**Unit testing native code with Test Explorer**

**Visual Studio 2013**

[Other Versions](javascript:;)

In Visual Studio, you can create unit tests for unmanaged code written in C++. Unmanaged code is sometimes referred to as native code.

The following procedure contains the essential information that will get you started. The later sections provide a walkthrough that describes the steps in more detail.

**To write unit tests for an unmanaged code DLL**

1. Use the Native Test Project template to create a separate Visual Studio project for your tests.

The project contains some sample test code.

1. Make the DLL accessible to the test project:
   * #include a .h file that contains declarations of the DLL’s externally-accessible functions.

The .h file should contain function declarations marked with \_declspec(dllimport). Alternatively, you can export the methods using a DEF file. For more information, see [Importing and Exporting](http://msdn.microsoft.com/en-us/library/9h658af8.aspx).

Your unit tests can access only functions that are exported from the DLL under test.

* + Add the DLL project to the References of the test project:

In the Properties of the test project, expand Common Properties, Framework and References, and choose Add Reference.

1. In the test project, create test classes and test methods by using the TEST macros and Assert class in the following way:

C++

#include "stdafx.h"

#include <CppUnitTest.h>

#include "..\MyProjectUnderTest\MyCodeUnderTest.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

TEST\_CLASS(TestClassName)

{

public:

TEST\_METHOD(TestMethodName)

{

// Run a function under test here.

Assert::AreEqual(expectedValue, actualValue, L"message", LINE\_INFO());

}

}

* + Assert contains several static functions that you can use to verify the result of a test.
  + The LINE\_INFO() parameter is optional. In cases where there is no PDB file, it allows the test runner to identify the location of a failure.
  + You can also write test setup and cleanup methods. For more information, open the definition of the TEST\_METHOD macro, and read the comments in CppUnitTest.h
  + You cannot nest test classes.

1. Use Test Explorer to run the tests:
   * On the View menu, choose Other Windows, Test Explorer.
   * Build the Visual Studio solution.
   * In Test Explorer, choose Run All.
   * To investigate any test in more detail in Test Explorer:
     1. Select the test name to see more details such as a failure message and stack trace.
     2. Open the test name (for example by double-clicking) to go to the failure location or to the test code.
     3. On the shortcut menu for a test, choose Debug Selected Test to run the test in the debugger.

[**Walkthrough: Developing an unmanaged DLL with Test Explorer**](javascript:void(0))

You can adapt this walkthrough to develop your own DLL. The principal steps are as follows:

1. [Create a Native Test Project](http://msdn.microsoft.com/en-us/library/hh270864.aspx#unitTestProject). The tests are created in a separate project from the DLL that you are developing.
2. [Create a DLL Project](http://msdn.microsoft.com/en-us/library/hh270864.aspx#createDllProject). This walkthrough creates a new DLL, but the procedure for testing an existing DLL is similar.
3. [Make the DLL functions visible to the tests](http://msdn.microsoft.com/en-us/library/hh270864.aspx#coupleProjects).
4. [Iteratively augment the tests](http://msdn.microsoft.com/en-us/library/hh270864.aspx#iterate). We recommend a "red-green-refactor" cycle, in which development of the code is led by the tests.
5. [Debug failing tests](http://msdn.microsoft.com/en-us/library/hh270864.aspx#debug). You can run tests in debug mode.
6. [Refactor while keeping the tests unchanged](http://msdn.microsoft.com/en-us/library/hh270864.aspx#refactor). Refactoring means improving the structure of the code without changing its external behavior. You can do it to improve the performance, extensibility, or readability of the code. Because the intention is not to change the behavior, you do not change the tests while making a refactoring change to the code. The tests help make sure that you do not introduce bugs while you are refactoring. You can therefore make such changes with much more confidence than if you did not have the tests.
7. [Check coverage](http://msdn.microsoft.com/en-us/library/fc8hec9e.aspx). Unit tests are more useful when they exercise more of your code. You can discover which parts of your code have been used by the tests.
8. [Isolate units from external resources](http://msdn.microsoft.com/en-us/library/hh270864.aspx#isolate). Typically, a DLL is dependent on other components of the system that you are developing, such as other DLLs, databases, or remote subsystems. It is useful to test each unit in isolation from its dependencies. External components can make tests run slowly. During development, the other components might not be complete.

