17

Exception Handling: A Deeper Look

Objectives

In this chapter you'll:

- Understand the exception-handling flow of control with `try`, `catch` and `throw`.
- Define new exception classes.
- Understand how stack unwinding enables exceptions not caught in one scope to be caught in another.
- Handle `new` failures.
- Use `unique_ptr` (a Standard Library "smart pointer" class) to prevent memory leaks.
- Understand the standard exception hierarchy.

Self-Review Exercises

17.1 List five common examples of exceptions.

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

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

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

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

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

17.4 What's a "resource leak"?

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

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

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

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

ANS: An exception thrown outside a `try` block causes stack unwinding.

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

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

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

ANS: This causes the search for a match to continue in the next enclosing `try` block if there is one. As this process continues, it might eventually be determined that there is no handler in the program that matches the type of the thrown object; in this case, the program terminates.

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

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

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

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

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

ANS: If a base-class handler is defined before a derived-class handler, the base-class handler would catch objects of all derived-class types.

17.12 Must throwing an exception cause program termination?

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

17.13 What happens when a `catch` handler `throw`s an exception?

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

17.14 What does the statement `throw;` do?

ANS: It rethrows the exception if it appears in a `catch` handler; otherwise, the program terminates.

Exercises

NOTE: Solutions to the programming exercises are located in the **`ch17solutions`** *folder.*

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

ANS: A few examples are: **Division by zero**—check the denominator to see if it is equal to zero, and throw an exception before the division occurs. **Array subscript out of bounds**—catch the exception, print an error message telling the user what index was trying to be referenced, and exit the program in a controlled manner. **Exhaustion of the free store**—catch the exception, and either deallocate enough other objects so that you can allocate memory for the new object or delete all objects and terminate the program. **Bad cast**—catch the exception and either cast the operand to the proper type, if that can be determined, or print an error message indicating what the bad cast was, and exit the program.

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

ANS: If there is no information in the object that is required in the handler, a parameter name is not required in the handler.

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

```
throw;
```

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

ANS: The statement would be found in an exception handler to rethrow an exception. If any `throw;` expression occurs outside a `catch` block, the function `unexpected` is called.

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

ANS: Exception handling enables the programmer to build more robust classes with built-in error-processing capabilities. Once created, such classes allow their clients to concentrate on using the classes rather than defining what should happen if an error occurs. Exception handling also allows a program that should not continue to exit in a controlled manner, returning any resources that it might have obtained, therefore preventing possible resource leaks, and it allows the program to display meaningful messages about the problem, instead of just terminating. Exception handling provides a single, uniform technique for processing errors; this helps programmers working on large projects understand each other's error-processing code.

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

ANS: Exception handling is designed to handle infrequently occurring situations that often result in program termination. Exceptions do not follow conventional forms of program control. Handling a larger number of exception cases can be cumbersome, and programs with a large number of exception cases can be difficult to read and maintain.

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

ANS: Create a base class for all related exceptions. From the base class, derive related exception classes. Once the exception class hierarchy is created, all exceptions from the hierarchy can be caught as the base class exception type.