Unit Testing C++ Code – CppUnit by Example

Venkat Subramaniam
venkats@agiledeveloper.com
http://www.agiledeveloper.com/download.aspx

Abstract

JUnit for Java popularized unit testing and developers using different languages are benefiting from appropriate tools to help with unit testing. In this article, I show–using examples–how to create unit tests for your C++ applications.

Why sudden interest in C++?

I have been asked at least three times in the past few weeks "Unit testing is cool, but how do I do that in my C++ application?" I figured, if my clients are interested in it, there should be general wider interest out there in that topic and so decided to write this article.

Assuming your familiarity with Unit Testing

In this article, I assume you are familiar with unit testing. There are some great books to read on that topic (I've listed my favorites[1, 2] in the references section). You may also refer to my article on Unit Testing[3].

How is Unit Testing different in C++?

Java and C# (VB.NET) have a feature that C++ does not – reflection. If you have used JUnit[4] or NUnit[5], you know how (easy it is) to write a test case. You either derived from a TestCase class in the case of Java or you mark your class with a TestFixture attribute in the case of .NET (With Java 5's annotation, you may also look forward, in JUnit 4.0, to marking your Java test case instead of extending TestCase). Using reflection the unit testing tool (JUnit/NUnit) finds your test methods dynamically.

Since C++ does not have support for reflection, it becomes a bit of a challenge to write a unit test in C++, at least the JUnit way. You will have to exploit some of the traditional features of C++ to get around the lack of reflection feature.

How to tell what your tests are?

In C++, how can you make the program know of the presence of a certain object dynamically or automatically? You can do that by registering an instance of your class with a static or global collection. Let's understand this with an example. If you are an expert in C++, browse through this section. If you will benefit from this example, then read it carefully and try it out for yourself.

Let's assume we have different types of Equipment: Equipment1, Equipment2, etc. More Equipment types may be added or some may be removed. We want to find what kind of equipment is available for us to use. The following code does just that. Let's take a look at it step by step.

```
#include <string>
using namespace std;
```

```
class Equipment
{
public:
	virtual string info() = 0;
};
```

`Equipment` is defined an abstract base class with a pure virtual function `info()`.

```
#include "Equipment.h"
#include <vector>

class EquipmentCollection
{
public:
	static void add(Equipment* pEquipment)
	{
		if (pEquipmentList == NULL)
		{
			pEquipmentList = new vector<Equipment*>();
		}

		pEquipmentList->push_back(pEquipment);
	}

	~EquipmentCollection()
	{
		delete pEquipmentList;
	}

	static vector<Equipment*>& get()
	{
		return *pEquipmentList;
	}
private:
	static vector<Equipment*>* pEquipmentList;
};
```

`EquipmentCollection` holds a collection (`vector`) of Equipment through a static pointer `pEquipmentList`. This static pointer is assigned to an instance of `vector<Equipment*>`, an instance of `vector` of `Equipment` pointers, the first time the `add()` method is called (the given code is not thread-safe). In the `add()` method, we add the pointer to the given `Equipment` to the collection. The collection is initialized in the EquipmentCollection.cpp file as shown below:

```
//EquipmentCollection.cpp
#include "EquipmentCollection.h"

vector<Equipment*>* EquipmentCollection::pEquipmentList = NULL;
```

How are elements placed into this collection? I use a class `EquipmentHelper` to do this:

```
#include "EquipmentCollection.h"

class EquipmentHelper
```

```
{
public:
	EquipmentHelper(Equipment* pEquipment)
	{
		EquipmentCollection::add(pEquipment);
	}
};
```

The constructor of `EquipmentHelper` receives `Equipment` and adds it to the collection.

`Equipment1` is shown below:

```
#include "EquipmentHelper.h"

class Equipment1 : public Equipment
{
public:
	virtual string info()
	{
		return "Equipment1";
	}
private:
	static EquipmentHelper helper;
};
```

The class holds a static instance of `EquipmentHelper`. The static instance is initialized as shown below in the Equipment1.cpp file:

```
//Equipment1.cpp
#include "Equipment1.h"

EquipmentHelper Equipment1::helper(new Equipment1());
```

The `helper` is provided with an instance of `Equipment1` which it registers with the collection in `EquipmentCollection`. Similarly, I have another class `Equipment2` with a static member variable named `helper` of type `EquipmentHelper` and it is given an instance of `Equipment2` (I have not shown the code as it is similar to the code for `Equipment1`). You may write other classes (like Equipment3, Equipment4, etc.) similarly.

