Introduction to C Unit Testing (CUnit)

Brian Nielsen
Arne Skou

{bnielsen | ask}@cs.auc.dk
Unit Testing

- Code that isn’t tested doesn’t work”
- “Code that isn’t regression tested suffers from code rot (breaks eventually)”
- A unit testing framework enables efficient and effective unit & regression testing
What is unit testing?

- **Unit testing**
  - Testing a ‘unit’ of code, usually a class

- **Integration testing**
  - Testing a module of code (e.g. a package)

- **Application testing**
  - Testing the code as the user would see it (black box)
Conventionally

- Ad hoc manner
  - Manual stimulation & observation
  - E.g. adding a main method to a class, which runs tests on the class
  - Uncommenting or deleting test code / drivers / printf / #ifdefs
  - Assert and debug builds

- **Code that isn’t tested doesn’t work**

- “If code has no automated test case written for it to prove that it works, it must be assumed not to work.”
Regression testing

- New code and changes to old code can affect the rest of the code base
  - “Affect” sometimes means “break”
- **Regression** = *Relapsed to a less perfect or developed state.*
- **Regression testing:** Test that code has not regressed
- Regression testing is required for a stable, maintainable code base
Refactoring

- **Refactoring** is a behavior preserving transformation
- Refactoring is an excellent way to break code.
- Regression testing allows developers to refactor safely – if the refactored code passes the test suite, it works
Running automated tests

- Regression testing “must” be automated
  - This requires they report pass/fail results in a standardized way

- Daily (Nightly) builds and testing
  - Clean & check out latest build tree
  - Run tests
  - Put results on a web page & send mail (if tests fail)
Why formalize unit testing?

- Ad hoc manner
  - Uncommenting or deleting test code / drivers printf
  - Manual stimulation & observation

- Axiom:
  - Code that isn’t tested doesn’t work
  - “If code has no automated test case written for it to prove that it works, it must be assumed not to work.”
What is a testing framework?

- A test framework is a software tool for writing and running unit-tests
- provides reusable test functionality which:
  - Is easier to use
  - Is standardized
  - Enables automatic execution for regression tests
What is a testing framework?

- A test framework is a software tool for writing and running unit-tests.
- Provides reusable test functionality which:
  - Is easier to use
  - Is standardized
  - Enables automatic execution for regression tests
Why Unit-testing Framework

- A test framework is a software tool for writing and running unit-tests
  - Most errors can be found by programmer
  - Lightweight tool that uses the same language and development environment as the programmer
  - Offers an easy, systematic, and comprehensive way of organizing and executing tests
    - It is practical to collect and re-use test cases
  - Automatic Regression Testing
  - GUI-test case browser/runner
  - Test report generation
CUnit Testing

- Each method is tested while developed
  - Create tests first
  - Start with simplest that works
  - Incrementally add code while testing
- Tests serve as benchmark
- Optimize and refactorize without worry
Basic Use of FrameWork

myUnit.c

C-compiler

cunit.lib

myUnitTests.c

myUnitTests.exe

Test-report.xml
Creating a Test

- Implement test functions
- Run the test using a TestRunner
- Group multiple TestCases using TestSuite
What is xUnit?

- A set of “Frameworks” for programming and automated execution of test-cases
- X stands for programming language
  - Most Famous is J-UNIT for Java
  - But exists for almost all programming languages
    - C-unit, Cutest, Cpp-Unit, JUnit N-unit, ...
- A framework is a collection of classes, procedures, and macros
xUNIT principles

- Write **test suite for each unit** in the program.
- All test can be executed (automatically) at any time.
- For each program modification **all tests must be passed** before the modification is regarded as complete - regression testing
- **Test First – implement later!**
- Originally based on “eXtreme Programming” principles:
  - Lightweight software development methodology – by programmers for programmers
- **TDD (Test Driven Development) cycle**
  1. Write test case, and check it fails
  2. Write the new code
  3. Check that the test passes (and maybe refactor, re-test)
Core parts

- Test runner
- GUI runner

TestResult \(\downarrow\) MyTest \(\downarrow\) TestFixture

- Collects results
- runs
- uses

Test Suite \(\rightarrow\) Test Case
Concepts

- **Assertions**
  - Boolean expression that compares expected and actual results
  - The basic and smallest building-block

- **Test Case**
  - A composition of concrete test procedures
  - May contain several assertions and test for several test objectives
  - E.g. all test of a particular function

