Faculty of Engineering
Cairo University

Final Report Submission

Software Testing

Submitted to:
Dr. Ahmed Sobeih
Eng. Yehia

Submitted by:

Name: Ahmed Saleh Email: abosalaah96@gmail.com

Name: Mohamed Hamada Email: homshamada@gmail.com

Name: Mina Magdy Email: mina_mego5@yahoo.com

Name: Omar Osama Email: omarosamasobeih@yahoo.com

Table of Contents

1) Overview

CppUTest is a C /C++ based unit xUnit test framework for unit testing and for test-driving your code. It is written in C++ but is used in C and C++ projects and frequently used in embedded systems but it works for any C/C++ project.

2) How to build it

- Windows Visual Studio
  - Create Visual Studio Project
  - Project Properties -> C/C++ -> General ->
    Additional Include Directories -> path of include folder in the library folder
  - Project Properties -> Linker -> General ->
    Additional Library Directories -> path of lib folder in the library folder
  - Project Properties -> Linker -> Input ->
    Additional Dependencies ->
    Add “CppUTestd.lib”
- Linux : using this command
  - apt-get install cpputest
- MacOSX : using this command
  - brew install cpputest

3) Assertions examples

i. CHECK_EQUAL

CHECK_EQUAL(expected, actual) - checks for equality between entities using “==” operator. So to use it we need to have this operator in the datatype we are comparing.

```
int add(int a, int b) {
    return a * b;
}
TEST(testgroup, test1)
{
    CHECK_EQUAL(5, add(2, 3));
}
```

Example:

In this example the function assertion check_equal will produce a failure as the function will return “6” not “5” then the test will fail.

ii. STRCMP_EQUAL

STRCMP_EQUAL(expected, actual) - compares two “ char * “ arrays using “strcmp” function.

```
string getFoo() {
    return "foo";
}
TEST(testgroup, test2)
{
    STRCMP_EQUAL("foo", getFoo().c_str());
}
```

Example:

In this example the test will pass as
the returned string equals to expected string.

iii. POINTERS_EQUAL

POINTERS_EQUAL(expected, actual) - compares two pointers and check if they are equal and equal here means they have the same address.

Example:

In this example the test will fail as the pointers don't equal each other even the elements of the arrays are the same

```
3
4   int a[] = {1,2,3};
5   int b[] = {1,2,3};
6
7   int main()
8   {
9       POINTERS_EQUAL(a,b);
10  }
11
```

4) CppUMock **introduces an efficient way that allows you to test how many times a particular function is being called and with what parameters and return values while executing a particular scenario.**

i. Simple Scenario:

check that one particular function call has happened. The following code is an example of this.

```
#include "CppUTest/TestHarness.h"
#include "CppUTestExt/MockSupport.h"
TEST_GROUP(MockDocumentation) {
    void teardown() {
        mock().clear();
    }
};
void productionCode() {
    mock().actualCall("productionCode");
}
TEST(MockDocumentation, SimpleScenario) {
```

```
    mock().expectOneCall("productionCode");
    productionCode();
    mock().checkExpectations();
}
```

- The TEST(MockDocumentation, SimpleScenario) contains the recording of the expectations as: `mock().expectOneCall("productionCode");`

- The test ends with checking whether the expectations have been met. This is done with the: `mock().checkExpectations();`

- the call to `mock().clear()` in the teardown is not needed when using the MockSupportPlugin, otherwise it is needed to clear the MockSupport. Without the clear, the memory leak detector will report the mock calls as leaks.

- If the call to productionCode wouldn't happen, then the test would fail with the following error message:

```
ApplicationLib/MockDocumentationTest.cpp:41: error: Failure in
TEST(MockDocumentation, SimpleScenario)
      Mock Failure: Expected call did not happen.
      EXPECTED calls that did NOT happen:
            productionCode -> no parameters

      ACTUAL calls that did happen:
            <none>
```

- Sometimes you expect several identical calls to the same function, for example five calls to productionCode. There is a convenient shorthand for that situation: `mock().expectNCalls(5, "productionCode");`

ii. Parameters:

just checking whether a function is called is not particularly useful when we cannot check the parameters. Recording parameters on a function is done like this:

```
mock().expectOneCall("function").onObject(object).withParameter("p1",2).w
 ithParameter("p2", "hah");
```

- The actual call is like:

```
mock().actualCall("function").onObject(this).withParameter("p1",
p1).withParameter("p2", p2);
```

- If the parameter isn't passed, it will give the following error

```
Mock Failure: Expected parameter for function "function" did not
happen.
      EXPECTED calls that DID NOT happen related to function: function
      (object address: 0x1)::function -> int p1: <2>, char* p2: <hah>
      ACTUAL calls that DID happen related to function: function
            <none>
      MISSING parameters that didn't happen:
            int p1, char* p2
```

iii. C Interface:

Sometimes it is useful to access the mocking framework from a .c file rather than a .cpp file. For example, perhaps, for some reason, the stubs are implemented in a .c file rather than a .cpp file. Instead of changing over all to .cpp, it would be easier if the mocking framework can be called via C. The C interface is exactly meant for this. The interface is based on the C++ one, so below is an example code

```
#include "CppUTestExt/MockSupport_c.h"

mock_c()->expectOneCall("foo")->withIntParameters("integer",
10)->andReturnDoubleValue(1.11);
return   mock_c()->actualCall("foo")->withIntParameters(
"integer", 10)->returnValue().value.doubleValue;

mock_c()->installComparator("type",  equalMethod,
toStringMethod);
mock_scope_c("scope")->expectOneCall("bar")->withParameterOfT
ype("type", "name", object);
mock_scope_c("scope")->actualCall("bar")->withParameterOfType
("type", "name", object);
mock_c()->removeAllComparators();

mock_c()->setIntData("important", 10);

mock_c()->checkExpectations();
mock_c()->clear();
```

- It is also now possible to specify the actual return value the same way as with C++ (v3.8)

- To specify a default return value in case of mocking is currently disabled when the actual call occurs

5) Memory Leak Detection

CppUTest has memory leak detection support on a per-test level. This means that it automatically checks whether the memory at the end of a test is the same as at the beginning of the test.

Steps :

Pre-setup -> Record the amount of memory used
Do setup
Run test
Do teardown
Post-teardown -> Check whether the amount of memory is the same

The memory leak detector consists of three parts:

Memory leak detector base (including linker symbols for operator new):
Macros overloading operator new for additional file and line info
Macros overloading malloc/free for memory leak detection in C

And all of these are on by default.

If you want to completely disable memory leak detection then you can do so by two ways:

Building CppUTest with “configure –disable-memory-leak-detection”
Passing -DCPPUTEST_MEM_LEAK_DETECTION_DISABLED to the compiler when compiling CppUTest.

6) References

http://cpputest.github.io/manual.html
http://cpputest.github.io/mocking_manual.html
https://github.com/cpputest/cpputest

7) Workload

Team Members	Activity List
Ahmed Saleh	Trie testing
Mohamed Hamada	Math_utilities testing
Mina Magdy	Segment tree testing
Omar Osama	Segment tree testing