choose—all exceptions, all exceptions of a certain type or all exceptions of a group of related types (e.g., exception types that belong to an inheritance hierarchy).
- Such flexibility reduces the likelihood that errors will be overlooked and thereby makes a program more robust.
- With programming languages that do not support exception handling, programmers often delay writing error-processing code or sometimes forget to include it.
- This results in less robust software products.
- C++ enables you to deal with exception handling easily from the inception of a project.

16.3 Example: Handling an Attempt to Divide by Zero

- Let's consider a simple example of exception handling (Figs. 16.1–16.2).
- The purpose of this example is to show how to prevent 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 positive or negative infinity is displayed as `INF` or `-INF`, respectively.

16.3 Example: Handling an Attempt to Divide by Zero (cont.)

- 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 promoted to type `double` for the calculation.
- So function `quotient` actually performs the division using two `double` values and returns a `double` result.

16.3 Example: Handling an Attempt to Divide by Zero (cont.)

- Although division by zero is 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 uses 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 to execute even after an improper value is entered, thus making the program more robust.

16.3 Example: Handling an Attempt to Divide by Zero (cont.)

- `DivideByZeroException.h` (Fig. 16.1) defines an exception class that represents the type of the problem that might occur in the example, and `fig16_02.cpp` (Fig. 16.2) defines the `quotient` function and the `main` function that calls it.
- Function `main` contains the code that demonstrates exception handling.

16.3 Example: Handling an Attempt to Divide by Zero (cont.)

- Figure 16.1 defines class `DivideByZeroException` as a derived class of Standard Library class `runtime_error` (defined in header file `<stdexcept>`).
- Class `runtime_error`—a derived class of Standard Library class `exception` (defined in header file `<exception>`)—is the C++ standard base class for representing runtime errors.
- Class `exception` is the standard C++ base class for all exceptions.
  - Section 16.13 discusses class `exception` and its derived classes in detail.

16.3 Example: Handling an Attempt to Divide by Zero (cont.)

- 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 are not required to derive a custom exception class, such as `DivideByZeroException`, from the standard exception classes provided by C++.
  - 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. 16.2 to indicate when an attempt is made to divide by zero.

```
1  // Fig. 16.1: DivideByZeroException.h
2  // Class DivideByZeroException definition.
3  #include <stdexcept> // stdexcept header file 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. 16.1 | Class `DivideByZeroException` definition.

16.3 Example: Handling an Attempt to Divide by Zero (cont.)

- The program in Fig. 16.2 uses exception handling to wrap code that might throw a "divide-by-zero" exception and to handle that exception, should one occur.
- Function `quotient` 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.
- However, if the user inputs `0` for the denominator, function `quotient` throws an exception.

```
1  // Fig. 16.2: Fig16_02.cpp
2  // A simple exception-handling example that checks for
3  // divide-by-zero exceptions.
4  #include <iostream>
5  #include "DivideByZeroException.h" // DivideByZeroException class
6  using namespace std;
7
8  // perform division and throw DivideByZeroException object if
9  // divide-by-zero exception occurs
10 double quotient( int numerator, int denominator )
11 {
12    // throw DivideByZeroException if trying to divide by zero
13    if ( denominator == 0 )
14       throw DivideByZeroException(); // terminate function
15
16    // return division result
17    return static_cast< double >( numerator ) / denominator;
18 } // end function quotient
19
```

Fig. 16.2 | Exception-handling example that throws exceptions on attempts to divide by zero. (Part 1 of 3.)

```
20  int main()
21  {
22     int number1; // user-specified numerator
23     int number2; // user-specified denominator
24     double result; // result of division
25
26     cout << "Enter two integers (end-of-file to end): ";
27
28     // enable user to enter two integers to divide
29     while ( cin >> number1 >> number2 )
30     {
31        // try block contains code that might throw exception
32        // and code that should not execute if an exception occurs
33        try
34        {
35           result = quotient( number1, number2 );
36           cout << "The quotient is: " << result << endl;
37        } // end try
38        catch ( DivideByZeroException &divideByZeroException )
39        {
40           cout << "Exception occurred: "
41              << divideByZeroException.what() << endl;
42        } // end catch
```

Fig. 16.2 | Exception-handling example that throws exceptions on attempts to divide by zero. (Part 2 of 3.)

```
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. 16.2 | Exception-handling example that throws exceptions on attempts to divide by zero. (Part 3 of 3.)

