Open source C/C++ unit testing tools, Part 2: Get to know CppUnit

Arpan Sen

January 12, 2010

In this second article in the series on open source unit testing utilities, get to know CppUnit, the `C++` port of the JUnit test framework.

View more content in this series

This article, the second in a series on open source tools for unit testing, introduces the very popular CppUnit—the `C++` port of the JUnit test framework originally developed by Eric Gamma and Kent Beck. The `C++` port was created by Michael Feathers and constitutes a variety of classes that help both white-box testing and creating your own regression suite. This article introduces some of the more useful CppUnit features, like TestCase, TestSuite, TestFixture, TestRunner, and the helper macros.

Frequently used acronyms

- **GUI:** Graphical user interface
- **XML:** Extensible Markup Language

Downloading and installing CppUnit

For purposes of this article, I download and installed CppUnit on a Linux® machine (kernel 2.4.21) with g++-3.2.3 and make-3.79.1. The installation is simple and standard: run the `configure` command followed by `make` and `make install`. Note that for certain platforms like cygwin, this process may not run smoothly, so be sure to look into the INSTALL-unix document that comes with the installation for details. If the installation is successful, you should see the include and lib folders for CppUnit in the install path—let's call that *CPPUNIT_HOME*. Listing 1 shows the file structure.

Listing 1. The CppUnit installation hierarchy

```
[arpan@tintin] echo $CPPUNIT_HOME
/home/arpan/ibm/cppUnit
[arpan@tintin] ls $CPPUNIT_HOME
bin   include   lib   man   share
```

To compile a test that uses CppUnit, you must build sources:

```
g++ <C/C++ file> -I$CPPUNIT_HOME/include –L$CPPUNIT_HOME/lib -lcppunit
```

Note that if you are using the shared library version of CppUnit, you may need to use the `–ldl` option to compile sources. After installation, you may also need to modify the UNIX® environment variable LD_LIBRARY_PATH to reflect the location of libcppunit.so.

Creating a basic test using CppUnit

The best way to learn CppUnit is to create a leaf-level test. CppUnit comes with a whole host of predefined classes that you'll make good use of while designing the tests. For the sake of continuity, recall the poorly designed string class discussed in part 1 of this series (see Listing 2).

Listing 2. An uninspiring string class

```
#ifndef _MYSTRING
#define _MYSTRING

class mystring {
  char* buffer;
  int length;
  public:
    void setbuffer(char* s) { buffer = s; length = strlen(s); }
    char& operator[ ] (const int index) { return buffer[index]; }
    int size( ) { return length; }
 };

#endif
```

Some of the typical string-related checks include verifying that an empty string has 0 size and accessing out of index from the string results in an error message/exception. Listing 3 uses CppUnit for such testing.

Listing 3. Unit tests for the string class

```
#include <cppunit/TestCase.h>
#include <cppunit/ui/text/TextTestRunner.h>

class mystringTest : public CppUnit::TestCase {
public:

  void runTest() {
    mystring s;
    CPPUNIT_ASSERT_MESSAGE("String Length Non-Zero", s.size() != 0);
  }
};

int main ()
{
  mystringTest test;
  CppUnit::TextTestRunner runner;
  runner.addTest(&test);

  runner.run();
  return 0;
}
```

The first class from the CppUnit code base you'll learn is `TestCase`. To create a unit test for the string class, you need to subclass the `CppUnit::TestCase` class and override the `runTest` method. Now that the test itself is defined, instantiate the `TextTestRunner` class, which is kind of a controller class to which you must add the individual tests (the `vide addTest` method). Listing 4 shows the output from the `run` method.

Listing 4. Output of the code from Listing 3

```
[arpan@tintin] ./a.out
!!!FAILURES!!!
Test Results:
Run:  1   Failures: 1   Errors: 0


1) test:  (F) line: 26 try.cc
assertion failed
- Expression: s.size() == 0
- String Length Non-Zero
```

Just to be sure the assertion works, negate the condition in the macro `CPPUNIT_ASSERT_MESSAGE`. Listing 5 shows the output from the code when the condition is `s.size() ==0`.

Listing 5. Output of the code from Listing 3 with condition s.size( ) == 0

```
[arpan@tintin] ./a.out

OK (1 tests)
```

Note that `TestRunner` is not the only way to run a single test or a test suite. CppUnit provides an alternate class hierarchy—namely, the templatized `TestCaller` class—to run tests. Instead of the `runTest` method, you can use the `TestCaller` class to execute any method. Listing 6 provides a small example.

