Googletest

Dr. Yihsiang Liow (December 19, 2022)

Contents

Chapter 1

Introduction

In software development, unit testing is a software testing method where individual units of source code are tested. Unit tests are typically automated tests written and run by software developers to ensure that a section of an application (known as the "unit") satisfy the intended behavior. In procedural programming, a unit could a function or procedure or an an entire module. In object-oriented programming, a unit can be a method, a class, or an interface.

Google Test, also known as gtest, is a unit testing library for the C/C++ programming language, based on the xUnit architecture. It can be compiled for a variety of POSIX and Windows platforms. The library is developed and used at Google. Google test was open sourced by Google in July 2008.

Besides being used at Google, many other projects implement Google Test as well. Here are some:

- Android Open Source Project operating system
- Chromium projects (behind the Chrome browser and Chrome OS)
- LLVM compiler
- Protocol Buffers (Google's data interchange format)
- OpenCV (computer vision library)
- ROS (Robotic operating system)
- Gromacs molecular dynamics simulation package

Chapter 2

Prerequisites

In terms of prerequisites (your background and the platform pre-requisites), I assume

- You know C++ as in you know CISS240, CISS245, CISS350. (At this point, CISS350 is not needed.)
- You know how to use our fedora virtual machines. Specifically, I'm using our Fedora 31 virtual machine.
- You know basic linux commands.
- You know how to write text files using a text editor (in the virtual machine). I'll be using emacs. You can use whatever you want.
- You know how to build executables and run them.
- I'll be using gtest 1.8.1, the default version of gtest for Fedora 31.

Chapter 3

Installation

In your bash shell, as root do the following:

```
dnf -y install gtest-devel
dnf -y install gmock-devel
```

Chapter 4

Testing functions

4.1 Testing with EQ, NE, LT, GT

Suppose you have a function `add` and you want to test it:

```
// File: main.cpp

int add(int x, int y)
{
    return x + y;
}


int main()
{
    // your actual program below that uses add

    return 0;
}
```

You do this:

```
// File: main.cpp
#include <limits.h>
#include "gtest/gtest.h"

int add(int x, int y)
{
    return x + y;
}


TEST(AddTest, PositiveNumbers)
{
    EXPECT_EQ(9, add(4, 5));
    EXPECT_EQ(101, add(1, 100));
}


TEST(AddTest, NegativeNumbers)
{
    ASSERT_EQ(-6, add(-1, -2)); // deliberate error
    EXPECT_EQ(-6, add(-2, -4));
}


TEST(AddTest, Zero)
{
    EXPECT_EQ(1, add(0, 5)); // deliberate error
    EXPECT_EQ(-5, add(-5, 0));
}
```

```
int main(int argc, char ** argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    int gtest = RUN_ALL_TESTS(); // 1 means there are errors

    // your actual program below that uses add

    return 0;
}
```

Then you compile:

```
[student@localhost example1] g++ *.cpp -lgtest
```

(The `l` in `-lgtest` is the lowercase letter l and not the number 1.) And you run the program:

```
[student@localhost example1] ./a.out
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from AddTest
[ RUN      ] AddTest.PositiveNumbers
[       OK ] AddTest.PositiveNumbers (0 ms)
[ RUN      ] AddTest.NegativeNumbers
main.cpp:20: Failure
Expected equality of these values:
  -6
  add(-1, -2)
    Which is: -3
[  FAILED  ] AddTest.NegativeNumbers (0 ms)
[ RUN      ] AddTest.Zero
main.cpp:27: Failure
Expected equality of these values:
  1
  add(0, 5)
    Which is: 5
[  FAILED  ] AddTest.Zero (0 ms)
[----------] 3 tests from AddTest (0 ms total)
[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 2 tests, listed below:
[  FAILED  ] AddTest.NegativeNumbers
[  FAILED  ] AddTest.Zero
```

```
2 FAILED TESTS
```

The `AddTest` is called a test suite and the `PositiveNumbers` is called a test. For the

```
EXPECT_EQ(expected, actual);
```

you are checking for equality between `expected` and `actual` (the actual), i.e., you pass this test case if `expected == actual`. Otherwise you fail this test case.

The name of test suite of test must be a valid C/C++ identifier. The recommendation is not to use underscore in case it conflicts with gtest, but that is probably very rare.

There's another macro that is similar:

```
ASSERT_EQ(9, add(4, 5));
```

The difference is that for `EXPECT_EQ` if you fail the test case, the testing continue. For `ASSERT_EQ`, if you fail a test case, the other test cases for that test suite (if any) are ignored.

`EXPECT_EQ` and `ASSERT_EQ` are C/C++ macros.

There are other `*_EQ` you can also compare two values with the following:.

	Success
`*_EQ(expected, actual)`	`expected == actual`
`*_NE(expected, actual)`	`expected != actual`
`*_LT(expected, actual)`	`expected < actual`
`*_LE(expected, actual)`	`expected <= actual`
`*_GT(expected, actual)`	`expected > actual`
`*_GE(expected, actual)`	`expected >= actual`

where `*` is `EXPECT` or `ASSERT`.