16.3 Example: Handling an Attempt to Divide by Zero (cont.)

- Exception handling is geared to situations in which the function that detects an error is unable to handle it.
- C++ provides `try` blocks to enable 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 be skipped if an exception occurs.
- 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 16.2

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.*

16.3 Example: Handling an Attempt to Divide by Zero (cont.)

- Exceptions are processed by `catch` handlers (also called exception handlers), which catch and handle exceptions.
- At least one `catch` handler (lines 38–42) must immediately follow each `try` block.
- Each `catch` handler begins with the keyword `catch` and specifies in parentheses an exception parameter that represents 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).

16.3 Example: Handling an Attempt to Divide by Zero (cont.)

- 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 16.1

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 16.2

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

Common Programming Error 16.3

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

16.3 Example: Handling an Attempt to Divide by Zero (cont.)

- 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.

16.3 Example: Handling an Attempt to Divide by Zero (cont.)

- 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 after the last `catch` handler following the `try` block.
- This is known as the termination model of exception handling.
- 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 16.4

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 16.2

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.

16.3 Example: Handling an Attempt to Divide by Zero (cont.)

- 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 16.8.

Common Programming Error 16.5

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.*

16.3 Example: Handling an Attempt to Divide by Zero (cont.)

- 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.
- Central characteristic of exception handling: A function should throw an exception before the error has an opportunity to occur.
- 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.
- The exception object is maintained by the exception-handling mechanism.

Performance Tip 16.2

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

Good Programming Practice 16.1

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

16.4 When to Use Exception Handling

- Exception handling is designed to process synchronous errors, which occur when a statement executes.
- Common examples of these errors are 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 16.3

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

Software Engineering Observation 16.4

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 16.5

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 16.6

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

16.4 When to Use Exception Handling (cont.)

- The exception-handling mechanism also is useful for processing problems that occur when a program interacts with software elements, such as member functions, constructors, destructors and classes.
- Rather than handling problems internally, 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 16.3

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 16.7

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

16.4 When to Use Exception Handling (cont.)

- 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.

16.5 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 (even partially) 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 16.6

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

16.5 Rethrowing an Exception (cont.)

- The program of Fig. 16.3 demonstrates rethrowing an exception.
- [Note: 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. 16.3: Fig16_03.cpp
2  // Demonstrating exception rethrowing.
3  #include <iostream>
4  #include <exception>
5  using namespace std;
6
7  // throw, catch and rethrow exception
8  void throwException()
9  {
10    // throw exception and catch it immediately
11    try
12    {
13       cout << "  Function throwException throws an exception\n";
14       throw exception(); // generate exception
15    } // end try
16    catch ( exception & ) // handle exception
17    {
18       cout << "  Exception handled in function throwException"
19          << "\n  Function throwException rethrows exception";
20       throw; // rethrow exception for further processing
21    } // end catch
22
23    cout << "This also should not print\n";
24 } // end function throwException
```

Fig. 16.3 | Rethrowing an exception. (Part 1 of 3.)

```
25
26 int main()
27 {
28    // throw exception
29    try
30    {
31       cout << "\nmain invokes function throwException\n";
32       throwException();
33       cout << "This should not print\n";
34    } // end try
35    catch ( exception & ) // handle exception
36    {
37       cout << "\n\nException handled in main\n";
38    } // end catch
39
40    cout << "Program control continues after catch in main\n";
41 } // end main
```

Fig. 16.3 | Rethrowing an exception. (Part 2 of 3.)

```
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. 16.3 | Rethrowing an exception. (Part 3 of 3.)

16.6 Exception Specifications

- An optional exception specification (also called a `throw list`) enumerates a list of exceptions that a function can throw.
- For example, consider the function declaration

```
int someFunction( double value )
   throw ( ExceptionA, ExceptionB, ExceptionC )
{
   // function body
}
```

- Indicates that function `someFunction` can throw exceptions of types `ExceptionA`, `ExceptionB` and `ExceptionC`.

16.6 Exception Specifications (cont.)

- A function can throw only exceptions of the types indicated by the specification or exceptions of any type derived from these types.
- If the function `throw`s an exception that does not belong to a specified type, the exception-handling mechanism calls function `unexpected`, which terminates the program.
- A function that does not provide an exception specification can `throw` any exception.
- Placing `throw()`—an empty exception specification—after a function's parameter list states that the function does not `throw` exceptions.
- If the function attempts to `throw` an exception, function `unexpected` is invoked.
- [*Note: Some compilers ignore exception specifications.*]

Common Programming Error 16.7

Throwing an exception that has not been declared in a function's exception specification causes a call to function `unexpected`.

