Chapter 16
Exception Handling

Yu-Shuen Wang, CS, NCTU

16.1 Introduction

- 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.

16.1 Introduction (cont.)

- 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
...
```

- 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.
- 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

- Here we 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.)

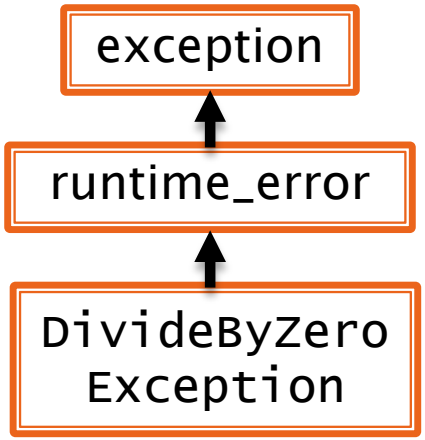
- If the divisor is zero, the function uses an exception to indicate to the caller that a problem occurred.
- The caller (`main` in this example) can then process the exception and allow the user to type two new values before calling function `quotient` again.
- In this way, the program can continue 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.)

- 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.

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 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.



```
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.

```
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, which consists of keyword `try` followed by braces `{}`.
- The `try` block encloses statements that might cause exceptions and statements that should be skipped if an exception occurs.

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, which catch and handle exceptions.
- At least one `catch` handler 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.

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

- 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.
- 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 the above 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.
- Next, the program searches for the first `catch` handler that can process 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 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 go out of scope.
- Control resumes with the first statement after the last `catch` handler following the `try` block.
- This is 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.

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

- If the `try` block completes its execution with no exceptions, 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.

Performance Tip 16.2

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

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, division by zero, invalid function parameters and unsuccessful memory allocation.

16.4 When to Use Exception Handling

- 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 16.4

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

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.*

16.5 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 to another exception handler.
- In either case, you achieve this by rethrowing the exception via the statement `throw`.

16.5 Rethrowing an Exception

- 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 16.6

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

```
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.

16.6 Exception Specifications (cont.)

- 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.

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.

16.7 Processing Unexpected Exceptions

- 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 is no exception currently being handled
  4. a call to function `unexpected` defaults to calling function `terminate`

16.7 Processing Unexpected Exceptions

- 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

- 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.

16.7 Processing Unexpected Exceptions

```
1  // set_unexpected example
2  #include <iostream>
3  #include <exception>
4  using namespace std;
5
6  void myunexpected () {
7    cerr << "unexpected called\n";
8    throw 0;      // throws int (in exception-specification)
9  }
10
11 void myfunction () throw (int) {
12   throw 'x';   // throws char (not in exception-specification)
13 }
14
15 int main (void) {
16   set_unexpected (myunexpected);
17   try {
18     myfunction();
19   }
20   catch (int) { cerr << "caught int\n"; }
21   catch (...) { cerr << "caught other exception (non-compliant compiler?)\n"; }
22   return 0;
23 }
```

Output:

```
unexpected called
caught int
```

16.7 Processing Unexpected Exceptions

```
1  // set_terminate example
2  #include <iostream>
3  #include <exception>
4  #include <cstdlib>
5  using namespace std;
6
7  void myterminate () {
8    cerr << "terminate handler called\n";
9    abort();  // forces abnormal termination
10 }
11
12 int main (void) {
13   set_terminate (myterminate);
14   throw 0;  // unhandled exception: calls terminate handler
15   return 0;
16 }
```

Possible output:

```
terminate handler called
Aborted
```

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.
- If no `catch` handler ever catches this exception, function `terminate` is called to terminate the program.

```
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.)

16.9 Constructors, Destructors and Exception Handling

- An object's constructor respond when "new" fails because it was unable to allocate required memory.
- The constructor cannot return a value to indicate an error.

16.9 Constructors, Destructors and Exception Handling (cont.)

- Offering an opportunity for the constructor to handle the failure.
- Before an exception is thrown by a constructor, destructors are called for every automatic object constructed in a `try` block.

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.
- When an exception occurs, the destructor for that object will be invoked and can free the resource.

```
class MyException
{
private:
        string msg;

public:
        MyException(string str)          { msg = str; }
        void printerrmsg()               { cout<<msg.c_str()<<endl; }
};

class A
{
private:
        int i;
        A() {
                i = 10;
                throw MyException("Exception thrown in constructor of A()");
        }
};
```

```
void main()
{
        try
        {
                A();
        }
        catch(MyException& e)
        {
                e.printerrmsg();
        }
}
```

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>`).

```
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.)

- 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.

16.11 Processing new Failures (cont.)

- In old versions of C++, operator `new` returned 0 when it failed to allocate memory.
- 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 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` 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 in the program, operator `new` does not throw `bad_alloc` on failure; rather, it defers the error handling to the `new`-handler function.

16.11 Processing new Failures (cont.)

- 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 file `<cstdlib>`) to terminate the program.

```
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.
- The C++ standard provides class template `auto_ptr` in header file `<memory>` to release memory automatically.

16.12 Class auto_ptr and Dynamic Memory Allocation (cont.)

- When an `auto_ptr` object destructor is called, it performs a `delete` operation on its pointer data member.
  - Usually when an `auto_ptr` object goes out of scope
- Class template `auto_ptr` provides overloaded operators `*` and `->` so that an `auto_ptr` object can be used just as a regular pointer variable is.

```
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.

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.

