



UnitTest - The C/C++ Source Code Testing Framework Guide

Author: Rajinder Yadav

Date: Sept 9, 2007

Revision: Jan 27, 2008

Web: <http://devmentor.org>

Email: rajinder@devmentor.org

UnitTest is developed on the principle of keeping things simple as possible. To that point, the source code testing framework is made to allow the developer to easily write white/grey-box unit test cases. There are only a hand-full of macros you need to know of, but all your job comes down to returning a boolean **'true'** or **'false'** value in your unit test method, that's all you really need to know!

UnitTest now has the provision to allow test input values to be modified at runtime with the use of a XML unit test input file called "**UnitTest.xml**". The format is very simple and allows the mapping of a (name, value) text pair. How to use the new XML input file is described later in document.

The **utgen.exe** utility is command line tool provided to painlessly generates the unit test class source code, it's makes life as a unit test coder simple and quick.

I can say this, once you get into the habit of writing unit test code, you will always want to unit test everything you code. As you will get into the habit of developing unit test in parallel with your new design and development, you will have the confidence that there will a test harness to do regression testing when changes are made, as new bugs are discovered and fixed you will added a new test case to cover it. With this in place someone else can come in and make changes run your unit tests and have the confidence they have not broken existing functionality. Likewise you can go in fearlessly and make changes and have a safety-net to catch issues early during those times when turn-around is so crucial to the project and things get broken when changes are made in a haste to satisfy a hot customer issues that's just landed on your lap.

You will also discover errors sooner as you write unit test code that will get your code to crash or fail. I make it a game to try to crash my methods and APIs. Only by doing this will know if your code is robust. Always run your unit test before changes are committed to a source control like cvs, and before any core-reviews, make this a practice! Your boss will love you for it.

To get unit test support from the **UnitTest** testing framework you will need to do the following things.

NOTE: The following steps are for general understanding, you can skim though with the reading and start unit testing you code right away! In practice you will use the **utgen.exe** utility rather than do the steps by hand.

Step 1

Create your test class and subclass your test class from class UnitTestRunner.

```
// File: MyUnitTest.h
```

```
class MyUnitTest : public UnitTestRunner
{
};
```

Step 2

You will need to register the unit test class with the **UnitTest** framework. To do this include the **REGISTER_TESTCLASS()** with the name of your test class as a parameter. The macro must be placed in the public scope of the class definition because the macro expands into a default class constructor that calls the base class **UnitTestRunner** constructor with the name of the test class.

```
class MyUnitTest : public UnitTestRunner
{
public:

    REGISTER_TESTCLASS( MyUnitTest )
};
```

Step 3

Since class **UnitTestRunner** implements the **IUnitTestRunner** interface you will need to support the following interface methods

```
struct IUnitTestRunner
{
    virtual void Setup() = 0;
    virtual void CleanUp() = 0;
    virtual void RunTest() = 0;
};

class MyUnitTest : public UnitTestRunner
{
public:

    REGISTER_TESTCLASS( MyUnitTest )

    void Setup(); // optional method
    void CleanUp(); // optional method

    void RunTest()
    {
    }
};
```

Note: the **Setup()** and **CleanUp()** methods are optional and your test class does not need to override these methods as the base class **UnitTestRunner** provides an empty placeholders. However your test class is required to override **RunTest()**, which will contain the calls to the class unit test methods.

Step 4

You will then need to add two more macros to initialize and terminate the unit test cases that the **UnitTest** framework will call. The following two macros must be placed inside the **RunTest()** method:

```
UNIT_TEST_START();  
UNIT_TEST_END();
```

At this point your test class header file should look like the one shown below.

```
class MyUnitTest : public UnitTestRunner  
{  
public:  
  
    REGISTER_TESTCLASS( MyUnitTest )  
  
    void Setup();  
    void Cleanup();  
  
    void RunTest()  
    {  
        UNIT_TEST_START();  
        UNIT_TEST_END();  
    }  
};
```

Step 5

You will now need to write your own unit test methods. A unit test method does not take any arguments and the returns type is boolean. A value of **true** signifies the unit test passed, whereas a **false** value indicates the unit test failed.

```
class MyUnitTest : public UnitTestRunner  
{  
    bool Load();  
    bool Save();  
    bool Connect();  
    bool Open();  
  
public:  
  
    REGISTER_TESTCLASS( MyUnitTest )  
  
    void Setup();  
    void Cleanup();  
  
    void RunTest()  
    {  
        UNIT_TEST_START();  
        UNIT_TEST_END();  
    }  
};
```

At this point the test class is almost complete, what's left is to add hooks that will allow the **UnitTest** framework to call each unit test method and monitor their result.

Step 6

To allow the **UnitTest** framework to monitor the unit test case methods, you will need to add the **UNIT_TEST()** macro that takes as a parameter the name of a unit test method. This macro is to be used inside the **Run()** method and placed between the **UNIT_TEST_START** and **UNIT_TEST_END** macros.

```
// File: MyUnitTest.h
class MyUnitTest : public UnitTestRunner
{
    bool Load();
    bool Save();
    bool Connect();
    bool Open();

public:

    REGISTER_TESTCLASS( MyUnitTest )

    void Setup();
    void CleanUp();

    void RunTest()
    {
        UNIT_TEST_START();

        UNIT_TEST(Load);
        UNIT_TEST(Save);
        UNIT_TEST(Connect);
        UNIT_TEST(Open);

        UNIT_TEST_END();
    }
};
```

Step 7

The final step is to create the .cpp [source file](#) for the test class with all the unit test case methods. That file will need to include the header file for the test class along with the **DECLARE_TESTRUNNER()** macro to register your unit test class with the **UnitTest** framework.

```
// File: MyUnitTest.cpp
#include <UnitTest.h> // includes all required UnitTest headers
#include "MyUnitTest.h" // your test class header file

DECLARE_TESTRUNNER( MyUnitTest ); // declare unit test class
```

When the global instance of the test class is created, using the macro above it will register itself with the **UnitTest** framework.

