**CS273 ASSIGNMENT #3: Exception Handling and Unit-Testing**

## MY NAME:

**Grade:**

|  |  |  |
| --- | --- | --- |
| **CATEGORY** | **POINTS** |  |
| EX03\_1 |  | 30 |
| EX03\_2 |  | 70 |
| **TOTAL** |  | 100 |

**Objectives:**

* **Gain hands-on practice with the concepts of exception handling and unit-testing.**
* Understand how to use the **standard exceptions** that are available in C++.
* Understand how to handle **user input** in **unit-testing**

**Please read Sections 2.2 and 2.3 of the textbook.**

**EX03\_1:** Complete programming exercise 2 (page 147). HINT: reuse the example **read\_int()**function that is already defined for you in example 2.8 (pages 144 & 145).

Define the prototype of the new function in **readint.h**:

#ifndef READINT\_H\_

#define READINT\_H\_

#include <string>

int read\_int(const std::string &prompt, int low, int high);

#endif

Implement your new function in **readint.cpp**:

#include "readint.h"

#include <stdexcept>

using namespace std;

int read\_int(const string &prompt, int low, int high)

{

// Implement this

}

You should use your **read\_int()** function in the following **main** function:

#include <iostream>

#include <stdexcept>

#include "readint.h"

using namespace std;

int main()

{

int low, high;

cout << "Enter the range of values to read.\n";

cout << "high: ";

cin >> high;

cout << "low: ";

cin >> low;

try

{

int number = read\_int("Please enter a number within the range: ", low, high);

cout << "You entered " << number << endl;

}

catch (invalid\_argument &e) {

cerr << "Exception: You supplied an invalid argument for read\_int!\n";

}

}

### Implementation Notes:

When a user enters a value at the command prompt outside of the specified **low** and **high** range, **read\_int()** should provide an informative message and re-prompt for another value (no need to throw an exception in this case).

For example, in the main function above, if the range specified is 0 to 5, then when the user types in a value outside the this range, i.e. 5, **read\_int()** should ask for another value within the range.

The function **read\_int()**will only throw an **invalid\_argument** exception if you specify invalid arguments when you call it, e.g. **read\_int("Enter a number: ", 4, 1).** In this case, the values for **low** and **high** are incorrect because they specify an empty range.

This means that in read\_int, you will need to **throw two different exceptions**. You will need to handle each in a different place. One exception will be handled in main’s try - catch() block (the **invalid\_argument** exception). You will also need to add a different std exception **range\_error** (see page 142) inside read\_int, that will deal with the user entering values outside of the range specified by high and low. This exception should be handled inside read\_int by read\_int’s internal exception handler. Which exception should read\_int throw first? Why?

Finally, the **ios\_base::failure** exception is **not thrown by you**. Instead, it is thrown by **cin** when the fail bit is set, i.e. when a user enters a non-integer value for it to read. You will need to **catch** the **ios\_base::failure** exception in the **read\_int()** function. This will enable the function to request that the user re-type in another value.

Ignore the question’s requirement for this exception to be thrown when the “***end of file is encountered***”.

Save your solution to your github repository

**EX03\_2:** I willdescribe four **unit test cases** you will need to implement using the **Test Explorer** framework. These test cases will help to ensure that the function read\_int(), created in EX03\_1, works correctly.

**Test case 1**: If **read\_int(“My prompt: ”, -3, 3)** AND input is **“Hello\n0”**

* Expect 0 will be returned by the function

**Test case 2**: If **read\_int(“My prompt: ”, 5, 1)**

* Expect **invalid\_argument** **exception** will be thrown

**Test case 3**: If **read\_int(“My prompt: ”, 0, 4)** AND input is **“-3\n5\n10\n-2\n3”**

* Expect 3 will be returned by the function

**Test case 4**: If **read\_int(“My prompt: ”, 0, 0)**

* Expect **invalid\_argument** **exception** will be thrown

Using Visual Studio Test Explorer, and following the steps in the Unit-Testing Tutorial, please implement the above four test cases.

Save your solution in your github repository

A number of additional concepts will be useful to enable you to properly test your read\_int() function.