Error-Prevention Tip 16.3

The compiler will not generate a compilation error if a function contains a `throw` *expression for an exception not listed in the function's exception specification. An error occurs only when that function attempts to throw that exception at execution time. To avoid surprises at execution time, carefully check your code to ensure that functions do not throw exceptions not listed in their exception specifications.*

Software Engineering Observation 16.8

It's generally recommended that you do not use exception specifications unless you're overriding a base-class member function that already has an exception specification. In this case, the exception specification is required for the derived-class member function.

16.7 Processing Unexpected Exceptions

- Function `unexpected` calls the function registered with function `set_unexpected` (defined in header file `<exception>`).
- If no function has been registered in this manner, function `terminate` is called by default.
- Cases in which function `terminate` is called include:
  - the exception mechanism cannot find a matching `catch` for a thrown exception
  - a destructor attempts to `throw` an exception during stack unwinding
  - an attempt is made to rethrow an exception when there is no exception currently being handled
  - a call to function `unexpected` defaults to calling function `terminate`

16.7 Processing Unexpected Exceptions (cont.)

- 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 16.8

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****.*

16.7 Processing Unexpected Exceptions (cont.)

- 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 after the other statements of the programmer-defined termination function are executed.

16.8 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. 16.4 demonstrates stack unwinding.

```
1  // Fig. 16.4: Fig16_04.cpp
2  // Demonstrating stack unwinding.
3  #include <iostream>
4  #include <stdexcept>
5  using namespace std;
6
7  // function3 throws runtime error
8  void function3() throw ( runtime_error )
9  {
10    cout << "In function 3" << endl;
11
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
```

Fig. 16.4 | Stack unwinding. (Part 1 of 3.)

```
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
```

Fig. 16.4 | Stack unwinding. (Part 2 of 3.)

```
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. 16.4 | Stack unwinding. (Part 3 of 3.)

16.8 Stack Unwinding (cont.)

- 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.

16.9 Constructors, Destructors and Exception Handling

- 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.

16.9 Constructors, Destructors and Exception Handling (cont.)

- 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.

16.9 Constructors, Destructors and Exception Handling (cont.)

- 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 16.4

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.*

16.10 Exceptions and Inheritance‘

- Various exception classes can be derived from a common base class, as we discussed in Section 16.3, 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 16.5

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.*

16.11 Processing new Failures

- The C++ standard specifies that, when operator `new` fails, it `throw`s a `bad_alloc` exception (defined in header file `<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. 16.5–16.6 allocate large amounts of dynamic memory, which could cause your computer to become sluggish.]*

16.11 Processing new Failures (cont.)

- Figure 16.5 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++).

16.11 Processing new Failures (cont.)

- 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 are using.

```
1  // Fig. 16.5: Fig16_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       {
18          ptr[ i ] = new double[ 50000000 ]; // may throw exception
19          cout << "ptr[" << i << "] points to 50,000,000 new doubles\n";
20       } // end for
21    } // end try
```

Fig. 16.5 | new throwing bad_alloc on failure. (Part 1 of 2.)

```
22      catch ( bad_alloc &memoryAllocationException )
23      {
24         cerr << "Exception occurred: "
25            << memoryAllocationException.what() << endl;
26      } // end catch
27   } // 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. 16.5 | new throwing `bad_alloc` on failure. (Part 2 of 2.)

16.11 Processing new Failures (cont.)

- In old versions of C++, operator `new` returned 0 when it failed to allocate memory.
- The C++ standard specifies that standard-compliant compilers can continue to use a version of `new` that returns `0` upon failure.
- For this purpose, header file `<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 `double`s.

Software Engineering Observation 16.9

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

16.11 Processing new Failures (cont.)

- Function `set_new_handler` (prototyped in standard header file `<new>`) 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` 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.

16.11 Processing new Failures (cont.)

- The C++ standard specifies that the `new`-handler function should perform one of the following tasks:
  - 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.
  - Throw an exception of type `bad_alloc`.
  - Call function `abort` or `exit` (both found in header file `<cstdlib>`) to terminate the program.
- Figure 16.6 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`.

```
1  // Fig. 16.6: Fig16_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 );
```

Fig. 16.6 | `set_new_handler` specifying the function to call when `new` fails. (Part 1 of 2.)

```
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. 16.6 | `set_new_handler` specifying the function to call when `new` fails. (Part 2 of 2.)

16.12 Class `auto_ptr` and Dynamic Memory Allocation

- 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 `auto_ptr` in header file `<memory>` to deal with this situation.

16.12 Class auto_ptr and Dynamic Memory Allocation (cont.)

- An object of class `auto_ptr` maintains a pointer to dynamically allocated memory.
- When an `auto_ptr` object destructor is called (for example, when an `auto_ptr` object goes out of scope), it performs a `delete` operation on its pointer data member.
- Class template `auto_ptr` provides overloaded operators `*` and `->` so that an `auto_ptr` object can be used just as a regular pointer variable is.
- Figure 16.9 demonstrates an `auto_ptr` object that points to a dynamically allocated object of class `Integer` (Figs. 16.7–16.8).