Listing 6. Using TestCaller to run tests

```
class ComplexNumberTest ... {
  public:
     void ComplexNumberTest::testEquality( ) { … }
};

CppUnit::TestCaller<ComplexNumberTest> test( "testEquality",
                                             &ComplexNumberTest::testEquality );
CppUnit::TestResult result;
test.run( &result );
```

In the above example, a class of type `ComplexNumberText` is defined with a method `testEquality` (to test the equality of two complex numbers). `TestCaller` is templatized with this class and, as in the case of `TestRunner`, you make a call to the `run` method for test execution. Using the `TestCaller` class as-is is not of much use, though: the `TextTestRunner` class automatically handles the display of the output. In the case of `TestCaller`, you have to use separate classes to handle the output. You'll see this type of code flow later in the article when you use the `TestCaller` class to define a customized test suite.

Using assertions

CppUnit provides several routines for the most common assertion scenarios. These are defined as public static methods of the `CppUnit::Asserter` class, available in the header Asserter.h. There also exist predefined macros for most of these classes in the header TestAssert.h. From Listing 2, here's how `CPPUNIT_ASSERT_MESSAGE` is defined (see Listing 7):

Listing 7. Definition of CPPUNIT_ASSERT_MESSAGE

```
#define CPPUNIT_ASSERT_MESSAGE(message,condition)                          \
  ( CPPUNIT_NS::Asserter::failIf( !(condition),                            \
                                  CPPUNIT_NS::Message( "assertion failed", \
                                                       "Expression: "      \
                                                       #condition,         \
                                                       message ),          \
                                  CPPUNIT_SOURCELINE() ) )
```

The declaration of the `failIf` method on which the assertion is based is provided in Listing 8.

Listing 8. Declaration of the failIf method

```
struct Asserter
{
…
  static void CPPUNIT_API failIf( bool shouldFail,
                                  const Message &message,
                                  const SourceLine &sourceLine = SourceLine() );
…
}
```

If the condition in the `failIf` method becomes True, then an exception is thrown. The `run` method handles this process internally. Yet another interesting and useful macro is `CPPUNIT_ASSERT_DOUBLES_EQUAL`, which checks for equality between two doubles with a tolerance value (so, `|expected – actual | ≤ delta`). Listing 9 provides the macro definition.

Listing 9. CPPUNIT_ASSERT_DOUBLES_EQUAL macro definition

```
void CPPUNIT_API assertDoubleEquals( double expected,
                                     double actual,
                                     double delta,
                                     SourceLine sourceLine,
                                     const std::string &message );
#define CPPUNIT_ASSERT_DOUBLES_EQUAL(expected,actual,delta)        \
  ( CPPUNIT_NS::assertDoubleEquals( (expected),            \
                                    (actual),              \
                                    (delta),               \
                                    CPPUNIT_SOURCELINE(),  \
                                    "" ) )
```

Test the string class once more

One way of testing different facets of the `mystring` class is to continue adding more checks inside the `runTest` method. However, except for the simplest of classes, doing so quickly becomes an unmanageable affair. This is where you need to define and use test suites. Look at Listing 10, which defines a test suite for your string class.

Listing 10. Making a test suite for a string class

```
#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/ui/text/TextTestRunner.h>
```

```
#include <cppunit/extensions/HelperMacros.h>

class mystringTest : public CppUnit::TestCase {
public:
  void checkLength() {
    mystring s;
    CPPUNIT_ASSERT_MESSAGE("String Length Non-Zero", s.size() == 0);
  }

  void checkValue() {
    mystring s;
    s.setbuffer("hello world!\n");
    CPPUNIT_ASSERT_EQUAL_MESSAGE("Corrupt String Data", s[0], 'w');
  }

  CPPUNIT_TEST_SUITE( mystringTest );
  CPPUNIT_TEST( checkLength );
  CPPUNIT_TEST( checkValue );
  CPPUNIT_TEST_SUITE_END();
};
```

That was simple enough. You use the `CPPUNIT_TEST_SUITE` macro to define the test suite. Individual methods that are part of the `mystringTest` class form unit tests in the test suite. We'll examine these macros and their contents in a moment, but first, take a look at the client code in Listing 11, which uses this test suite.

Listing 11. Client code using the test suite for the mystring class

```
CPPUNIT_TEST_SUITE_REGISTRATION ( mystringTest );

int main ()
{
  CppUnit::Test *test =
    CppUnit::TestFactoryRegistry::getRegistry().makeTest();
  CppUnit::TextTestRunner runner;
  runner.addTest(test);

  runner.run();
  return 0;
}
```

Listing 12 shows the output when the code in Listing 11 is run.

