**Unit Testing**

As we explained in the introduction, unit testing is the process of testing the code in small units. Usually these units correspond to functions, but that is not necessarily always the case. Each unit test feeds the function under test specific values and then compares the return value to the expected output. If the value produced by the function under test is identical to that that is expected, then the test has passed. If the value is different, then the test fails.

While it is entirely possible to write tests manually by simply writing code which invokes functions and checks to see that the values output by the functions are correct, it is more common to use a unit test framework. A unit test framework requires that you write your unit tests in a specific format and then it automatically runs the tests for you and records whether they have passed or failed. Most testing frameworks give you some kind of an interface that allows you to display the results of the tests and readily determine which tests have passed or failed.

The previous chapter described how we could select test data for unit tests. You should follow the procedures outlined in that chapter to carefully select your test data before you build your unit tests. Once the test data has been selected, you can lay it out as a series of unit tests.

Most unit tests frameworks group unit tests into a test suite. The test suite could test one particular function and it would use different tests to pass each piece of data to the function.

Most unit tests use some type of an assertion to determine if the result of running a function is correct or not. An assertion is a type of statement which compares the expected value with the value produced and, if they satisfy the comparison, indicates that the test has passed. Many assertions allow an error message to be associated with it so you can see a message describing exactly where the test failed.

There are many different unit test frameworks, but they all follow this basic model. In the next section we will start to look in detail at the unit test framework provided by Microsoft Visual Studio.

**Unit Testing in Visual Studio**

Visual Studio provides its own test framework for C and C++ programs. The framework is written in C++, but can be adapted to test C code as well as C++ code. While the framework can be used to test any existing project, one of the common ways to use the test framework is to add the test code itself as a new project to the solution that is under test. We will look at testing the following code.

**mathfuncs.h**[**​**](https://software-testing.sdds.ca/C-UnitTesting/vs-test-2#mathfuncsh)

#pragma once  
#ifndef MATHFUNCS\_H  
#define MATHFUNCS\_H  
  
double square(double n);  
double cube(double n);  
  
#endif

Copy

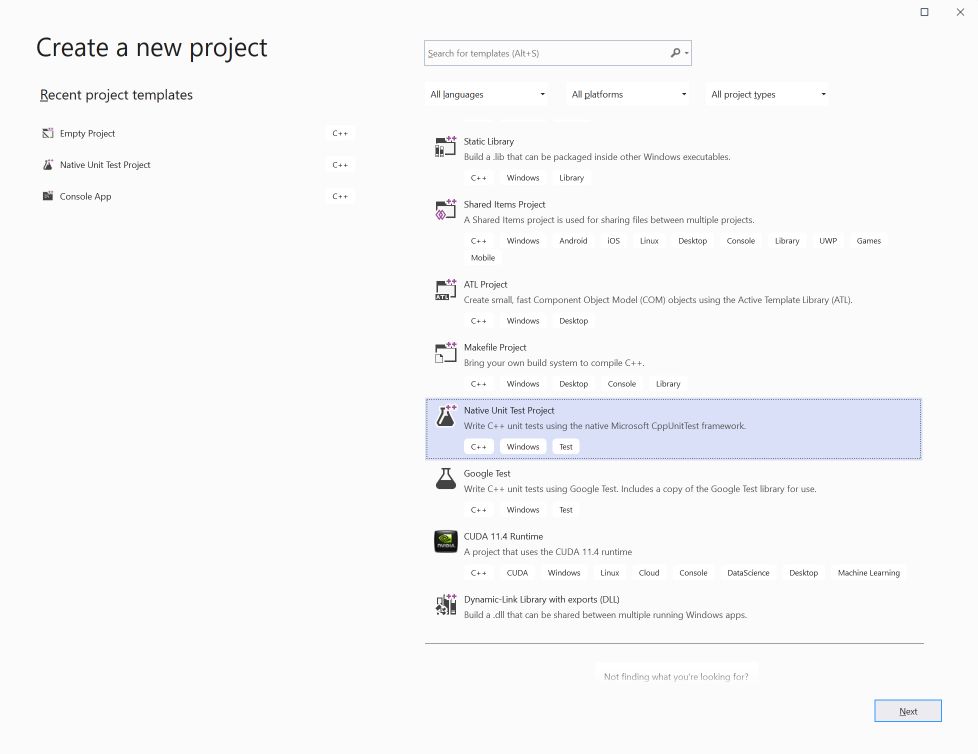
**mathfuncs.c**[**​**](https://software-testing.sdds.ca/C-UnitTesting/vs-test-2#mathfuncsc)