Exercise 4.1.1. Change one of the deliberate errors to `ASSERT`. Compile, run, and check out the output. □

Exercise 4.1.2. Run the following and see if you can use `EXPECT_EQ` for character values:

```
// File: main.cpp
#include <limits.h>
```

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

TEST(A, B)
{
    EXPECT_EQ('a', 'a');
    EXPECT_EQ('B', 'b');
    EXPECT_EQ('C', 'C');
}

int main(int argc, char ** argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    int gtest = RUN_ALL_TESTS();
    return 0;
}
```

□

If you are going to mix your actual program and the tests, then you probably want to do the following so that the tests are collected together in a `test` function:

```
// File: main.cpp

#include <limits.h>
#include "gtest/gtest.h"

int add(int x, int y)
{
    return x + y;
}


TEST(AddTest, PositiveNumbers)
{
    EXPECT_EQ(9, add(4, 5));
    EXPECT_EQ(5, add(2, 3));
    EXPECT_EQ(101, add(1, 100));
}

... and other TEST ...

class UnitTestError
{};
void test()
{
    ::testing::InitGoogleTest(&argc, argv);
    int gtest = RUN_ALL_TESTS(); // 1 means there are errors
    if (gtest != 0)
    {
        std::cout << "Error(s)!!! Halting ...'' << std::endl;
        throw UnitTestError();
    }
}


int main(int argc, char ** argv)
{
    test();

    // your actual program below that uses add

    return 0;
}
```

Of course you might have more than one function to test.

The above is the fastest (quick and dirty) way to add tests to a simple program. In a more complex program, you don't really want to do the above.

You can add your own output when there's an error:

```
EXPECT_EQ(-6, add(-1, -2)) << "FIX YOUR ERROR!!!";
```

Exercise 4.1.3. You are write a chess game. You have the following function

```
bool is_white(char board[8][8], int row, int col);
bool is_black(char board[8][8], int row, int col);
bool is_occupied(char board[8][8], int row, int col);
```

that does the obvious. Use the following to fill your chess board for all tests:

```
               //01234567
char t[8][9] = {"    q  r", // 0
                " K      ", // 1
                "     n  ", // 2
                "    Q  B", // 3
                " R p    ", // 4
                "      k ", // 5
                "  N    P", // 6
                "   b    "};// 7
```

The following is the expected output of your tests:

```
[==========] Running 29 tests from 3 test cases.
[----------] Global test environment set-up.
[----------] 8 tests from IsWhiteTest
[ RUN      ] IsWhiteTest.K
[       OK ] IsWhiteTest.K (0 ms)
[ RUN      ] IsWhiteTest.B
[       OK ] IsWhiteTest.B (0 ms)
[ RUN      ] IsWhiteTest.P
[       OK ] IsWhiteTest.P (0 ms)
[ RUN      ] IsWhiteTest.N
[       OK ] IsWhiteTest.N (0 ms)
[ RUN      ] IsWhiteTest.R
[       OK ] IsWhiteTest.R (0 ms)
[ RUN      ] IsWhiteTest.Q
[       OK ] IsWhiteTest.Q (0 ms)
[ RUN      ] IsWhiteTest.SPACE
[       OK ] IsWhiteTest.SPACE (0 ms)
[ RUN      ] IsWhiteTest.BLACK
[       OK ] IsWhiteTest.BLACK (0 ms)
[----------] 8 tests from IsWhiteTest (0 ms total)

[----------] 8 tests from IsBlackTest
[ RUN      ] IsBlackTest.k
[       OK ] IsBlackTest.k (0 ms)
[ RUN      ] IsBlackTest.b
```

```
[       OK ] IsBlackTest.b (0 ms)
[ RUN      ] IsBlackTest.p
[       OK ] IsBlackTest.p (0 ms)
[ RUN      ] IsBlackTest.n
[       OK ] IsBlackTest.n (0 ms)
[ RUN      ] IsBlackTest.r
[       OK ] IsBlackTest.r (0 ms)
[ RUN      ] IsBlackTest.q
[       OK ] IsBlackTest.q (0 ms)
[ RUN      ] IsBlackTest.SPACE
[       OK ] IsBlackTest.SPACE (0 ms)
[ RUN      ] IsBlackTest.WHITE
[       OK ] IsBlackTest.WHITE (0 ms)
[----------] 8 tests from IsBlackTest (1 ms total)