Let's take a closer look at what's going on. When a C++ program starts up, before the code in `main()` executes, all static members of all the classes are initialized. The order in which these are initialized is, however, unpredictable. The members are initialized to 0 unless some other value is specified. In the example code above, before main starts, the helper within Equiment1, Equipment2, etc. are all initialized, and as a result, the collection of `Equipment` is setup with an instance of these classes. Now, let's take a look at the `main()` function.

```
#include "EquipmentCollection.h"
#include <iostream>
using namespace std;
```

```
int main(int argc, char* argv[])
{
	cout << "List contains " << endl;

	vector<Equipment*> equipmentList = EquipmentCollection::get();
	vector<Equipment*>::iterator iter = equipmentList.begin();
	while(iter != equipmentList.end())
	{
		cout << (*iter)->info() << endl;
		iter++;
	}

	return 0;
}
```

In `main()`, I fetch an instance of the collection and iterate over its contents, getting instances of different kinds of Equipment. I invoke the `info()` method on each Equipment and display the result. The output from the above program is shown below:

```
List contains
Equipment1
Equipment2
```

If you are curious about the above example, go ahead, copy and paste the code above and give it a try. Write your own `Equipment3` class and see if `main()` prints information about it without any change.

Now that we have figured out how to dynamically recognize classes being added to our code, let's take a look at using macros to do the job. If you are going to write another `Equipment` class, like `Equipment3`, you will have to follow certain steps:

1. You must declare a static member of `EquipmentHelper`
2. You must initialize it with an instance of you class

That is really not that hard to do, however, macros may make it a tad easier. Let's take a look at first defining two macros.

```
#include "EquipmentCollection.h"

#define DECLARE_EQUIPMENT() private: static EquipmentHelper helper;
#define DEFINE_EQUIPMENT(TYPE) EquipmentHelper TYPE::helper(new
TYPE());

class EquipmentHelper
{
public:
	EquipmentHelper(Equipment* pEquipment)
	{
		EquipmentCollection::add(pEquipment);
	}
};
```

Now, I can use these macros to write a class `Equipment3` as shown below:

```
#include "EquipmentHelper.h"

class Equipment3 : public Equipment
{
public:
      virtual string info()
      {
            return "Equipment3";
      }

DECLARE_EQUIPMENT()
};

//Equipment3.cpp
#include "Equipment3.h"

DEFINE_EQUIPMENT(Equipment3)
```

Of course we did not make any change to `main()`. The output from the program is shown below:

```
List contains
Equipment1
Equipment2
Equipment3
```

There is no guarantee that the program will produce the output in this nice order. You may get a different ordering from what's shown above.

