16

Exception Handling

I never forget a face, but in your case I' ll make an exception.

—Groucho Marx

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

O! throw away the worser part of it, And live the purer with the other half.

—William Shakespeare

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

O infinite virtue! com’st thou smiling from the world’s great snare uncaught?

—William Shakespeare

OBJECTIVES

In this chapter you will 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.

Outline

16.1 Introduction

- Exceptions
  - Indicate problems that occur during a program’s execution
  - Occur infrequently
- Exception handling
  - Can resolve exceptions
    - Allow a program to continue executing or
    - Notify the user of the problem and
    - Terminate the program in a controlled manner
  - Makes programs robust and fault-tolerant

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

- **Intermixing program and error-handling logic**
  - **Pseudocode example**

    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
    …
  - **Makes the program difficult to read, modify, maintain and debug**

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
  - Removes error-handling code from the program execution’s “main line”
  - Programmers can handle any exceptions they choose
    - All exceptions,
    - All exceptions of a certain type or
    - All exceptions of a group of related types

16.3 Example: Handling an Attempt to Divide by Zero

- Class `exception`
  - Is the standard C++ base class for all exceptions
  - Provides its derived classes with `virtual` function `what`
    - Returns the exception's stored error message

```
// Fig. 16.1: DivideByZeroException.h
// Class DivideByZeroException definition.
#include <stdexcept> // stdexcept header file contains runtime_error
using std::runtime_error; // standard C++ library class runtime_error

// DivideByZeroException objects should be thrown by functions
// upon detecting division-by-zero exceptions
class DivideByZeroException : public runtime_error
{
public:
   // constructor specifies default error message
   DivideByZeroException::DivideByZeroException()
      : runtime_error( "attempted to divide by zero" ) {}
}; // end class DivideByZeroException
```

Outline

DivideByZeroException.h

(1 of 1)

Outline

Fig16_02.cpp

(1 of 3)

```
// Fig. 16.2: Fig16_02.cpp
// A simple exception-handling example that checks for
// divide-by-zero exceptions.
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include "DivideByZeroException.h" // DivideByZeroException class

// 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 should not execute if an exception occurs
      try
      {
         result = quotient( number1, number2 );
         cout << "The quotient is: " << result << endl;
      } // end try

      // exception handler handles a divide-by-zero exception
      catch ( DivideByZeroException &divideByZeroException )
      {
         cout << "Exception occurred: "
            << divideByZeroException.what() << endl;
      } // end catch

      cout << "\nEnter two integers (end-of-file to end): ";
   } // end while

   cout << endl;
   return 0; // terminate normally
} // 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
```

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

- `try` Blocks
  - Keyword `try` followed by braces (`{}`)
  - Should enclose
    - Statements that might cause exceptions and
    - Statements that should be skipped in case of an exception

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.)

- `catch` handlers
  - Immediately follow a `try` block
    - One or more `catch` handlers for each `try` block
  - Keyword `catch`
  - Exception parameter enclosed in parentheses
    - Represents the type of exception to process
    - Can provide an optional parameter name to interact with the caught exception object
  - Executes if exception parameter type matches the exception thrown in the `try` block
    - Could be a base class of the thrown exception's class

Common Programming Error 16.1

It is a syntax error to place code between a `try` block and its corresponding `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 is 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.)

- Termination model of exception handling
  - `try` block expires when an exception occurs
    - Local variables in `try` block go out of scope
  - The code within the matching `catch` handler executes
  - Control resumes with the first statement after the last `catch` handler following the `try` block
    - Control does not return to throw point
- Stack unwinding
  - Occurs if no matching `catch` handler is found
  - Program attempts to locate another enclosing `try` block in the calling function

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 is called mission-critical computing or business-critical computing.