Listing 12. Output of the code from Listings 10 and 11

```
[arpan@tintin] ./a.out
!!!FAILURES!!!
Test Results:
Run:  2   Failures: 2   Errors: 0


1) test: mystringTest::checkLength (F) line: 26 str.cc
assertion failed
- Expression: s.size() == 0
- String Length Non-Zero


2) test: mystringTest::checkValue (F) line: 32 str.cc
equality assertion failed
- Expected: h
- Actual  : w
- Corrupt String Data
```

The `CPPUNIT_ASSERT_EQUAL_MESSAGE` is defined in the header TestAssert.h and checks whether the expected and actual arguments match. In the negative, the message specified is displayed. The `CPPUNIT_TEST_SUITE` macro, defined in HelperMacros.h, simplifies creating a test suite and adding individual tests to it. Internally, a templatized object of type `CppUnit::TestSuiteBuilderContext` is created (this is the equivalent of a test suite in the CppUnit context), and each call to `CPPUNIT_TEST` adds the corresponding class method to this suite. Needless to say, it is the class method that serves as the unit test to the code. Note the ordering of the macros: for the code to compile individual `CPPUNIT_TEST` macros, it must be between the `CPPUNIT_TEST_SUITE` and `CPPUNIT_TEST_SUITE_END` macros.

Incorporating new tests

Developers keep adding functionality to the code over time, and this necessitates further testing. Continuing to add tests to the same test suite adds to clutter over time, and the incremental nature of the changes for which the tests were developed in the first place gets lost. Thankfully, CppUnit has a useful macro called `CPPUNIT_TEST_SUB_SUITE` that you can use to extend existing test suites. Listing 13 uses this macro.

Listing 13. Extending test suites

```
class mystringTestNew : public mystringTest {
public:
  CPPUNIT_TEST_SUB_SUITE (mystringTestNew, mystringTest);
  CPPUNIT_TEST( someMoreChecks );
  CPPUNIT_TEST_SUITE_END();

  void someMoreChecks() {
    std::cout << "Some more checks...\n";
  }
};

CPPUNIT_TEST_SUITE_REGISTRATION ( mystringTestNew );
```

Note that the new class `mystringTestNew` derives from the previous `myStringTest` class. The `CPPUNIT_TEST_SUB_SUITE` macro accepts the new class and its super class as the two arguments. On the client side, you just register the new class instead of both of the classes. That's it: the rest of the syntax is much the same for creating test suites.

Customizing tests using fixtures

A *fixture,* or a `TestFixture` in the CppUnit context, is meant to provide clean setup and exit routines for individual tests. To use fixtures, you derive your test class from `CppUnit::TestFixture` and override the predefined `setUp` and `tearDown` methods. The `setUp` method is called before execution of a unit test, and `tearDown` is called when the test is executed. Listing 14 shows how a `TestFixture` is used.

Listing 14. Making a test suite with test fixtures

```
#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/ui/text/TextTestRunner.h>
#include <cppunit/extensions/HelperMacros.h>
```

```
class mystringTest : public CppUnit::TestFixture {
public:
  void setUp() {
     std::cout << “Do some initialization here…\n”;
  }

  void tearDown() {
      std::cout << “Cleanup actions post test execution…\n”;
  }

  void checkLength() {
    mystring s;
    CPPUNIT_ASSERT_MESSAGE("String Length Non-Zero", s.size() == 0);
  }

  void checkValue() {
    mystring s;
    s.setbuffer("hello world!\n");
    CPPUNIT_ASSERT_EQUAL_MESSAGE("Corrupt String Data", s[0], 'w');
  }

  CPPUNIT_TEST_SUITE( mystringTest );
  CPPUNIT_TEST( checkLength );
  CPPUNIT_TEST( checkValue );
  CPPUNIT_TEST_SUITE_END();
};
```

Listing 15 shows the output of the code from Listing 14.

Listing 15. Output of the code in Listing 14

```
[arpan@tintin] ./a.out
. Do some initialization here…
FCleanup actions post test execution…
. Do some initialization here…
FCleanup actions post test execution…

!!!FAILURES!!!
Test Results:
Run:  2   Failures: 2   Errors: 0


1) test: mystringTest::checkLength (F) line: 26 str.cc
assertion failed
- Expression: s.size() == 0
- String Length Non-Zero


2) test: mystringTest::checkValue (F) line: 32 str.cc
equality assertion failed
- Expected: h
- Actual  : w
- Corrupt String Data
```

As you can see from this output, the setup and teardown routine messages appear once each per unit test execution.

