Installing the CUnit Unit Test Framework for Eclipse C/C++ IDE

Philip Gust

2017-09-11

Modern development environments provide a way for programmers to verify the implementation of code by creating unit tests that compare the results of calling a collection of functions with expected results for key input and output values. Developing a suite of unit tests helps ensure that as much of the code as possible is “covered” by unit tests. Full test coverage is always a goal, although for more complex systems it is not easy to achieve. If a bug is ever discovered that was not detected by the current suite of tests, additional unit tests should be added that would have caught the bug.

A unit test suite also helps guard against future changes that introduce bugs that would otherwise be difficult to detect. Running the unit tests for a collection of functions when changes are made ensures that the code continues to perform as expected. If new functionality is added, corresponding unit tests should be added to the suite to cover it.

Learning to create good unit tests and practicing “test-drive” development is a valuable skill to learn and practice. Test-driven development often speeds up the the development, delivery, and maintenance of code. In many companies, test-driven development is mandatory, and code cannot be committed to the source code control system without first passing its unit tests.

Although C or C++ do not provide built-in unit test frameworks, there are a number of external frameworks that are widely used. We will use one called CUnit to create and manage unit tests. CUnit is a popular unit test framework that enables creating suites of tests for collections of code, managing the execution of the tests, and recording test results for analysis.

Requirements

To install Eclipse, you will need to already have the Eclipse IDE and the C/C++ Development Tools (CDT) already installed. CUnit is integrated into individual project by configuring the project library and include paths.

Download and Install CUnit

The first step is to download and install the CUnit unit test framework. There are pre-configured versions for MacOS and Windows, and it is easy to build from source on Linux.

MacOS

Install CUnit from the Home Brew package manager by executing the commands

```
brew install cunit
```

Windows

A version of CUnit for Windows is available at

http://www.ccis.northeastern.edu/home/pgust/classes/cs5001/2017/Fall/resources/windows/CUnit-2.1-2.zip

Download this ZIP archive file to your *C:\* directory, then double-click on it in File Explorer to unpack it as the directory *C:\CUnit-2.1.2*.

Linux

Download the source code for CUnit from

https://sourceforge.net/projects/cunit/files/CUnit/2.1-2/

and select the CUnit-2.1-2-src.tar.bz2 link to start the download. Unpack this archive file into a directory CUnit-2.1-2. Change to this directory and run the following commands to build and install it.

```
./configure
make
make-install
```

Integrating and Configuring Unit Testing for a Project

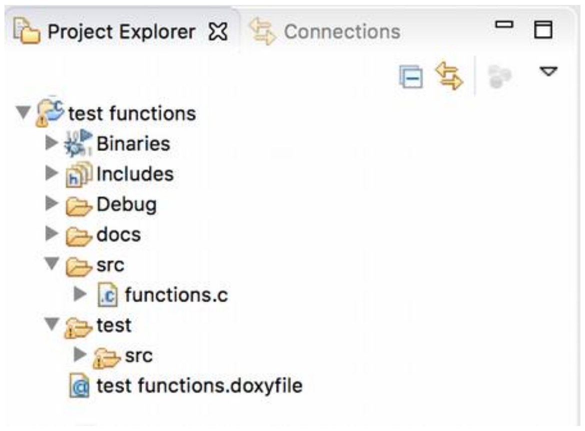
You will need to integrate and configure CUnit for each project with unit tests. Once you create a project, open the "Project" > "Properties" dialog. Under "C++ General" select "Paths and Symbols."

- On the "Includes" tab, select the C language option, and add the directory where CUnit include files are located. For MacOS and Linux, this is */usr/local/include*. For Windows, this is *C:\CUnit-2.1.2\includes* .
- On the "Library Paths" tab, add */usr/local/lib* for MacOS and Linux, and *C:\CUnit-2.1.2\lb* for Windows.
- On the "Libraries" tab, add the name *cunit* as the name of the CUnit library.

Adding Tests to Your Project

You can add unit tests to any C or C++ file whose functionality you want to test. However, a better way to manage tests is to create them in their own file. For example, if you want to create a unit test for functions in a C file named "functions.c" you would create a corresponding C file named "test-functions.c".

A common practice is to put the tests in a parallel"test/src" directory that is a sibling of the "src" directory where the application C or C++ files are located. If your unit tests require include files, they would go in the "test/include" directory. This helps keep the code being developed separate from the code that tests it, and enables you to manage the two separately.



Here is a function *fib_i(int n)* in “src/functions.c”.

```
/**
*  Uses iteration to implement Fibbonacci sequence
 *                  f(n) = f(n-2)+f(n-1) | n>=2,
 *                  f(n) = 1 | n = 0, 1
 * @param n the input number to compute fib(n)
 * @return the nth fibonacci number f(n)
 */
long fib_i(long n) {
        // fn is f(n); fn1 is f(n-1), fn2 is f(n-2)
        long fn = 1, fn1 = 1, fn2 = 1;
        while (n > 1) {
                fn = fn2 + fn1;

                fn2=fn1;
                fn1=fn;
                n--;
        }
        return fn;
}
```

Here is the corresponding unit test in test/src/*test-functions.c*, along with the code that sets up the test suite and its unit tests.

```
#include "CUnit/CUnit.h"
#include "CUnit/Basic.h"
…

/**
 * Test {@link fib_i(long)} function.
 */
static void test_fib_i(void) {
        CU_ASSERT_EQUAL(fib_i(0L), 1);
        CU_ASSERT_EQUAL(fib_i(1L), 1);
        CU_ASSERT_EQUAL(fib_i(8L), 34);
}

…

/**
 * Test all the functions for this project.
 *
 * @return test error code
 */
static int test_all(void) {
	// initialize the CUnit test registry – only once per application
	CU_initialize_registry();

	// add a suite to the registry with no init or cleanup
	CU_pSuite pSuite = CU_add_suite("function_tests", NULL, NULL);

	// add the tests to the suite
	CU_add_test(pSuite, "test_pwr_i", test_pwr_i);
...
	// run all suites using the basic interface that echoes to the console in this example
	CU_basic_set_mode(CU_BRM_VERBOSE);
	CU_basic_run_tests();

	// display information on failures that occurred
	CU_basic_show_failures(CU_get_failure_list());

	// Clean up registry and return status
	CU_cleanup_registry();
	return CU_get_error();
}
...
/**
 * Main program to invoke test functions
 *
 * @return the exit status of the program
 */
int main(void) {

	// test all the functions
	CU_ErrorCode code = test_all();

	return (code == CUE_SUCCESS) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

CUnit can run all registered test suites at once, or only selected suites corresponding to code that has changed. Output can be directed to the console as in this case, or test results can be captured as XML and stored for later analysis.

Here is console output for a suite that includes the test for the *fib_i(long)* function shown earlier.

```
      CUnit - A unit testing framework for C - Version 2.1-3
     http://cunit.sourceforge.net/

Suite: function_tests
  Test: test_pwr_i ...passed
  Test: test_pwr_r ...passed
  Test: test_fib_i ...passed
  Test: test_fib_r ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      4      4      4      0        0
             asserts     14     14     14      0      n/a

Elapsed time =    0.000 seconds
```

We will discuss testing strategies and ways to organize and run test suites for the data structures we study throughout the class.

See the CUnit Guide for complete information about test functions, managing suites, and options for running tests:

http://cunit.sourceforge.net/doc/index.html