



GLOBAL
EDITION



Chapter 18

Exception Handling

Absolute C++

SIXTH EDITION
Walter Savitch

ALWAYS LEARNING

PEARSON

Copyright © 2017 Pearson Education, Ltd.
All rights reserved.

Learning Objectives

- Exception Handling Basics
 - Defining exception classes
 - Multiple throws and catches
 - Exception specifications
- Programming Techniques for Exception Handling
 - When to throw exceptions
 - Exception class hierarchies

Introduction

- Typical approach to development:
 - Write programs assuming things go as planned
 - Get "core" working
 - Then take care of "exceptional" cases
- C++ exception-handling facilities
 - Handle "exceptional" situations
 - Mechanism "signals" unusual happening
 - Another place in code "deals" with exception

Exception-Handling Basics

- Meant to be used sparingly
 - In "involved" situations
- Difficult to teach such large examples
- Approach:
 - Simple toy examples, that would not normally use exception-handling
 - Keep in mind "big picture"

Toy Example

- Imagine: people rarely run out of milk:
cout << "Enter number of donuts:";
cin >> donuts;
cout << "Enter number of glasses of milk:";
cin >> milk
dpg = donuts/static_cast<double>(milk);
cout << donuts << "donuts.\n";
 << milk << "glasses of milk.\n";
 << "You have " << dpg
 << "donuts for each glass of milk.\n";
- Basic code assumes never run out of milk

Toy Example if-else

- Notice: If no milk → divide by zero error!
- Program should accommodate unlikely situation of running out of milk
 - Can use simple if-else structure:

```
if (milk <= 0)
    cout << "Go buy some milk!\n";
else
    {...}
```
- Notice: no exception-handling here

Toy Example with Exception Handling: **Display 18.2** Same Thing Using Exception Handling

```
9      try
10     {
11         cout << "Enter number of donuts:\n";
12         cin >> donuts;
13         cout << "Enter number of glasses of milk:\n";
14         cin >> milk;
15
16         if (milk <= 0)
17             throw donuts;
18
19         dpg = donuts/static_cast<double>(milk);
20         cout << donuts << " donuts.\n"
21             << milk << " glasses of milk.\n"
22             << "You have " << dpg
23             << " donuts for each glass of milk.\n";
24     }
25     catch(int e)
26     {
27         cout << e << " donuts, and No Milk!\n"
28             << "Go buy some milk.\n";
29     }
```

Toy Example Discussion

- Code between keywords *try* and *catch*
 - Same code from ordinary version, except if statement simpler:
if (milk <= 0)
 throw donuts;
 - Much cleaner code
 - If "no milk" → do something exceptional
- The "something exceptional" is provided after keyword *catch*

Toy Example try-catch

- Try block
 - Handles "normal" situation
- Catch block
 - Handles "exceptional" situations
- Provides separation of normal from exceptional
 - Not big deal for this simple example, but important concept

try block

- Basic method of exception-handling is try-throw-catch
- Try block:

```
try
{
    Some_Code;
}
```

 - Contains code for basic algorithm when all goes smoothly

throw

- Inside try-block, when something unusual happens:

```
try
{
    Code_To_Try
    if (exceptional_happened)
        throw donuts;
    More_Code
}
```

- Keyword *throw* followed by exception type
- Called "throwing an exception"

catch-block

- When something thrown → goes somewhere
 - In C++, flow of control goes from try-block to catch-block
 - try-block is "exited" and control passes to catch-block
 - Executing catch block called "catching the exception"
- Exceptions must be "handled" in some catch block

catch-block More

- Recall:

```
catch(int e)
{
    cout << e << " donuts, and no milk!\n";
    << " Go buy some milk.\n";
}
```
- Looks like function definition with int parameter!
 - Not a function, but works similarly
 - Throw like "function call"

catch-block Parameter

- Recall: `catch(int e)`
- "e" called catch-block parameter
 - Each catch block can have at most ONE catch-block parameter
- Does two things:
 1. type name specifies what kind of thrown value the catch-block can catch
 2. Provides name for thrown value caught; can "do things" with value

Defining Exception Classes

- throw statement can throw value of any type
- Exception class
 - Contains objects with information to be thrown
 - Can have different types identifying each possible exceptional situation
 - Still just a class
 - An "exception class" due to how it's used