Note within the test class you will need to contain one or more instance of the classes that the unit testing will be directed at. I will refer to them as **subject** classes. These subject classes can be created inside the **Setup()** function and destroyed inside the **CleanUp()** function. The other thing to do is simply make the subject classes permanent members of the test class. Since the Setup() and CleanUp() methods are called before and after each test case is run, you get the advantage of always having fresh subjects to unit test.

Configuring Runtime Test Input Value

As mentioned earlier, The UnitTest Framework now allows test input values to be provided using an XML file called "**UnitTest.xml**". The format of this XML file is very simple and it allows multiple (name, value) text pair to be listed. The basic format of the XML schema is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<UnitTest>
  <Input field="Test1.txt">G:\UnitTest\TestInput\Test.txt</Input>
  <Input field="root">G:\UnitTest\TestInput</Input>
</UnitTest>
```

The Input element has one attribute called, "field". The field value is the name of the input text, while the text of the Input element is the value. In the above XML, two input fields "Test1.txt" and "root" have been defined.

Before all the unit test cases are executed, the UnitTest Framework parses the XML unit test input file, and makes available the input values using the following API.

```
wchar_t* UnitTestInputs( wchar_t* pwzField ) const;
```

The API takes the name of the field as the input and returns a string of the value. To get the "root" value of "**G:\UnitTest\TestInput**" the API would be called like this from inside your unit test class.

```
wchar_t* pwzRootFolder = UnitTestInputs( L"root" );
```

Using the XML unit test input file, one no longer needs to worry about hard coding filenames, paths and other values that cause unit test cases to break.

Automated unit test code generation with utgen.exe

As mentioned above, since the process of writing unit test class header and source files follows a pattern, this process has been automated by the utgen utility. It is a command line tool that will generate both the header and source file on your behalf so you can get started with writing your unit test cases.

To generate the above example file in this document, the utgen utility would be used with the following switches.

```
utgen.exe -n MyUnitTest -t Load Save Connect Open
```

Here are the following switches that are available

```
/s <subject class>  
/n <class name>  
/o <output directory>  
/a <author name>  
/c <copyright>  
/t <test case> ...
```

The subject class is the class you want to unit test.

The '/t' switch can take one or more names of the unit test case methods to be include in the test class. The names should be space separated and the resulting name will be post fixed with a '**_UT**', this is done to avoid compiler error resulting from name conflicts between the subject class and the test class.

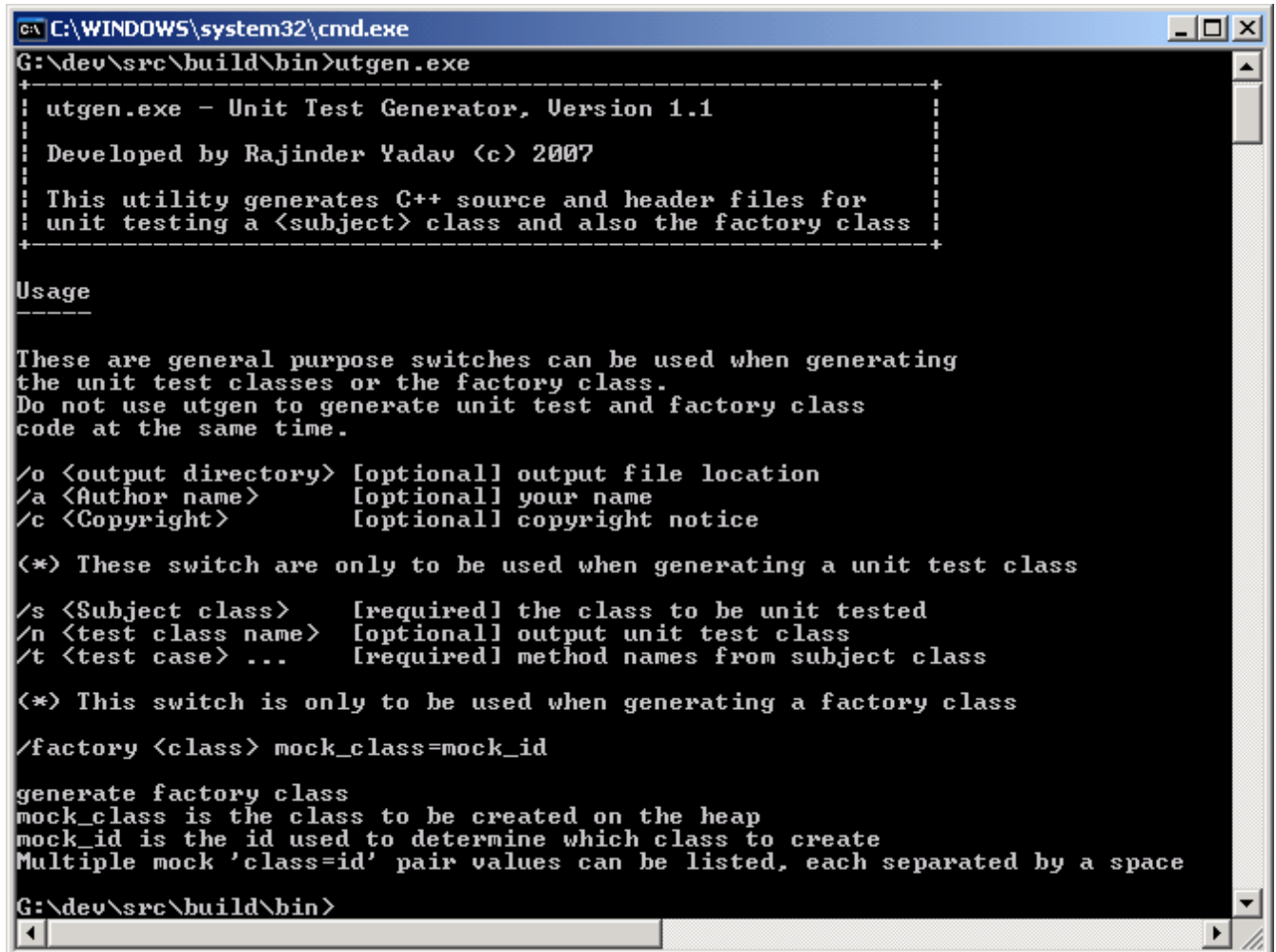
switch '/n', '/o', '/a', '/c' are optional.

If a output class name is not provided, then a name will be created from the subject class name by appending "**_UnitTest**".

If the output directory is not provided, the local directory will contain the generated files.

If you forget these options simply type in the name of the utility at the command prompt and it will display them for you like this.

NOTE: utgen.exe produces mock classes and mock factory classes, at the time of this writing mock class testing with UnitTest is still in the experimental stages.



