OOP (C++): Testing

Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iaşi, Romania
`dlucanu@info.uaic.ro`

Object Oriented Programming 2020/20201

Introduction

- In this lesson we learn how to use a unit-testing framework.
- The framework we use is Google Test `https://github.com/google/googletest/`
- Usually, a unit-testing framework is used together with a build automation tool.
- We use CMake, a cross platform builder: `https://cmake.org/`

Plan

Brief Description

- A CMake-based buildsystem is organized as a set of high-level logical targets.
- Each target corresponds to an executable, or a library, or is a custom target that contains custom commands.
- Dependencies between targets are expressed in the generation system to determine the order of generation and the rules for regeneration when changes are made.
- CMake commands are written in CMake Language and included in CMakeLists.txt files or with the .cmake extension.

Resources

- An introduction:
  `https://cmake.org/cmake/help/v3.20/manual/cmake.1.html`

- Detailed description:
  `https://cmake.org/cmake/help/latest/manual/cmake-buildsystem.7.html`

- CMake Language is described in
  `https://cmake.org/cmake/help/latest/manual/cmake-language.7.html`

First Example: Folder Structure

This is the most simple example using CMake.

```
...
└─ex0_cmake
   ├─CMakeLists.txt
   └─main.cpp
```

First Example: CMakeLists.txt

```
# CMake minimum version required
cmake_minimum_required(VERSION 3.16)

# the name of the project
project(ex0)

# C++ minimum version required
# GoogleTest requires at least C++11
set(CMAKE_CXX_STANDARD 14)

# add an executable target
add_executable(ex0_main main.cpp)
```

First Example: main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

First Example: Demo with CLion

First Example: Command line building

We work with a copy ex0_cmake_cmd

```
cd ex0_cmake_cmd  // go to the example folder
cmake -H. -Bbuild  // create the scripts in the "build" folder
cd build
cmake -build .  // the building process
ls
CMakeCache.txt          Makefile                ex0_main
CMakeFiles              cmake_install.cmake
./ex0_main  // run the generated binary
```

Third Example: Folder Structure

Let’s elaborate a bit the folder structure.

```
...
└ ex2_cmake
   ├ CMakeLists.txt
   ├ includes
   │  └ Date.h
   ├ lib
   │  └ libex2_library.a
   └ src
      ├ Date.cpp
      └ main.cpp
```

Third Example: New commands in CMakeLists.txt

```
add_executable(ex2_date src/main.cpp src/Date.cpp)

# for additional headers available for including
#   from the sources of all targets
include_directories(includes)

# add a library target
add_library(ex2_library STATIC src/Date.cpp)

# linking the library
find_library(EX2_LIBRARY ex2_library lib)
target_link_libraries(ex2_main LINK_PUBLIC ${EX2_LIBRARY})
```

Third Example: Date.h

```
//...
class Date {
private:
    int year;
    int month;
    int day;
public:
    void setToday(int aYear, int aMonth, int aDay);
    string to_string();
    string today();
    string tomorrow();
private:
    static bool isLeapYear(int year);
    static int daysNo(Date& d);
};
//...
```

Third Example: Demo with CLion

Plan

Testing Frameworks fro C++

- Boost Test Library `https://www.boost.org/doc/libs/1_66_0/libs/test/doc/html/index.html`
- CppUnit `http://cppunit.sourceforge.net/doc/cvs/cppunit_cookbook.html`
- CUTE (C++ Unit Testing Easier) `http://cute-test.com/`
- Google C++ Mocking Framework `https://github.com/google/googletest/tree/master/googlemock`
- Google Test `https://github.com/google/googletest/tree/master/googletest`
- Microsoft Unit Testing Framework for C++ `https://msdn.microsoft.com/en-us/library/hh598953.aspx`
- ...

On Google Test

From Google Test primer:

1. Tests should be independent and repeatable.
2. Tests should be well organized and reflect the structure of the tested code.
3. Tests should be portable and reusable.
4. When tests fail, they should provide as much information about the problem as possible.
5. The testing framework should liberate test writers from housekeeping chores and let them focus on the test content.
6. Tests should be fast.

