

[CppUnit project page](#)[FAQ](#)[CppUnit home page](#)

[Main Page](#) | [Modules](#) | [Namespace List](#) | [Class Hierarchy](#) |
[Alphabetical List](#) | [Class List](#) | [File List](#) | [Namespace Members](#) |
[Class Members](#) | [File Members](#) | [Related Pages](#)

CppUnit Cookbook

Here is a short cookbook to help you get started.

Simple Test Case

You want to know whether your code is working.

How do you do it?

There are many ways. Stepping through a debugger or littering your code with stream output calls are two of the simpler ways, but they both have drawbacks. Stepping through your code is a good idea, but it is not automatic. You have to do it every time you make changes. Streaming out text is also fine, but it makes code ugly and it generates far more information than you need most of the time.

Tests in CppUnit can be run automatically. They are easy to set up and once you have written them, they are always there to help you keep confidence in the quality of your code.

To make a simple test, here is what you do:

Subclass the **TestCase** class. Override the method **runTest()**. When you want to check a value, call **CPPUNIT_ASSERT(bool)** and pass in an expression that is true if the test succeeds.

For example, to test the equality comparison for a Complex number class, write:

```
class ComplexNumberTest : public CppUnit::TestCase {
public:
    ComplexNumberTest( std::string name ) : CppUnit::TestCas

    void runTest() {
```

```
    CPPUNIT_ASSERT( Complex (10, 1) == Complex (10, 1) );  
    CPPUNIT_ASSERT( !(Complex (1, 1) == Complex (2, 2)) );  
  }  
};
```

That was a very simple test. Ordinarily, you'll have many little test cases that you'll want to run on the same set of objects. To do this, use a fixture.

Fixture

A fixture is a known set of objects that serves as a base for a set of test cases. Fixtures come in very handy when you are testing as you develop.

Let's try out this style of development and learn about fixtures along the way. Suppose that we are really developing a complex number class. Let's start by defining a empty class named Complex.

```
class Complex {};
```

Now create an instance of ComplexNumberTest above, compile the code and see what happens. The first thing we notice is a few compiler errors. The test uses operator `==`, but it is not defined. Let's fix that.

```
bool operator==( const Complex &a, const Complex &b)  
{  
    return true;  
}
```

Now compile the test, and run it. This time it compiles but the test fails. We need a bit more to get an operator `==` working correctly, so we revisit the code.

```
class Complex {  
    friend bool operator ==(const Complex& a, const Complex&  
        double real, imaginary;  
public:  
    Complex( double r, double i = 0 )  
        : real(r)  
          , imaginary(i)  
    {  
    }  
};
```

```
bool operator ==( const Complex &a, const Complex &b )
{
    return a.real == b.real  &&  a.imaginary == b.imaginary;
}
```

If we compile now and run our test it will pass.

Now we are ready to add new operations and new tests. At this point a fixture would be handy. We would probably be better off when doing our tests if we decided to instantiate three or four complex numbers and reuse them across our tests.

Here is how we do it:

- Add member variables for each part of the **fixture**
- Override **setUp()** to initialize the variables
- Override **tearDown()** to release any permanent resources you allocated in **setUp()**

```
class ComplexNumberTest : public CppUnit::TestFixture {
private:
    Complex *m_10_1, *m_1_1, *m_11_2;
public:
    void setUp()
    {
        m_10_1 = new Complex( 10, 1 );
        m_1_1 = new Complex( 1, 1 );
        m_11_2 = new Complex( 11, 2 );
    }

    void tearDown()
    {
        delete m_10_1;
        delete m_1_1;
        delete m_11_2;
    }
};
```

Once we have this fixture, we can add the complex addition test case any any others that we need over the course of our development.

Test Case

How do you write and invoke individual tests using a fixture?