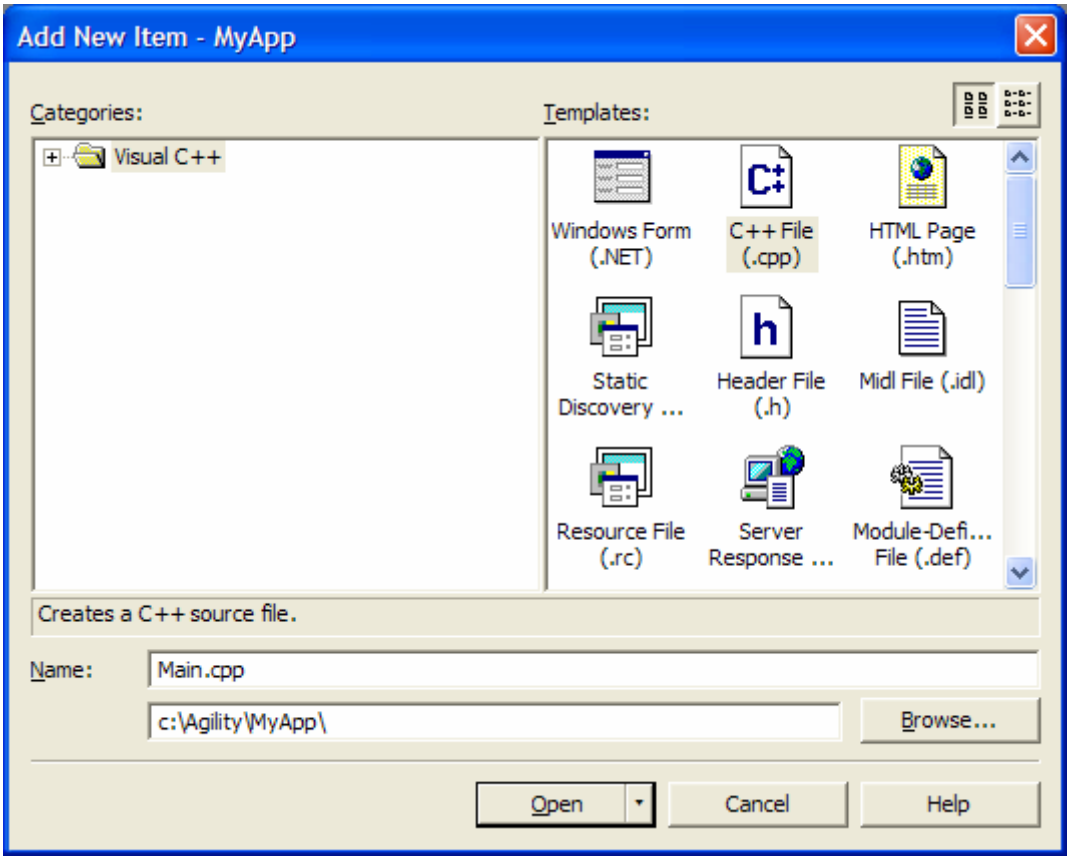
OK, let's move on to what we are here for, to see how to write unit tests with CPPUnit.

Setting up CPPUnit

I am using CPPUnit 1.10.2[6] in this example. Download cppunit-1.10.2.tar.gz and uncompress it on your system. On mine, I have uncompressed it in c:\programs. Open CppUnitLibraries.dsw which is located in cppunit-1.10.2\src directory and compile it. I had to build it twice to get a clean compile. After successful build, you should see cppunit-1.10.2\lib directory.

Setting up Your Project

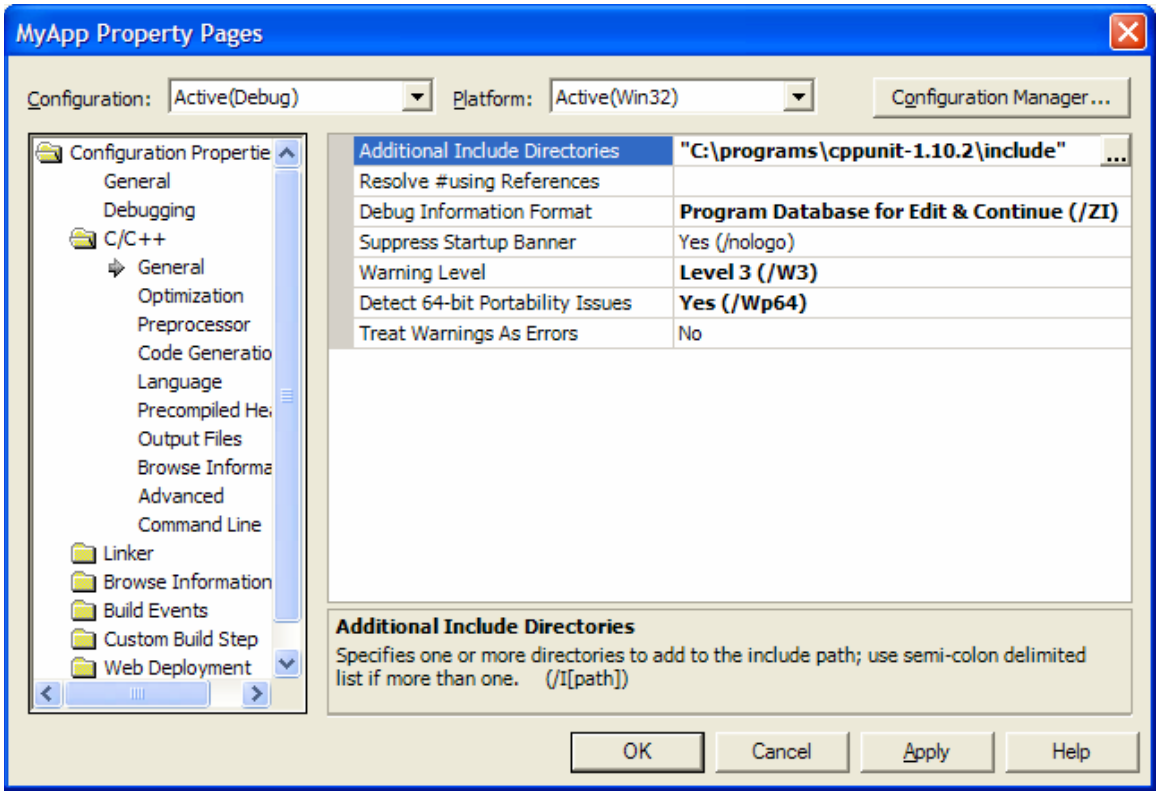
We will use Visual Studio 2003 to build this example – unmanaged C++. Created a console application named MyApp. In Solution Explorer, right click on "Source Files" and select Add | Add New Item…. Create a file named Main.cpp as shown here:



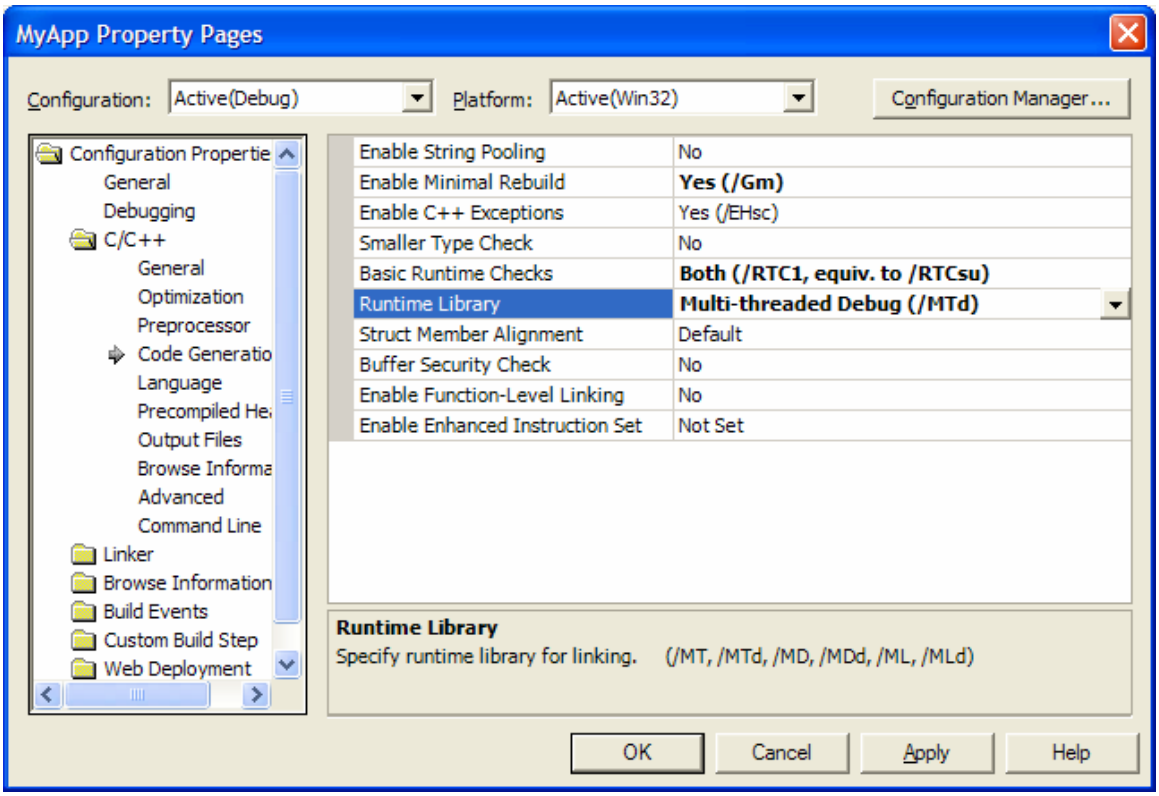
Write a dummy `main()` method for now:

```
int main(int argc, char* argv[])
{
	return 0;
}
```