```
C:\WINDOWS\system32\cmd.exe
G:\dev\src\build\bin>utgen.exe
-----
| utgen.exe - Unit Test Generator, Version 1.1
| Developed by Rajinder Yadav (c) 2007
| This utility generates C++ source and header files for
| unit testing a <subject> class and also the factory class
|-----
Usage
-----
These are general purpose switches can be used when generating
the unit test classes or the factory class.
Do not use utgen to generate unit test and factory class
code at the same time.

/o <output directory> [optional] output file location
/a <Author name>      [optional] your name
/c <Copyright>       [optional] copyright notice

(*) These switch are only to be used when generating a unit test class
/s <Subject class>   [required] the class to be unit tested
/n <test class name> [optional] output unit test class
/t <test case> ...   [required] method names from subject class

(*) This switch is only to be used when generating a factory class
/factory <class> mock_class=mock_id

generate factory class
mock_class is the class to be created on the heap
mock_id is the id used to determine which class to create
Multiple mock 'class=id' pair values can be listed, each separated by a space
G:\dev\src\build\bin>
```

A Test Run with utgen.exe

To see how it works and what the generated output files look like, type is the following at the command prompt:

```
utgen.exe /s MyApp /t Init Term Open Close
```

This utility will output 2 files, "MyApp_UnitTest.h" and "MyApp_UnitTest.cpp"

Below is what the source file will look like. Notice how "_UT" has been appended to the tech case method names you provided, and how "_UnitTest" has been appended to the subject class name.

```
// This unit test header file was generated by the utgen.exe utility
//
// Source: MyApp_UnitTest.h
// Author:
// Date:   July 14, 2007
//
//
//

class MyApp_UnitTest : public UnitTestRunner
{
    MyApp* m_pSubject; // Class object that unit tests get applied too!!!

    bool Init_UT();
    bool Term_UT();
    bool Open_UT();
    bool Close_UT();

public:

    REGISTER_TESTCLASS( MyApp_UnitTest )

    void Setup();
    void CleanUp();

    void RunTest()
    {
        UNIT_TEST_START();

        UNIT_TEST( Init_UT );
        UNIT_TEST( Term_UT );
        UNIT_TEST( Open_UT );
        UNIT_TEST( Close_UT );

        UNIT_TEST_END();
    }
};
```

Below is what the source file will look like. Noticed by default the methods are designed to fail as the initial boolean return value of bRet is set to false. The utility tries to do the right thing inside the method **Setup** and **CleanUp**, but most likely this will need to be modified, or simply removed altogether.

```
// This unit test source file was generated by the utgen.exe utility
//
// Source: MyApp_UnitTest.cpp
// Author:
// Date: July 14, 2007
//
//
//

#include "stdafx.h"
#include <UnitTest.h>
#include "MyApp_UnitTest.h"

DECLARE_TESTRUNNER( MyApp_UnitTest );

void MyApp_UnitTest::Setup()
{
    // TODO: Add code to perform initialization before each unit test is executed.
    //       Allocate all the resources here to allow the unit test to execute..

    m_pSubject = new MyApp(); // TODO: use proper class constructor
}

void MyApp_UnitTest::CleanUp()
{
    // TODO: Add code to clean up after each unit test.
    //       Release all resource allocated in method SetUp()

    delete m_pSubject;
    m_pSubject = 0;
}

bool MyApp_UnitTest::Init_UT()
{
    bool bRet = false;
    // TODO: add testing code here, return true for pass, false for fail.
    return bRet;
}

bool MyApp_UnitTest::Term_UT()
{
    bool bRet = false;
    // TODO: add testing code here, return true for pass, false for fail.
    return bRet;
}

bool MyApp_UnitTest::Open_UT()
{
    bool bRet = false;
    // TODO: add testing code here, return true for pass, false for fail.
    return bRet;
}

bool MyApp_UnitTest::Close_UT()
{
    bool bRet = false;
    // TODO: add testing code here, return true for pass, false for fail.
    return bRet;
}
```

Now you should have a better idea about utgen.

Executing Your Unit Test Classes

With the test class written you will need to create a testing project. In the test project you will need to declare an instance of the Test class. The basic outline will be like the one shown below.