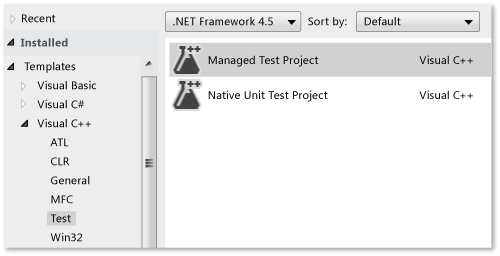
**Create a native unit test project**

1. On the File menu, choose New, Project.

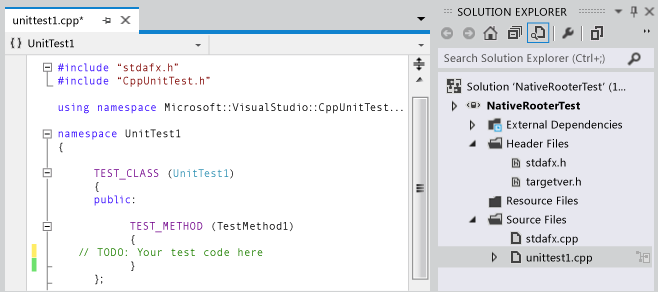
In the dialog box, expand Installed, Templates, Visual C++, Test.

Choose the Native Test Project template.

In this walkthrough, the test project is named NativeRooterTest.



1. In the new project, inspect **unittest1.cpp**



Notice that:

* + Each test is defined by using TEST\_METHOD(YourTestName){...}.

You do not have to write a conventional function signature. The signature is created by the macro TEST\_METHOD. The macro generates an instance function that returns void. It also generates a static function that returns information about the test method. This information allows the test explorer to find the method.

* + Test methods are grouped into classes by using TEST\_CLASS(YourClassName){...}.

When the tests are run, an instance of each test class is created. The test methods are called in an unspecified order. You can define special methods that are invoked before and after each module, class, or method. For more information, see [Organizing C++ Tests](http://msdn.microsoft.com/en-us/library/hh270864.aspx#organizing).

1. Verify that the tests run in Test Explorer:
   * Insert some test code:

C++

TEST\_METHOD(TestMethod1)

{

Assert::AreEqual(1,1);

}

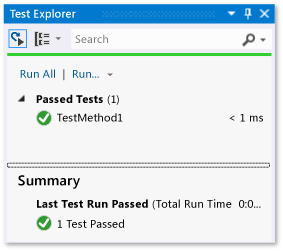
Notice that the Assert class provides several static methods that you can use to verify results in test methods.

* + On the Test menu, choose Run , All Tests.

The test builds and runs.

Test Explorer appears.

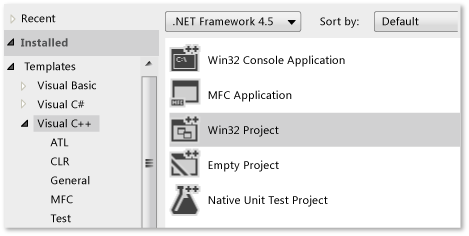
The test appears under Passed Tests.



**Create an Unmanaged DLL project**

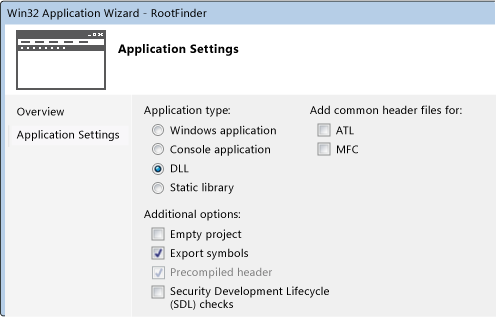
1. Create a Visual C++ project by using the Win32 Project template.

