Lesson 14 - Exception Handling
Recovering From Errors

Summary

In this lesson we describe the C++ Exception mechanism; the throwing and handling of exceptions to detect and recover from erroneous program execution.

New Concepts

Exceptions, try/catch blocks, reading from a file.

Exceptions

In this lesson we describe the Exception mechanism provided by C++ for handling run-time errors that invariably crop up during the execution of sophisticated programs. Exception handling can prove rather complex sometimes, so we'll start by explaining why we need an Exception mechanism, before we talk about how it works.

In C Programs, the language predecessor of C++, the method of handling errors at run-time involved returning an integer code from a function. The integer code (or error code) would inform the calling function that something had gone awry in a certain way. For example, let's suppose a function attempted to perform division by zero, which mathematically speaking is not a defined operation. The function could detect the zero divisor and return an integer code perhaps signalling a 'bad argument' to the function or a 'bad arithmetic instruction' requested. Problems arise with C style return codes however. The first issue is consistency, where some functions may return the value 1 for success while others 0. Returning an integer from a function as a return code prevents the function from returning any other value too, and return codes can be ignored by an inexperienced or belligerent programmer.

The Exception Handling mechanism has been incorporated into C++ to address the short-comings of the return-code method. Exceptions provide a consistent but versatile method of handling errors. In addition they do not require or preclude functions (or methods) from returning integer values. Finally, an exception that is 'thrown' cannot be ignored by the executing program. All that said, however, Exceptions have their problems too: Exception handling can be complicated; memory-leaks can be introduced without due care and a slight performance penalty may be incurred. You can avoid them altogether and handle errors 'the C way' with a few modifications to code and compilation. Bear in mind, however, that certain C++ library functions use Exceptions, so either way it makes sense to acquaint yourself with them.

Without further ado, let's demonstrate Exception Handling in a real program, see the `exceptions.cpp` listing below. First we begin by including the `stdexcept` header file which contains the necessary header information for using Exceptions (line 3). Note also that we include the headers `string`, `fstream` and `stdlib.h` on lines 4 to 6. We'll explain `fstream` and `stdlib.h` shortly, but for now a word about Strings. The `string` header allows the use of the C++ `string` Class. A `string` is simply a sequence of characters forming a word or sentence. In the C language, strings are simply represented as an array of `char`s, but C++ provides an object representation for convenient operations on strings via class methods and overloaded operators. For instance, we can concatenate strings with the addition operator (`+`) or even compare strings with equality/greater-than/less-than etc (`=,>,<`). On line 10 we declare a constant global string to represent the name of a file we will use in the program.

```
1 #include <iostream>
2 #include <vector>
3 #include <stdexcept>
4 #include <string>
5 #include <fstream>
```

```
6  #include <stdlib.h>
7
8  using namespace std;
9
10 const string file_name = "data.txt";
```

exceptions.cpp

Next we define our first function `read_scores` on line 11. The purpose of this function is to open the file `data.txt`, via the `file_name` string. Then it will read in the contents of that file, storing the results to a vector called `scores`. Note that we pass the vector as a reference parameter to the function. On line 12 we create an `ifstream` object called `data_file` on the Stack. An `ifstream` object is an Input File Stream (the reason we needed to include the header `fstream` on line 5). An Input File Stream allows us to read into our executing program, the contents of a file stored on disk.

To read in the data, first we need to call the `open` method of `data_file`, our `ifstream` object (see line 15). Notice that we call the `c_str` method on our `file_name` string. This returns a C String representation of the file name which `data_file` requires for the `open` method. On line 20 we read in the contents of the file, but first we call the `fail` method on `data_file` to detect any errors when opening the file.