```
#include "MyUnitTest.h"

int _tmain(int argc, _TCHAR* argv[])
{
    // begin the unit test
    UnitTestAssembly::GetInstance().Run();

    // free singleton unit test framework
    UnitTestAssembly::ReleaseInstance();
    return 0;
}
```

All that is required then is to tell the **UnitTest** framework to begin the unit test process. This is done by calling it's **Run()** method. You will first need to get an instance of the **UnitTest** framework. This can be done with a call of the UnitTestAssembly class static method:

```
UnitTestAssembly::GetInstance()
```

Our testing project does not do much as we still need to be able to view the results! To do this will need to create an instance of a logger class and attach it to the test class to be notified of unit test results as well as user generated message and other test events. The code below shows how to get file and console logging to take place. The file logger is asked to create a 'unit_test.log' log file, and to place this file in the 'C:\testlog' directory.

```
int _tmain(int argc, _TCHAR* argv[])
{
    // setup file logging
    StreamLogger logger("c:\\testlog\\unit_test.log");

    // setup console logging
    ConsoleLogger display;

    UnitTestAssembly::GetInstance().RegisterObserver( logger );
    UnitTestAssembly::GetInstance().RegisterObserver( display );

    // begin the unit test
    UnitTestAssembly::GetInstance().Run();

    // free singleton unit test framework
    UnitTestAssembly::ReleaseInstance();

    return 0;
}
```

The above code can be greatly simplified by using the macros below.

```
int _tmain(int argc, _TCHAR* argv[])
{
    // setup file logging
    StreamLogger logger("c:\\testlog\\unit_test.log");

    // setup console logging
    ConsoleLogger display;

    // setup test result observers
    DECLARE_OBSERVER( logger );
    DECLARE_OBSERVER( display );

    // begin unit testing
    UnitTestManager::Start();

    return 0;
}
```

Below is a sample of the test Log output.

```
-----
Test Runner: RefCount_UnitTest
-----

Unit Test Started
[passed] Test case: RefCount_UnitTest::Default_UT1()
[passed] Test case: RefCount_UnitTest::Default_UT2()
[passed] Test case: RefCount_UnitTest::Default_UT3()

...

[passed] Test case: RefCount_UnitTest::RefCount_UT1()
[passed] Test case: RefCount_UnitTest::RefCount_UT2()
[passed] Test case: RefCount_UnitTest::AddRef_UT()
[passed] Test case: RefCount_UnitTest::Release_UT1()
Unit Test complete.

Test Summary: Tests(15) Fails(0) Exceptions(0)

...

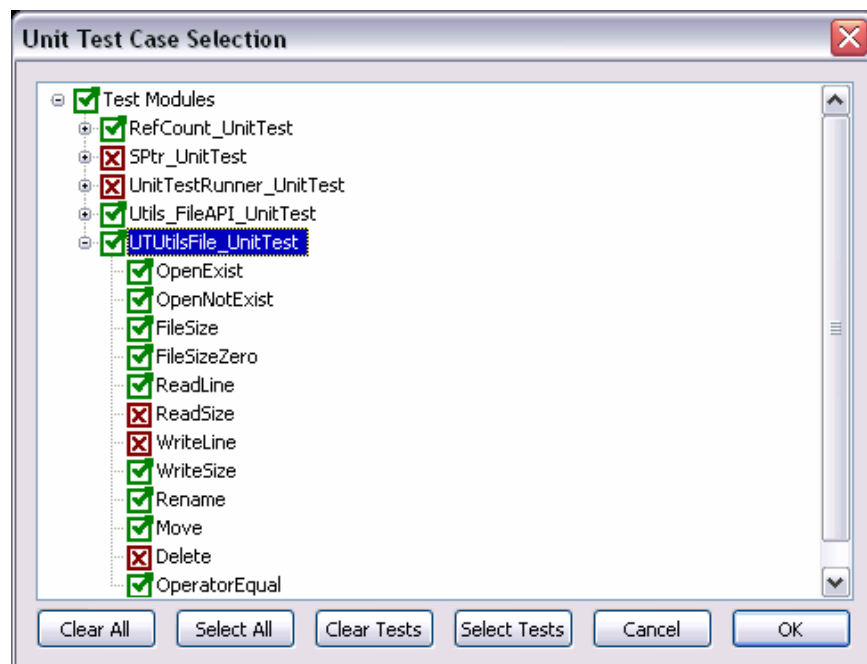
-----
Overall Summary
-----
Total Run: 170
Total Failed: 2
Total Exceptions: 0
```

UnitTest Monitor

For those wanting sometime visual and some control over what classes gets unit tested, there is the "UnitTest Monitor" GUI app.

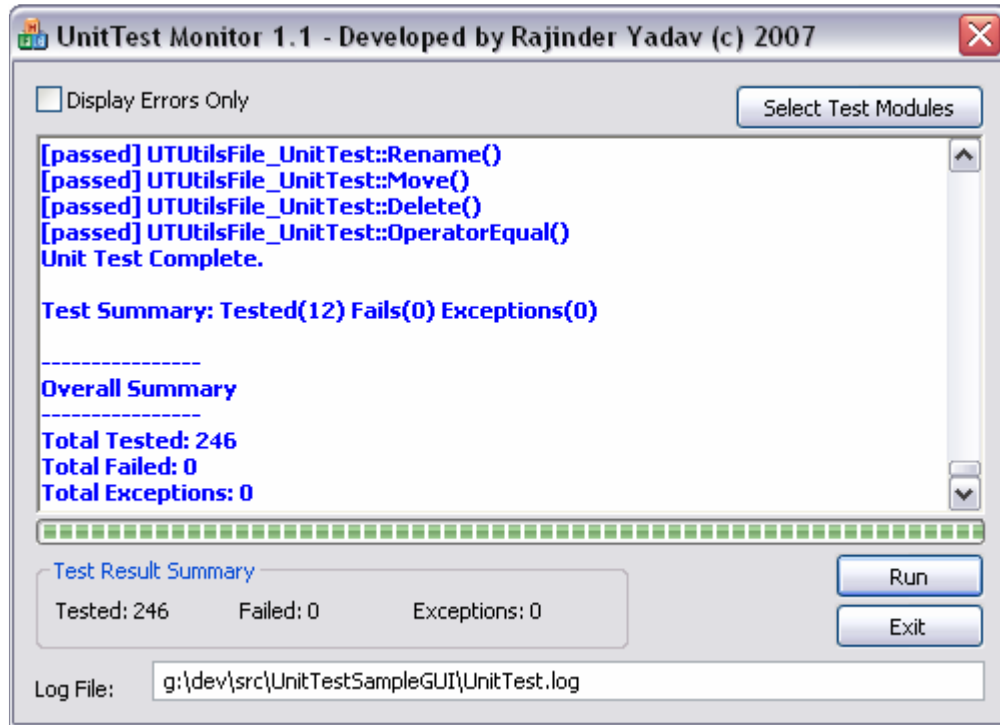


The UnitTest Monitor GUI tool shows you a list of the registered unit test classes. The list can be viewed using the "Unit Test Case Selection" dialog box. The dialog box uses a treeview to present all the registered *subject* classes, and under each subject class is a list of the test cases.



Once the test are run the GUI tool will generate a display of a log report in the "Test Result" window similar to what you would see in the log file. Also basic stats of the test results are shown in the "Test Result Summary" area.

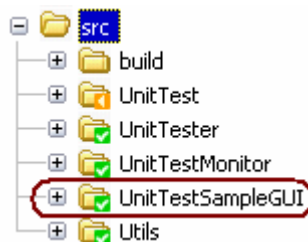
(*) The test results also get saved to a log file as, "**c:\unit_test.log**"



To get started using the UnitTest Monitor, which is just another unit test observer such as ConsoleLogger and StreamLogger, you will need to create a MFC dialog based application. Refer to the **UnitTestSampleGUI** project to get an idea of what is required.

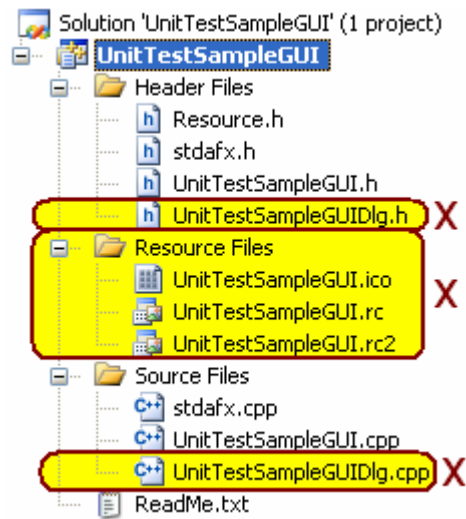
NOTE: you can remove the dialog class source and header files that are generated by the App Wizard from the project since we will be using the **UnitTestMonitorDlg** class.

I am going to assume the following project folder hierarchy:



You will simply create the MFC unit test app at the same level as "UnitTest" like "UnitTestSampleGUI", if you don't then you will need to modify the include and lib paths that are described further down.

After you have created the MFC Dialog based app, you can clean up the project files by removing the resources and the Wizard generated dialog box class as shown below for Project **UnitTestSampleGUI**.



Your next step is to edit the **InitInstance()** method of your MFC dialog based application. Inside the method, after the call to **AfxEnableControlContainer()** remove the Wizard Generated code that calls the dialog box, then add the call **InitUTMonitor()**