16.13 Standard Library Exception Hierarchy

- Experience has shown that exceptions fall nicely into a number of categories.
- 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.

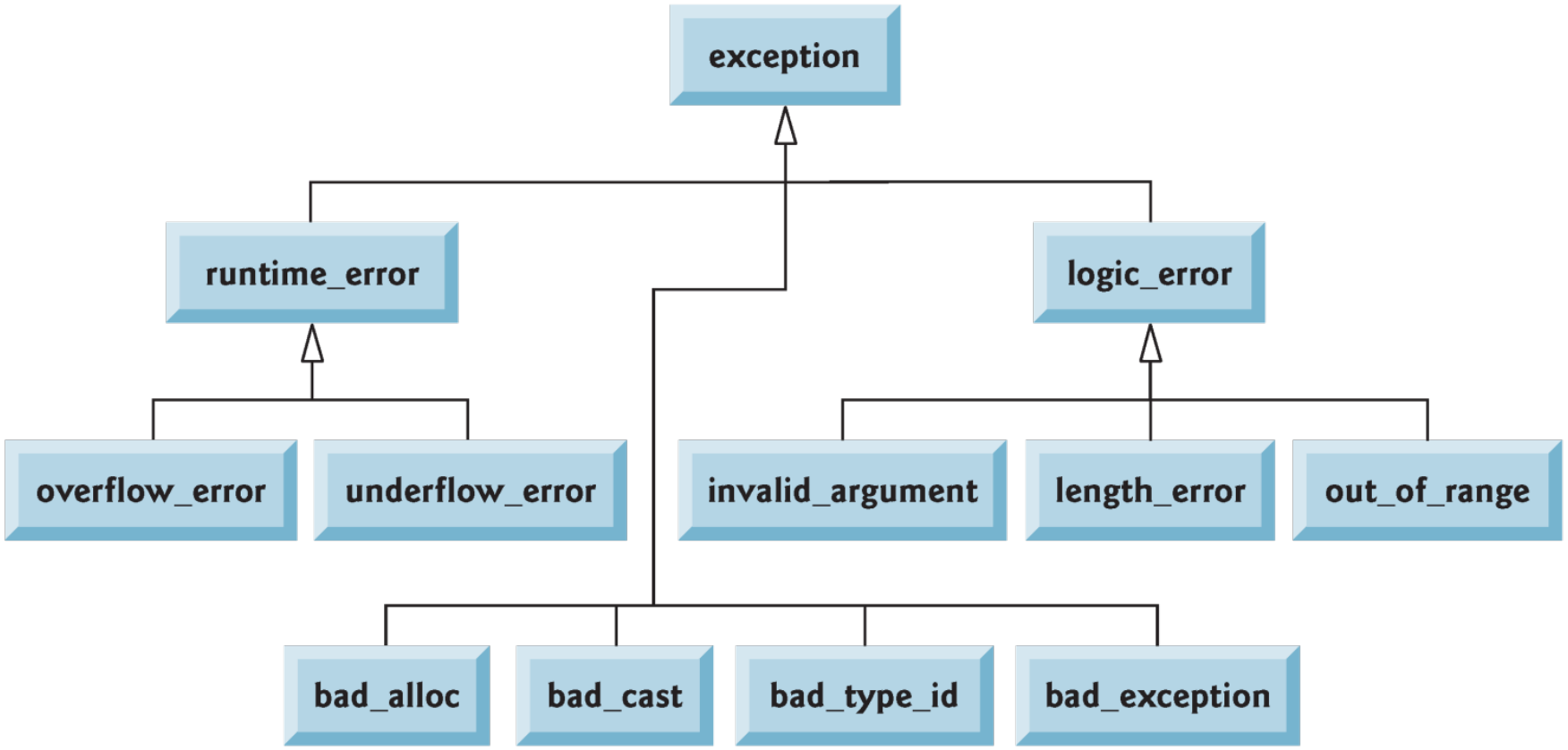


Fig. 16.10 | Some of the Standard Library exception classes.

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`, `bad_cast` is thrown by `dynamic_cast` and `bad_typeid` is thrown by `typeid`.

```
class Polymorphic {
        virtual void Member()        {}
};

int main ()
{
        try {
                Polymorphic * pb = 0;
                std::cout << typeid(*pb).name();
        }
        catch (std::bad_typeid& bt)
        {
                std::cerr << "bad_typeid caught: " << bt.what() << '\n';
        }

        return 0;
}
```

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.
- Class length_error indicates that a length larger than 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 as a subscript into an array, exceeded its allowed range of values.

```
// length_error example
#include <iostream>       // std::cerr
#include <stdexcept>      // std::length_error
#include <vector>         // std::vector

int main (void) {
  try {
    // vector throws a length_error if resized above max_size
    std::vector<int> myvector;
    myvector.resize(myvector.max_size()+1);
  }
  catch (const std::length_error& le) {
            std::cerr << "Length error: " << le.what() << '\n';
  }
  return 0;
}
```

16.13 Standard Library Exception Hierarchy (cont.)

- Class `runtime_error` is the base class of several other standard exception classes that indicate execution-time errors.
  - `overflow_error`
  - `underflow_error`

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 16.11

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

- The following summarizes these and other error-handling techniques:
  1. If an exception occurs, the program might fail as a result of the uncaught exception.
  2. But, for software developed for your own purposes, ignoring many kinds of errors is common.
  3. Abort the program to prevent a program from running to completion and producing incorrect results.
     - potentially misleading you to think that the program functioned correctly.