COMP 10050
Software Engineering Project 1
Assignment 3
Unit Testing
Dé Máirt, 4 Bealtaine 2021

The goal of this assignment is to perform Unit Testing on some small programs using the CUnit library.

This assignment must be done individually.

Introduction

Software Testing is a hugely important aspect of software development. Three stages of testing can be identified:

- **Development Testing**, where the system is tested during development to discover bugs and defects;
- **Release Testing**, where a different team, separate to the development team, tests a complete version of the software system before its release to check that the system meets the requirements of the stakeholders; and
- **User Testing** (or **Beta Testing**), where the end-users test the system in their own environment and decide whether the software should be accepted or whether further development is required.

Development Testing aims to discover bugs in the software. Hence it is interleaved with the debugging process and it is carried out by the software programmers themselves. There are three levels of granularity:

- **Unit Testing**, where individual program units are tested and the functionality of methods/functions/routines is tested;
- **Component Testing**, where component interfaces are tested; and
- **System Testing**, where some or all the components in a system are integrated and the system as a whole is tested and the interactions between components are tested.

This assignment focuses on Unit Testing.

The CUnit framework

In Unit Testing, it is preferable to automate the execution of test cases; to assist in this, a **Unit-Testing Framework** is used. This assignment uses the **CUnit** framework. The main concepts in a unit test are the following:

Assertions

- These are the basic and smallest building-blocks
- They are typically Boolean expressions that compare expected and actual results

Test Case

- This is a number of concrete test procedures
- It may contain several assertions and test for several test objectives

Test Suite

- This is a collection of related test cases
- It can often be executed automatically in a single command

The main operations that are required in a unit test using CUnit are the following:

- `CU_initialize_registry()`: Initialise the test registry, which contains the test suites and associated tests
- `CU_add_suite()`: Create a new test suite (collection of tests)
- `CU_add_test()`: Add a new test to the test suite
- `CU_basic_set_mode()`: Set the mode for the amount of information provided
- `CU_basic_run_tests()`: Run all tests in the test registry
- `CU_cleanup_registry()`: Clean up the test registry

CUnit examples

Consider the following C function, `maxi`, that we wish to test:

```
int maxi(int i1, int i2)
{
    if (i1 > i2)
        return i1;
    else
        return i2;
}
```

We create the following test function to test our `max` function:

```
void test_maxi(void)
{
    CU_ASSERT(maxi(0, 2) == 2);
    CU_ASSERT(maxi(0, -2) == 9);
    CU_ASSERT(maxi(1, 2) == 2);
}
```

Clearly, we would expect that the first and last of these assertions would be true, whereas the second one will be false. A test function should have neither arguments nor return values.

This test function is added to a test suite and the tests are carried out in the following program:

```
int main()
{
    CU_initialize_registry();
    CU_pSuite suite = CU_add_suite("maxi_test", 0, 0);
    CU_add_test(suite, "maxi_fun", test_maxi);
    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();
    return 0;
}
```

An alternative technique is to package the CUnit function calls into a function and have a single function call in the main program, as follows:

```
void runAllTests()
{
    CU_initialize_registry();
    CU_pSuite suite = CU_add_suite("maxi_test", 0, 0);
    CU_add_test(suite, "maxi_fun", test_maxi);
    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();
    return;
}

int main()
{
    runAllTests();
    return 0;
}
```

The advantage of doing it this way is that the same program structure can be used for all tests and, for tests on other functions, only the arguments to `CU_add_suite()` and `CU_add_test()` need to be changed.

Consider another function, that checks whether a triangle is a right-angle triangle, an equilateral triangle, an isosceles triangle or a scalene triangle (Recall the caveats I mentioned regarding this function in the lecture).

```
char * checkTriangle(int a, int b, int c)
{
    if (a == 90  ||  b == 90  ||  c == 90)
        return "Right";
    if (a == 60  &&  b == 60  &&  c == 60)
        return "Equilateral";
    if (a == b || b == c || c == a)
        return "Isosceles";
    return "Scalene";
}
```

A possible test function for this function is the following:

```
void triangle_testcase1(void)
{
    CU_ASSERT(strcmp("Equilateral", checkTriangle(60,60,60)) == 0);
    CU_ASSERT_STRING_EQUAL("Right", checkTriangle(40,90,50));
    CU_ASSERT_STRING_NOT_EQUAL("Isosceles", checkTriangle(80,80,50));
}
```

We would expect that the first and second of these assertions would be true, whereas the third one will be false.