```
BOOL CUnitTestSampleGUIApp::InitInstance()
{
    // InitCommonControlEx() is required on Windows XP if an application
    // manifest specifies use of ComCtl32.dll version 6 or later to enable
    // visual styles. Otherwise, any window creation will fail.
    INITCOMMONCONTROLSEX InitCtrls;
    InitCtrls.dwSize = sizeof(InitCtrls);
    // Set this to include all the common control classes you want to use
    // in your application.
    InitCtrls.dwICC = ICC_WIN95_CLASSES;
    InitCommonControlEx(&InitCtrls);
    CWinApp::InitInstance();

    AfxEnableControlContainer();
    InitMonitor();

    // Since the dialog has been closed, return FALSE so that we exit the
    // application, rather than start the application's message pump.
    return FALSE;
}
```

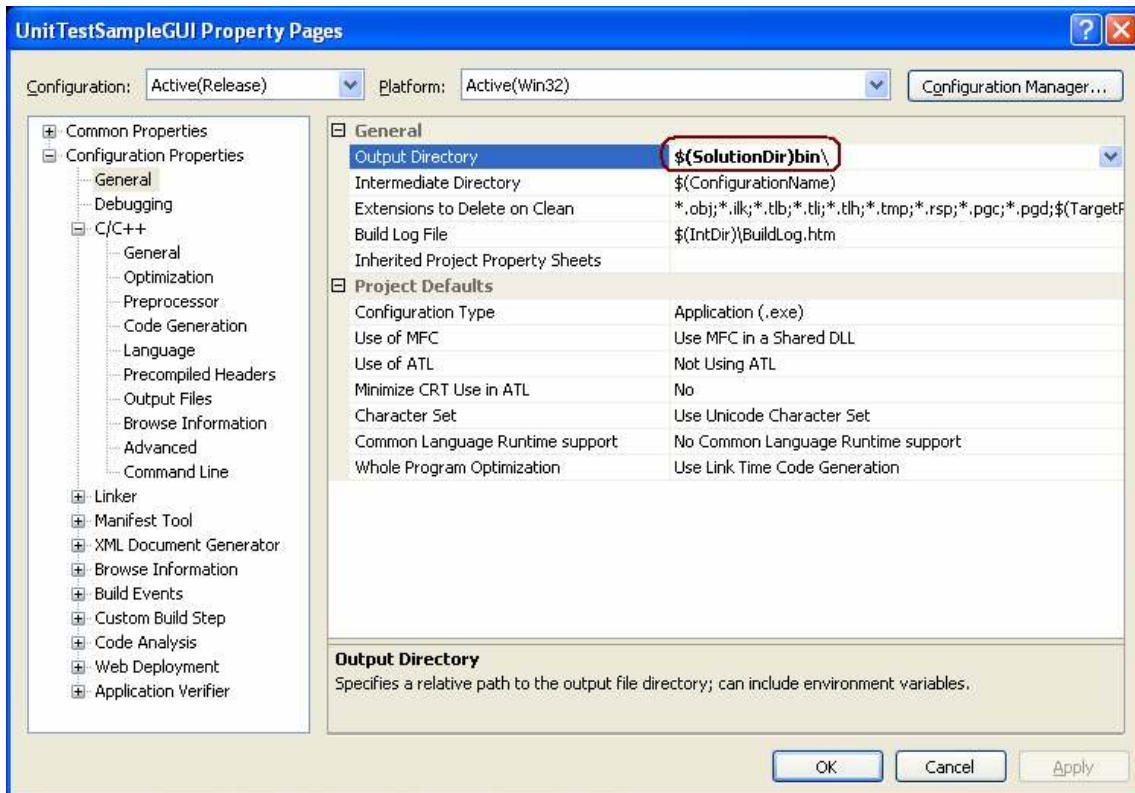
You will need to remove **#include "UnitTestSampleGUIDlg.h"**

In the same source file, add the following includes.