There are two steps to this process:

- Write the test case as a method in the fixture class
- Create a **TestCaller** which runs that particular method

Here is our test case class with a few extra case methods:

```
class ComplexNumberTest : public CppUnit::TestFixture {
private:
    Complex *m_10_1, *m_1_1, *m_11_2;
public:
    void setUp()
    {
        m_10_1 = new Complex( 10, 1 );
        m_1_1 = new Complex( 1, 1 );
        m_11_2 = new Complex( 11, 2 );
    }

    void tearDown()
    {
        delete m_10_1;
        delete m_1_1;
        delete m_11_2;
    }

    void testEquality()
    {
        CPPUNIT_ASSERT( *m_10_1 == *m_10_1 );
        CPPUNIT_ASSERT( !( *m_10_1 == *m_11_2 ) );
    }

    void testAddition()
    {
        CPPUNIT_ASSERT( *m_10_1 + *m_1_1 == *m_11_2 );
    }
};
```

One may create and run instances for each test case like this:

```
CppUnit::TestCaller<ComplexNumberTest> test( "testEquality
                                             &ComplexNumbe
CppUnit::TestResult result;
```

```
test.run( &result );
```

The second argument to the test caller constructor is the address of a method on `ComplexNumberTest`. When the test caller is run, that specific method will be run. This is not a useful thing to do, however, as no diagnostics will be displayed. One will normally use a **TestRunner** (see below) to display the results.

Once you have several tests, organize them into a suite.

Suite

How do you set up your tests so that you can run them all at once?

CppUnit provides a **TestSuite** class that runs any number of `TestCases` together.

We saw, above, how to run a single test case.

To create a suite of two or more tests, you do the following:

```
CppUnit::TestSuite suite;
CppUnit::TestResult result;
suite.addTest( new CppUnit::TestCaller<ComplexNumberTest>(
    "testEquality",
    &ComplexNumberTest::testEquality )
suite.addTest( new CppUnit::TestCaller<ComplexNumberTest>(
    "testAddition",
    &ComplexNumberTest::testAddition )
suite.run( &result );
```

TestSuites don't only have to contain callers for `TestCases`. They can contain any object that implements the **Test** interface. For example, you can create a **TestSuite** in your code and I can create one in mine, and we can run them together by creating a **TestSuite** that contains both:

```
CppUnit::TestSuite suite;
CppUnit::TestResult result;
suite.addTest( ComplexNumberTest::suite() );
suite.addTest( SurrealNumberTest::suite() );
suite.run( &result );
```

TestRunner

How do you run your tests and collect their results?

Once you have a test suite, you'll want to run it. CppUnit provides tools to define the suite to be run and to display its results. You make your suite accessible to a **TestRunner** program with a static method *suite* that returns a test suite.

For example, to make a `ComplexNumberTest` suite available to a **TestRunner**, add the following code to `ComplexNumberTest`:

```
public:
    static CppUnit::Test *suite()
    {
        CppUnit::TestSuite *suiteOfTests = new CppUnit::TestSuite;
        suiteOfTests->addTest( new CppUnit::TestCaller<ComplexNumberTest>
                               "testEquality",
                               &ComplexNumberTest::testEquality );
        suiteOfTests->addTest( new CppUnit::TestCaller<ComplexNumberTest>
                               "testAddition",
                               &ComplexNumberTest::testAddition );
        return suiteOfTests;
    }
```

To use the text version, include the header files for the tests in `Main.cpp`:

```
#include <cppunit/ui/text/TestRunner.h>
#include "ExampleTestCase.h"
#include "ComplexNumberTest.h"
```

And add a call to **`addTest(CppUnit::Test *)`** in the `main()` function:

```
int main( int argc, char **argv )
{
    CppUnit::TextUi::TestRunner runner;
    runner.addTest( ExampleTestCase::suite() );
    runner.addTest( ComplexNumberTest::suite() );
    runner.run();
    return 0;
}
```

The **TestRunner** will run the tests. If all the tests pass, you'll get an informative message. If any fail, you'll get the following information:

- The name of the test case that failed
- The name of the source file that contains the test
- The line number where the failure occurred
- All of the text inside the call to **CPPUNIT_ASSERT()** which detected the failure

CppUnit distinguishes between *failures* and *errors*. A failure is anticipated and checked for with assertions. Errors are unanticipated problems like division by zero and other exceptions thrown by the C++ runtime or your code.

Helper Macros

As you might have noticed, implementing the fixture static `suite()` method is a repetitive and error prone task. A **Writing test fixture** set of macros have been created to automatically implements the static `suite()` method.