**Firstly**, you will need to be able to test that your read\_int() function throws the correct exception when it is given incorrect arguments. An example of how to do this is shown below:

#include <stdexcept> // remember to include in unittest1.cpp

…

TEST\_CLASS(READINT\_TEST) // Test suite for read\_int()

{

TEST\_METHOD(TestCase2) // Test case 2: for invalid argument

{

// define a C++11 Lambda function to be called by your test

auto func = []() {

// call with incorrect arguments (test case 2)

read\_int("My prompt: ", 5, 1);

};

// We expect an invalid\_argument exception to be thrown when we call func!

Assert::ExpectException<std::invalid\_argument>(func);

}

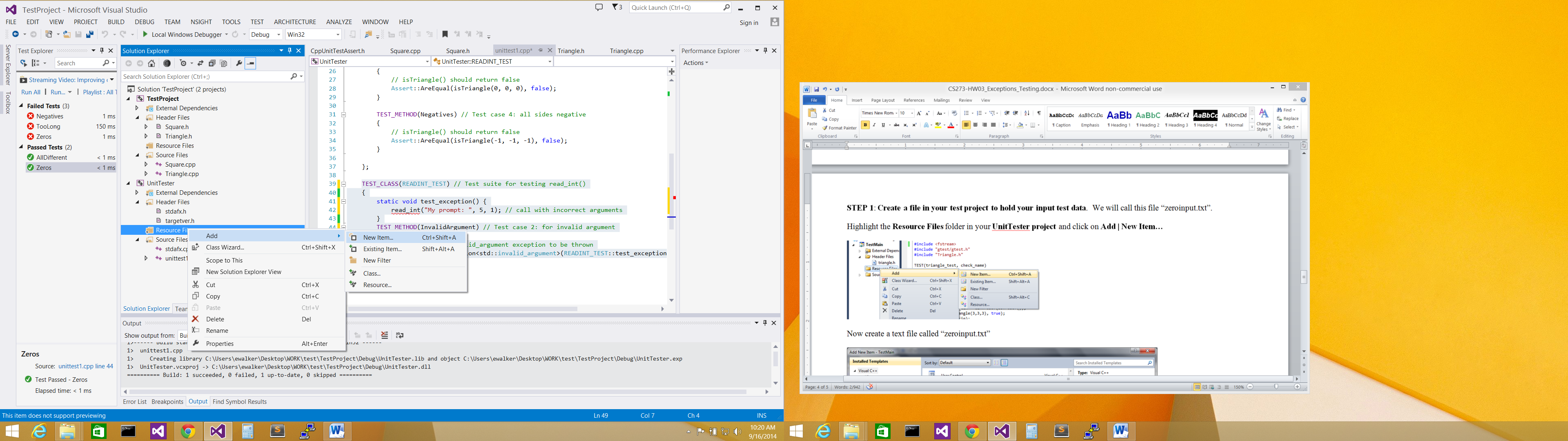
};

**Secondly**, since read\_int()is a function that requires user input, we will need to provide the test input automatically. After all, this test might be run in the middle of the night, and we don’t want to require a person to have to actually type in something at the keyboard when the test is run at midnight. That would really suck! There are two approaches. First, we'll see how to read input through a file to allow **Test Explorer** to automate the testing of your function. Second, we'll see how to use stringstream objects to provide input