#include "mathfuncs.h"  
  
double square(double n)  
{  
 return n \* n;  
}  
  
double cube(double n)  
{  
 return n \* n \* n;  
}

Copy

As you can see, we create two functions: one to square a number and another to cube a number. Testing that these functions work is a simple matter but seeing how it is done will illustrate all the steps in creating a test project for an existing project.

Open the project you want to test and then click on the name of the solution at the top of the solution explorer. Then, from the File menu, select **New | Project**. This will display the new project menu and you should select a **native unit test project for C++ on Windows**, as shown below.



Once the project has been created, you will see it at the bottom of the solution explorer. A skeleton unit test program has been created for you called UnitTest1.cpp (or whatever name you selected). I have modified this file to look as shown below.

#include "pch.h"  
#include "CppUnitTest.h"  
#include "mathfuncs\_r.h"  
  
using namespace Microsoft::VisualStudio::CppUnitTestFramework;  
  
namespace MathTestSuite  
{  
 TEST\_CLASS(MathTest)  
 {  
 public:  
  
 TEST\_METHOD(SquareTest)  
 {  
 double d = square(8.0);  
 Assert::AreEqual(64.0, d);  
 }  
  
 TEST\_METHOD(CubeTest)  
 {  
 double d = cube(3.0);  
 Assert::AreEqual(27.0, d);  
 }  
 };  
  
 TEST\_CLASS(MathIntegrationTest)  
 {  
 public:  
  
 TEST\_METHOD(AdditionTest)  
 {  
 double d = square(8.0);  
 double d1 = cube(3.0);  
 Assert::AreEqual(91.0, d + d1);  
 }  
 };  
}

Copy

I modified the namespace to be more meaningful and called it **MathTestSuite**. Within the namespace, I have created two test classes. Each class would usually represent the tests for a single function or feature. Placing tests in classes provides a convenient way to group tests together. Inside each class, there are test methods which each test one particular aspect of the code. In this example, I have one class to test the functions and a second class to test how the functions might work together. There is no set way to organize this and you are encouraged to layout your tests in the most organized fashion you can for your particular project.

Inside the tests, you call your functions with specific values and then use an assertion to determine if the result is correct by comparing it to a known value. The Assert class has many methods that allow you to compare values. If the two values pass the comparison, the test passes, otherwise it fails.

The methods of the Assert class are summarized in the following table.

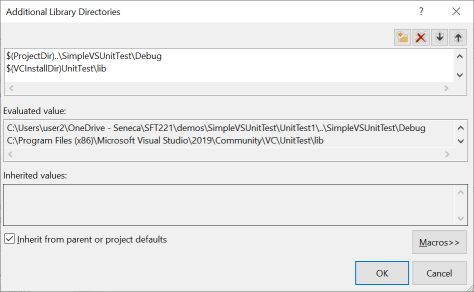
| **Method** | **Description** |
| --- | --- |
| AreEqual(v1, v2 [, "error message"] ) | Compares v1 to v2 and throws an exception if they are not equal. If they are not equal, the optional error string will be displayed. |
| AreNotEqual(v1, v2 [, "error message"] ) | Compares v1 to v2 and throws an exception if they are equal. If they are equal, the optional error string will be displayed. |
| IsTrue(b1 [, "error message"] ) | If the Boolean b1 is not true it throws an exception. If not true, the optional error string will be displayed. |
| IsFalse(b1 [, "error message"] ) | If the Boolean b1 is not false it throws an exception. If not false, the optional error string will be displayed. |
| Fail([ "error message"] ) | This causes the test to fail and throws an exception. If called, the optional error string will be displayed. |

**Accessing the Software Under Test**[**​**](https://software-testing.sdds.ca/C-UnitTesting/vs-test-2#accessing-the-software-under-test)

We have now created two projects in the same solution. The testing project needs access to:

* the .lib or .obj file for the software under test,
* the header files for the project or modified header files if the project is in pure C.