```
1  // Fig. 16.7: Integer.h
2  // Integer class definition.
3
4  class Integer
5  {
6  public:
7     Integer( int i = 0 ); // Integer default constructor
8     ~Integer(); // Integer destructor
9     void setInteger( int i ); // functions to set Integer
10    int getInteger() const; // function to return Integer
11 private:
12    int value;
13 }; // end class Integer
```

Fig. 16.7 | `Integer` class definition.

```
1  // Fig. 16.8: Integer.cpp
2  // Integer member function definitions.
3  #include <iostream>
4  #include "Integer.h"
5  using namespace std;
6
7  // Integer default constructor
8  Integer::Integer( int i )
9     : value( i )
10 {
11    cout << "Constructor for Integer " << value << endl;
12 } // end Integer constructor
13
14 // Integer destructor
15 Integer::~Integer()
16 {
17    cout << "Destructor for Integer " << value << endl;
18 } // end Integer destructor
19
20 // set Integer value
21 void Integer::setInteger( int i )
22 {
23    value = i;
24 } // end function setInteger
```

Fig. 16.8 | Member function definitions of class `Integer`. (Part 1 of 2.)

```
25
26  // return Integer value
27  int Integer::getInteger() const
28  {
29     return value;
30  } // end function getInteger
```

Fig. 16.8 | Member function definitions of class `Integer`. (Part 2 of 2.)

```
1  // Fig. 16.9: Fig16_09.cpp
2  // Demonstrating auto_ptr.
3  #include <iostream>
4  #include <memory>
5  using namespace std;
6
7  #include "Integer.h"
8
9  // use auto_ptr to manipulate Integer object
10 int main()
11 {
12    cout << "Creating an auto_ptr object that points to an Integer\n";
13
14    // "aim" auto_ptr at Integer object
15    auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );
16
17    cout << "\nUsing the auto_ptr to manipulate the Integer\n";
18    ptrToInteger->setInteger( 99 ); // use auto_ptr to set Integer value
19
20    // use auto_ptr to get Integer value
21    cout << "Integer after setInteger: " << ( *ptrToInteger ).getInteger()
22 } // end main
```

Fig. 16.9 | `auto_ptr` object manages dynamically allocated memory. (Part 1 of 2.)

```
Creating an auto_ptr object that points to an Integer
Constructor for Integer 7

