Lab 1: Exception Handling

Information

In-class labs are meant to introduce you to a new topic and provide some practice with that new topic.

Topics: Exception handling

Solo work: Labs should be worked on by each individual student, though asking others for help is permitted. Do not copy code from other sources, and do not give your code to other students. **Students who commit or aid in plagiarism will receive a 0% on the assignment and be reported.**

Building and running: If you are using Visual Studio, make sure to run with debugging. **Do not run without debugging!**

Using the debugger will help you find errors.
To prevent a program exit, use this before `return 0;`

```
cin.ignore();      cin.get();
```

Turn in: Once you're ready to turn in your code, prepare the files by doing the following: **(1)** Make a copy of your project folder and name it `LASTNAME-FIRSTNAME-LABNAME`. (Example: `HOPPER-GRACE-LAB-UNIT-TESTS`) **(2)** Make sure that all source files (.cpp, .hpp, and/or .h files) and the `Makefile` files are all present. **(3)** Remove all Visual Studio files - I only want the source files and Makefiles. **(4)** Zip your project folder as `LASTNAME-FIRSTNAME-LABNAME.zip`

Never turn in Visual Studio files!

Starter files: Download from GitHub.

Grading: Grading is based on completion, if the program functions as intended, and absense of errors. **Programs that don't build will receive 0%.** Besides build errors, runtime errors, logic errors, memory leaks, and ugly code will reduce your score.

Contents

1.1 About 3
    1.1.1 Setting up the project 3
1.2 Lab specifications 4
    1.2.1 Adding error checking 4
    1.2.2 Adding try/catch 6

1.1 About

Exception handling helps us design better code that is intended to be reused over and over. When creating a data structure that other programmers will be using, it can be important to report errors with your structure to them so that they can decide how to resolve the error; you can't always make the best design decisions on your own, it's up to each programmer to figure out the needs for their project.

You will start with the following code, which contains an object that handles file streams, as well as the main source file that contains tests. At the moment, it should compile and run, but it will crash during the tests because the RecordManager object doesn't have error checking.

In this lab, you will implement the exception handling within the RecordManager, as well as checking for exceptions from the main file.

1.1.1 Setting up the project

Download the starter zip file, `LAB-EXCEPTION-HANDLING.zip`, off GitHub. This contains:

- `RecordManager.hpp`
- `main.cpp`

1.2 Lab specifications

1.2.1 Adding error checking

Within the RecordManager class functions, you will add error checking and `throw` commands. Outside the class, in main and the tests, you will add the `try/catch` blocks.

Throw, try, and catch

INSIDE a function is where the `throw` command is used. After checking for some error state, the response is to `throw` the type of exception that has occurred, as well as an error message.

OUTSIDE a function (at the function-call level) is where `try` and `catch` commands are used. When a function that may throw an exception is being called, it should be wrapped within a `try` block. Then, the `catch` block(s) follow, to catch different types of exceptions, resolve them, clean up, and allow the program to continue.

Updating RecordManager::OpenOutputFile

In this function, add an error check before `m_outputs[ index ]` is accessed: Check to see if the *index* value is equal to -1. If so, throw a `runtime_error` like this:

```
if ( index == -1 )
{
    throw runtime_error( "No available files" );
}
```

Updating RecordManager::CloseOutputFile

This function should also throw a `runtime_error` if the returned index is -1.

Updating RecordManager::WriteToFile

This function should also throw a `runtime_error` if the returned index is -1.

Updating RecordManager::DisplayAllOpenFiles

This function won't cause any exceptions. Therefore, you should add the function specifier, `noexcept`, to the end of the signature - both in the declaration and the definition.

Updating RecordManager::FindAvailableFile

Add the `noexcept` function specifier.

Updating RecordManager::FindFilenameIndex

Add the `noexcept` function specifier.

Testing it out

Now when you run the program, it won't flat out crash like before, but it will abort the program once the first exception is hit.

```
-------------------------------------
TEST 1: Open one file and write to it

Open files:
0. Test1.txt

END OF TEST 1

-------------------------------------
TEST 2: Open 5 files and write to them

Open files:
0. Test2_A.txt
1. Test2_B.txt
2. Test2_C.txt
3. Test2_D.txt
4. Test2_E.txt

END OF TEST 2

-------------------------------------
TEST 3: Write to a file that isn't opened

Open files:
terminate called after throwing an instance of 'std::
   runtime_error'
  what():  File Test2.txt is not open
Aborted
```

(Note: This is the example output for the program running in Linux, with the g++ compiler. The result text might look different in Visual Studio.)

Next, `try/catch` blocks will need to be added in our tests so that the program will complete its execution, even if exceptions are discovered.

1.2.2 Adding try/catch

When we are calling a function that may cause an exception, that's when we should have a `try/catch` statement. The functions from `RecordManager` that may cause exceptions are:

- `OpenOutputFile`
- `CloseOutputFile`
- `WriteToFile`

Test1 and Test2 themselves won't cause any exceptions to be thrown, but if we were to add the try/catch to Test1, it would look like this:

```
// (...)
try
{
    record.DisplayAllOpenFiles();

    record.WriteToFile( "Test1.txt", "Hello world!" );

    record.CloseOutputFile( "Test1.txt" );
}
catch( runtime_error& ex )
{
    cout << "Error: " << ex.what() << endl;
}
// (...)
```

We could wrap each individual function in a try/catch, or wrap all three of them together. This is a design decision to make. In general, it is considered good design to wrap your try/catch around the smallest possible amount of code, but it also doesn't need to wrap just one line at a time. Since these three functions are related, we can wrap the three in a single try/catch block and it would be fairly clean.

Mainly, don't wrap an entire function in a try/catch - only a section working with "risky" areas.

Updating Test3, Test4, and Test5

For each of these tests, surround only the function calls that may return with an exception. Remember that any functions marked as `noexcept` will never throw an exception.

Once you've updated the entire program, the output should look like this:

```
-------------------------------------
TEST 1: Open one file and write to it

Open files:
0. Test1.txt

END OF TEST 1

-------------------------------------
TEST 2: Open 5 files and write to them

Open files:
0. Test2_A.txt
1. Test2_B.txt
2. Test2_C.txt
3. Test2_D.txt
4. Test2_E.txt

END OF TEST 2

-------------------------------------
TEST 3: Write to a file that isn't opened

Open files:
Error: File Test2.txt is not open

END OF TEST 3

-------------------------------------
TEST 4: Close a file that isn't opened

Open files:
Error: File Test3.txt is not open
```

```

END OF TEST 4

-------------------------------------
TEST 5: Try to open more than max # of files

Error: No available files
Open files:
0. Test5_A.txt
1. Test5_B.txt
2. Test5_C.txt
3. Test5_D.txt
4. Test5_E.txt

END OF TEST 5
```