[----------] 13 tests from IsOccupiedTest
[ RUN      ] IsOccupiedTest.K
[       OK ] IsOccupiedTest.K (0 ms)
[ RUN      ] IsOccupiedTest.B
[       OK ] IsOccupiedTest.B (0 ms)
[ RUN      ] IsOccupiedTest.P
[       OK ] IsOccupiedTest.P (0 ms)
[ RUN      ] IsOccupiedTest.N
[       OK ] IsOccupiedTest.N (0 ms)
[ RUN      ] IsOccupiedTest.R
[       OK ] IsOccupiedTest.R (0 ms)
[ RUN      ] IsOccupiedTest.Q
[       OK ] IsOccupiedTest.Q (0 ms)
[ RUN      ] IsOccupiedTest.k
[       OK ] IsOccupiedTest.k (0 ms)
[ RUN      ] IsOccupiedTest.b
[       OK ] IsOccupiedTest.b (0 ms)
[ RUN      ] IsOccupiedTest.p
[       OK ] IsOccupiedTest.p (0 ms)
[ RUN      ] IsOccupiedTest.n
[       OK ] IsOccupiedTest.n (0 ms)
[ RUN      ] IsOccupiedTest.r
[       OK ] IsOccupiedTest.r (0 ms)
[ RUN      ] IsOccupiedTest.q
[       OK ] IsOccupiedTest.q (0 ms)
[ RUN      ] IsOccupiedTest.SPACE
[       OK ] IsOccupiedTest.SPACE (0 ms)
[----------] 13 tests from IsOccupiedTest (1 ms total)

[----------] Global test environment tear-down
[==========] 29 tests from 3 test cases ran. (3 ms total)
[  PASSED  ] 29 tests.
```

Clearly `IsWhiteTest.K` (test suite `IsWhiteTest`, test `K`) tests if the board has a white king at the given row, column. `IsWhiteTest.SPACE` tests is the

`is_white` returns `false` when the character at the given row, column is a space. `IsWhiteTest.BLACK` tests is the `is_white` returns `false` when the character at the given row, column is a black piece. Etc. Note that for each test you have to setup the same chess board. So for the `is_white` function, there are 8 test cases, all with the same initialization code to setup the board. Same for `is_black` and `is_occupied`. That's pretty tedious! You can use a global chess board. But that's probably not good idea. Later I'll create a class for storing the initialization of data (an an object) for various test cases.

4.2 Testing with TRUE, FALSE

For testing boolean values, you can do

```
EXPECT_EQ(true, greater_than(42, 0));
```

(where `greater_than` is the obvious function). Another way to do the above is to do this:

```
EXPECT_TRUE(greater_than(42, 0));
```

or this:

```
EXPECT_FALSE(greater_than(0, 42));
```

Try this:

```
// File: main.cpp

#include <limits.h>
#include "gtest/gtest.h"

bool positive(int x)
{
    return (x >= 0);
}


TEST(PositiveTest, PositiveNumbers)
{
    EXPECT_TRUE(positive(1));
    EXPECT_EQ(true, positive(2));
}


TEST(PositiveTest, NegativeNumbers)
{
    EXPECT_FALSE(positive(-1));
    EXPECT_EQ(false, positive(-42));
}


TEST(PositiveTest, Zero)
{
    EXPECT_TRUE(positive(0));
}


int main(int argc, char ** argv)
{
    ::testing::InitGoogleTest(&argc, argv);
```

```
    int gtest = RUN_ALL_TESTS();

    return 0;
}
```

Of course both

```
EXPECT_EQ(6, add(2, 3));
```

and

```
EXPECT_TRUE(6 == add(2, 3));
```

will more or less have the same effect except that the outputs are slightly different.

Exercise 4.2.1. Back to the chess game. The three functions `is_white`, `is_black`, and `is_occupied` are boolean functions. Rewrite your tests using `EQUAL_TRUE` and `EQUAL_FALSE`.

4.3 Testing floating point values

Go ahead and test this:

```
EXPECT_EQ(4.56, 4.0 + 0.56);
```

Recall (from CISS240) that floating point computations are not exact. There might be rounding errors.

For floating point type equality comparisons, you have

	Success
`*_FLOAT_EQ(expected, actual)`	`expected == actual` as floats
`*_DOUBLE_EQ(expected, actual)`	`expected == actual` as doubles
`*_NEAR(expected, actual, error-bound)}`	`|expected - actual| < error-bound`

where `*` is `EXPECT` or `ASSERT`.

Go ahead and try

```
EXPECT_EQ(4.56, 4.0 + 0.56);
```

with `EXPECT_DOUBLE_EQ`

4.4 Testing arrays

The following *tries* to test if two integer arrays are the same:

```
// File: main.cpp

#include <limits.h>
#include "gtest/gtest.h"


TEST(A, B)
{
    int x[] = {2, 3, 5};
    int y[] = {2, 3, 5};
    EXPECT_EQ(x, y);
}


int main(int argc, char ** argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    int gtest = RUN_ALL_TESTS();
    return 0;
}
```

Run it and you'll see the following output:

```
[student@localhost examplearraywrong] g++ *.cpp -lgtest; ./a.out
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from A
[ RUN      ] A.B
main.cpp:11: Failure
Expected equality of these values:
  x
    Which is: 0x7fff17c4a294
  y
    Which is: 0x7fff17c4a288
[  FAILED  ] A.B (0 ms)
[----------] 1 test from A (0 ms total)
[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] A.B
 1 FAILED TEST
```

The two arrays being tested are obviously the same and yet the test says they are *not* the same. You should know from CISS240 that you should not do `x == y` when `x` and `y` are arrays and you should know from CISS245 that `x == y` compares the addresses of `x` and `y` and not the contents. The output you get when you run the above program says as much.

Run this and study the code:

```
// File: main.cpp

#include <limits.h>
#include "gtest/gtest.h"


TEST(A, B)
{
    int x[] = {2, 3, 5};
    int y[] = {2, 1, 5}; // deliberate error at index 1
    for (int i = 0; i < 3; ++i)
    {
        EXPECT_EQ(x[i], y[i]) << "Error at index " << i;
    }
}


int main(int argc, char ** argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    int gtest = RUN_ALL_TESTS();
    return 0;
}
```

Here's the output:

```
[student@localhost examplearraycorrect] g++ *.cpp -lgtest; ./a.out
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from A
[ RUN      ] A.B
main.cpp:13: Failure
Expected equality of these values:
  x[i]
    Which is: 3
  y[i]
    Which is: 1
Error at index 1
[  FAILED  ] A.B (0 ms)
```

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

Exercise 4.4.1. Write a chess game. You have a function

```
bool is_valid_move(char board[8][8],
                   int from_row, int from_col,
                   int to_row, int to_col,
                   bool white_king_has_moved = false,
                   bool black_king_has_moved = false,
                   char promotion_piece = '?');
```

4.5 Testing STL containers

Note that C++ STL containers have `operator==` and `operator!=` which compares their contents. So although you cannot use `EXPECT_EQ` on two integer arrays, you can do `EXPECT_EQ` on for instance `std::vector` of integers. Run this

```
// File: main.cpp

#include <vector>
#include <limits.h>
#include "gtest/gtest.h"


TEST(VectorTest, A)
{
    std::vector< int > x {2, 3, 5};
    std::vector< int > y {2, 1, 5}; // deliberate error at index 1
    EXPECT_EQ(x, y);
}


int main(int argc, char ** argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    int gtest = RUN_ALL_TESTS();
    return 0;
}
```

and study the output:

```
[student@localhost examplevector] g++ *.cpp -lgtest; ./a.out
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from VectorTest
[ RUN      ] VectorTest.A
main.cpp:12: Failure
Expected equality of these values:
  x
    Which is: { 2, 3, 5 }
  y
    Which is: { 2, 1, 5 }
[  FAILED  ] VectorTest.A (0 ms)
[----------] 1 test from VectorTest (0 ms total)
[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 0 tests.
```

```
[  FAILED  ] 1 test, listed below:
[  FAILED  ] VectorTest.A
 1 FAILED TEST
```

There's another way to do the above using the gmock library. Study this source file:

```
// File: main.cpp

#include <vector>
#include <limits.h>
//#include "gtest/gtest.h" // this is now redundant
#include "gmock/gmock.h"


TEST(VectorTest, A)
{
    std::vector< int > x {2, 3, 5};
    std::vector< int > y {2, 1, 5}; // deliberate error at index 1
    EXPECT_EQ(x, y);
    EXPECT_THAT(x, ::testing::ContainerEq(y));
}


int main(int argc, char ** argv)
{
    //::testing::InitGoogleTest(&argc, argv); // this is now redundant
    ::testing::InitGoogleMock(&argc, argv);
    int gtest = RUN_ALL_TESTS();
    return 0;
}
```

Compile (look at the `-l` option) and study the output:

```
[student@localhost examplevector2] g++ *.cpp -lgtest -lgmock; ./a.out
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from VectorTest
[ RUN      ] VectorTest.A
main.cpp:13: Failure
Expected equality of these values:
  x
    Which is: { 2, 3, 5 }
  y
    Which is: { 2, 1, 5 }
main.cpp:14: Failure
Value of: x
```

```
Expected: equals { 2, 1, 5 }
  Actual: { 2, 3, 5 }, which has these unexpected elements: 3,
and doesn't have these expected elements: 1
[  FAILED  ] VectorTest.A (0 ms)
[----------] 1 test from VectorTest (0 ms total)
[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] VectorTest.A
 1 FAILED TEST
```

See the difference?

There's yet another way to do the above: study and run this:

```
// File: main.cpp

#include <vector>
#include <limits.h>
//#include "gtest/gtest.h" // this is now redundant
#include "gmock/gmock.h"


TEST(VectorTest, A)
{
    std::vector< int > x {2, 3, 5};
    std::vector< int > y {2, 1, 5}; // deliberate error at index 1
    //EXPECT_EQ(x, y);
    //EXPECT_THAT(x, ::testing::ContainerEq(y));
    EXPECT_THAT(x, ::testing::ElementsAreArray({2, 1, 5}));
}


int main(int argc, char ** argv)
{
    //::testing::InitGoogleTest(&argc, argv); // this is now redundant
    ::testing::InitGoogleMock(&argc, argv);
    int gtest = RUN_ALL_TESTS();
    return 0;
}
```

Notice that here you do not need to create the variable `y`.

(`ElementsAreArray` is part of a larger collection of `ElementsAre*` macros.)

4.6 Filtering and repetitions

You can run selected tests. For instance suppose you have an `AddTest` suite and there are `PositiveNumbers`, `NegativeNumbers`, `Zero` tests. If your executable is `a.out`, you can do this to run only the test cases in `PositiveNumbers` of `AddTest`:

```
./a.out --gtest_filter=AddTest.PositiveNumbers
```

To run the test cases of `PositiveNumbers` and `NegativeNumbers` of `AddTest`, do this:

```
./a.out --gtest_filter=AddTest.*Numbers
```

You can also run the tests multiple times:

```
./a.out --gtest_repeat=100
```

In this case you run all the tests 100 times. This is helpful if for instance your function performs some randomization.

Chapter 5

Testing a classes

5.1 Testing a class

If you have a class, you can test each method using the ideas of testing a function. For instance if you have a rectangle class `Rect` that has an `area()` method, you might have

```
TEST(RectTest, area)
{
    EXPECT_EQ(6, Rect(2, 3).area());
}
```

In many cases, when you want to test a class, you create the same collection of objects. To save time on writing the code to create the same objects again and again, usually the unit test library will provide a way to create the same objects in one place and will also provide a way to perform cleanup in case the objects created live in the memory heap and requires memory deallocation. In the example below, there's a `Rect` class. To test the methods in the `Rect` class,

1. Create a `RectTest` class. `RectTest` must publicly inherit from `::testing::Test`.
2. Make `RectTest` a friend class in `Rect`.
3. Create test object(s) in `RectTest`. In the example, `r` is the `Rect` object used in test cases.
4. In `RectTest::SetUp()`, set `r` to a rectangle of width 2 and height 3.
5. In `RectTest::TearDown()`, we don't do anything since the memory for `r` is deallocated automatically.
6. The tests are in `TEST_F` (not `TEST`).

Note that for each `TEST_F`, `RectTest::Setup()` is executed once for each before `TEST_F` runs and `RectTest::TearDown()` is executed once after `TEST_F` ends. Run this

```
// File: main.cpp
#include <limits.h>
#include "gtest/gtest.h"
#include <string>

class RectTest;
class Rect
{
public:
    Rect()
    {}

    Rect(int width, int height)
        : width_(width), height_(height)
    {}
```

```
    void set(int width, int height)
    {
        width_ = width;
        height_ = height;
    }
    int area() const
    {
        return width_ * height_;
    }
    int perimeter() const
    {
        return 2 * (width_ + height_);
    }

    friend class RectTest;

private:
    int width_;
    int height_;
};


class RectTest : public ::testing::Test
{
public:
    RectTest()
        : r(0, 0)
    {}
protected:
    void SetUp()
    {
        std::cout << "RectTest::SetUp() ... set r to (2, 3)\n";
        r.set(2, 3);
    }
    void TearDown()
    {
        std::cout << "RectTest::TearDown() ...\n";
    }

    Rect r;
};


TEST_F(RectTest, area)
{
    EXPECT_EQ(7, r.area()); // deliberate error
    EXPECT_EQ(6, r.area());
}

TEST_F(RectTest, perimeter)
{
```

```
    EXPECT_EQ(11, r.perimeter());
}


int main(int argc, char ** argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    int gtest = RUN_ALL_TESTS();

    return 0;
}
```

to get this output:

```
[student@localhost example-class] g++ *.cpp -lgtest -lgmock; ./a.out
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from RectTest
[ RUN      ] RectTest.area
RectTest::SetUp() ... set r to (2, 3)
main.cpp:61: Failure
Expected equality of these values:
  7
  r.area()
    Which is: 6
RectTest::TearDown() ...
[  FAILED  ] RectTest.area (0 ms)
[ RUN      ] RectTest.perimeter
RectTest::SetUp() ... set r to (2, 3)
main.cpp:67: Failure
Expected equality of these values:
  11
  r.perimeter()
    Which is: 10
RectTest::TearDown() ...
[  FAILED  ] RectTest.perimeter (0 ms)
[----------] 2 tests from RectTest (0 ms total)
[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 2 tests, listed below:
[  FAILED  ] RectTest.area
[  FAILED  ] RectTest.perimeter
 2 FAILED TESTS
```

Note that there are times when you need some initial setup and final teardown for *all* the test suites. In that case you need `SetUpTestCase()` and `TearDownTestCase()`. Run this:

```
// File: main.cpp
#include <limits.h>
#include "gtest/gtest.h"
#include <string>

class RectTest;
class Rect
{
public:
    Rect()
    {}

    Rect(int width, int height)
        : width_(width), height_(height)
    {}
    void set(int width, int height)
    {
        width_ = width;
        height_ = height;
    }
    int area() const
    {
        return width_ * height_;
    }
    int perimeter() const
    {
        return 2 * (width_ + height_);
    }

    friend class RectTest;

private:
    int width_;
    int height_;
};


class RectTest : public ::testing::Test
{
public:
    RectTest()
        : r(0, 0)
    {}
protected:
    void SetUp()
    {
```