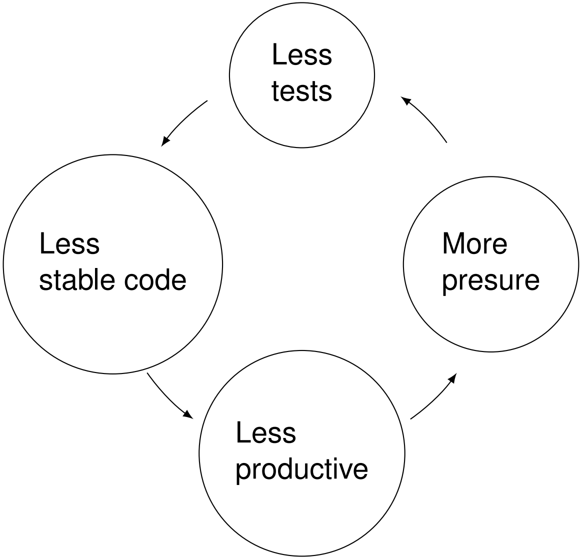
Unit Testing Terminology

Assertion	A statement that checks if a condition holds (is true).
Test	A collection of assertions that check the behaviour of a piece of code.
Test suite	A group of test. The test suite should reflect the structure of the code.
Test program	A set of test suites.

Advantages of using unit testing (UT)

- UT helps to fix bugs in early stahes.
- UT helps to understand better the code testimg.
- UT helps to understand better what functionality should be provided by a unit code.
- UT allows to test some parts of the code without waiting the other ones to be finished.
- UT helps to reuse the code.

Unit testing myth



The structure of a Google test

```
TEST(TestSuiteName, TestName) {
  ... test body ...
}
```

Six Steps to Follow

- Step 1: Initialize the project
- Step 2: Incorporate the Google Test project
- Step 3: Add the source files to the project
- Step 4: Add tests
- Step 5: Complete the files with the test configuration information
- Step 6: Build and execute the project

First Example with tests: Folder Structure

This is the most simple example using CMake.

```
...
└─ex0_cmake
   ├─CMakeLists.txt
   ├─hello_test.cpp
   └─main.cpp
```

First Example with tests: New commands in CMakeLists.txt

```
# fetch Google Test
include(FetchContent)
FetchContent_Declare(
        googletest
        URL https://github.com/google/googletest/archive/refs/tags/release-1.10.0.zip
)
# For Windows: Prevent overriding the parent project’s compiler/linker settings
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(googletest)

enable_testing()

# add an executable test target
add_executable(
        hello_test
        hello_test.cpp
)

# link the Google Test library
target_link_libraries(
        hello_test
        gtest_main
)

# enable CMake’s test runner to discover the tests included in the binary,
# using the GoogleTest CMake module
include(GoogleTest)
gtest_discover_tests(hello_test)
```

First Example with tests: hello_test.cpp

```
#include <gtest/gtest.h>

// Demonstrate some basic assertions.
TEST(HelloTest, BasicAssertions) {
  // Expect two strings not to be equal.
  EXPECT_STRNE("mine", "yours");
  // Expect equality.
  EXPECT_EQ(2 + 2, 4);
}
```

First Example with tests: Demo with CLion

First Example with tests: Command line building

We work with a copy ex0_cmake_cmd

```
cd ex0_cmake_cmd  // go to the example folder
cmake -H. -Bbuild  // create the scripts in the "build" folder
cd build
cmake -build .  // the building process
ls
CMakeCache.txt                    cmake_install.cmake
CMakeFiles                        ex0_main
CTestTestfile.cmake               hello_test
Makefile                          hello_test[1]_include.cmake
_deps                             hello_test[1]_tests.cmake
bin                               lib
./hello_test // run the generated tests
...
[==========] Running 1 test from 1 test suite.
[-------] Global test environment set-up.
[-------] 1 test from HelloTest
[ RUN      ] HelloTest.BasicAssertions
[       OK ] HelloTest.BasicAssertions (0 ms)
[-------] 1 test from HelloTest (0 ms total)

[-------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
```

Second Example with tests: Folder Structure

This is the most simple example using CMake.

```
...
└ ex1_cmake_with_tests
    ├ CMakeLists.txt
    ├ includes
    │   └ fnctns.h
    ├ lib
    │   └ libex1_library.a
    ├ src
    │   └ fnctns.cpp
    └ tests
        └ test1.cpp
```