In this walkthrough, the project is named RootFinder.

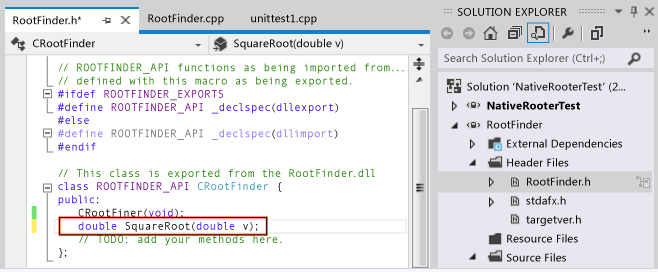


1. Select DLL and Export Symbols in the Win32 Application Wizard.

The Export Symbols option generates a convenient macro that you can use to declare exported methods.



1. Declare an exported function in the principal .h file:



The declarator \_\_declspec(dllexport) causes the public and protected members of the class to be visible outside the DLL. For more information, see [Using dllimport and dllexport in C++ Classes](http://msdn.microsoft.com/en-us/library/81h27t8c.aspx).

1. In the principal .cpp file, add a minimal body for the function:

C++

// Find the square root of a number.

double CRootFinder::SquareRoot(double v)

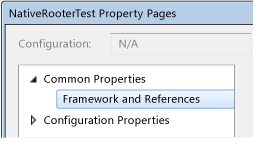
{

return 0.0;

}

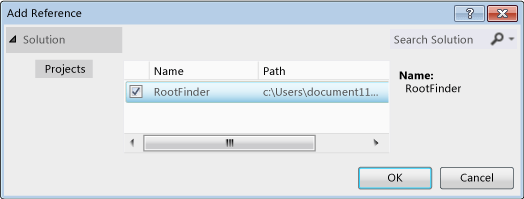
**Couple the test project to the DLL project**

1. Add the DLL project to the project references of the test project:
   1. Open the properties of the test project and choose Common Properties, Framework and References.



* 1. Choose Add New Reference.

In the Add Reference dialog box, select the DLL project and choose Add.



1. In the principal unit test .cpp file, include the .h file of the DLL code:

C++

#include "..\RootFinder\RootFinder.h"

1. Add a basic test that uses the exported function:

C++

TEST\_METHOD(BasicTest)

{

CRootFinder rooter;

Assert::AreEqual(

// Expected value:

0.0,

// Actual value:

rooter.SquareRoot(0.0),

// Tolerance:

0.01,

// Message:

L"Basic test failed",

// Line number - used if there is no PDB file:

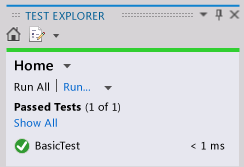
LINE\_INFO());

}

1. Build the solution.

The new test appears in Test Explorer.

1. In Test Explorer, choose Run All.



You have set up the test and the code projects, and verified that you can run tests that run functions in the code project. Now you can begin to write real tests and code.

**Iteratively augment the tests and make them pass**

1. Add a new test:

C++

TEST\_METHOD(RangeTest)

{

CRootFinder rooter;

for (double v = 1e-6; v < 1e6; v = v \* 3.2)

{

double actual = rooter.SquareRoot(v\*v);

Assert::AreEqual(v, actual, v/1000);

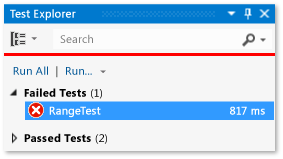
}

}

|  |
| --- |
| **TipTip** |
| We recommend that you do not change tests that have passed. Instead, add a new test, update the code so that the test passes, and then add another test, and so on.  When your users change their requirements, disable the tests that are no longer correct. Write new tests and make them work one at a time, in the same incremental manner. |

1. Build the solution, and then in Test Explorer, choose Run All.

The new test fails.