```
        std::cout << "RectTest::SetUp() ... set r to (2, 3)\n";
        r.set(2, 3);
    }
    void TearDown()
    {
        std::cout << "RectTest::TearDown() ...\n";
    }
    static void SetUpTestCase()
    {
        std::cout << "RectTest::SetUpTestCase() ... \n";
    }
    static void TearDownTestCase()
    {
        std::cout << "RectTest::TearDownTestCase() ...\n";
    }

    Rect r;
};


TEST_F(RectTest, area)
{
    EXPECT_EQ(7, r.area()); // deliberate error
    EXPECT_EQ(6, r.area());
}

TEST_F(RectTest, perimeter)
{
    EXPECT_EQ(10, r.perimeter()); // deliberate error
    EXPECT_EQ(11, r.perimeter());
}


int main(int argc, char ** argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    int gtest = RUN_ALL_TESTS();

    return 0;
}
```

and you'll get this output:

```
[student@localhost example-class-2] g++ *.cpp -lgtest -lgmock
[student@localhost example-class-2] ./a.out
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from RectTest
RectTest::SetUpTestCase() ...
```

```
[ RUN      ] RectTest.area
RectTest::SetUp() ... set r to (2, 3)
main.cpp:69: Failure
Expected equality of these values:
  7
  r.area()
    Which is: 6
RectTest::TearDown() ...
[  FAILED  ] RectTest.area (0 ms)
[ RUN      ] RectTest.perimeter
RectTest::SetUp() ... set r to (2, 3)
main.cpp:76: Failure
Expected equality of these values:
  11
  r.perimeter()
    Which is: 10
RectTest::TearDown() ...
[  FAILED  ] RectTest.perimeter (0 ms)
RectTest::TearDownTestCase() ...
[----------] 2 tests from RectTest (0 ms total)
[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 2 tests, listed below:
[  FAILED  ] RectTest.area
[  FAILED  ] RectTest.perimeter
 2 FAILED TESTS
```

Read the output very carefully. Note that `Setup` and `TearDown` are run for each of the two test cases. However `SetupTestCase` and `TearDownTestCase` are each only run `once`. `SetupTestCase` is run once before all the test cases are executed and `TearDownTestCase` is run once after all the test cases are done.

(I hope you realize the above is confusimg: The name `SetupTestCase` and `TearupTestCase` should really be `SetupTestSuite` and `TearupTestSuite`. Remember a test suite is a collection of test cases. This was a mistake on google's part. In a later version of googletest, the names were fixed.)

Exercise 5.1.1. Refer to the previous chess game functions and test cases. I have the following functions: `is_white`, `is_black`, `is_occupied`. I'm going to move these into a class. Here's a chess game class:

```
class ChessGameState
{
private:
    char board_[8][8];
    char player_to_move_; // If 'W', then white to move. Otherwise 'B'.
    bool K_moved_; // white king has moved (for castling)
    bool k_moved_; // black king has moved (for castling)
};
```

This a snap shot of a chess game (i.e., it does not records all the moves and all the states of the game). Move the functions `is_white`, `is_black`, `is_occupied` into `ChessGameState` and make them methods.

The first goal is to rewrite the `is_white` test cases with less code, i.e. the initialization code of the board is written once in a class.

Follow the example above and create a class called `ChessGameStateIsWhiteTest` for all the test cases for the original `is_white` function. An object of `ChessGameStateIsWhiteTest` has a `ChessGameState` called `chess`. Create a `SetUp()` in `ChessGameState` to set `chess.board_` to the following just like the earlier chess exercise:

```
                //01234567
char t[8][9] = {"    q  r", // 0
                " K      ", // 1
                "     n  ", // 2
                "    Q  B", // 3
                " R p    ", // 4
                "      k ", // 5
                "  N    P", // 6
                "   b    "};// 7
```

The `ChessGameStateIsWhiteTest` looks like this:

```
class ChessGameStateIsWhiteTest : public ::testing::Test
{
public:
    ChessGameStateIsWhiteTest()
    {}
protected:
    void SetUp()
    {
      // TODO
    }

    void TearDown()
    {}
```

```
    ChessGameState chess;
};
```

This is the expected output

```
[==========] Running 8 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 8 tests from ChessGameStateIsWhiteTest
[ RUN      ] ChessGameStateIsWhiteTest.K
[       OK ] ChessGameStateIsWhiteTest.K (0 ms)
[ RUN      ] ChessGameStateIsWhiteTest.B
[       OK ] ChessGameStateIsWhiteTest.B (0 ms)
[ RUN      ] ChessGameStateIsWhiteTest.P
[       OK ] ChessGameStateIsWhiteTest.P (0 ms)
[ RUN      ] ChessGameStateIsWhiteTest.N
[       OK ] ChessGameStateIsWhiteTest.N (0 ms)
[ RUN      ] ChessGameStateIsWhiteTest.R
[       OK ] ChessGameStateIsWhiteTest.R (0 ms)
[ RUN      ] ChessGameStateIsWhiteTest.Q
[       OK ] ChessGameStateIsWhiteTest.Q (0 ms)
[ RUN      ] ChessGameStateIsWhiteTest.SPACE
[       OK ] ChessGameStateIsWhiteTest.SPACE (0 ms)
[ RUN      ] ChessGameStateIsWhiteTest.BLACK
[       OK ] ChessGameStateIsWhiteTest.BLACK (0 ms)
[----------] 8 tests from ChessGameStateIsWhiteTest (0 ms total)