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

- Throwing an exception
  - Use keyword `throw` followed by an operand representing the type of exception
    - The `throw` operand can be of any type
      - If the `throw` operand is an object, it is called an exception object
  - The `throw` operand initializes the exception parameter in the matching `catch` handler, if one is found

Common Programming Error 16.5

Use caution when `throw`ing the result of a conditional expression (`?:`), because 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 conditional expression converts the `int` to a `double`. However, the `catch` handler always catches the result as a `double`, rather than catching the result as a `double` when a `double` is thrown, and catching the result as an `int` when an `int` is thrown.

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

- When to use exception handling
  - To process synchronous errors
    - Occur when a statement executes
  - Not to process asynchronous errors
    - Occur in parallel with, and independent of, program execution
  - To process problems arising in predefined software elements
    - Such as predefined functions and classes
    - Error handling can be performed by the program code to be customized based on the application’s needs

Software Engineering Observation 16.3

Incorporate your exception-handling strategy into your system from the design process’ s 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 working 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 simplifies combining software components and enables them to work together effectively by enabling predefined components to communicate problems to application-specific components, which can then process the problems in an application-specific manner.

Performance Tip 16.3

When no exceptions occur, exception-handling code incurs little or no performance penalties. 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.

16.5 Rethrowing an Exception

- Rethrowing an exception
  - Empty `throw;` statement
  - Use when a `catch` handler cannot or can only partially process an exception
  - Next enclosing `try` block attempts to match the exception with one of its `catch` handlers

Common Programming Error 16.6

Executing an empty `throw` statement that is situated outside a `catch` handler causes a call to function terminate, which abandons exception processing and terminates the program immediately.

Outline

Fig16_03.cpp

(1 of 2)

```
// Fig. 16.3: Fig16_03.cpp
// Demonstrating exception rethrowing.
#include <iostream>
using std::cout;
using std::endl;

#include <exception>
using std::exception;

// 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
```

Rethrow the exception

Outline

Fig16_03.cpp

(2 of 2)

```cpp

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";
   return 0;
} // end main
```

Catch rethrown exception

```
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
```

16.6 Exception Specifications

- Exception specifications (a.k.a. `throw` lists)
  - Keyword `throw`
  - Comma-separated list of exception classes in parentheses
  - Example

```
int someFunction( double value )
   throw ( ExceptionA, ExceptionB,
           ExceptionC )
{
   …
}
```

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

16.6 Exception Specifications (Cont.)

- Exception specifications (Cont.)
  - A function can `throw` only exceptions of types in its specification or types derived from those types
    - If a function `throw`s a non-specification exception, function `unexpected` is called
      - This normally terminates the program
  - No exception specification indicates the function can `throw` any exception
  - An empty exception specification, `throw()`, indicates the function can not `throw` any exceptions

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.

16.7 Processing Unexpected Exceptions

- Function `unexpected`
  - Called when a function `throws` an exception not in its exception specification
  - Calls the function registered with function `set_unexpected`
  - Function `terminate` is called by default
- Function `set_unexpected` of `<exception>`
  - Takes as argument a pointer to a function with no arguments and a `void` return type
  - Returns a pointer to the last function called by `unexpected`
    - Returns `0` the first time

16.7 Processing Unexpected Exceptions (Cont.)

- Function `terminate`
  - Called when
    - No matching `catch` is found for a thrown exception
    - A destructor attempts to `throw` an exception during stack unwinding
    - Attempting to rethrow an exception when no exception is being handled
    - Calling function `unexpected` before registering a function with function `set_unexpected`
  - Calls the function registered with function `set_terminate`
  - Function `abort` is called by default

16.7 Processing Unexpected Exceptions (Cont.)

- Function `set_terminate`
  - Takes as argument a pointer to a function with no arguments and a `void` return type
  - Returns a pointer to the last function called by `terminate`
    - Returns `0` the first time
- Function `abort`
  - Terminates the program without calling destructors for automatic or static storage class objects
    - Could lead to resource leaks