The following code is a rewrite of `ComplexNumberTest` using those macros:

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

class ComplexNumberTest : public CppUnit::TestFixture {
```

First, we declare the suite, passing the class name to the macro:

```
CPPUNIT_TEST_SUITE( ComplexNumberTest );
```

The suite created by the static `suite()` method is named after the class name. Then, we declare each test case of the fixture:

```
CPPUNIT_TEST( testEquality );
CPPUNIT_TEST( testAddition );
```

Finally, we end the suite declaration:

```
CPPUNIT_TEST_SUITE_END();
```

At this point, a method with the following signature has been implemented:

```
static CppUnit::TestSuite *suite();
```

The rest of the fixture is left unchanged:

```
private:
    Complex *m_10_1, *m_1_1, *m_11_2;
public:
    void setUp()
    {
        m_10_1 = new Complex( 10, 1 );
        m_1_1 = new Complex( 1, 1 );
        m_11_2 = new Complex( 11, 2 );
    }

    void tearDown()
    {
        delete m_10_1;
        delete m_1_1;
        delete m_11_2;
    }

    void testEquality()
    {
        CPPUNIT_ASSERT( *m_10_1 == *m_10_1 );
        CPPUNIT_ASSERT( !( *m_10_1 == *m_11_2 ) );
    }

    void testAddition()
    {
        CPPUNIT_ASSERT( *m_10_1 + *m_1_1 == *m_11_2 );
    }
};
```

The name of the **TestCaller** added to the suite are a composition of the fixture name and the method name.

In the present case, the names would be:
"ComplexNumberTest.testEquality" and

"ComplexNumberTest.testAddition".

The **helper macros** help you write common assertions. For example, to check that ComplexNumber throws a MathException when dividing a number by 0:

- add the test to the suite using CPPUNIT_TEST_EXCEPTION, specifying the expected exception type.
- write the test case method

```
CPPUNIT_TEST_SUITE( ComplexNumberTest );
// [...]
CPPUNIT_TEST_EXCEPTION( testDivideByZeroThrows, MathExcept
CPPUNIT_TEST_SUITE_END( );

// [...]

void testDivideByZeroThrows()
{
    // The following line should throw a MathException.
    *m_10_1 / ComplexNumber(0);
}
```

If the expected exception is not thrown, then an assertion failure is reported.

TestFactoryRegistry

The **TestFactoryRegistry** was created to solve two pitfalls:

- forgetting to add your fixture suite to the test runner (since it is in another file, it is easy to forget)
- compilation bottleneck caused by the inclusion of all test case headers (see **previous example**)

The **TestFactoryRegistry** is a place where suites can be registered at initialization time.

To register the ComplexNumber suite, in the .cpp file, you add:

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

```
CPPUNIT_TEST_SUITE_REGISTRATION( ComplexNumberTest );
```

Behind the scene, a static variable type of **AutoRegisterSuite** is declared. On construction, it will **register** a **TestSuiteFactory** into the **TestFactoryRegistry** . The **TestSuiteFactory** returns the **TestSuite** returned by `ComplexNumber::suite()`.

To run the tests, using the text test runner, we don't need to include the fixture anymore:

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

int main( int argc, char **argv)
{
    CPPUNIT::TextUi::TestRunner runner;
```

First, we retrieve the instance of the **TestFactoryRegistry** :

```
CppUnit::TestFactoryRegistry &registry = CppUnit::TestFa
```

Then, we obtain and add a new **TestSuite** created by the **TestFactoryRegistry** that contains all the test suite registered using **CPPUNIT_TEST_SUITE_REGISTRATION()**.

```
runner.addTest( registry.makeTest() );
runner.run();
return 0;
}
```

Post-build check

Well, now that we have our unit tests running, how about integrating unit testing to our build process ?

To do that, the application must returns a value different than 0 to indicate that there was an error.

TestRunner::run() returns a boolean indicating if the run was successful.

Updating our main programm, we obtains:

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

int main( int argc, char **argv)
{
    CppUnit::TextUi::TestRunner runner;
    CppUnit::TestFactoryRegistry &registry = CppUnit::TestFa
runner.addTest( registry.makeTest() );
    bool wasSuccessful = runner.run( "", false );
    return wasSuccessful;
}
```

Now, you need to run your application after compilation.

With Visual C++, this is done in *Project Settings/Post-Build step*, by adding the following command: . It is expanded to the application executable path. Look up the project examples/cppunittest/CppUnitTestMain.dsp which use that technic.

Original version by Michael Feathers. Doxygen conversion and update by Baptiste Lepilleur.



hosts this
site.

Send comments to:
[CppUnit Developers](#)