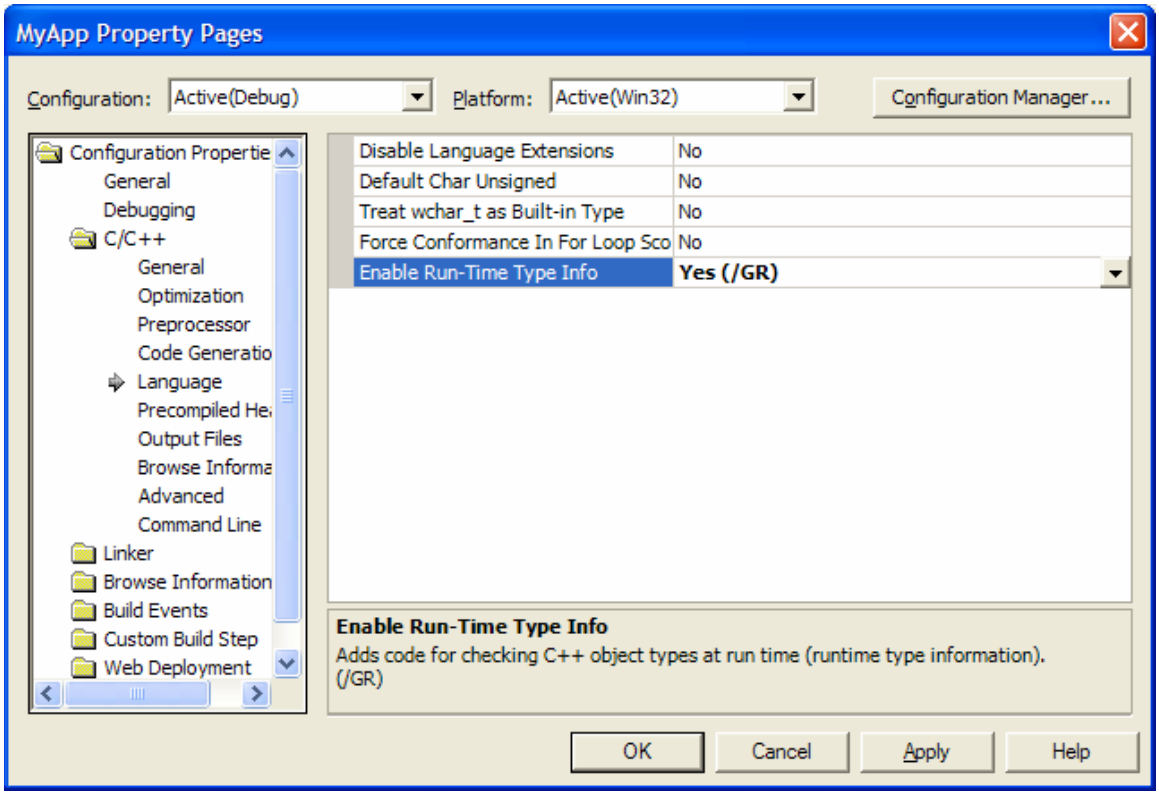
Right click on the project in Solutions Explorer and select Properties. For the “Additional Include Directories” for C/C++ properties, enter the directory where you have CPPUnit installed on your machine. I have it under c:\programs directory. So, I entered the following:



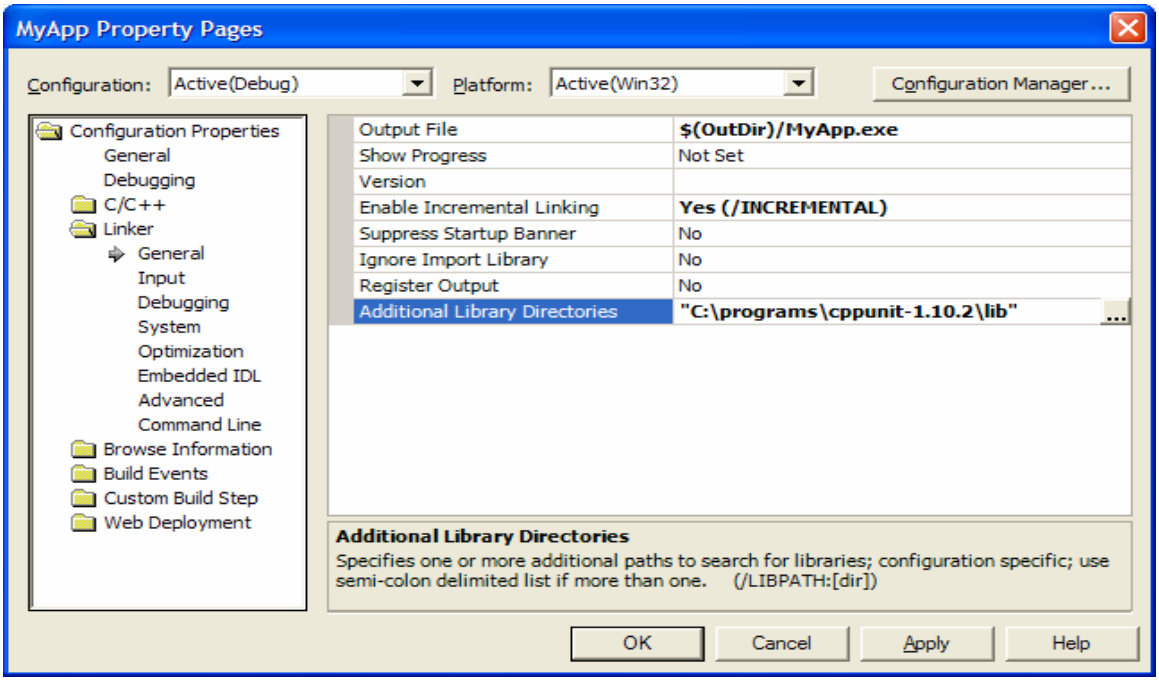
Modify the “Runtime Library” in “Code Generation” as shown below:



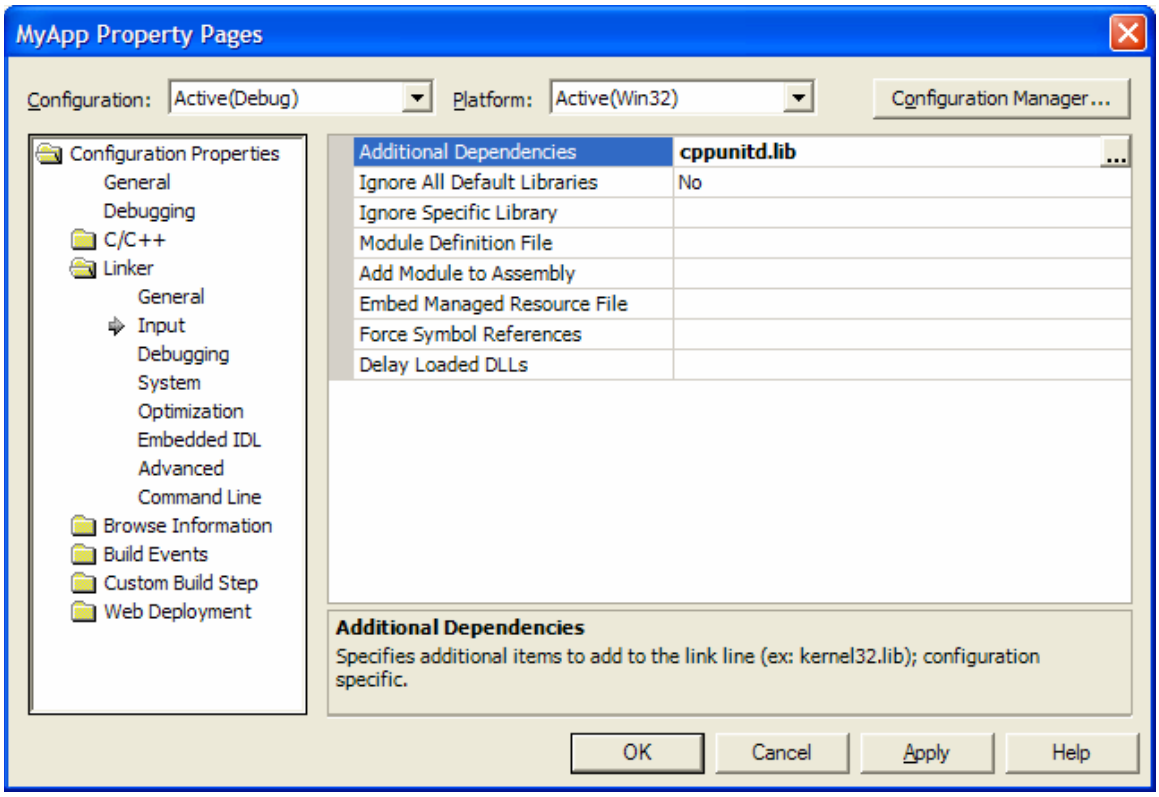
Set the “Enable Run-Time Type Info” in “Language” properties as shown below:



Two more steps before we can write code. Under “Linker,” for “General” properties, set “Additional Library Directories” to point to the lib directory under CPPUnit as shown below:



Finally, set the “Additional Dependencies” in “Input” as shown below:



Running your unit test

Let’s now create a Unit Test. We will start in Main.cpp that we have created in the last section. Modify that file as shown below:

```
1: #include <cppunit/CompilerOutputter.h>
2: #include <cppunit/extensions/TestFactoryRegistry.h>
3: #include <cppunit/ui/text/TestRunner.h>
4:
5: using namespace CppUnit;
6:
7: int main(int argc, char* argv[])
8: {
9:   CppUnit::Test* suite =
10:         CppUnit::TestFactoryRegistry::getRegistry().makeTest();
11:
12:   CppUnit::TextUi::TestRunner runner;
13:   runner.addTest( suite );
14:
15:   runner.setOutputter( new CppUnit::CompilerOutputter(
16:                                   &runner.result(), std::cerr ) );
17:
18:   return runner.run() ? 0 : 1;
19: }
```

In lines 1 to 3, we have included the necessary header files from CPPUnit. The `CompilerOutputter` class is useful to display the results from the run of a test. The `TestFactoryRegistry` class keeps a collection of tests (this is like the `EquipmentCollection` class we saw in our example). The `TestRunner` class will execute each test from the test classes we register with the `TestFactoryRegistry`. This is like the code that iterated though the `Equipment` collection in our example. In that example, we iterated and printed the result of `info()` method invocation. `TestRunner` does something useful – runs your tests.

In lines 9 and 10, I invoke the `TestFactoryRegistry`'s `getRegistry()` method and invoke its `makeTest()`method. This call will return all the tests found in your application. In lines 12 through 16, I have created a `TestRunner` object, the one that will take care of running the tests and reporting the output using the `CompilerOutputter`. The `CompilerOutputter` will format the error messages in the compiler compatible format – the same format as the compiler errors – so you can easily identify the problem areas and also this facilitates IDEs to jump to the location of the error in your test run.

Finally in line 18, I ask the tests to run. We haven't written any tests yet. That is OK, let's go ahead and give this a try. Go ahead, compile the code and run it. You should get the following output:

```
OK (0)
```

This says that all is well (after all there is not a whole lot we have done to mess up!) and that no tests were found.

Writing unit tests

We will take the test first driven approach[1,3] to writing our code. We will build a simple calculator with two methods, `add()` and `divide()`. Let's start by creating a Test class as shown below:

```
// CalculatorTest.h

#include <cppunit/extensions/HelperMacros.h>

using namespace CppUnit;

class CalculatorTest : public TestFixture
{
      CPPUNIT_TEST_SUITE(CalculatorTest);
      CPPUNIT_TEST_SUITE_END();
};

// CalculatorTest.cpp

#include "CalculatorTest.h"

CPPUNIT_TEST_SUITE_REGISTRATION(CalculatorTest);
```

You may study the macros I have used in the header file. I will expand here on the macro I have used in the cpp file, that is `CPPUNIT_TEST_SUITE_REGISTRATION`:

```
#define CPPUNIT_TEST_SUITE_REGISTRATION( ATestFixtureType )      \
  static CPPUNIT_NS::AutoRegisterSuite< ATestFixtureType >       \
            CPPUNIT_MAKE_UNIQUE_NAME(autoRegisterRegistry__ )
```

Notice the creation of a static object. This object will take care of registering the test fixture object with the test registry.

Now, let's go ahead and write a test:

```
// CalculatorTest.h

#include <cppunit/extensions/HelperMacros.h>

using namespace CppUnit;

class CalculatorTest : public TestFixture
{
      CPPUNIT_TEST_SUITE(CalculatorTest);
      CPPUNIT_TEST(testAdd);
      CPPUNIT_TEST_SUITE_END();

public:
      void testAdd()
      {
      }
};
```

I have created a method `testAdd()` and have declared a macro `CPPUNIT_TEST` to introduce that method as a test. Let's compile and execute this code. The output is shown below:

```
.

OK (1)
```

The above output shows that one test was executed successfully.