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

Destructor for Integer 99
```

Fig. 16.9 | `auto_ptr` object manages dynamically allocated memory. (Part 2 of 2.)

16.12 Class auto_ptr and Dynamic Memory Allocation (cont.)

- Because `ptrToInteger` is a local automatic variable in `main`, `ptrToInteger` is destroyed when `main` terminates.
- The `auto_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.

16.12 Class auto_ptr and Dynamic Memory Allocation (cont.)

- Only one `auto_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, an `auto_ptr` can transfer ownership of the dynamic memory it manages.
- The last `auto_ptr` object that maintains the pointer to the dynamic memory will delete the memory.
- This makes `auto_ptr` an ideal mechanism for returning dynamically allocated memory to client code.
- When the `auto_ptr` goes out of scope in the client code, the `auto_ptr`'s destructor deletes the dynamic memory.

Common Programming Error 16.9

Because `auto_ptr` *objects transfer ownership of memory when they are copied, they cannot be used with Standard Library container classes like* `vector`*. Container classes often make copies of objects. This causes ownership of a container element to be transferred to another object, which might then be accidentally deleted when the copy goes out of scope. The* `Boost.Smart_ptr` *library (Section 23.6) provides memory management features similar* `auto_ptr` *that* can *be used with containers.*

16.13 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. 16.10.
- As we first discussed in Section 16.3, this hierarchy is headed by base-class `exception` (defined in header file `<exception>`), which contains `virtual` function `what`, which derived classes can override to issue appropriate error messages.

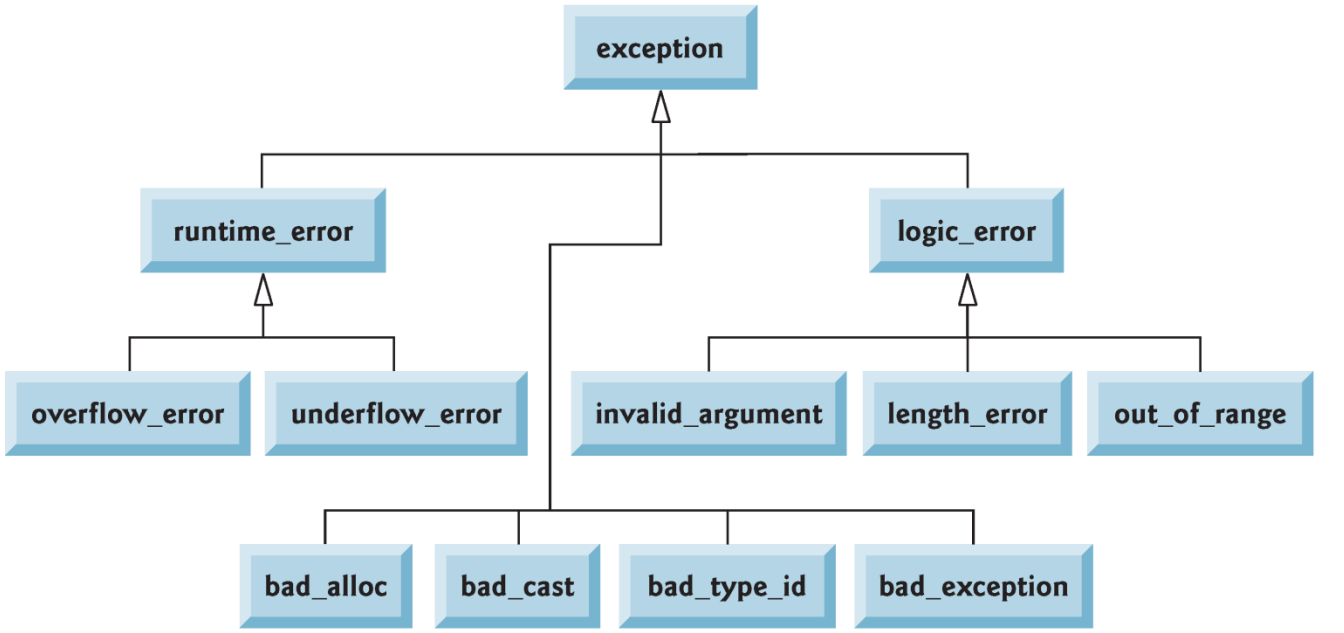

Fig. 16.10 | Some of the Standard Library exception classes.

16.13 Standard Library Exception Hierarchy (cont.)

- 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 16.11), `bad_cast` is thrown by `dynamic_cast` (Chapter 13) and `bad_typeid` is thrown by `typeid` (Chapter 13).
- 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 16.10

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.*

16.13 Standard Library Exception Hierarchy (cont.)

- 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.

16.13 Standard Library Exception Hierarchy (cont.)

- Class `runtime_error`, which we used briefly in Section 16.8, 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 16.11

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 16.6

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 is no way to refer to the exception object inside the exception handler.*

Software Engineering Observation 16.10

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 16.11

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.*

16.14 Other Error-Handling Techniques

- We've discussed several ways to deal with exceptional situations prior to this chapter.
- The following summarizes these and other error-handling techniques:
  - Ignore the exception. If an exception occurs, the program might fail as a result of the uncaught exception. This is devastating for commercial software products and special-purpose mission-critical software, but, for software developed for your own purposes, ignoring many kinds of errors is common.

16.14 Other Error-Handling Techniques (cont.)

- Abort the program. This, of course, prevents a program from running to completion and producing incorrect results. For many types of errors, this is appropriate, especially for nonfatal errors that enable a program to run to completion (potentially misleading you to think that the program functioned correctly). This strategy is inappropriate for mission-critical applications. Resource issues also are important here—if a program obtains a resource, the program should release that resource before program termination.

16.14 Other Error-Handling Techniques (cont.)

- Set error indicators. The problem with this approach is that programs might not check these error indicators at all points at which the errors could be troublesome. Another problem is that the program, after processing the problem, might not clear the error indicators.
- Test for the error condition, issue an error message and call `exit` (in `<cstdlib>`) to pass an appropriate error code to the program's environment.

16.14 Other Error-Handling Techniques (cont.)

- Certain kinds of errors have dedicated capabilities for handling them. For example, when operator `new` fails to allocate memory, a `new_handler` function can be called to handle the error. This function can be customized by supplying a function name as the argument to `set_new_handler`, as we discussed in Section 16.11.