16.8 Stack Unwinding

- Stack unwinding
  - Occurs when a thrown exception is not caught in a particular scope
  - Unwinding a function terminates that function
    - All local variables of the function are destroyed
    - Control returns to the statement that invoked the function
  - Attempts are made to `catch` the exception in outer `try`…`catch` blocks
  - If the exception is never caught, function `terminate` is called

Outline

Fig16_04.cpp

(1 of 3)

```
// Fig. 16.4: Fig16_04.cpp
// Demonstrating stack unwinding.
#include <iostream>
using std::cout;
using std::endl;

#include <stdexcept>
using std::runtime_error;

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

   // no try block, stack unwinding occur, return control to function2
   throw runtime_error( "runtime_error in function3" );
} // end function3

// function2 invokes function3
void function2() throw ( runtime_error )
{
   cout << "function3 is called inside function2" << endl;
   function3(); // stack unwinding occur, return control to function1
} // end function2
```

```cpp

// function1 invokes function2
void function1() throw ( runtime_error )
{
   cout << "function2 is called inside function1" << endl;
   function2(); // stack unwinding occur, return control to main
} // end function1

// demonstrate stack unwinding
int main()
{
   // invoke function1
   try
   {
      cout << "function1 is called inside main" << endl;
      function1(); // call function1 which throws runtime_error
   } // end try
   catch ( runtime_error &error ) // handle run-time error
   {
      cout << "Exception occurred: " << error.what() << endl;
      cout << "Exception handled in main" << endl;
   } // end catch

   return 0;
} // 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
```

Outline

Fig16_04.cpp

(3 of 3)

16.9 Constructors, Destructors and Exception Handling

- Exceptions and constructors
  - Exceptions enable constructors, which cannot return values, to report errors to the program
  - Exceptions thrown by constructors cause any already-constructed component objects to call their destructors
    - Only those objects that have already been constructed will be destructed
- Exceptions and destructors
  - Destructors are called for all automatic objects in the terminated `try` block when an exception is thrown
    - Acquired resources can be placed in local objects to automatically release the resources when an exception occurs
  - If a destructor invoked by stack unwinding throws an exception, function `terminate` is called

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

- **Inheritance with exception classes**
  - **New exception classes can be defined to inherit from existing exception classes**
  - **A catch handler for a particular exception class can also catch exceptions of classes derived from that class**

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 the programmer forgets to test explicitly for one or more of the derived-class pointer or reference types.

16.11 Processing new Failures

- `new` failures
  - Some compilers `throw` a `bad_alloc` exception
    - Compliant to the C++ standard specification
  - Some compilers return `0`
    - C++ standard-compliant compilers also have a version of `new` that returns `0`
      - Use expression `new( nothrow )`, where `nothrow` is of type `nothrow_t`
  - Some compilers `throw` `bad_alloc` if `<new>` is included

Outline

Fig16_05.cpp

(1 of 2)

```
// Fig. 16.5: Fig16_05.cpp
// Demonstrating pre-standard new returning 0 when memory
// is not allocated.
#include <iostream>
using std::cerr;
using std::cout;

int main()
{
   double *ptr[ 50 ];

   // allocate memory for ptr
   for ( int i = 0; i < 50; i++ )
   {
      ptr[ i ] = new double[ 50000000 ];

      if ( ptr[ i ] == 0 ) // did new fail to allocate memo
      {
         cerr << "Memory allocation failed for ptr[ " << i << " ]\n";
         break;
      } // end if
      else // successful memory allocation
         cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
   } // end for

   return 0;
} // end main
```