Take a look at line 18. If we fail to open the file (because it doesn't exist or we don't have the permission to open it), then the statement instructs the program to '`throw invalid_argument`'. This is the first part of Exception handling. We're saying if for some reason, we cannot open the file for reading, then we will create an Exception of type 'invalid_argument'. In the Constructor we provide a string describing the problem. We then 'throw' this Exception. If we reach this situation in our program, the function will immediately end, as if we had a `return` statement on line 18. We are 'throwing' the 'invalid_argument' exception out of the `read_scores` function. Note on line 11 we declare that `read_scores` can throw an `invalid_argument`. Once an Exception is thrown, we shall discover that we need to 'catch' it at some point in the program, or the program will terminate.

```
11 void read_scores(vector<int>& scores) throw (invalid_argument) {
12    ifstream data_file;
13    int temp;
14
15    data_file.open(file_name.c_str());
16
17    if(data_file.fail())
18       throw invalid_argument("no file exists " + file_name);
19
20    while(data_file >> temp)
21       scores.push_back(temp);
22
23    data_file.close();
24 }
```

exceptions.cpp

After the `read_scores` function we define another function called `find_average`. This time we state that the `find_average` function may throw a `runtime_error`, another type of Exception. On line 26 we test to see if the supplied divisor is zero. If so then we cannot perform the division. If we reach line 27 we 'throw' a `runtime_error`, immediately ending this function and skipping the statement on line 28.

```
25 double find_average(const int sum, const int divisor) throw (runtime_error) {
26    if(!divisor)
27       throw runtime_error("divisor is zero");
28    return (sum / (double) divisor);
29 }
```

exceptions.cpp

Now let's use these functions within our `main` method, to introduce the rest of the Exception Handling syntax. Note on line 34 we open a set of curly brackets with the `try` keyword. We are informing the compiler that we will 'try' the code within the brackets. If the enclosed code throws an Exception, we try and 'catch' the Exception within a subsequent `catch` block, which we define on lines 40 to 46. Exceptions can always be thrown within a program, but providing a 'try/catch' block is the only means of handling them without allowing the program to terminate.

Note that in the `try` block we call the `read_scores` and `find_average` functions. Recall that the `read_scores` function can throw an `invalid_argument` object. If this takes place then execution jumps to the `catch` statement on line 40. Notice that we stipulate the type of Exception we're

attempting to catch in brackets after the `catch` keyword, in this case a constant reference to an `invalid_argument` object. You can catch Exceptions by value, reference, const reference or pointer, but be sure if you throw an exception by value, you provide a corresponding `catch` statement that requires a value, reference or constant reference.

If an `invalid_argument` exception is caught on line 40, the program executes the statements in the subsequent `catch` block, denoted by the curly brackets. In this case we display a message to screen via `cout` explaining that we were unable to read the data. We also call the `what` method on the exception object. This returns the string which we gave to this object upon its construction, which you'll recall included the name of the file (see line 18). We then exit the program, using the `exit` library function defined in the `stdlib.h` file included on line 6. The value 1 passed to this function indicates that an error has occurred.

If we read the contents of the data file without error, then the next step of the program is to compute the sum of the scores (lines 36-38) and average by calling the `find_average` function (see line 39). Recall that if size is zero, then the `find_average` function will throw a `run_time` exception object (line 27). If this occurs then the corresponding `catch` statement on line 43 will be executed. Again we display to screen a helpful message to inform the user what went wrong before terminating the program.

```
30 int main() {
31    vector<int> scores;
32    int sum = 0;
33
34    try {
35       read_scores(scores);
36       for(int i = 0; i < scores.size(); ++i) {
37          sum += scores[i];
38       }
39       cout << "avg = " << find_average(sum, scores.size()) << "\n";
40    } catch (const invalid_argument& iae) {
41       cout << "unable to read data: " << iae.what() << "\n";
42       exit(1);
43    } catch (const runtime_error& rte) {
44       cout << "unable to compute average: " << rte.what() << "\n";
45       exit(1);
46    }
47
48    for(int i = 0; i < scores.size(); ++i)
49       cout << "score " << i << " = " << scores[i]  << "\n";
50
51    return 0;
52 }
```

