Lab : Unit Tests

Information

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

Topics: Designing and implementing unit tests

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

Unit tests are small tests meant to validate functions. As functions, at a basic level, simply take inputs and result in some output, we can generally come up with a list of expected outputs, given some inputs.

For example, if we have a function like this:

```
int ReturnOneGreater( int n );
```

and we know that the function is supposed to return $n + 1$ for any n passed in, we can then build test cases like this:

- Input: 1 Output: 2
- Input: -1 Output: 0
- Input: -5 Output: -4

And we can validate the function by testing each of these scenarios. If, for some reason, the actual output we receive differs from the expected output, this suggests that something is wrong with the function.

Unit tests come in handy as you're building up large projects. As you implement one feature at a time, you can also build tests along with those features. Then, after each subsequent feature you add, you can run all your tests to ensure that everything still works - that you didn't inadvertantly break anything.

1.2 Setting up the project

Download the starter zip file, `LAB-UNIT-TESTS.zip`, off GitHub. This zip file contains the following:

- `function1.hpp`
- `function2.hpp`
- `function3.hpp`
- `function4.hpp`
- `tester_program.cpp`

- `Makefile`

All of these source files belong in one project and are not separate programs.

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 testing_program.cpp!

Warning!

Common error!
It is very common for students who are new to the concept of unit tests to make the following mistake:

REWRITING THE TEST TO PASS,
INSTEAD OF WRITING THE TEST TO BE CORRECT.

The tests you write should be logically sound, and if the tests fail, you should be investigating the function being tested to find something to fix. You should NOT, however, rewrite your test so that it passes - this means your function is still incorrect and the test is incorrect, too.

Example: Let's say you have a function that takes three inputs: a, b, and c. The result should be $a + b + c$. You write a test that sets $a = 2$, $b = 3$, and $c = 4$. The expected output is 9, but instead the function returns 24. Something's wrong!

INCORRECT FIX: You change the test so that the expected output is 24.

CORRECT FIX: You investigate the function that should be adding the values, find that its math is wrong, and fix the math.

1.2.1 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.

1.3 Lab specifications

1.3.1 Function 1: SumThree

Each of these .hpp files contain a **function to be tested** and a **test function** with one test already written.

The function for the first file is:

```
int SumThree( int a, int b, int c )
{
    return a + b + b;
}
```

You can probably already see the logic error here, but don't fix it yet; you will fix it after you write all the tests.

Inside the `Test_SumThree` function, there is one test already written:

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

actualOutput = SumThree( input1, input2, input3 );
if ( actualOutput == expectedOutput )
{
    cout << "Test_AddThree: Test 1 passed!" << endl <<
  endl;
}
else
{
    cout << "Test_AddThree: Test 1 FAILED! \n\t"
    // ... more info ..
}
```

This test has the following data:

Test #	Input 1	Input 2	Input 3	Expected output
1	1	1	1	3
2				
3				

But, a single test isn't enough to fully cover this function, we need more. Build out the table so that there are at least three total test cases before continuing.

Before programming in your tests, ask yourself the following:

- Do your test cases cover both positive and negative input values?
- Do your test cases cover both positive and negative output values?
- Do your test cases have different values for inputs 1, 2, and 3?

Make sure your test cases are varied enough; your new tests shouldn't just be "1 + 1 + 1 = 3", "2 + 2 + 2 = 6", "3 + 3 + 3 = 9", and so on...

Once your test cases are done, it's time to program them in. There are two additional tests in the `Test_SumThree` function. To set them up, simply replace the values of the input and expectedOutput variables:

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

After you've updated these, run the tests for the `SumThree` function again. Your tests *should* fail because the logic error hasn't been fixed yet. While the error is pretty obvious for this function, logic errors are not always obvious. This is where unit tests *really* come in handy.

Fix the logic error in the function, re-run the tests, and make sure everything passes. Once it does, move on to the next file.

1.3.2 Function 2: SumArray

```
int SumArray( int arr[], int size )
{
    for ( int i = 0; i <= size; i++ )
    {
        int sum = 0;
        sum += arr[i];
    }
    return sum;
}
```

Again, each of these functions has a logic error. Plan out your test cases, then implement them, test, and try to locate the error and fix it.

Function description:

`SumArray function` This function should take an array of integers, `arr[]`, as well as the amount of items stored in that array, `size`.

The return value is the sum of all of the elements in the array.

Test cases:

Test #	Input array	Expected output
1	{ 1, 2, 3, 4 }	10
2		
3		

1.3.3 Function 3: IsOverdrawn

```
bool IsOverdrawn( float balance )
{
    if ( balance <= 0 )
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Function description:

`IsOverdrawn function` This function should take a float, `balance`, as the input.

The function will return `true` if the account is overdrawn and `false` if the account is not overdrawn. An account is only overdrawn if the account balance is negative.

Test cases:

Test #	Input	Expected output
1	0	false
2		
3		

1.3.4 Function 4: GetLength

```
int GetLength(string word)
{
    int length = 3;
    word.size();
    return length;
}
```

Function description:

`IsOverdrawn function` This function should take a string, `word`, as the input.

The function will return the amount of letters in the word given.

Test cases:

Test #	Input	Expected output
1	"cat"	3
2		
3		