Allocate **50000000** **double** values

new will have returned **0** if the memory allocation operation failed

```
Allocated 50000000 doubles in ptr[ 0 ]
Allocated 50000000 doubles in ptr[ 1 ]
Allocated 50000000 doubles in ptr[ 2 ]
Memory allocation failed for ptr[ 3 ]
```

```cpp
// Fig. 16.6: Fig16_06.cpp
// Demonstrating standard new throwing bad_alloc when memory
// cannot be allocated.
#include <iostream>
using std::cerr;
using std::cout;
using std::endl;

#include <new> // standard operator new
using std::bad_alloc;

int main()
{
   double *ptr[ 50 ];

   // allocate memory for ptr
   try
   {
      // allocate memory for ptr[ i ]; new throws bad_alloc on failure
      for ( int i = 0; i < 50; i++ )
      {
         ptr[ i ] = new double[ 50000000 ]; // may throw exception
         cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
      } // end for
   } // end try
```

Allocate `50000000` **`double`** values

Outline

new throws a **bad_alloc** exception if the memory allocation operation failed

```

   // handle bad_alloc exception
   catch ( bad_alloc &memoryAllocationException )
   {
      cerr << "Exception occurred: "
         << memoryAllocationException.what() << endl;
   } // end catch

   return 0;
} // end main
```

```
Allocated 50000000 doubles in ptr[ 0 ]
Allocated 50000000 doubles in ptr[ 1 ]
Allocated 50000000 doubles in ptr[ 2 ]
Exception occurred: bad allocation
```

Fig16_06.cpp

(2 of 2)

Software Engineering Observation 16.8

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

16.11 Processing new Failures (Cont.)

- `new` failures (Cont.)
  - Function `set_new_handler`
    - Registers a function to handle `new` failures
      - The registered function is called by `new` when a memory allocation operation fails
    - Takes as argument a pointer to a function that takes no arguments and returns `void`
    - C++ standard specifies that the `new`-handler function should:
      - Make more memory available and let `new` try again,
      - Throw a `bad_alloc` exception or
      - Call function `abort` or `exit` to terminate the program

```cpp
// Fig. 16.7: Fig16_07.cpp
// Demonstrating set_new_handler.
#include <iostream>
using std::cerr;
using std::cout;

#include <new> // standard operator new and set_new_handler
using std::set_new_handler;

#include <cstdlib> // abort function prototype
using std::abort;

// handle memory allocation failure
void customNewHandler()
{
   cerr << "customNewHandler was called";
   abort();
} // end function customNewHandler

// using set_new_handler to handle failed memory allocation
int main()
{
   double *ptr[ 50 ];

   // specify that customNewHandler should be called on
   // memory allocation failure
   set_new_handler( customNewHandler );
```

Create a user-defined **new**-handler function **customNewHandler**

Register **customNewHandler** with **set_new_handler**

<u>Outline</u>

```

   // allocate memory for ptr[ i ]; customNewHandler will be
   // called on failed memory allocation
   for ( int i = 0; i < 50; i++ )
   {
      ptr[ i ] = new double[ 50000000 ]; // may throw exception
      cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
   } // end for

   return 0;
} // end main
```

Allocate **50000000 double** values

Fig16_07.cpp

(2 of 2)

```
Allocated 50000000 doubles in ptr[ 0 ]
Allocated 50000000 doubles in ptr[ 1 ]
Allocated 50000000 doubles in ptr[ 2 ]
customNewHandler was called
```

16.12 Class auto_ptr and Dynamic Memory Allocation

- Class template `auto_ptr`
  - Defined in header file `<memory>`
  - Maintains a pointer to dynamically allocated memory
    - Its destructor performs `delete` on the pointer data member
      - Prevents memory leaks by deleting the dynamically allocated memory even if an exception occurs
    - Provides overloaded operators `*` and `->` just like a regular pointer variable
    - Can pass ownership of the memory via the overloaded assignment operator or the copy constructor
      - The last `auto_ptr` object maintaining the pointer will `delete` the memory

Outline

```
// Fig. 16.8: 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
```

Integer.h

(1 of 1)

Outline

```
// Fig. 16.9: Integer.cpp
// Integer member function definition.
#include <iostream>
using std::cout;
using std::endl;

#include "Integer.h"

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

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

Integer.cpp

(1 of 2)

Outline