Exception Class for Toy Example

- Consider:
class NoMilk
{
public:
 NoMilk() { }
 NoMilk(int howMany) : count(howMany) { }
 int getcount() const { return count; }
private:
 int count;
};
- throw NoMilk(donuts);
 - Invokes constructor of NoMilk class

Multiple Throws and Catches

- try-block typically throws any number of exception values, of differing types
- Of course only one exception thrown
 - Since throw statement ends try-block
- But different types can be thrown
 - Each catch block only catches "one type"
 - Typical to place many catch-blocks after each try-block
 - To catch "all-possible" exceptions to be thrown

Catching

- Order of catch blocks important
- Catch-blocks tried "in order" after try-block
 - First match handles it!
- Consider:
catch (...) { }
 - Called "catch-all", "default" exception handler
 - Catches any exception
 - Ensure catch-all placed AFTER more specific exceptions!
 - Or others will never be caught!

Trivial Exception Classes

- Consider:
class DivideByZero
{ }
- No member variables
- No member functions (except default constructor)
- Nothing but it's name, which is enough
 - Might be "nothing to do" with exception value
 - Used simply to "get to" catch block
 - Can omit catch block parameter

Throwing Exception in Function

- Function might throw exception
- Callers might have different "reactions"
 - Some might desire to "end program"
 - Some might continue, or do something else
- Makes sense to "catch" exception in calling function's try-catch-block
 - Place call inside try-block
 - Handle in catch-block after try-block

Throwing Exception in Function Example

- Consider:

```
try
{
    quotient = safeDivide(num, den);
}
catch (DivideByZero)
{ ... }
```

- `safeDivide()` function throws `DividebyZero` exception
 - Handled back in caller's catch-block

Exception Specification

- Functions that don't catch exceptions
 - Should "warn" users that it could throw
 - But it won't catch!
- Should list such exceptions:
`double safeDivide(int top, int bottom)`
`throw (DividebyZero);`
 - Called "exception specification" or "throw list"
 - Should be in declaration and definition
 - All types listed handled "normally"
 - If no throw list → all types considered there

Throw List

- If exception thrown in function NOT in throw list:
 - No errors (compile or run-time)
 - Function unexpected() automatically called
 - Default behavior is to terminate
 - Can modify behavior
- Same result if no catch-block found

Throw List Summary

- void someFunction()
 throw(DividebyZero, OtherException);
**//Exception types DividebyZero or OtherException
//treated normally. All others invoke unexpected()**
- void someFunction() throw ();
**//Empty exception list, all exceptions invoke
unexpected()**
- void someFunction();
//All exceptions of all types treated normally

Derived Classes

- Remember: derived class objects also objects of base class
- Consider:
D is derived class of B
- If B is in exception specification →
 - Class D thrown objects will also be treated normally, since it's also object of class B
- Note: does not do automatic type cast:
 - double will not account for throwing an int

unexpected()

- Default action: terminates program
 - No special includes or using directives
- Normally no need to redefine
- But you can:
 - Use `set_unexpected`
 - Consult compiler manual or advanced text for details

When to Throw Exceptions

- Typical to separate throws and catches
 - In separate functions
- Throwing function:
 - Include throw statements in definition
 - List exceptions in throw list
 - In both declaration and definition
- Catching function:
 - Different function, perhaps even in different file

Preferred throw-catch Triad: throw

- `void functionA() throw (MyException)`
 {
 ...
 throw MyException(arg);
 ...
 }
- Function throws exception as needed

Preferred throw-catch Triad: catch

- Then some other function:

```
void functionB()
{
    ...
    try
    {
        ...
        functionA();
        ...
    }
    catch (MyException e)
    { // Handle exception
    }
    ...
}
```

Uncaught Exceptions

- Should catch every exception thrown
- If not → program terminates
 - terminate() is called
- Recall for functions
 - If exception not in throw list: unexpected() is called
 - It in turn calls terminate()
- So same result

Overuse of Exceptions

- Exceptions alter flow of control
 - Similar to old "goto" construct
 - "Unrestricted" flow of control
- Should be used sparingly
- Good rule:
 - If desire a "throw": consider how to write program without throw
 - If alternative reasonable → do it

Exception Class Hierarchies

- Useful to have; consider:
DivideByZero class derives from:
 ArithmeticError exception class
 - All catch-blocks for ArithmeticError also catch DivideByZero
 - If ArithmeticError in throw list, then DividebyZero also considered there