Creating a test suite without using macros

It is possible to create a test suite without using any of the helper macros. There is no particular benefit to using one style over the other, but the non-macro style of coding makes debugging easier. To create a test suite without macros, instantiate `CppUnit::TestSuite`, and then add individual tests to the suite. Finally, pass the suite itself on to `CppUnit::TextTestRunner` before an invocation of the `run` method. The client-side code remains pretty much the same, as you can see in Listing 16.

Listing 16. Creating a test suite without helper macros

```
int main ()
{
  CppUnit::TestSuite* suite = new CppUnit::TestSuite("mystringTest");
  suite->addTest(new CppUnit::TestCaller<mystringTest>("checkLength",
              &mystringTest::checkLength));
  suite->addTest(new CppUnit::TestCaller<mystringTest>("checkValue",
              &mystringTest::checkLength));

  // client code follows next
  CppUnit::TextTestRunner runner;
  runner.addTest(suite);

  runner.run();
  return 0;
}
```

To understand what's going on in Listing 16, you need to understand two classes from the CppUnit namespace: `TestSuite` and `TestCaller`, declared in TestSuite.h and TestCaller.h, respectively. When the `runner.run()` call is executed, the `runTest` method is called internal to CppUnit for each individual `TestCaller` object, which in turn calls the same routine that was passed to the `TestCaller<mystringTest>` constructor. Listing 17 shows the code (from CppUnit sources) that illustrates how individual tests are called for each suite.

Listing 17. Individual tests executed from the suite

```
void
TestComposite::doRunChildTests( TestResult *controller )
{
  int childCount = getChildTestCount();
  for ( int index =0; index < childCount; ++index )
  {
    if ( controller->shouldStop() )
      break;

    getChildTestAt( index )->run( controller );
  }
}
```

The `TestSuite` class is derived from `CppUnit::TestComposite`.

Understanding pointers in CppUnit

It's important that the test suite is declared on the heap, because CppUnit internally deletes the `TestSuite` pointer in the `TestRunner` destructor. This might not be the best design decision, though, and is not apparent from the CppUnit documentation.

Running multiple test suites

You can create multiple test suites and run them using the `TextTestRunner` object in a single operation. All you need to do is create each test suite as you did in Listing 16, and then add the same `addTest` method to the `TextTestRunner`, as shown in Listing 18.

Listing 18. Running multiple suites using the TextTestRunner

```
CppUnit::TestSuite* suite1 = new CppUnit::TestSuite("mystringTest");
suite1->addTest(…);
…
CppUnit::TestSuite* suite2 = new CppUnit::TestSuite("mymathTest");
…
suite2->addTest(…);
CppUnit::TextTestRunner runner;
runner.addTest(suite1);
runner.addTest(suite2);
…
```

Custom formatting of the output

So far, the output from the testing has been default-generated by the `TextTestRunner` class. However, CppUnit allows you to use custom formatting on the output. One of the classes for doing so is `CompilerOutputter`, declared in the header CompilerOutputter.h. Among other things, this class lets you specify the format for displaying file name-line number information in the output. Also, you can save the log directly in a file as opposed to dumping it on screen. Listing 19 provides an example of the output being dumped into a file. Observe the format `%p:%l`: the former denotes the path of the file, and the latter shows the line number. A typical output will look like /home/arpan/work/str.cc:26 when you use this format.

Listing 19. Redirecting the test output into a log file with customized formatting

```
#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/ui/text/TextTestRunner.h>
#include <cppunit/extensions/HelperMacros.h>
#include <cppunit/CompilerOutputter.h>

int main ()
{
  CppUnit::Test *test =
    CppUnit::TestFactoryRegistry::getRegistry().makeTest();
  CppUnit::TextTestRunner runner;
  runner.addTest(test);

  const std::string format("%p:%l");
  std::ofstream ofile;
  ofile.open("run.log");
  CppUnit::CompilerOutputter* outputter = new
    CppUnit::CompilerOutputter(&runner.result(), ofile);
  outputter->setLocationFormat(format);
  runner.setOutputter(outputter);

  runner.run();
  ofile.close();
  return 0;
}
```

`CompilerOutputter` has a host of other useful methods, like `printStatistics` and `printFailureReport`, which you can use to get a subset of the overall information it dumps.