```
#include <UnitTest.h>
#include <UnitTestMonitorDlg.h>
```

Now we will have to set the project settings, so bring up the project properties dialog box.

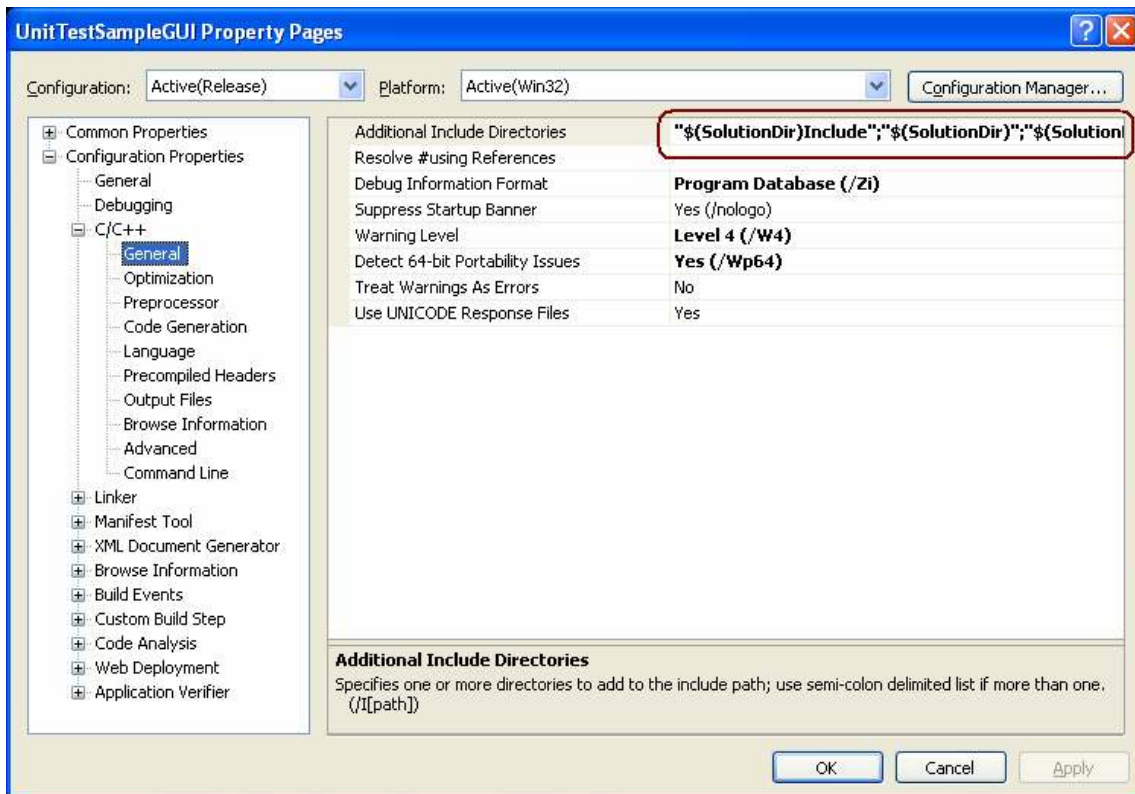
Modify the Output Directory as shown, this will put the exe where the UnitTest DLLs are that the unit test application will need to run.



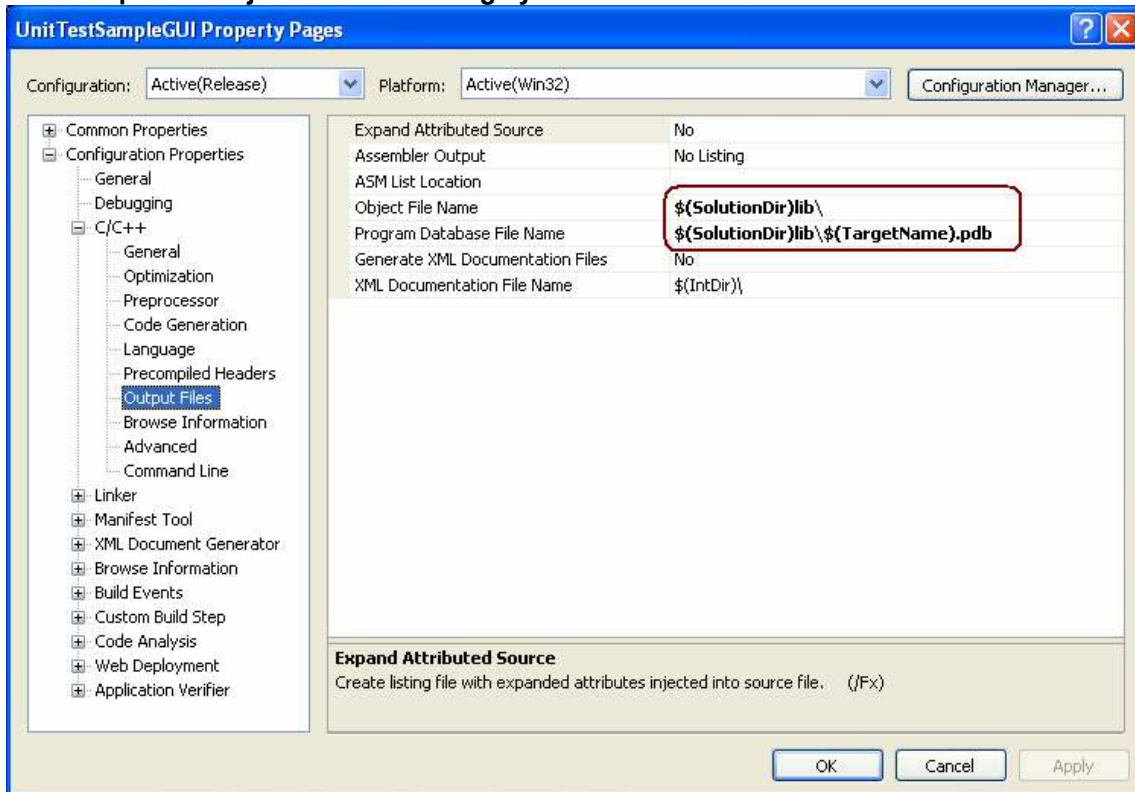
Inside the C/C++ General section, enter the include path shown below:

```
%Source_Dir%\UnitTest\Include;  
%Source_Dir%\UnitTest\UTUtils;  
%Source_Dir%\Utils\Include;  
%Source_Dir%\UnitTestMonitor
```

Replace %Source_Dir% with the location to the **UnitTest** extracted source file folder

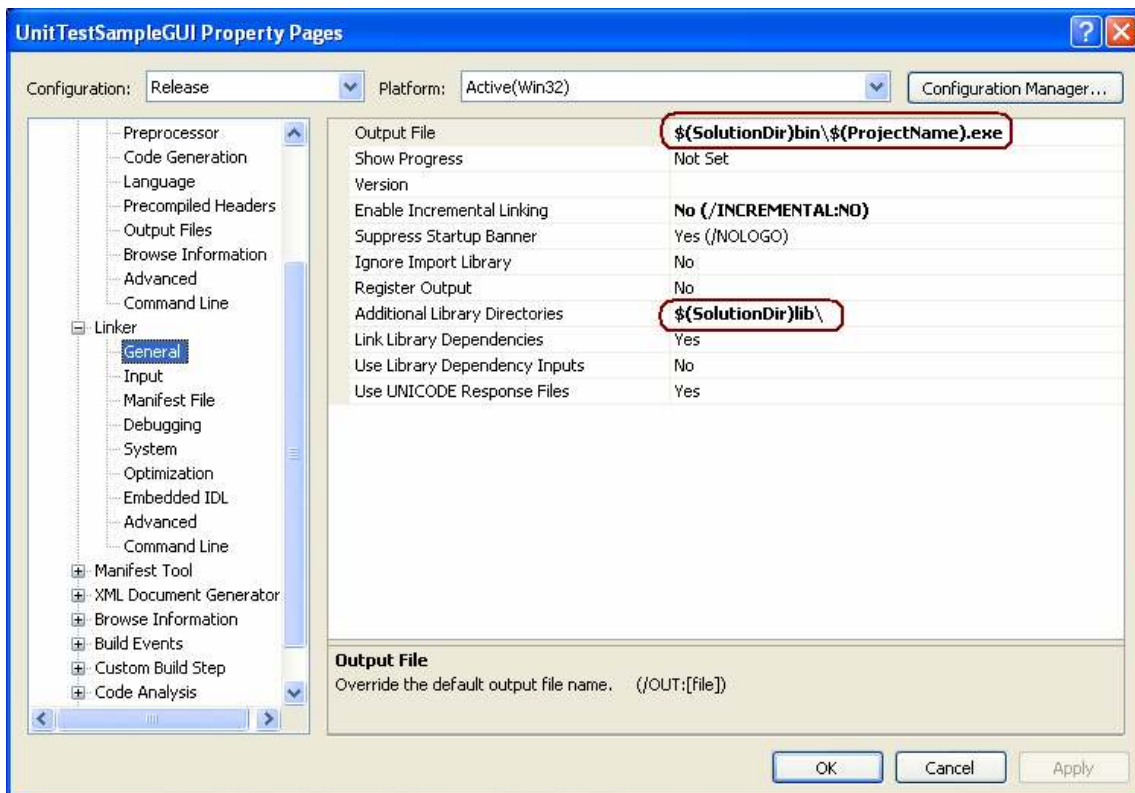


This will put the object files and debug symbols into the lib folder

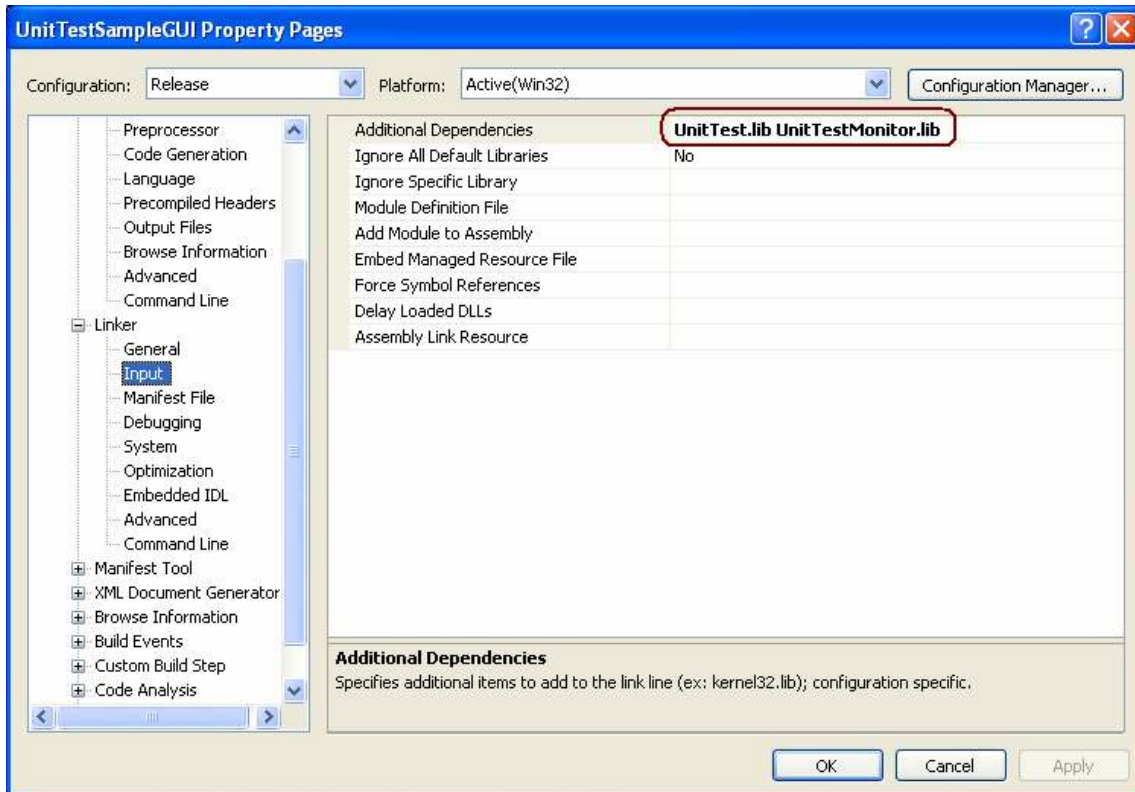