exceptions.cpp

Finally a word on memory-leaks. When you declare any memory on the heap, bear in mind that throwing an exception before that memory is released may result in a memory-leak. Recall that when an exception is thrown, the remainder of the current function and the calling function is not executed. The exception is thrown until it is caught or the program terminates. See the `clean_up` listing below. On line 1-3 we define a function that simply throws a `runtime_error` exception. The problem is the `memory_leak` function defined on line 5 allocates memory on the heap for a String object (line 6). When it calls `throwing_ftn` this memory never gets released because the exception is thrown, terminating this function at line 8. Notice that the method declaration of `memory_leak` doesn't include a throw list like `throwing_ftn` does (line 1). In C++ this means, somewhat awkwardly that `memory_leak` can possibly throw any exception. To state that a function does not throw any exception you would provide a throw list with empty brackets after the function or method declaration (`throw ()`).

```
1  void throwing_ftn() throw (runtime_error) {
2     throw runtime_error("something went wrong\n");
3  }
4
5  void memory_leak() {
6     string* s = new string("hello");
7
8     throwing_ftn();
9
10    delete s;
11 }
```

clean_up

In lines 12-22 we provide the solution to this problem in the function `no_memory_leak`. See line 15 and notice when we call the `throwing_ftn` we enclose this call within a try/catch block. Even though

`no_memory_leak` may not know how to handle the exception, we can catch any exception using the '`(...)`' syntax on line 17. On line 18 we 'clean up' by calling `delete` on the String object, before re-throwing the exception to the calling function on line 19. Calling `throw` without an exception simply throws the last caught exception. If you use exceptions, always remember you need to clean up any allocated memory if there's a chance that an exception could be thrown which would result in the omission of corresponding delete statements.

```
12 void no_memory_leak() {
13    string* s = new string("hello");
14
15    try {
16       throwing_ftn();
17    } catch (...) {
18       delete s;
19       throw;
20    }
21    delete s;
22 }
```

clean_up

C++ Style

We have seen how the Exception Handling in C++ provides an optional mechanism to programmers when encountering exceptional and unexpected states ('C style' return codes are another). The way you employ these mechanisms comes down to your personal choice or style, but there are a few guidelines to help you.
Firstly, consider where it makes the most sense to deal with an exception. Functions and methods tend to be most readable, and easier to debug when they are written to complete one specific task. It's generally considered good practise, therefore, to separate the code which catches and handles an exception to the code which detects and throws it. A significant caveat to this rule, however, is ensuring that you release any allocated memory by possibly catching, deleting and re-throwing any exceptions.
Consider also, what would be the most reasonable approach to dealing with the various type of exceptions that may occur. How should your program respond, for example, to exhaustion of memory compared to a divide by zero? While the former is typically irrecoverable (and a symptom of greater problems such as a memory leak gone wild!) it may be possible to recover from the latter. Bear in mind that your game will provide a pretty frustrating experience for the player if your whole application simply terminates without the option to save any player data or return to a safe point in the code. Keep the end user in mind and, in most situations, if it's possible to recover from an exception, do so!

Exercises

1. Explore the documentation for your compiler so that you know how to create programs which do not use the exception handling mechanism.

2. Change the name of the file 'data.txt' and run the program, notice how the program responds as it now cannot locate the file.

3. Change the name of the file back to 'data.txt'. Amend line 39 so that the second argument of the `find_average` function is zero. Again, note the response of the divide by zero exception.

4. Create another data file called 'data2.txt'. In 'data2.txt' place 6 random integers values as in 'data.txt'. Now amend the code so that it reads in both files and adds the values in 'data.txt' and 'data2.txt' before computing the average.

5. Create your own exception class which is thrown in the event that the `size` of the scores vector is less than 10 after the values have been read and stored in the scores vector. Amend the code to include a test for this exception and test that it works.