Second Example with tests: Specific commands in CMakeLists.txt

```
...
# add an executable target
add_executable(ex1_main src/main.cpp)
add_executable(ex1_fnctns src/main.cpp src/fnctns.cpp)
...
# add a library target
add_library(ex1_library STATIC src/fnctns.cpp)

# linking the library
find_library(EX1_LIBRARY ex1_library lib)
target_link_libraries(ex1_main LINK_PUBLIC $EX1_LIBRARY)
...
# fetch Google Test
...
enable_testing()

# add an executable test target
add_executable(ex1_test1 tests/test1.cpp src/fnctns.cpp)

# link the Google Test library
target_link_libraries(ex1_test1 gtest_main)

# enable CMake's test runner to discover the tests
...
gtest_discover_tests(ex1_test1)
```

Second Example with tests: test1.cpp

```
#include <gtest/gtest.h>
#include "../includes/fnctns.h"
#include <climits>

// Tests for inc2
TEST(Ex1Test, BasicAssertionsInc2) {
    EXPECT_EQ(5, inc2(3));
    ASSERT_TRUE(inc2(7) > 7);
}

// Tests for sqr
TEST(Ex1Test, BasicAssertionsSqr) {
    EXPECT_EQ(9, sqr(3));
    ASSERT_TRUE(sqr(-5) > 0);
}

// Tests for combinations of the two
TEST(Ex1Test, CombinedAssertions) {
    EXPECT_EQ(inc2(2), sqr(2));
    ASSERT_TRUE(sqr(7) > inc2(7));
    ASSERT_TRUE(inc2(1) > sqr(1));
}

// Tests for boundary cases
TEST(Ex1Test, BondaryAssertions) {
    ASSERT_TRUE(inc2(INT_MAX) > INT_MAX);
    ASSERT_TRUE(sqr(INT_MAX/5) > INT_MAX/5);
}
```

Second Example with tests: Demo with CLion

Second Example with tests: Command line building

We work with a copy ex1_cmake_wit_tests_cmd

```
cd ex1_cmake_with_tests_cmd  // go to the example folder
cmake -H. -Bbuild  // create the scripts in the "build" folder
cd build
cmake -build .  // the building process
./ex1_test1
...
[==========] Running 4 tests from 1 test suite.
[-------] Global test environment set-up.
[-------] 4 tests from Ex1Test
[ RUN      ] Ex1Test.BasicAssertionsInc2
[       OK ] Ex1Test.BasicAssertionsInc2 (0 ms)
[ RUN      ] Ex1Test.BasicAssertionsSqr
[       OK ] Ex1Test.BasicAssertionsSqr (0 ms)
[ RUN      ] Ex1Test.CombinedAssertions
[       OK ] Ex1Test.CombinedAssertions (0 ms)
[ RUN      ] Ex1Test.BondaryAssertions
.../examples/ex1_cmake_with_tests_cmd/tests/test1.cpp:30: Failure
Value of: inc2(INT_MAX) > INT_MAX
  Actual: false
Expected: true
[  FAILED  ] Ex1Test.BondaryAssertions (0 ms)
[-------] 4 tests from Ex1Test (0 ms total)

[-------] Global test environment tear-down
[==========] 4 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 3 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] Ex1Test.BondaryAssertions

 1 FAILED TEST
```

Plan

What Is a Test Fixture

- Allows the use of the same object configuration for multiple tests
- It is recommended for testing classes
- The steps to follow are:
  1. Derive a class from :: testing :: Test.
  2. Inside the class, state any objects you intend to use.
  3. If necessary, write a default constructor or a SetUp() function to prepare the objects for each test. A common mistake is to write SetUp() as Setup(), with a small u - don't let this happen.
  4. If necessary, write a destructor or TearDown() function to free up the resources allocated in SetUp().
  5. If necessary, define methods to allow test sharing.

Further Info

`https://github.com/google/googletest/blob/master/googletest/docs/FAQ.md`

TEST_F() Macro

- it must be used whenever we have "test fixture" type objects
- template:

```
TEST_F(test_case_name, test_name) {
 ... test body ...
}
```

- the first parameter must be the name of the "test fixture" class
- you must first define a test fixture class before using it in a TEST_F()

How TEST_F() Works

For each test defined with TEST_F(), Google Test will:

1. create a new “test fixture” at runtime
2. initialize with SetUp ()
3. run the test
4. free memory by calling TearDown()
5. delete "test fixture".

Keep in mind that different tests in the same test have different test objects, and Google Test always deletes one test item before creating the next one.

Google Test does not reuse the same "fixture test" for multiple tests. Any changes that a test makes to the "test fixture" do not affect other tests.