In the Linker General section, set Output File to the bin folder and set the lib path to:

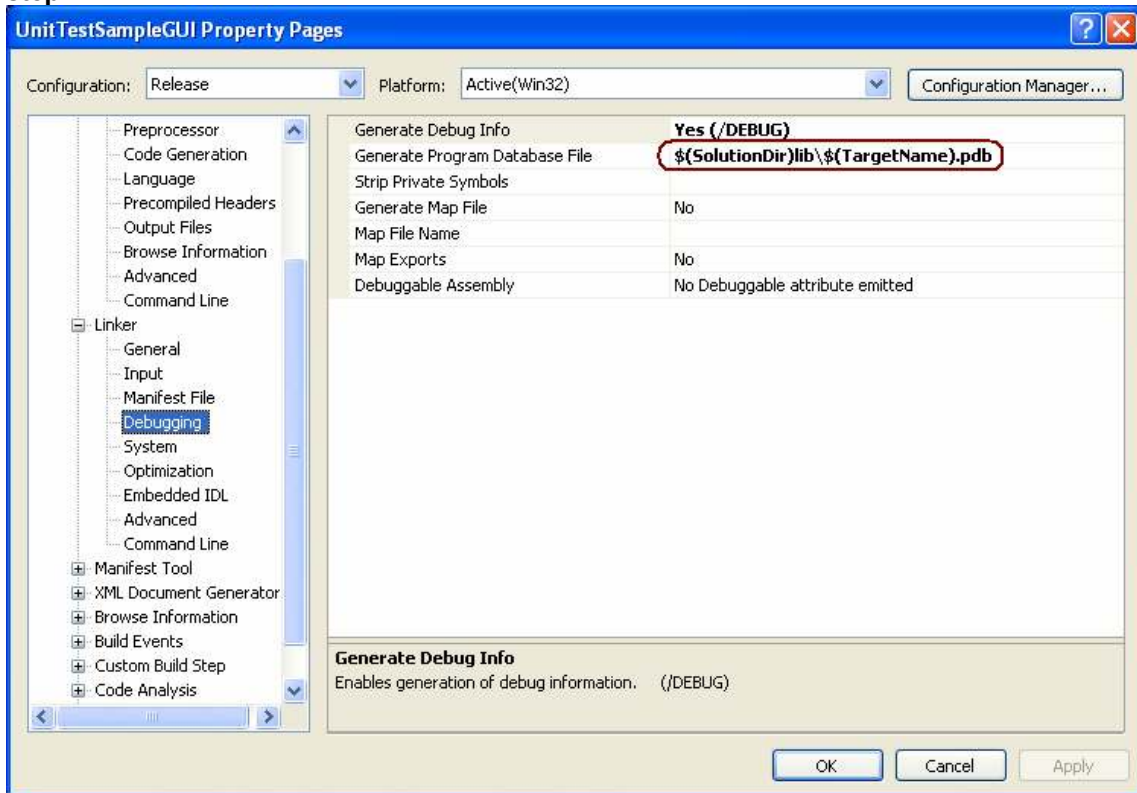
`%Source_Dir%\UnitTest\lib`



We need to tell the linker to link to “UnitTest.lib” and “UnitTestMonitor.lib”



This setting is optional, it merges the debug symbols with that generated in the C/C++ section by the preprocessor. Should you want to step into a test case, I recommend this step.



That's it, have fun testing!