Lecture 33:
Exception Handling

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
May 24th, 2010

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.

16.11 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.*

16.13 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.

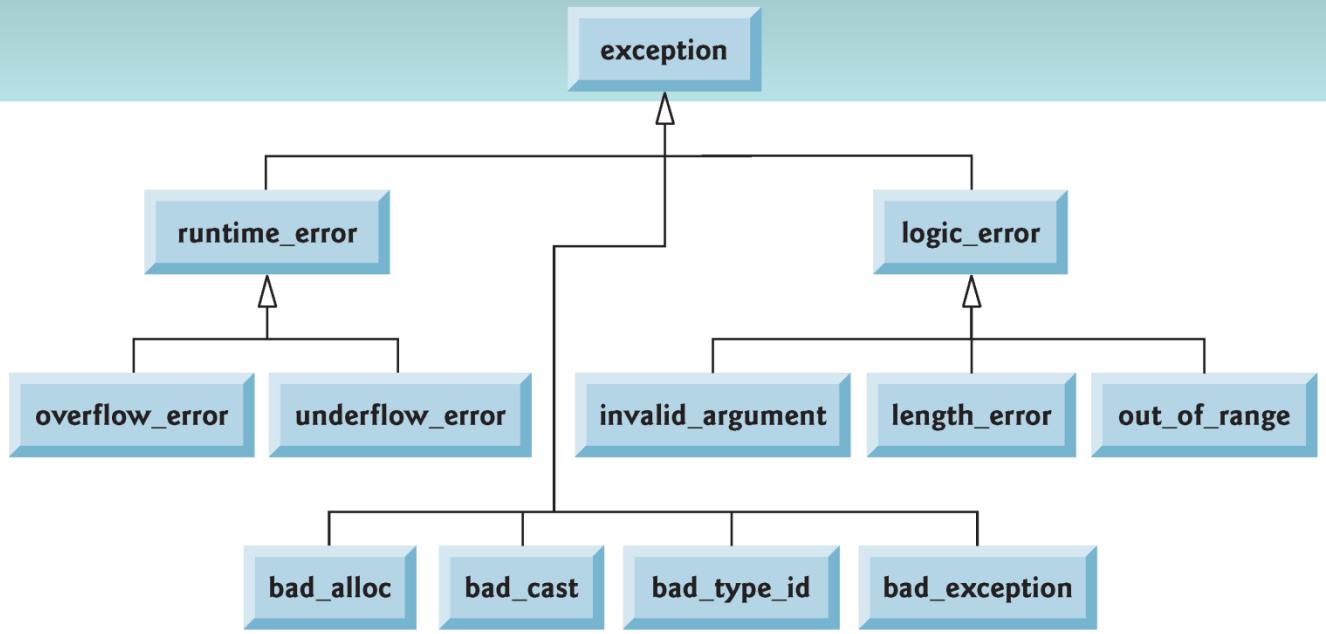

Fig. 16.10 | Some of the Standard Library exception classes.

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`.

Questions