Third Example: Folder Structure

Let's elaborate a bit the folder structure.

```
...
└ ex2_cmake
    ├ CMakeLists.txt
    ├ includes
    │   └ Date.h
    ├ lib
    │   └ libex2_library.a
    ├ src
    │   ├ Date.cpp
    │   └ main.cpp
    └ tests
        └ DateTest.cpp
```

Third Example: New commands in CMakeLists.txt

```
...
# fetch Google Test
...
# add an executable test target
add_executable(ex2_test1 tests/DateTest.cpp src/Date.cpp)

# link the Google Test library
target_link_libraries(ex2_test1 gtest_main)

# enable CMake's test runner to discover the tests included in
...
gtest_discover_tests(ex2_test1)
```

Third Example: DateTest.cpp

```
#include "gtest/gtest.h"
#include "Date.h"

class DateTest : public testing::Test {
protected:  // You should make the members protected s.t. they can be
    // accessed from sub-classes.

    // virtual void SetUp() will be called before each test is run.  You
    // should define it if you need to initialize the variables.
    // Otherwise, this can be skipped.
    virtual void SetUp() {
        d1_.setToday(2021, 5, 10);
        d2_.setToday(2020, 2, 28);
        d3_.setToday(2020, 12, 31);
    }

    // virtual void TearDown() will be called after each test is run.
    // You should define it if there is cleanup work to do.  Otherwise,
    // you don't have to provide it.
    //
    // virtual void TearDown() {
    // }

    // Declares the variables your tests want to use.
    Date d1_, d2_, d3_;
};

// When you have a test fixture, you define a test using TEST_F
// instead of TEST.

// Tests for today().
TEST_F(DateTest, Today) {
    ASSERT_STREQ("10.5.2021", d1_.today().c_str());
}
```

What Happen When a Test is Executed

For the Today test, Google Test builds a DateTest object (let's call it t1). Then the following steps are performed.

- t1.SetUp () initializes t1.
- The first test (DefaultConstructor) is run over t1.
- t1.TearDown () cleans up after the test (if any).
- t1 is destroyed.

The above steps are repeated with another DateTest object, in this case the Tomorrow test.

Third Example: Demo with CLion

Third Example with tests: Command line building

We work with a copy ex2_cmake_wit_tests_cmd

```
cd ex2_cmake_with_tests_cmd  // go to the example folder
cmake -H. -Bbuild  // create the scripts in the "build" folder
cd build
cmake -build .  // the building process
./ex2_test1
...
[==========] Running 2 tests from 1 test suite.
[-------] Global test environment set-up.
[-------] 2 tests from DateTest
[ RUN      ] DateTest.Today
[       OK ] DateTest.Today (0 ms)
[ RUN      ] DateTest.Tomorrow
[       OK ] DateTest.Tomorrow (0 ms)
[-------] 2 tests from DateTest (0 ms total)

[-------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 2 tests.
```

Plan

A Giants's Dialog

From SOFTWARE ENGINEERING TECHNIQUES Report on a conference sponsored by the NATO SCIENCE COMMITTEE Rome, Italy, 27th to 31st October 1969:

Hoare: One can construct convincing proofs quite readily of the ultimate futility of exhaustive testing of a program and even of testing by sampling. So how can one proceed? The role of testing, in theory, is to establish the base propositions of an inductive proof. You should convince yourself, or other people, as firmly as possible that if the program works a certain number of times on specified data, then it will always work on any data.

. . .

Dijkstra: Testing shows the presence, not the absence of bugs.

`http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF`

On TDD

“One of the primary reasons I switched to TDD is for improved test coverage, which leads to 40%-80% fewer bugs in production. This is my favorite benefit of TDD. It’s like a giant weight lifting off your shoulders.”

(Eric Elliott, TDD Changed My Life)

Test-driven development (TDD): Definition

“Test-driven development” refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring).