[----------] Global test environment tear-down
[==========] 8 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 8 tests.
```

Each test case should now be very simple. Here's the first test case (the new version):

```
TEST_F(ChessGameStateIsWhiteTest, K)
{
    EXPECT_TRUE(chess.is_white(1, 1));
}
```

Exercise 5.1.2. Once you are done with the 8 tests for `is_white`, now do `is_black` and `is_occupied`.

Exercise 5.1.3. But wait ... the chess game state for `is_white`, `is_black`, and `is_occupied` are the *same*. Can you write *one* setup code that works for the test cases of all three test suites? The expected output should be

```
[==========] Running 29 tests from 3 test cases.
[----------] Global test environment set-up.
[----------] 8 tests from ChessGameStateIsWhiteTest
[ RUN      ] ChessGameStateIsWhiteTest.K
[       OK ] ChessGameStateIsWhiteTest.K (0 ms)
[ RUN      ] ChessGameStateIsWhiteTest.B
[       OK ] ChessGameStateIsWhiteTest.B (0 ms)
[ RUN      ] ChessGameStateIsWhiteTest.P
[       OK ] ChessGameStateIsWhiteTest.P (0 ms)
[ RUN      ] ChessGameStateIsWhiteTest.N
[       OK ] ChessGameStateIsWhiteTest.N (0 ms)
[ RUN      ] ChessGameStateIsWhiteTest.R
[       OK ] ChessGameStateIsWhiteTest.R (0 ms)
[ RUN      ] ChessGameStateIsWhiteTest.Q
[       OK ] ChessGameStateIsWhiteTest.Q (0 ms)
[ RUN      ] ChessGameStateIsWhiteTest.SPACE
[       OK ] ChessGameStateIsWhiteTest.SPACE (0 ms)
[ RUN      ] ChessGameStateIsWhiteTest.BLACK
[       OK ] ChessGameStateIsWhiteTest.BLACK (0 ms)
[----------] 8 tests from ChessGameStateIsWhiteTest (0 ms total)

[----------] 8 tests from ChessGameStateIsBlackTest
[ RUN      ] ChessGameStateIsBlackTest.k
[       OK ] ChessGameStateIsBlackTest.k (0 ms)
[ RUN      ] ChessGameStateIsBlackTest.b
[       OK ] ChessGameStateIsBlackTest.b (0 ms)
[ RUN      ] ChessGameStateIsBlackTest.p
[       OK ] ChessGameStateIsBlackTest.p (0 ms)
[ RUN      ] ChessGameStateIsBlackTest.n
[       OK ] ChessGameStateIsBlackTest.n (0 ms)
[ RUN      ] ChessGameStateIsBlackTest.r
[       OK ] ChessGameStateIsBlackTest.r (0 ms)
[ RUN      ] ChessGameStateIsBlackTest.q
[       OK ] ChessGameStateIsBlackTest.q (0 ms)
[ RUN      ] ChessGameStateIsBlackTest.SPACE
[       OK ] ChessGameStateIsBlackTest.SPACE (0 ms)
[ RUN      ] ChessGameStateIsBlackTest.WHITE
[       OK ] ChessGameStateIsBlackTest.WHITE (0 ms)
[----------] 8 tests from ChessGameStateIsBlackTest (1 ms total)

[----------] 13 tests from ChessGameStateIsOccupiedTest
[ RUN      ] ChessGameStateIsOccupiedTest.K
[       OK ] ChessGameStateIsOccupiedTest.K (0 ms)
[ RUN      ] ChessGameStateIsOccupiedTest.B
[       OK ] ChessGameStateIsOccupiedTest.B (0 ms)
[ RUN      ] ChessGameStateIsOccupiedTest.P
[       OK ] ChessGameStateIsOccupiedTest.P (0 ms)
[ RUN      ] ChessGameStateIsOccupiedTest.N
[       OK ] ChessGameStateIsOccupiedTest.N (0 ms)
[ RUN      ] ChessGameStateIsOccupiedTest.R
```

