Unit Testing

# 

**Table of Contents**

[I. OVERVIEW 4](#_Toc477536674)

[II. REQUIREMENTS BASED TESTING 4](#_Toc477536675)

[II.1 Stages in Requirements based Testing 4](#_Toc477536676)

[II.2 Requirements testing process: 4](#_Toc477536677)

[III. ORIENTED UNIT TESTING 5](#_Toc477536678)

[IV. TEST CASE 7](#_Toc477536696)

[IV.1 Prepare for a test 7](#_Toc477536697)

[IV.2 Requirement of test cases 7](#_Toc477536702)

[V. TEST CODE COVERAGE 8](#_Toc477536708)

[V.1 What is test code coverage? 8](#_Toc477536709)

[V.2 Requirement 9](#_Toc477536710)

[V.3 Technique 10](#_Toc477536711)

[V.4 Tool 10](#_Toc477536712)

[1. On Windows 10](#_Toc477536713)

[2. On Linux 12](#_Toc477536714)

[VI. GOOGLE C++ TESTING FRAMEWORK 13](#_Toc477536715)

[VI.1 Google C++ overview 13](#_Toc477536716)

[VI.2 Features 14](#_Toc477536717)

[VI.3 Platforms 15](#_Toc477536718)

[VI.4 Setup project unit test with Google C++ Framework 15](#_Toc477536719)

[VII. SUMMARY 18](#_Toc477536720)

[VII.1 Create new project using GoogleTest (gtest) 18](#_Toc477536721)

[On windows 18](#_Toc477536722)

[On Linux 21](#_Toc477536724)

[VII.2 Check code coverage with OpenCppCoverage tool 24](#_Toc477536725)

[On windows: 24](#_Toc477536726)

[On Linux 25](#_Toc477536727)

[VIII. Reference Links 27](#_Toc477536738)

# OVERVIEW

Unit testing is a [level of software testing](http://softwaretestingfundamentals.com/software-testing-levels/) where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed.

A unit test is code that exercises a specific portion of your codebase in a particular context. Typically, each unit test sends a specific input to a method and verifies that the method returns the expected value, or takes the expected action. Unit tests prove that the code you are testing does in fact do what you expect it to do.

# REQUIREMENTS BASED TESTING

Requirements-based testing is a testing approach in which test cases, conditions and data are derived from requirements. It includes functional tests and also non-functional attributes such as performance, reliability or usability.

## II.1 Stages in Requirements based Testing

* Defining Test Completion Criteria: Testing is completed only when all the functional and non-functional testing is complete.
* Design Test Cases:  A Test case has five parameters namely the initial state or precondition, data setup, the inputs, expected outcomes and actual outcomes.
* Execute Tests: Execute the test cases against the system under test and document the results.
* Verify Test Results: Verify if the expected and actual results match each other.
* Verify Test Coverage: Verify if the tests cover both functional and non-functional aspects of the requirement.
* Track and Manage Defects: Any defects detected during the testing process goes through the defect life cycle and are tracked to resolution. Defect Statistics are maintained which will give us the overall status of the project.

## II.2 Requirements testing process:

* Testing must be carried out in a timely manner.
* Testing process should add value to the software life cycle, hence it needs to be effective.
* Testing the system exhaustively is impossible hence the testing process needs to be efficient as well.
* Testing must provide the overall status of the project, hence it should be manageable.

# ORIENTED UNIT TESTING

# Create unit tests when object design is completed

* Black-box test: Test the functional model.
* White-box test: Test the dynamic model.

# Develop the test cases

* Goal: Find effective number of test cases

# Cross-check the test cases to eliminate duplicates

* Don't waste your time!

## Desk check your source code

* Sometimes reduces testing time

# Create a test harness

* Test drivers and test stubs are needed for integration testing

# Describe the test oracle

* Often the result of the first successfully executed test

# Execute the test cases

* Re-execute test whenever a change is made (“regression testing”)

# Keep unit tests small and fast

* Ideally the entire test suite should be executed before every code check in. Keeping the tests fast reduce the development turnaround time.

# Unit tests should be fully automated and non-interactive

* The test suite is normally executed on a regular basis and must be fully automated to be useful. If the results require manual inspection the tests are not proper unit tests

# Make unit tests simple to run

* Configure the development environment so that single tests and test suites can be run by a single command or a one button click.

# Measure the tests

* Apply coverage analysis to the test runs so that it is possible to read the exact execution coverage and investigate which parts of the code is executed and not.