It can be succinctly described by the following set of rules:

- write a “single” unit test describing an aspect of the program
- run the test, which should fail because the program lacks that feature
- write “just enough” code, the simplest possible, to make the test pass
- “refactor” the code until it conforms to the simplicity criteria
- repeat, “accumulating” unit tests over time

`https://www.agilealliance.org/`

TDD: Expected Benefits

- many teams report significant reductions in defect rates, at the cost of a moderate increase in initial development effort
- the same teams tend to report that these overheads are more than offset by a reduction in effort in projects' final phases
- although empirical research has so far failed to confirm this, veteran practitioners report that TDD leads to improved design qualities in the code, and more generally a higher degree of "internal" or technical quality, for instance improving the metrics of cohesion and coupling

https://www.agilealliance.org/

TDD: Common Pitfalls 1/2

Typical individual mistakes include:

- forgetting to run tests frequently
- writing too many tests at once
- writing tests that are too large or coarse-grained
- writing overly trivial tests, for instance omitting assertions
- writing tests for trivial code, for instance accessors

`https://www.agilealliance.org/`

TDD: Common Pitfalls 2/2

Typical individual mistakes include:

- partial adoption – only a few developers on the team use TDD
- poor maintenance of the test suite – most commonly leading to a test suite with a prohibitively long running time
- abandoned test suite (i.e. seldom or never run) – sometimes as a result of poor maintenance, sometimes as a result of team turnover

`https://www.agilealliance.org/`

TDD: Signs of Use

- “code coverage” is a common approach to evidencing the use of TDD; while high coverage does not guarantee appropriate use of TDD, coverage below 80% is likely to indicate deficiencies in a team’s mastery of TDD

- version control logs should show that test code is checked in each time product code is checked in, in roughly comparable amounts

`https://www.agilealliance.org/`

Skill Levels: Beginner

- able to write a unit test prior to writing the corresponding code
- able to write code sufficient to make a failing test pass

`https://www.agilealliance.org/`

Skill Levels: Intermediate

- practices “test driven bug fixing”: when a defect is found, writes a test exposing the defect before correction
- able to decompose a compound program feature into a sequence of several unit tests to be written
- knows and can name a number of tactics to guide the writing of tests (for instance “when testing a recursive algorithm, first write a test for the recursion terminating case”)
- able to factor out reusable elements from existing unit tests, yielding situation-specific testing tools

`https://www.agilealliance.org/`

Skill Levels: Advanced

- able to formulate a “roadmap” of planned unit tests for a macroscopic features (and to revise it as necessary)
- able to “test drive” a variety of design paradigms: object-oriented, functional, event-drive
- able to “test drive” a variety of technical domains: computation, user interfaces, persistent data access…

`https://www.agilealliance.org/`

Plan

First Step 1/3

- From Wikipedia:

In mathematics, the greatest common divisor (gcd) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers. For example, the gcd of 8 and 12 is 4.

1. The definition already gives us a test, which we include in the file ./tests/gcd-unittest.cc

```
TEST(GcdTest, Positive) {
  EXPECT_EQ(4, Gcd(8,12));
}
```

2. add the file gcd-unittest.cc in ./tests/CMakeList.txt
3. add in ./src/myfsttstprj.h the prototype of Gcd:

```
int Gcd(int a, int b);
```

First Step 2/3

4. add to the ./src/ folder the gcd.cc file in which we write the empty GCD function:

   ```
   int Gcd(int a, int b) {
     // nothing
   }
   ```

5. add the file gcd.cc in ./src/CMakeList.txt and comment isprime_unittest.cc (for optimisation)
6. build the binaries
7. Ignore the warning

First Step 2/3

8. test:

```
Running main() from gtest_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from GcdTest
[ RUN      ] GcdTest.Positive
.../myfsttstprj/tests/gcd_unittest.cc:88: Failure
Expected equality of these values:
  4
  Gcd(8,12)
    Which is: 1
[  FAILED  ] GcdTest.Positive (0 ms)
[----------] 1 test from GcdTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (1 ms total)
[ PASSED ] 0 tests.
[ FAILED ] 1 test, listed below:
[ FAILED ] GcdTest.Positive

 1 FAILED TEST
```

First Refinement 1/3

1. add the Euclid algorithm:

```
int Gcd(int a, int b) {
  while (a != b)
    if (a > b)
      a = a - b;
    else
      b = b - a;
  return a;
}
```

First Refinement 2/3

2. build and test:

```
Running main() from gtest_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from GcdTest
[ RUN      ] GcdTest.Positive
[       OK ] GcdTest.Positive (0 ms)
[----------] 1 test from GcdTest (0 ms total)
[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
```

First Refinement 3/3

3. add more positive tests:

```
TEST(GcdTest, Positive) {
  EXPECT_EQ(4, Gcd(8,12));
  EXPECT_EQ(7, Gcd(28,21));
  EXPECT_EQ(1, Gcd(23,31));
  EXPECT_EQ(1, Gcd(48,17));
}
```