```
// 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
```

`Integer.cpp`

(2 of 2)

Outline

Fig16_10.cpp

(1 of 2)

```
// Fig. 16.10: Fig16_10.cpp
// Demonstrating auto_ptr.
#include <iostream>
using std::cout;
using std::endl;

#include <memory>
using std::auto_ptr; // auto_ptr class definition

#include "Integer.h"

// use auto_ptr to manipulate Integer object
int main()
{
   cout << "Creating an auto_ptr object that points to an Integer\n";

   // "aim" auto_ptr at Integer object
   auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );

   cout << "\nUsing the auto_ptr to manipulate the Integer\n";
   ptrToInteger->setInteger( 99 ); // use auto_ptr to set Integer val

   // use auto_ptr to get Integer value
   cout << "Integer after setInteger: " << ( *ptrToInteger ).getInteger()
   return 0;
} // end main
```

Create an **auto_ptr** to point to a dynamically allocated **Integer** object

Manipulate the **auto_ptr** as if it were a pointer to an **Integer**

The dynamically allocated memory is automatically deleted by the **auto_ptr** when it goes out of scope

```
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

Terminating program
Destructor for Integer 99
```

Outline

Fig16_10.cpp

(2 of 2)

Software Engineering Observation 16.9

An `auto_ptr` has restrictions on certain operations. For example, an `auto_ptr` cannot point to an array or a standard-container class.

16.13 Standard Library Exception Hierarchy

- Exception hierarchy classes
  - Base-class `exception`
    - Contains `virtual` function `what` for storing error messages
    - Exception classes derived from `exception`
      - `bad_alloc` – thrown by `new`
      - `bad_cast` – thrown by `dynamic_cast`
      - `bad_typeid` – thrown by `typeid`
      - `bad_exception` – thrown by `unexpected`
        - Instead of terminating the program or calling the function specified by `set_unexpected`
        - Used only if `bad_exception` is in the function's `throw` list

Common Programming Error 16.8

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.

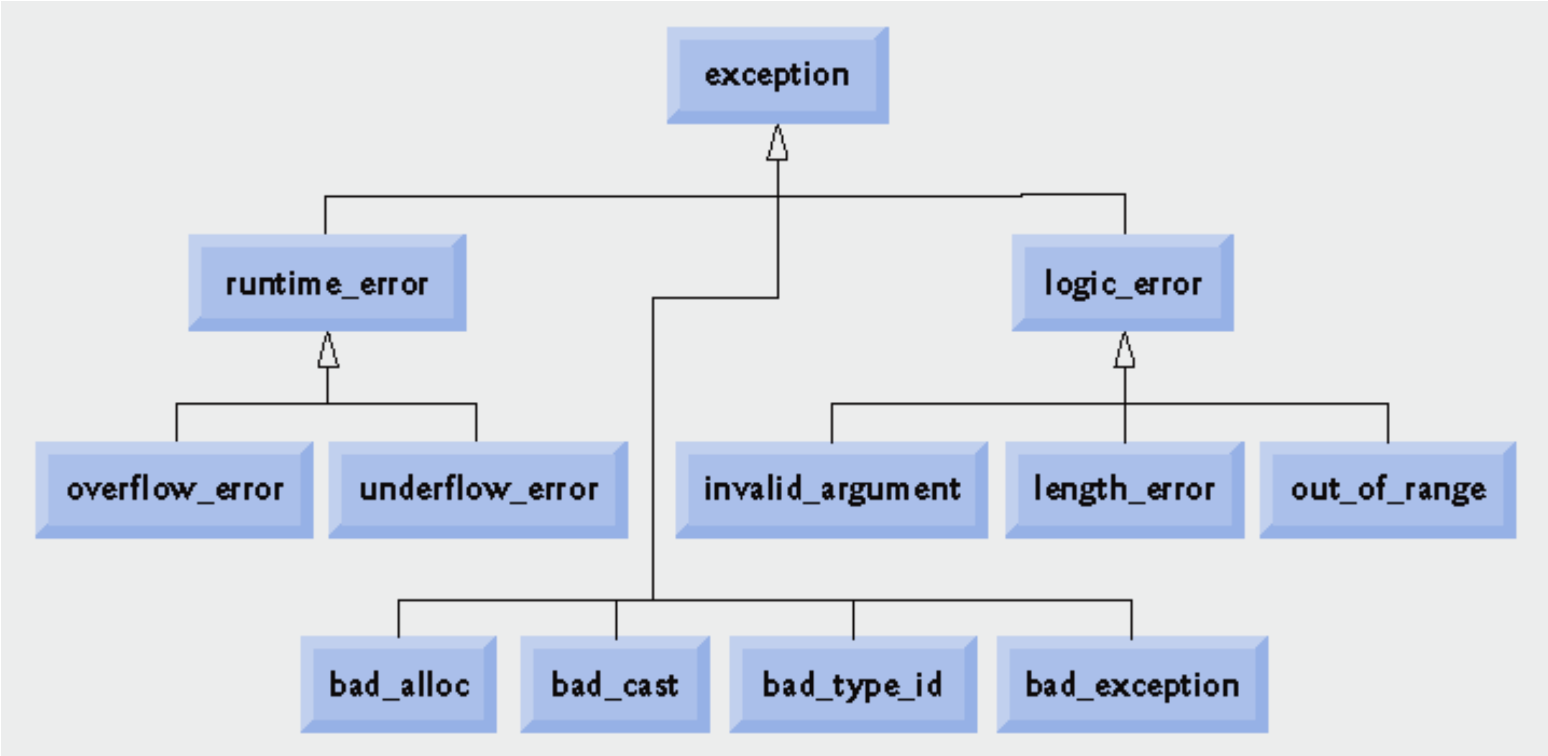

Fig. 16.11 | **Standard Library exception classes.**

16.13 Standard Library Exception Hierarchy (Cont.)

- Exception hierarchy classes (Cont.)
  - Class `logic_error`, derived from `exception`