```
[       OK ] ChessGameStateIsOccupiedTest.R (0 ms)
[ RUN      ] ChessGameStateIsOccupiedTest.Q
[       OK ] ChessGameStateIsOccupiedTest.Q (0 ms)
[ RUN      ] ChessGameStateIsOccupiedTest.k
[       OK ] ChessGameStateIsOccupiedTest.k (0 ms)
[ RUN      ] ChessGameStateIsOccupiedTest.b
[       OK ] ChessGameStateIsOccupiedTest.b (0 ms)
[ RUN      ] ChessGameStateIsOccupiedTest.p
[       OK ] ChessGameStateIsOccupiedTest.p (0 ms)
[ RUN      ] ChessGameStateIsOccupiedTest.n
[       OK ] ChessGameStateIsOccupiedTest.n (0 ms)
[ RUN      ] ChessGameStateIsOccupiedTest.r
[       OK ] ChessGameStateIsOccupiedTest.r (0 ms)
[ RUN      ] ChessGameStateIsOccupiedTest.q
[       OK ] ChessGameStateIsOccupiedTest.q (0 ms)
[ RUN      ] ChessGameStateIsOccupiedTest.SPACE
[       OK ] ChessGameStateIsOccupiedTest.SPACE (0 ms)
[----------] 13 tests from ChessGameStateIsOccupiedTest (0 ms total)

[----------] Global test environment tear-down
[==========] 29 tests from 3 test cases ran. (2 ms total)
[  PASSED  ] 29 tests.
```

Read the output very carefully and cross reference with the earlier example. (Hint: Inheritance.)

5.2 Testing a class: calling a method of the test suite

It's somewhat tedious to write test case by test case. For instance suppose you want to test your function in the chess games that checks for valid white pawn moves:

```
bool is_valid_white_pawn_move(int from_row, int from_col,
                              int to_row, int to_col);
```

Consider the case when a white pawn wants to move one square forward. There are $8 \times 6 = 48$ cases for `is_valid_white_pawn_move` to return `true`, i.e., the white pawn is unobstructed. Including the case where the white pawn is not able to move one square ahead, i.e., `is_valid_white_pawn_move` returns `false`, there would be 48×2 cases. And if you want to test both cases where the white pawn is obstructed by a white or a black piece, there would be 48×3 cases.

Even if you want to test just 10 cases, it's still pretty tedious.

Now it turns out that you can write one test class and in a test case, you can call a method of that test class to give you a different setup. In other words, you can have multiple test case scenarios with one single test class.

First go have to the `RectTest` class in the previous section and add a dummy method like this:

```
class RectTest : public ::testing::Test
{
public:
    // ...
protected:
    void helloworld()
    {
        std::cout << "hello world\n";
    }
    // ...
};
```

Now call `helloworld` in the area test case:

```
TEST_F(RectTest, area)
{
    helloworld(); //
    EXPECT_EQ(7, r.area()); // deliberate error
```

```
    EXPECT_EQ(6, r.area());
}
```

Run the test cases again and read the output. Get it?

Now let's write a test case that contains multiple cases. Let's use our chess game as an example.

Because I will need to set the characters in the chess board in a method of the test class, write this in the `ChessGameState` class:

```
class ChessGameState
{
public:
    ...

    void set(int r, int c, char p)
    {
        board_[r][c] = p;
    }
...
};
```

Next, write a method

```
bool is_valid_white_pawn_move(int from_row, int from_col,
                              int to_row, int to_col);
```

We are now going to test this method.

Write a test suite `ChessGameStateIsValidWhitePawnMoveTest` In the `Setup`, complete fill the chess board with `'K'` and `'k'`.

Next in this class, write a method for testing white pawn moves:

```
class ChessGameStateIsValidWhitePawnMoveTest
{
protected:
    void Setup()
    {
        // completely fill chess board with 'K' and 'k' randomly.
    }
    void setup(int row, int col, int num_empty)
    {
        // put 'P' at rol, col in chess board and make sure
        // there are num_empty squares in front of this pawn.
```

```
    }
    // ...
};
```

Then write this test case:

```
TEST_F(ChessGameStateIsValidWhitePawnMoveTest, Test6050one)
{
    setup(6, 0, 1);
    EXPECT_TRUE(chess.is_valid_white_pawn_move(6, 0, 5, 0));
}
```

which is a test for white pawn moving from (6, 0) to (5, 0) with one unoccupied space in front of (6, 0). The output for this test case if

```
[----------] 1 test from ChessGameStateIsValidWhitePawnMoveTest
[ RUN      ] ChessGameStateIsValidWhitePawnMoveTest.Test6050one
[       OK ] ChessGameStateIsValidWhitePawnMoveTest.Test6050one (0 ms)
[----------] 1 test from ChessGameStateIsValidWhitePawnMoveTest (0 ms total)
```

Get it? Now you can easily create more test cases.

Exercise 5.2.1. Add more test cases to test `is_valid_white_pawn_move`:

- `Test6050zero`: test moving pawn at (6, 0) moving to (5, 0) with zero empty squares in front. Function should return `false`.
- `Test6040two`: test moving pawn at (6, 0) moving to (4, 0) with 2 empty squares in front. Function should return `true`.
- `Test6040one`: test moving pawn at (6, 0) moving to (4, 0) with one empty squares in front. Function should return `false`.
- `Test6040zero`: test moving pawn at (6, 0) moving to (4, 0) with zero empty squares in front. Function should return `false`.
- `Test6030four`: test moving pawn at (6, 0) moving to (3, 0) with four empty squares in front. Function should return `false`.