4. build and test:

```
Running main() from gtest_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from GcdTest
[ RUN      ] GcdTest.Positive
[       OK ] GcdTest.Positive (0 ms)
[----------] 1 test from GcdTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[ PASSED ] 1 test.
```

Second Refinement 1/3

1. add trivial tests

   ```
   TEST(GcdTest, Trivial) {
     EXPECT_EQ(18, Gcd(18,0));
     EXPECT_EQ(24, Gcd(0,24));
     EXPECT_EQ(19, Gcd(19,19));
     EXPECT_EQ(0, Gcd(0,0));    // undefined
   }
   ```

2. build and generate

```
Running main() from gtest_main.cc
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from GcdTest
[ RUN      ] GcdTest.Positive
[       OK ] GcdTest.Positive (0 ms)
[ RUN      ] GcdTest.Trivial
^C
```

Oooops! The execution does not terminate ...

Second Refinement 2/3

3. proceed by elimination and see that Gcd(18,0) is the problem
4. analyze the problem:

```
int Gcd(int a, int b) {
  while (a != b)
    if (a > b)
      a = a - b; // if b == 0, a is not modified!!!
    else
      b = b - a;
  return a;
}
```

5. fix it:

```
int Gcd(int a, int b) {
  if (b == 0) return a;
  if (a == 0) return b;
  while (a != b)
    if (a > b)
      a = a - b;
    else
      b = b - a;
  return a;
}
```

Second Refinement 3/3

6. build and test it:

```
Running main() from gtest_main.cc
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from GcdTest
[ RUN      ] GcdTest.Positive
[       OK ] GcdTest.Positive (0 ms)
[ RUN      ] GcdTest.Trivial
[       OK ] GcdTest.Trivial (0 ms)
[----------] 2 tests from GcdTest (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (1 ms tota
[  PASSED  ] 2 tests.
```

Third Refinement 1/3

1. add negative tests

   ```
   TEST(GcdTest, Negative) {
     EXPECT_EQ(6, Gcd(18, -12));
     EXPECT_EQ(4, Gcd(-28, 32));
     EXPECT_EQ(1, Gcd(-29, -37));
   }
   ```

2. build and generate

```
Running main() from gtest_main.cc
[==========] Running 3 tests from 1 test case.
...
.../gcd_unittest.cc:104: Failure
Expected equality of these values:
  6
  Gcd(18, -12)
    Which is: -2147483638
^C
```

Oooops! The execution does not terminate again ...

Third Refinement 2/3

Back to the documentation:
"$d \mid a$ if and only if $d \mid -a$. Thus, the fact that a number is negative does not change its list of positive divisors relative to its positive counterpart. . . .
Therefore, GCD(a, b) = GCD(|a|, |b|) for any integers a and b, at least one of which is nonzero."

3. include the case when the numbers are negative:

```
int Gcd(int a, int b) {
  if (b == 0) return a;
  if (a == 0) return b;
  if(a < 0) a = -a;
  if(b < 0) b = -b;
  while (a != b)
    if (a > b)
      a = a - b;
    else
      b = b - a;
  return a;
}
```

Third Refinement 3/3

4. build and test it:

```
Running main() from gtest_main.cc
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from GcdTest
[ RUN      ] GcdTest.Positive
[       OK ] GcdTest.Positive (0 ms)
[ RUN      ] GcdTest.Trivial
[       OK ] GcdTest.Trivial (0 ms)
[ RUN      ] GcdTest.Negative
[       OK ] GcdTest.Negative (0 ms)
[----------] 3 tests from GcdTest (0 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 3 tests.
```

Conclusion

- TDD, like any other software development concept or method, requires practice.
- The more you use TDD, the easier TDD becomes.
- Don’t forget to keep the tests simple.
- The tests, which are simple, are easy to understand and easy to maintain.
- Tools like Google Test, Google Mock, CppUnit, JustCode, JustMock, NUnit and Ninject are important and help facilitate the practice of TDD.
- But it is good to know that TDD is a practice and a philosophy that goes beyond the use of tools.
- Experience with tools and frameworks is important, the skills acquired will inspire confidence in the application developer.
- But it is good to know that the tools should not be the center of attention.
- Equal time must be given to the idea of "test first".