Using the second technique described above, where the CUnit function calls are packaged into a `runAllTests()` function and that function is called in the main program, the `checkTriangle()` function can be tested as follows:

```
void runAllTests()
{
    CU_initialize_registry();
    CU_pSuite suite = CU_add_suite("triangle_suite", 0, 0);
    CU_add_test(suite, "triangle_test", triangle_testcase1);
    CU_basic_set_mode(
    CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();
    return;
}


int main()
{
    runAllTests();
    return 0;
}
```

Exercises

The following exercises should be completed.

1. Consider the `average()` function in the file `avg_and_max.c`. Create assertions to test this function as follows:

   - For the array {-1.3, -1.45, -220, -100, -0.1, -0.1234, -200}, test that `average()` returns -74.7104857142857.
   - For the array {-1.3, 1.45, 220, 100, 0, 200, 0.1234}, test that `average()` returns 74.3247714285714.

   Save your solution as `ass3q1.c` and save the report as `ass3q1.out`.

2. Note that there is a logical error in this function (This should be apparent from the results of the tests). Correct this error and use the assertions to test your revised function. Save your solution as `ass3q2.c` and save the report as `ass3q2.out`.

3. Consider the `max()` function in the file `avg_and_max.c`. Create assertions to test this function as follows:

   - For the array {-1.3, -1.45, -220, -100, -0.1, -0.1234, -200}, test that `max()` returns -0.1.
   - For the array {-1.3, 1.45, 220, 100, 0, 200, 0.1234}, test that `max()` returns 220.

   Save your solution as `ass3q3.c` and save the report as `ass3q3.out`.

4. Note that there is a logical error in this function (Again, this should be apparent from the results of the tests). Correct this error and use the assertions to test your revised function. Save your solution as `ass3q4.c` and save the report as `ass3q4.out`.

5. Consider the `factorial()` function in the file `fact.c`. Create assertions to test this function as follows:

   - Test that `factorial(0)` is equal to 1
   - Test that `factorial(1)` is equal to 1
   - Test that `factorial(4)` is equal to 24
   - Test that `factorial(6)` is equal to 720

   Save your solution as `ass3q5.c` and save the report as `ass3q5.out`.

6. One of the oldest algorithms in common use is *Euclid's Algorithm*, to calculate the **Greatest common divisor** (**GCD**) of two integers$x$ and y, often written $\gcd(x, y)$. Donald Knuth had the following comment on Euclid's Algorithm:

   > "[The Euclidean algorithm] is the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day."

   Donald Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 2nd edition (1981), Page 318.

Consider the following three algorithms, all versions of Euclid's Algorithm:
Division-based algorithm

```
function gcd1(a, b)
    while b ≠ 0
        t = b
        b = a mod b
        a = t
    return a
```

Subtraction-based algorithm

```
function gcd2(a, b)
    while a ≠ b
        if a > b
            a = a - b
        else
            b = b - a
    return a
```

Recursive algorithm

```
function gcd3(a, b)
    if b == 0
        return a
    else
        return gcd3(b, a mod b)
```

Implement each of these functions, `gcd1()`, `gcd2()` and `gcd3()`.
Create assertions to test each of these functions as follows:

- Test that $\gcd(42, 56)$ is equal to 14
- Test that $\gcd(48, 18)$ is equal to 6
- Test that $\gcd(270, 192)$ is equal to 6
- Test that $\gcd(1237, 24)$ is equal to 1
- Test that $\gcd(4200000, 3780000)$ is equal to 420 000
- Test that $\gcd(0, 0)$ is equal to 0

Save your solutions as `ass3q6a.c`, `ass3q6b.c` and `ass3q6c.c`, respectively (one solution for each algorithm), and save the reports as `ass3q6a.out`, `ass3q6b.out` and `ass3q6c.out`, respectively.

You must upload your submission by 23:59 on Wednesday, 19 May.