    - Indicates errors in program logic
    - Exception classes derived from `logic_error`
      - `invalid_argument`
        - Indicates an invalid argument to a function
      - `length_error`
        - Indicates a length larger than the maximum size for some object was used
      - `out_of_range`
        - Indicates a value, such as an array subscript, exceeded its allowed range

16.13 Standard Library Exception Hierarchy (Cont.)

- **Exception hierarchy classes (Cont.)**
  - Class `runtime_error`, derived from `exception`
    - Indicates execution-time errors
    - Exception classes derived from `runtime_error`
      - `overflow_error`
        - Indicates an arithmetic overflow error – an arithmetic result is larger than the largest storable number
      - `underflow_error`
        - Indicates an arithmetic underflow error – an arithmetic result is smaller than the smallest storable number

Common Programming Error 16.9

Programmer-defined exception classes need not be derived from class `exception`. Thus, writing `catch( exception anyException )` 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. Programmers can build programs that can `throw` standard exceptions, `throw` exceptions derived from the standard exceptions or `throw` their 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

- Other error-handling techniques
  - Ignore the exception
    - Devastating for commercial and mission-critical software
  - Abort the program
    - Prevents a program from giving users incorrect results
    - Inappropriate for mission-critical applications
    - Should release acquired resources before aborting
  - Set error indicators
  - Issue an error message and pass an appropriate error code through `exit` to the program's environment

Common Programming Error 16.10

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."

16.14 Other Error-Handling Techniques (Cont.)

- **Other error-handling techniques (Cont.)**
  - Use functions `setjump` and `longjump`
    - Defined in library `<csetjmp>`
    - Used to jump immediately from a deeply nested function call to an error handler
      - Unwind the stack without calling destructors for automatic objects
  - Use a dedicated error-handling capability
    - Such as a `new_handler` function registered with `set_new_handler` for operator `new`