- **Test Suite**
  - Collection of related test cases
  - Can be executed automatically in a single command
Test Case / suite

- A collection of concrete test methods
- A suite is a collection of test cases

// Registers the fixture into the 'registry'

CU_pSuite getTriangleSuite(){

CU_pSuite pSuite = NULL;

if ((NULL == CU_add_test(pSuite, "Tests classification of valid triangles", validClassification)) ||
   (NULL == CU_add_test(pSuite, "Tests classification of invalid triangles", invalidClassification)) ||
   (NULL == CU_add_test(pSuite, "Tests for string conversion", invalidClassification)) ||
   (NULL == CU_add_test(pSuite, "Tests triangle main driver", testCheckTriangle))
){ . . .}
Assertion Examples

- `CU_ASSERT_EQUAL(rectangularTriangle, classifyTriangle(13,12,5));`
- `int actual_val; CU_ASSERT(stringToInt("+0", &actual_val)); CPPUNIT_ASSERT_EQUAL(0, actual_val);`
- `char* argv4[4] = {programName,"1","1","2"}; CU_ASSERT_EQUAL(string( "Isosceles Triangle"), string(checkTriangle(4,argv4))));`
Test Cases Imp.

void validClassification(){
    CU_ASSERT_EQUAL(rectangularTriangle, classifyTriangle(13,12,5) );
    CU_ASSERT_EQUAL(scaleneTriangle, classifyTriangle(15,10,5) );
    ..
Driver File

```c
int RunAllTests(void)
{
    CU_pSuite pSuite = NULL;
    pSuite = getTriangleSuite();

    CU_set_output_filename("TriangleTest");
    CU_list_tests_to_file();
    CU_automated_run_tests();
}

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

- Collection of test cases (or other test suites) in a logical unit
- Test Suites can be executed automatically
Test Reports

C:\NovoUnitTest\TriangleDemo\cppunitDemo>Debug\cppunitDemo.exe
.F...

c:\novounittest\triangledemo\testtriangle\testtriangle.cpp(30):Assertion
Test name: TriangleTests::validClassification
equality assertion failed
  - Expected: 1
  - Actual : 4

Failures !!!
Run: 4   Failure total: 1   Failures: 1   Errors: 0

Test Report

Failed Tests

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>FailureType</th>
<th>Location</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TriangleTests::validClassification</td>
<td>line #30 in \novounittest\triangledemo\testtriangle\testtriangle.cpp</td>
<td>equality assertion failed</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Expected: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Actual : 4</td>
</tr>
</tbody>
</table>

Statistics

<table>
<thead>
<tr>
<th>Status</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tests</td>
<td>4</td>
</tr>
<tr>
<td>Failures Total</td>
<td>1</td>
</tr>
<tr>
<td>Errors</td>
<td>0</td>
</tr>
<tr>
<td>Failures</td>
<td>1</td>
</tr>
</tbody>
</table>
### CUnit - A Unit testing framework for C.

http://cunit.sourceforge.net/

Running Suite Suite_1

<table>
<thead>
<tr>
<th>Type</th>
<th>Total</th>
<th>Run</th>
<th>Succeeded</th>
<th>Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suites</td>
<td>1</td>
<td>1</td>
<td>- NA -</td>
<td>0</td>
</tr>
<tr>
<td>Test Cases</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Assertions</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

File Generated By CUnit v2.1.0 at Thu Mar 15 16:14:33 2007
Advice: xUnit style

- Test cases exhibits isolation
- Sets up an independent environment / scenario and perform a distinct check
- One check per test method $\Rightarrow$ one `assert` per test method
- BUT consider amount of test code declarations to be written (when a `assert` fails the test method is stopped and no further asserts are checked).
- Test expected errors and exceptions
Advice: Application

- Design and program for testability
- Directly applicable to
  - Pure function libraries
  - API
- (With some footwork also user interfaces, network-, web-, and database applications)
Advice: Version Control

- Keep test code in a separate directory
- Keep both tests-sources and implementation-source in version control
- Don’t check in unless version passes all tests
Conclusions

- Code that isn’t tested doesn’t work”
- “Code that isn’t regression tested suffers from code rot (breaks eventually)”
- A unit testing framework enables efficient and effective unit & regression testing
- Use xUNIT to store and maintain all the small tests that you write anyway
- Write tests instead of playing with debugger and printf – tests can be automatically repeated