To make the project we are testing visible to the test code, we need to modify the values of the properties of the test project. You can do this by right clicking on the test project in the solution explorer and selecting properties. Once the properties are displayed, select **Linker | General** and then **Additional Library Directories**. This will display the dialog shown below.



The goal is to add the debug directory in the project under test. This should be two levels higher in the directory structure and then into the project under test. Start by clicking the symbol for a new directory at the left of the top series of buttons. This will create a new row into which you can type the directory. While you can use the threee dots to navigate, this will result in an absolute path which will make it difficult to move the project to a different location. A better way is to use the macro **$(ProjectDir)** which will be set to the directory containing the source code for the test project. If the project is moved, this macro will be automatically updated.

Next, we need to provide the name of the file(s) containing the compiled version of the software under test. If you just have an object file, like we do here, then you can just enter the name, **mathfuncs.obj**. If you had build a library, then you would enter the name of the library.

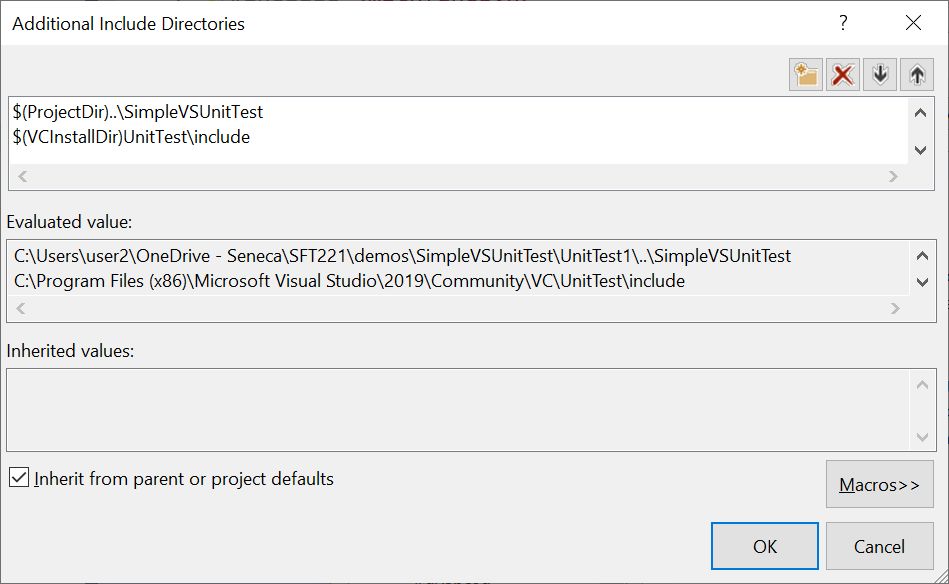
The final step is to tell the test project how to find the header files for the project being tested. If this is a C++ project we are testing then we navigate to **C/C++ | General | Additional Include Directories** and add the include directory for the project under test. If the project you are testing is pure C, then the process is more complicated.

C++ compilers alter names of functions inside classes so that they can determine the class that a particular method belongs to. C compilers do not do this, and this creates a problem when we are trying to link C++ code with pure C code. In order to get the two language is to be compiled together, we have to tell the C++ compiler which functions are actually pure C so that it will not alter the names of them, but just use the original names. The way we can do this, is to use the **extern “C”** declaration as demonstrated in the next piece of code.

