Lab 4: Testing

Information

Topics: Unit tests, debugging

Turn in: All source files (.cpp and .hpp).

Starter files: Download on GitHub or D2L.

```
Lab 04 - Testing/
├── lab04_testing.cpp ...  Contains main()
├── lab04_function1.hpp ...  First testing challenge
├── lab04_function2.hpp ...  Second testing challenge
├── lab04_function3.hpp ...  Third testing challenge
├── lab04_function4.hpp ...  Fourth testing challenge
└── CodeBlocks Project/
    └── CodeBlocks Project.cbp ...  If you're using
                                    Code::Blocks instead
                                    of Visual Studio,
                                    you can open up this
                                    project file to get
                                    started
```

If you're using Visual Studio [1] you'll have to create a new **Empty Project** and add these files in. For more information on how to use Visual Studio, please reference
`https://github.com/Rachels-Courses/Course-Common-Files/`
and go to **Course-Common-Files / STUDENT_REFERENCE / HOW_TO / Visual_Studio.md**

[1] Why is the Code::Blocks project included? Because I work in Linux so I don't use Visual Studio. ;P

Getting started

For this lab, there are already several functions included. **Each of these functions have logic errors.** You will be writing unit tests to test out the expected functionality, and to locate *where* the error is, then fix it.

Function stubs are also already provided - you just have to fill them in. The program is built so that it should launch and automatically run the tests without you having to do any coding outside of the tests themselves.

Do not modify main() or anything in lab04_testing.cpp!

Tests

Within each of the *function* hpp files, there will be a function that needs fixing at the top.

For example, in `lab04_function1.hpp`, we have:

```
1 int AddThree( int a, int b, int c )
2 {
3     return a + b + b;
4 }
```

And there is a tester function set up, with one test already written to give you an example.

For a test, first we figure out a **test case**. A test case here will be, "For these inputs, what is the expected output?" We can write this in code by creating variables for all the input variables, creating a variable to store what the *expected* output is, and then calling the function and storing its return value in an *actual output* variable.

Then, we compare the *expected output* with the *actual output*, and if they're the same - the test passed! If they're not the same, then the test failed. **You are not duplicating the functionality of the original function in the test! That defeats the purpose of the test!** You're merely examining the output obtained by calling the function, and making sure it returns the correct information.

Here's the first test written for the `AddThree` function:

```
void Test_AddThree()
{
    cout << "************ Test_AddThree ************" <<
    endl;

    int input1, input2, input3;
    int expectedOutput;
    int actualOutput;

    /* TEST 1
   ******************************************/
    input1 = 1; input2 = 1; input3 = 1;
    expectedOutput = 3;

    actualOutput = AddThree( input1, input2, input3 );
    if ( actualOutput == expectedOutput )
    {
        cout << "Test_AddThree: Test 1 passed!" << endl;
    }
    else
    {
        cout << "Test_AddThree: Test 1 FAILED!" << endl;
    }
    // ... etc ...
}
```

One test isn't enough to cover all possibilities, so you will need to add additional tests to make sure the `AddThree` function works for all reasonable cases.

For example, the existing test checks that three numbers with the same values (all 1's) result in a 3. It does, but there's still an error with the original function! You might implement tests for three different numbers, or positive and negative numbers, in order to cover all reasonable cases. You don't have to test over a wide range of numbers, just a sufficiently different set of numbers.

Your task for the test of the `Test_AddThree` function is simply to fill in the inputs and outputs variables...

```
input1 = 0;              // change me
input2 = 0;              // change me
input3 = 0;              // change me
expectedOutput = -1;     // change me
```

...and afterwards the code is already there to call the function and compare the outputs.

```
actualOutput = AddThree( input1, input2, input3 );
if ( actualOutput == expectedOutput )
{
    cout << "Test_AddThree: Test 2 passed!" << endl;
}
else
{
    cout << "Test_AddThree: Test 2 FAILED!" << endl;
}
```

Running the tests

When you run the program, it will have a menu with options to test each function:

```
****************************************
**               TESTER               **
****************************************
 1. Test AddThree
 2. Test IsOverdrawn
 3. Test TranslateWord
 4. Test GetLength
 5. Quit

Test which function?
```

When you select one of the functions, it will run the tests and display whether each test passed or failed.

```
************ Test_AddThree ************
Test_AddThree: Test 1 passed!
Test_AddThree: Test 2 FAILED!
Test_AddThree: Test 3 FAILED!
```

Based on which test failed, you can look at the inputs and outputs to help you figure out where the logic error is at in these functions.

Function 1: AddThree

```
int AddThree( int a, int b, int c )
```

Expected functionality: This function should take in three integer variables as input: a, b, and c. Within the function, it should be adding a, b, and c together, and returning the sum.

Function 2: IsOverdrawn

```
bool IsOverdrawn( float balance )
```

Expected functionality: This function takes in some float variable, the user's balance, as the input. Its result is either **true** if the account is overdrawn, or **false** if not. A negative balance is considered overdrawn, while $0 or positive dollar values are considered not overdrawn.

Function 3: TranslateWord

```
string TranslateWord(string word)
```

Expected functionality: This function takes in some English word as a string. Its output is another string, but in Esperanto. The function should support the following translations:

English	Esperanto
cat	kato
dog	hundo
mouse	muso
bird	birdo

And if the word passed in as input is not known, it will return "unknown".

Function 4: GetLength

```
int GetLength(string word)
```

Expected functionality: This function takes a string as input and returns the amount of characters in it, so it returns an int. Make sure to test with several words of different lengths to make sure the function works.

Turn in

You will write the tests, and also fix the logic errors of the functions. You will turn in all the source files.