Testing Available Memory

- new operator throws bad_alloc exception if insufficient memory:

```
try
{
    NodePtr pointer = new Node;
}
catch (bad_alloc)
{
    cout << "Ran out of memory!";
    // Can do other things here as well...
}
```

- In library <new>, std namespace

Rethrowing an Exception

- Legal to throw exception IN catch-block!
 - Typically only in rare cases
- Throws to catch-block "farther up chain"
- Can re-throw same or new exception
 - rethrow;
 - Throws same exception again
 - throw newExceptionUp;
 - Throws new exception to next catch-block

Example – High Score

- Throwing an exception in a function is especially helpful when the exception has no relation to the return value of the function.
- Consider a function that scans through a text file of high scores and returns the highest score.
 - What should the function return if the file cannot be opened?
 - One strategy is to return a special value, such as a negative number.

High Score – No Exception Handling (1 of 3)

```
// Program that outputs the high score from a high scores file.  
// Does not use exception handling.  
#include <iostream>  
#include <fstream>  
#include <string>  
using std::cout;  
using std::endl;  
using std::ifstream;  
  
//Function prototypes  
int getHighscore();
```

High Score – No Exception Handling (2 of 3)

// Returns the high score in the scores.txt file

```
int getHighscore()
```

```
{
```

```
    ifstream f;
```

```
    int high = -1;
```

```
    f.open("scores.txt");
```

```
    // Check if the file did not open
```

```
    if (f.fail())
```

```
    {
```

```
        cout << "File could not be opened." << endl;
```

```
        return -1;
```

```
    }
```

```
    int num;
```

```
    // Scan through each number in the file and return the largest
```

```
    f >> high;
```

```
    while (f >> num)
```

```
    {
```

```
        if (num > high)
```

```
            high = num;
```

```
    }
```

```
    f.close();
```

```
    return high;
```

```
} Copyright © 2017 Pearson Education, Ltd. All rights reserved.
```

High Score – No Exception Handling (3 of 3)

```
int main()
{
    int highscore = getHighscore();
    cout << "The high score is " << highscore << endl;
    return 0;
}
```

Sample Dialogue 1 (file exists with values 10, 50, 30)

The high score is 50

Sample Dialogue 2 (the file does not exist)

File could not be opened.

The high score is -1

But what if negative scores are possible? No way to distinguish high scores from no scores!

High Score Solution – Throw Exception (1 of 3)

- Throw an exception if there is an IO error and catch it in main

```
// Program that outputs the high score from a high scores file.
```

```
// Uses exception handling.
```

```
#include <iostream>
```

```
#include <fstream>
```

```
#include <string>
```

```
using std::cout;
```

```
using std::endl;
```

```
using std::ifstream;
```

```
class FileIOError
```

```
{};
```

```
//Function prototypes
```

```
int getHighscore() throw (FileIOError);
```

High Score Solution – Throw Exception (2 of 3)

```
/ Returns the high score in the scores.txt file
// but throws an exception if the file could not be opened.
// This eliminates possible confusion over the return value.
int getHighscore() throw (FileNotFoundException)
{
    ifstream f;
    int high = -1;

    f.open("cores.txt");
    // Check if the file did not open
    if (f.fail())
    {
        throw FileNotFoundException();
    }
    int num;
    // Scan through each number in the file and return the largest
    f >> high;
    while (f >> num)
    {
        if (num > high)
            high = num;
    }
    f.close();
    return high;
}
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

High Score Solution – Throw Exception (3 of 3)

```
int main()
{
    try
    {
        int highscore = getHighscore();
        cout << "The high score is " << highscore << endl;
    }
    catch (FileIOError)
    {
        cout << "Could not open the scores file." << endl;
    }
    return 0;
}
```

Summary 1

- Exception handling allows separation of "normal" cases and "exceptional" cases
- Exceptions thrown in try-block
 - Or within a function whose call is in try-block
- Exceptions caught in catch-block
- try-blocks typically followed by more than one catch-block
 - List more specific exceptions first

Summary 2

- Best used with separate functions
 - Especially considering callers might handle differently
- Exceptions thrown in but not caught in function, should be listed in throw list
- Exceptions thrown but never caught → program terminates
- Resist overuse of exceptions
 - Unrestricted flow of control