#pragma once  
#ifndef MATHFUNCS\_R\_H  
#define MATHFUNCS\_R\_H  
  
extern "C"  
{  
#include <mathfuncs.h>  
}  
  
#endif

Copy

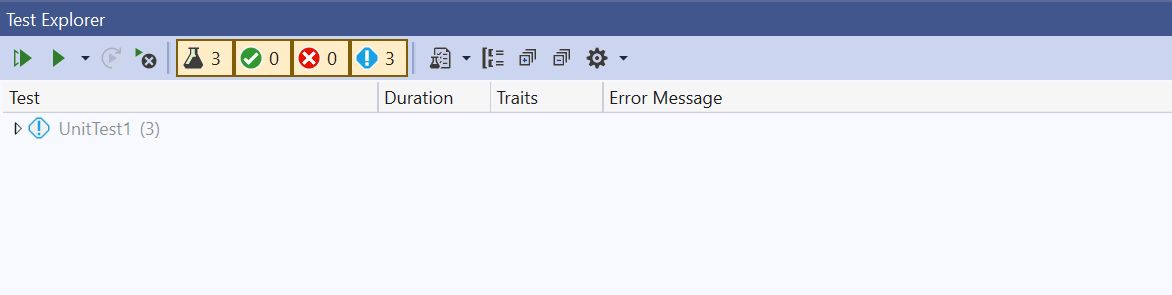
What I have done in this piece of code is to include the header file for the software under test inside an **extern “C”** block. This prevents the C compiler from altering the names and allows the two languages to be linked together. I called this file **mathfuncs\_r.h** and placed it in the test project. The **\_r** is used to indicate it references the original header file. Next, you need to tell the test project where to find the header files for the project under test. This is done by editing the test project properties under **C++ | General | Additional Include Directories**. You should add paths to include the directories in which the header files are stored for the code you are testing, as well as the test project. Once done, the include paths should look like:



The final step is to go the **Build** menu at the top and select \*Build All\*\* . You should ckeck the output window at the bottom to be sure that the solution compiled correctly. It should say that two projects were compiled and zero failed.

**Running the Tests**[**​**](https://software-testing.sdds.ca/C-UnitTesting/vs-test-2#running-the-tests)

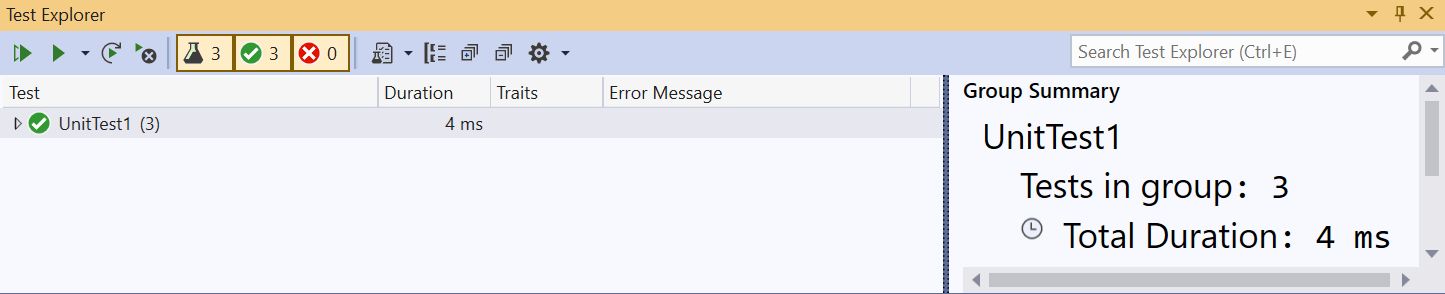
The next step is to actually run the tests. Go to the **Test** menu at the top and select **Test Explorer**. This will display a new window to control the tests, as shown below. This wildow can be left floating or docked into the Visual Studio window.



The Test Explorer shows:

* the list of tests,
* controls to run the tests,
* the number of tests available to run, the number passed, the number failed, and the number which have not been run,
* a sub-window at the right that will show the output of the tests.

With the test explorer, you can highlight the test(s) you want to run and then press the *play* button to run the tests. The results of the test are shown as you can see in the diagram below.



**Setting Up and Tearing Down Tests**[**​**](https://software-testing.sdds.ca/C-UnitTesting/vs-test-2#setting-up-and-tearing-down-tests)

You can create additional functions to set up and tear down the tests. This would be needed if you required data structures to be set up before the tests were run and torn down after the tests were completed. There are functions for **TEST\_MODULE\_INITIALIZE** and **TEST\_MODULE\_CLEANUP** which are run before any of the class tests are run and after all class tests are complete. In addition, there are methods to go inside a test class called **TEST\_CLASS\_INITIALIZE** and **TEST\_CLASS\_CLEANUP**. These can be used to setup and tear down data structures required for each test class. There is a **Logger** class which is used to write simple messages to the test output. This is demonstrated in the code below:

#include "pch.h"  
#include "CppUnitTest.h"  
#include "mathfuncs\_r.h"  
  
using namespace Microsoft::VisualStudio::CppUnitTestFramework;  
  
namespace MathTestSuite  
{  
 TEST\_MODULE\_INITIALIZE(ModuleInitialize)  
 {  
 Logger::WriteMessage("In Module Initialize");  
 }  
  
 TEST\_MODULE\_CLEANUP(ModuleCleanup)  
 {  
 Logger::WriteMessage("In Module Cleanup");  
 }  
  
 TEST\_CLASS(MathTest)  
 {  
 public:  
 TEST\_CLASS\_INITIALIZE(ClassInitialize)  
 {  
 Logger::WriteMessage("In Class Initialize");  
 }  
  
 TEST\_CLASS\_CLEANUP(ClassCleanup)  
 {  
 Logger::WriteMessage("In Class Cleanup");  
 }  
  
 TEST\_METHOD(SquareTest)  
 {  
 Logger::WriteMessage("In Square test");  
 double d = square(8.0);  
 Assert::AreEqual(64.0, d);  
 }  
  
 TEST\_METHOD(CubeTest)  
 {  
 Logger::WriteMessage("In Cube test");  
 double d = cube(3.0);  
 Assert::AreEqual(27.0, d);  
 }  
 };  
  
 TEST\_CLASS(MathIntegrationTest)  
 {  
 public:  
  
 TEST\_METHOD(AdditionTest)  
 {  
 double d = square(8.0);  
 double d1 = cube(3.0);  
 Logger::WriteMessage("In Integration test");  
 Assert::AreEqual(91.0, d + d1);  
 }  
 };  
}

Copy

We can run this code using the command vstest.console.exe UnitTest1.dll. This command should be executed in a Windows cmd shell and will produce the output shown below. Note the order in which the initialize and cleanup functions are run.

vstest.console.exe UnitTest1.dll  
Microsoft (R) Test Execution Command Line Tool Version 16.11.0  
Copyright (c) Microsoft Corporation. All rights reserved.  
  
Starting test execution, please wait...  
A total of 1 test files matched the specified pattern.  
In Module Initialize  
In Integration test  
In Class Initialize  
In Square test  
In Cube test  
In Class Cleanup  
In Module Cleanup  
 Passed AdditionTest [< 1 ms]  
 Passed SquareTest [< 1 ms]  
 Passed CubeTest [< 1 ms]  
  
Test Run Successful.  
Total tests: 3  
 Passed: 3  
 Total time: 0.2394 Seconds

Copy

[Previous](https://software-testing.sdds.ca/C-UnitTesting/unit-testing-1)

[« Unit Testing](https://software-testing.sdds.ca/C-UnitTesting/unit-testing-1)

[Next](https://software-testing.sdds.ca/C-UnitTesting/testing-coverage-3)

[Testing Coverage »](https://software-testing.sdds.ca/C-UnitTesting/testing-coverage-3)

* [mathfuncs.h](https://software-testing.sdds.ca/C-UnitTesting/vs-test-2#mathfuncsh)
* [mathfuncs.c](https://software-testing.sdds.ca/C-UnitTesting/vs-test-2#mathfuncsc)
* [Accessing the Software Under Test](https://software-testing.sdds.ca/C-UnitTesting/vs-test-2#accessing-the-software-under-test)
* [Running the Tests](https://software-testing.sdds.ca/C-UnitTesting/vs-test-2#running-the-tests)
* [Setting Up and Tearing Down Tests](https://software-testing.sdds.ca/C-UnitTesting/vs-test-2#setting-up-and-tearing-down-tests)

Software Testing

* [Contents](https://software-testing.sdds.ca/)
* [Install as an App](https://software-testing.sdds.ca/C-UnitTesting/vs-test-2)

Copyright © 2022 Seneca

**Testing Coverage**

We design our tests based upon black box and white box testing principles. This means that the test data has been selected to prove that certain inputs provide the correct outputs for black box testing as well as to drive the code through certain branches based upon white box testing. Unfortunately, we can still miss code which has never been executed by any tests. Code which has never been executed often has bugs in it that have never been found, for the obvious reason that it was never executed. The goal of testing coverage is to find the code which was executed as well as the code which has never been executed. Once we identify the code which has never been executed, we can then go on to design tests to exercise that particular piece of code.