Let's modify the test code to use the Calculator:

```
      void testAdd()
      {
            Calculator calc;

            CPPUNIT_ASSERT_EQUAL(5, calc.add(2, 3));
      }
```

I am using the `CPPUNIT_ASSERT_EQUAL` to assert that the method add returns the expected value of 5. Let's create the Calculator class and write the `add`() method as shown below:

```
#pragma once

class Calculator
{
public:
	Calculator(void);
	virtual ~Calculator(void);

	int add(int op1, int op2)
	{
		return 0;
	}
};
```

Let's execute the test and see what the output is:

```
.F

c:\...\calculatortest.h(20) : error : Assertion
Test name: CalculatorTest::testAdd
equality assertion failed
- Expected: 5
- Actual  : 0

Failures !!!
Run: 1   Failure total: 1   Failures: 1   Errors: 0
```

The output shows us that my test failed (obviously as I am returning a 0 in the `add()` method). Let's fix the code now to return op1 + op2. The output after the fix is:

```
.

OK (1)
```

Let's write a test for the `divide()` method as shown below:

```
	void testDivide()
	{
		Calculator calc;

		CPPUNIT_ASSERT_EQUAL(2.0, calc.divide(6, 3));
	}
```

Remember to add

```
	CPPUNIT_TEST(testDivide);
```

as well.

Now the `divide()` method in the `Calculator` class looks like this:

```
double divide(int op1, int op2)
{
	return op1/op2;
}
```

The output is shown below:

```
..

OK (2)
```

It shows us that two tests were executed successfully.

Let's write one more test, this time a negative test:

```
void testDivideByZero()
{
	Calculator calc;

	calc.divide(6, 0);
}
```

We are testing for division by zero. Intentionally I have not added any asserts yet. If we run this, we get:

```
...E

##Failure Location unknown## : Error
Test name: CalculatorTest::testDivideByZero
uncaught exception of unknown type

Failures !!!
Run: 3   Failure total: 1   Failures: 0   Errors: 1
```

CppUnit tells us that there was an exception from the code. Indeed, we should get an exception. But the success of this test is the method divide throwing the exception. So, the test must have reported a success and not a failure. How can we do that? This is where `CPP_FAIL` comes in:

```
void testDivideByZero()
{
	Calculator calc;

	try
	{
		calc.divide(6, 0);
		CPPUNIT_FAIL(
		 "Expected division by zero to throw exception!!!!");
	}
	catch(Exception ex)
	{
		// Fail throws Exception,
		//we will throw in back to CppUnit to handle
```

```
                throw;
        }
        catch(...)
        {
                // Division by Zero in this case. Good.
        }
}
```

If the method `divide()` throws an exception, then we are good. If the method does not throw exception, then we force fail this test by call to the `CPPUNIT_FAIL`.

Let's run the test now:

```
...

OK (3)
```

We have passing tests.

Reexamining the Divide by Zero

Let's make a change to the `divide()` method in Calculator–we will make the method take `double` instead of `int` as parameters.

```
double divide(double op1, double op2)
{
        return op1/op2;
}
```

Let's rerun out tests and see what we get:

```
...F

c:\agility\myapp\calculatortest.h(39) : error : Assertion
Test name: CalculatorTest::testDivideByZero
forced failure
- Expected division by zero to throw exception!!!!

Failures !!!
Run: 3   Failure total: 1   Failures: 1   Errors: 0
```

CppUnit tells us that the division by zero did not throw the exception as expected! The reason is the behavior of floating point division is different from the integer division. While integer division by zero throws an exception, division by zero for double returns infinity. This further illustrates how `CPPUNIT_FAIL` helps.

Conclusion

If you are a C++ developer, you can take advantage of unit testing using CppUnit. Since C++ does not provide some features of Java and .NET languages, you have to work around those limitations. However, CppUnit addresses these by providing helper macros. We first took time to see how in C++ we can identify classes automatically. Then we delved into examples of using CppUnit.

References

1. Test Driven Development: By Example, Kent Beck.
2. Pragmatic Unit Testing in C# with NUnit, Andy Hunt, Dave Thomas.
3. Test Driven Development – Part I: TFC, http://www.agiledeveloper.com/download.aspx
4. http://www.junit.org
5. http://www.sourceforge.net/projects/nunit
6. http://www.sourceforge.net/projects/cppunit