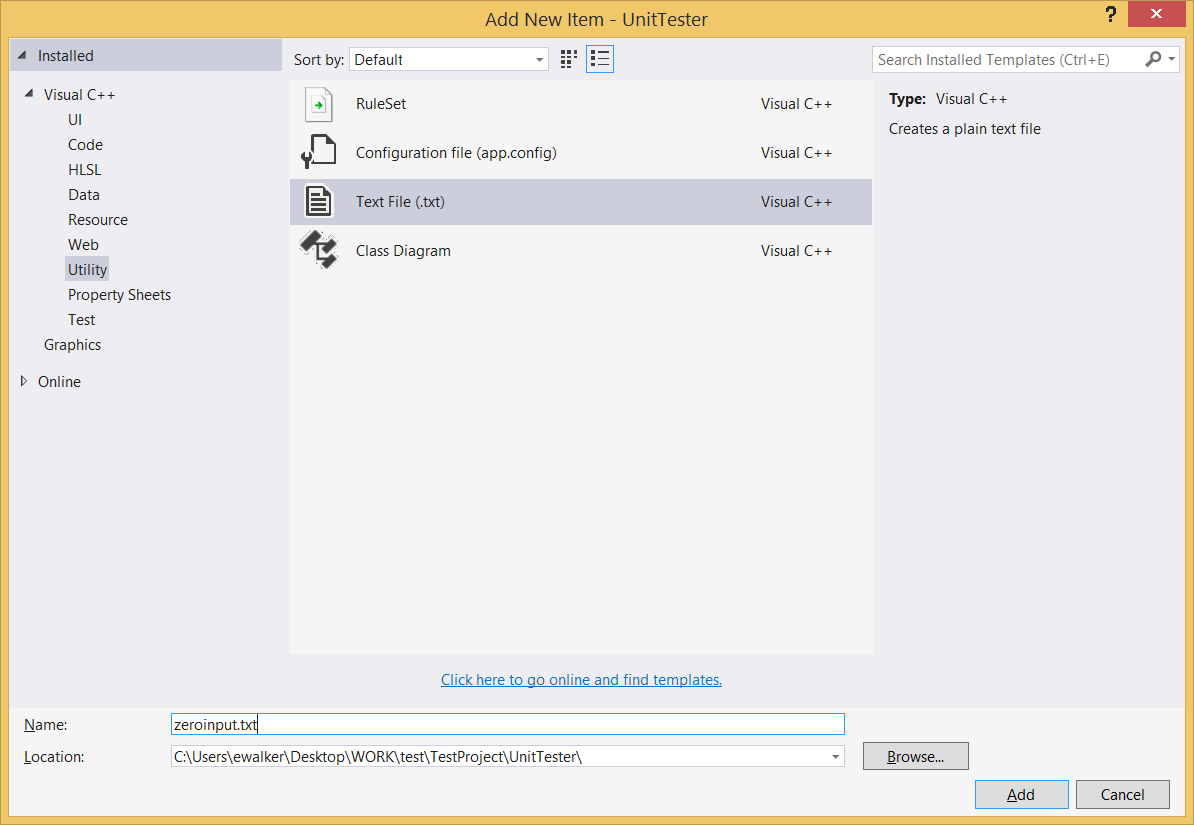
As an example on how to use file input, let’s assume we want to create a test to check the behavior of read\_int() when a user enters a string followed by the numeric value zero (i.e. test case 1).

**STEP 1**: **Create a file in your test project to hold your input test data**. We will call this file “zeroinput.txt”.

Highlight the **Resource Files** folder in your **UnitTester project** and click on **Add | New Item…**