Testing coverage is a facility offered by many compilers. While Visual Studio does have this capability, it is only available in the enterprise version of the tool. This means that it is not available to many programmers and therefore we will use the tool provided by GCC. While it would be preferable to do our tests in the same environment that we do our coverage analysis, it is complicated by the fact that we are using the testing tool from Visual Studio and they do not make their coverage analysis tool available to us. As a result, we will largely be demonstrating how to use coverage analysis tool without actually applying it to our testing regime.

In order to do testing coverage, you normally have to compile with special flags or options that tell your compiler to build in additional code that actually counts the number of times every line is executed. While theoretically, you could write your own coverage analysis tool by writing out a message every time a line was executed, this would be labor intensive to do and, even if you automated it, you would have to maintain two different versions of the code.

We are going to demonstrate how to do coverage using GCC with the following program which inserts one string into another.

#include <stdio.h>  
#include <string.h>  
  
#define STR\_SIZE 20  
  
void insert(char dest[], int posn, char src[])  
{  
 int dlen = strlen(dest);  
 int slen = 0, i;  
  
 if(posn < dlen)  
 {  
 slen = strlen(src);  
 for(i = dlen; i >= posn; i--)  
 {  
 dest[i + slen] = dest[i];  
 }  
  
 for(i = 0; i < slen; i++)  
 {  
 dest[posn +i] = src[i];  
 }  
 }  
 else  
 {  
 strcat(dest, src);  
 }  
}  
  
int main(void)  
{  
 char dest[STR\_SIZE] = {"The fox"};  
  
 insert(dest, 4, "red ");  
 printf("%s\n", dest);  
  
 return 0;  
}

Copy

When we compile this code, we have to provide the additional flags to the compiler instructing it to build in the instructions to count how many times each line is executed. The command to compile the program shown above, insert.c, is: gcc -Wall -fprofile-arcs -ftest-coverage -o insert insert.c After we compile the code with these flags, we then have to execute the program once to generate data that is stored in various files. These files contain the information needed to count how many times each line was executed. To prepare the files for viewing by humans, we use the gcov command: gcov insert.c This command produces a new file called insert.c.gcov which is shown next.

-: 0:Source:insert.c  
 -: 0:Graph:insert.gcno  
 -: 0:Data:insert.gcda  
 -: 0:Runs:1  
 -: 1:#include <stdio.h>  
 -: 2:#include <string.h>  
 -: 3:  
 -: 4:#define STR\_SIZE 20  
 -: 5:  
 1: 6:void insert(char dest[], int posn, char src[])  
 -: 7:{  
 1: 8: int dlen = strlen(dest);  
 1: 9: int slen = 0, i;  
 -: 10:  
 1: 11: if(posn < dlen)  
 -: 12: {  
 1: 13: slen = strlen(src);  
 5: 14: for(i = dlen; i >= posn; i--)  
 -: 15: {  
 4: 16: dest[i + slen] = dest[i];  
 -: 17: }  
 -: 18:  
 5: 19: for(i = 0; i < slen; i++)  
 -: 20: {  
 4: 21: dest[posn +i] = src[i];  
 -: 22: }  
 -: 23: }  
 -: 24: else  
 -: 25: {  
 #####: 26: strcat(dest, src);  
 -: 27: }  
 1: 28:}  
 -: 29:  
 1: 30:int main(void)  
 -: 31:{  
 1: 32: char dest[STR\_SIZE] = {"The fox"};  
 -: 33:  
 1: 34: insert(dest, 4, "red ");  
 1: 35: printf("%s\n", dest);  
 -: 36:  
 1: 37: return 0;  
 -: 38:}

Copy

This file contains the original source code and line numbers. The first column, before the line numbers, is an indication of how many times that line was executed. The symbols "-:" indicate this is not an executable line and was never executed. Hash signs (#) indicate executable lines which were never executed. These are what you should be looking for in order to find code which was not tested. The numbers in the first column are the counts of how many times each executable line was executed.