|  |
| --- |
| **TipTip** |
| Verify that each test fails immediately after you have written it. This helps you avoid the easy mistake of writing a test that never fails. |

1. Enhance the code under test so that the new test passes:

C++

#include <math.h>

...

double CRootFinder::SquareRoot(double v)

{

double result = v;

double diff = v;

while (diff > result/1000)

{

double oldResult = result;

result = result - (result\*result - v)/(2\*result);

diff = abs (oldResult - result);

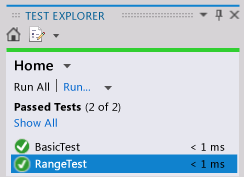
}

return result;

}

1. Build the solution and then in Test Explorer, choose Run All.

Both tests pass.



|  |
| --- |
| **TipTip** |
| Develop code by adding tests one at a time. Make sure that all the tests pass after each iteration. |

**Debug a failing test**

1. Add another test:

C++

#include <stdexcept>

...

// Verify that negative inputs throw an exception.

TEST\_METHOD(NegativeRangeTest)

{

wchar\_t message[200];

CRootFinder rooter;

for (double v = -0.1; v > -3.0; v = v - 0.5)

{

try

{

// Should raise an exception:

double result = rooter.SquareRoot(v);

\_swprintf(message, L"No exception for input %g", v);

Assert::Fail(message, LINE\_INFO());

}

catch (std::out\_of\_range ex)

{

continue; // Correct exception.

}

catch (...)

{

\_swprintf(message, L"Incorrect exception for %g", v);

Assert::Fail(message, LINE\_INFO());

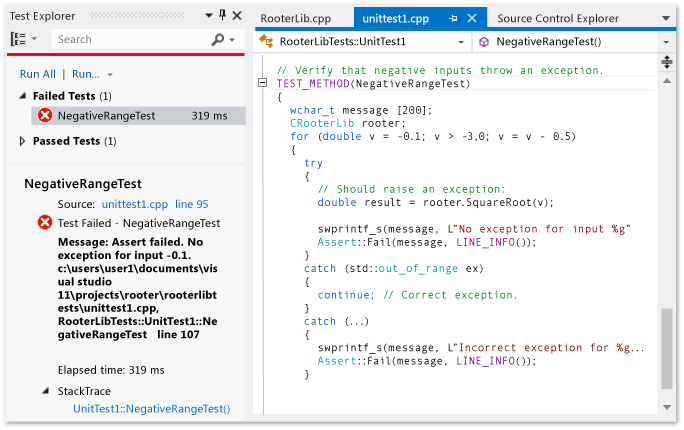
}

}

}

1. Build the solution and choose Run All.
2. Open (or double-click) the failed test.

The failed assertion is highlighted. The failure message is visible in the detail pane of Test Explorer.



1. To see why the test fails, step through the function:
   1. Set a breakpoint at the start of the SquareRoot function.
   2. On the shortcut menu of the failed test, choose Debug Selected Tests.

When the run stops at the breakpoint, step through the code.

1. Insert code in the function that you are developing:

C++

#include <stdexcept>

...

double CRootFinder::SquareRoot(double v)

{

// Validate parameter:

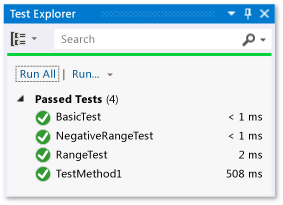
if (v < 0.0)

{

throw std::out\_of\_range("Can't do square roots of negatives");

}

1. All tests now pass.



**Refactor the code without changing tests**

1. Simplify the central calculation in the SquareRoot function:
2. // old code:
3. // result = result - (result\*result - v)/(2\*result);
4. // new code:
5. result = (result + v/result)/2.0;
6. Build the solution and choose Run All, to make sure that you have not introduced an error.

|  |
| --- |
| **TipTip** |
| A good set of unit tests gives confidence that you have not introduced bugs when you change the code.  Keep refactoring separate from other changes. |