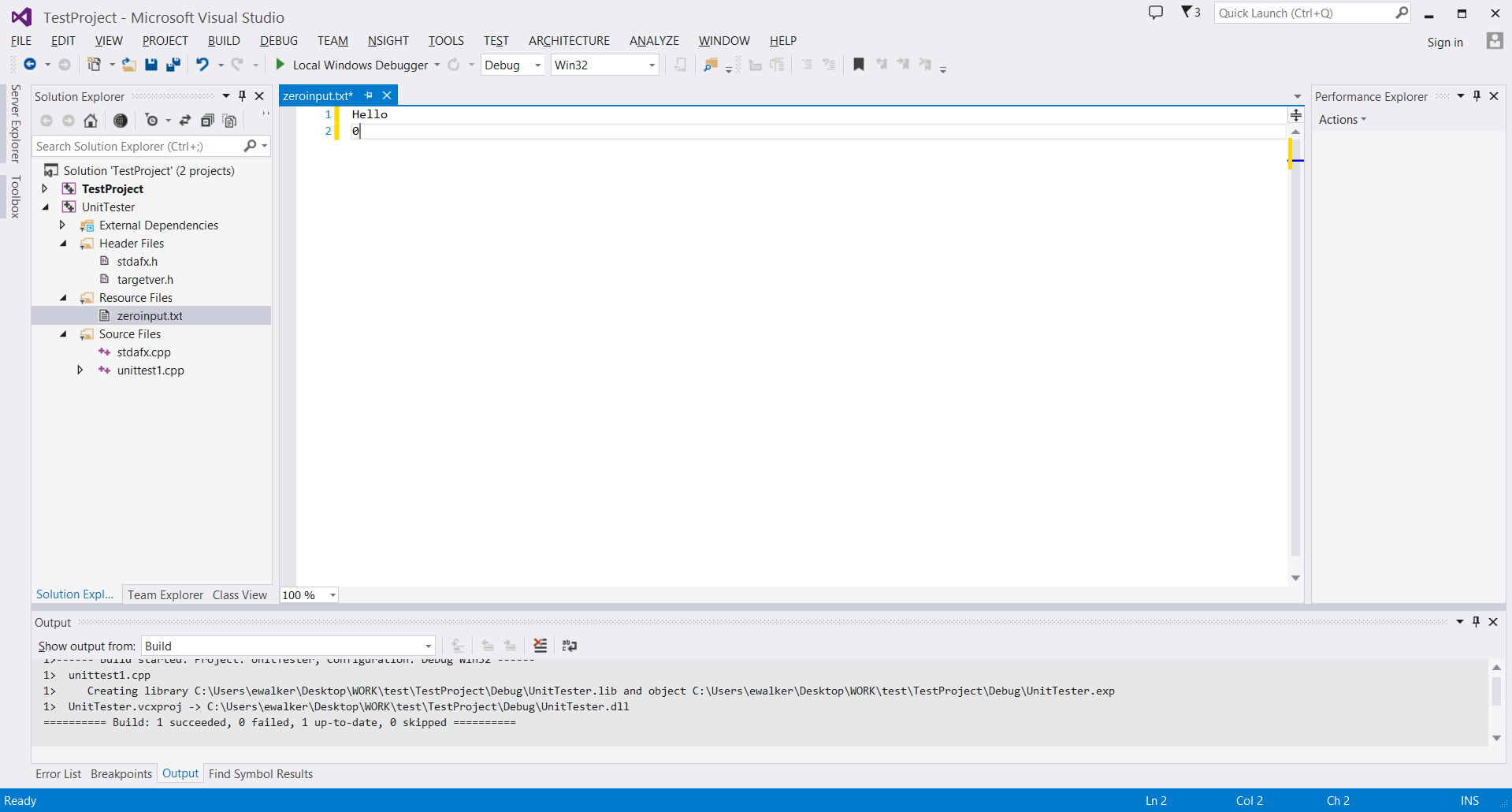


Now create a text file called “zeroinput.txt”



Click **Add** to create the file in your **Resource Files** folder.

You should now have a new file called “zeroinput.txt” in your resource folder. Double-click the file name, and enter two lines: “Hello” and “0”. This mimics the user typing Hello and then 0.



**STEP 2**: Use the file you have just created, “zeroinput.txt”, to **mimic user input.** We will implement **test case 1** to demonstrate how to do this:

#include <fstream> // remember to include in unittest1.cpp

#include <iostream>

using namespace std;

…

TEST\_METHOD(TestCase1) // Test case 1: invalid input

{

// Open a file stream to read the file zeroinput.txt (remember CS-172)

// Replace "UnitTester" with the name of your Native Unit Test project

ifstream ss("..\\UnitTester\\zeroinput.txt");

// Check if we opened the file stream successfully

if (ss.fail())

throw int(-1); // throw an integer with value -1

// Replace the cin read buffer with the read buffer from the file stream

streambuf \*orig\_cin = cin.rdbuf(ss.rdbuf());

// Perform the read\_int() test.

// cin will now read from your file and not from the keyboard.

// We expect the correct value returned is 0, ignoring the Hello string.

Assert::AreEqual(read\_int("My prompt: ", -3, 3), 0);

// Restore cin to the way it was before

cin.rdbuf(orig\_cin);

// Close the file stream

ss.close();

}

**Alternatively**, you can also pipe input through cin by using a istringstream (rather than an ifstream). This approach is more straightforward, but less reusable. You can reuse your input files for multiple tests, and even change the file contents in one place and effect multiple tests. The approach is similar – set up a istringstream object, and the tell cin to read from that buffer:

TEST\_METHOD(TestCase1) // Test case 1: invalid input

{

// crate a string stream object with the desired input

istringstream ss("Hello\n0");

// Replace the cin read buffer with the read buffer from the string stream

streambuf \*orig\_cin = cin.rdbuf(ss.rdbuf());

// Perform the read\_int() test.

// cin will now read contents from your stringstream and not from the keyboard.

// We expect the correct value returned is 0, ignoring the Hello string.

Assert::AreEqual(read\_int("My prompt: ", -3, 3), 0);

// Restore cin to the way it was before

cin.rdbuf(orig\_cin);

// Close the file stream

ss.close();

}