More customizations: tracking test time

So far, you've been using `TextTestRunner` as the default to run your tests. The pattern has been easy enough: instantiate an object of type `TextTestRunner`, add tests and the outputter to it, and then invoke the `run` method. Let's deviate from this flow now by using `TestRunner` (the superclass for `TextTestRunner`) and a new category of classes called (quite aptly) *listeners.* Say you intend to track how much time each individual test is taking—something quite common for developers doing performance benchmarking. Before any further explanation, look at Listing 20. This code uses three classes: `TestRunner`, `TestResult`, and `myListener`, which is derived from `TestListener`. You use the same `mystringTest` class from Listing 10.

Listing 20. Learning about the TestListener class

```
class myListener : public CppUnit::TestListener {
public:
  void startTest(CppUnit::Test* test) {
    std::cout << "starting to measure time\n";
  }
  void endTest(CppUnit::Test* test) {
    std::cout << "done with measuring time\n";
  }
};

int main ()
{
  CppUnit::TestSuite* suite = new CppUnit::TestSuite("mystringTest");
  suite->addTest(new CppUnit::TestCaller<mystringTest>("checkLength",
                &mystringTest::checkLength));
  suite->addTest(new CppUnit::TestCaller<mystringTest>("checkValue",
                &mystringTest::checkLength));

  CppUnit::TestRunner runner;
  runner.addTest(suite);

  myListener listener;
  CppUnit::TestResult result;
  result.addListener(&listener);

  runner.run(result);
  return 0;
}
```

Listing 21 shows the output from Listing 20.

Listing 21. Output of the code from Listing 20

```
 [arpan@tintin] ./a.out
starting to measure time
done with measuring time
starting to measure time
done with measuring time
```

The `myListener` class is subclassed from `CppUnit::TestListener`. You need to override the `startTest` and `endTest` methods accordingly, and these would be executed before and after every test, respectively. You can easily extend these methods to check the time individual tests are taking. So, why don't you add this functionality in the setup/teardown routines? You could, but that would mean duplicating the code in the setup/teardown methods of each test suite.

Next, look at the runner object, which is an instance of the `TestRunner` class, which in turn takes an argument of type `TestResult` in the `run` method, and the listener is added to the `TestResult` object.

Finally, whatever happened to your output? `TextTestRunner` had been displaying a lot of information after the `run` method, but the `TestRunner` does none of it. What you need is an outputter object that displays the information that the listener object has gathered during test execution. Listing 22 shows what you need to change from Listing 20.

Listing 22. Adding outputter to display test execution information

```
runner.run(result);
CppUnit::CompilerOutputter outputter( &listener, std::cerr );
outputter.write();
```

But wait: this too is insufficient to get the code compiled. The constructor for `CompilerOutputter` expects an object of type `TestResultCollector`, and because `TestResultCollector` itself is derived from `TestListener` (see Related topics for a link to the CppUnit class hierarchy for details), all you need to do is derive `myListener` from `TestResultCollector`. Listing 23 shows the compilation.

Listing 23. Deriving your listener class from TestResultCollector

```
class myListener : public CppUnit::TestResultCollector {
…
};

int main ()
{
  …

  myListener listener;
  CppUnit::TestResult result;
  result.addListener(&listener);

  runner.run(result);

  CppUnit::CompilerOutputter outputter( &listener, std::cerr );
  outputter.write();

  return 0;
}
```

The output is shown in Listing 24.

Listing 24. Output of the code from Listing 23

```
[arpan@tintin] ./a.out
starting to measure time
done with measuring time
starting to measure time
done with measuring time
str.cc:31:Assertion
Test name: checkLength
assertion failed
- Expression: s.size() == 0
- String Length Non-Zero

str.cc:31:Assertion
Test name: checkValue
assertion failed
- Expression: s.size() == 0
- String Length Non-Zero

Failures !!!
Run: 0   Failure total: 2   Failures: 2   Errors: 0
```

Conclusion

This article focused on certain specific classes of the CppUnit framework: `TestResult`, `TestListener`, `TestRunner`, `CompilerOutputter`, and so on. CppUnit as a stand-alone unit testing framework has a lot more to offer. There are classes in CppUnit for XML output generation (`XMLOutputter`) and running tests in GUI mode (`MFCTestRunner` and `QtTestRunner`) as well as a plug-in interface (`CppUnitTestPlugIn`). Be sure to look into the CppUnit documentation for its class hierarchy and the examples that come with the installation for further details.

Related topics

- CppUnit documentation: Visit the project page at Sourceforge.com.
- CppUnit on Wikipedia: Wikipedia has good information on CppUnit as well as links to other unit testing frameworks.
- Download CppUnit: Get the latest version of CppUnit.
- IBM product evaluation versions: Get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Evaluate XL C/C++ for AIX