# Fix failing tests immediately

* Each developer should be responsible for making sure a new test runs successfully upon check in, and that all existing tests runs successfully upon code check in.
* If a test fails as part of a regular test execution the entire team should drop what they are currently doing and make sure the problem gets fixed.

# Keep testing at unit level

* Unit testing is about testing classes. There should be one test class per ordinary class and the class behaviour should be tested in isolation. Avoid the temptation to test an entire work-flow using a unit testing framework, as such tests are slow and hard to maintain. Work-flow testing may have its place, but it is not unit testing and it must be set up and executed independently.

# Keep tests independent

* To ensure testing robustness and simplify maintenance, tests should never rely on other tests nor should they depend on the ordering in which tests are executed.

# Focus on execution coverage first

* Differentiate between execution coverage and actual test coverage. The initial goal of a test should be to ensure high execution coverage. This will ensure that the code is actually executed on some input parameters.

# Cover boundary cases

* Make sure the parameter boundary cases are covered. For numbers, test negatives, 0, positive, smallest, largest etc. For strings test empty string, single character string, non-ASCII string, multi-MB strings etc. For collections test empty, one, first, last, etc. For dates, test January 1, February 29, December 31 etc. The class being tested will suggest the boundary cases in each specific case. The point is to make sure as many as possible of these are tested properly as these cases are the prime candidates for errors.

# Provide negative tests

* Negative tests intentionally misuse the code and verify robustness and appropriate error handling.

# TEST CASE

## IV.1 Prepare for a test

# 1. Test plan (Test Leader or construction manager)

* Define the scope of testing, test strategy.
* Identify the risks and the element of surprise, the testing operation is handmade, which activity is automatic or both.
* Estimated cost of testing and construction testing schedule.
* Identification of environmental testing.

# 2. Test analysis & design

* Design testcase from the functional requirements and non-functional requirements of software.
* The testcase should cover all aspects of testing for each software requirements and all requests in the test strategy.

# 3. Test the executing

# 4. Test report

* Create error reports.
* Evaluation of test results and statistical requirements change.
* Calculate and distribute information measurement testing activities
* Create a summary assessment testing activities bug.
* Determine if achieved success criteria and completed testing yet

## IV.2 Requirement of test cases

# 1. As far as possible, write test cases in such a way that you test only one thing at a time. Do not overlap or complicate test cases.

# 2. Ensure that all positive scenarios and negative scenarios are covered.

* Testing Normal Conditions.
* Testing Unexpected Conditions.
* Bad Input Values.
* Boundary Conditions

# 4. Characteristics of a good test case:

* Accurate: Exacts the purpose.
* Economical: No unnecessary steps or words.
* Traceable: Capable of being traced to requirements.
* Repeatable: Can be used to perform the test over and over.
* Reusable: Can be reused if necessary.

# 5. Test Cases need to be simple and transparent:

* Create test cases that are as simple as possible. They must be clear and concise as the author of test case may not execute them.

# 6. Avoid test case repetition.

* Do not repeat test cases. If a test case is needed for executing some other test case, call the test case by its test case id in the pre-condition column

7. Ensure 100% Coverage

* Make sure you write test cases to check all software requirements mentioned in the specification document. Use Traceability Matrix to ensure no functions/conditions is left untested.

8. Test Cases must be identifiable.

* Name the test case id such that they are identified easily while tracking defects or identifying a software requirement at a later stage.

# TEST CODE COVERAGE

## V.1 What is test code coverage?

* Coverage analysis closes that feedback loop by reporting on the comprehensiveness of your unit tests. Code coverage reports allow developers to quickly find code that is not executed by the test suite.
* Test coverage measures the amount of testing performed by a set of test. Wherever we can count things and can tell whether or not each of those things has been tested by some test, then we can measure coverage and is known as test coverage.
* Test coverage is a useful tool for finding untested parts of a codebase. Test coverage is of little use as a numeric statement of how good your tests are.
* Criteria:
* Function coverage: each function in the program been executed or not.
* Statement coverage: each line of the source code been executed or not.
* Decision coverage (also known as Branch coverage): each control structure such as an if statement has been evaluated both to true and false.
* Condition coverage: each boolean sub-expression has been evaluated for both true and false.
* Path coverage: every possible route through a given part of the code been executed or not.
* Entry/exit coverage: every possible call and return of the function been executed or not

## V.2 Requirement

* Code coverage measures how much code of the application is being exercised when the tests are run.  This makes sure that the tests being run are actually testing the code of application. The basic measure of Code Coverage is the “Coverage Item” , which can by anything we have been able to count and see whether it has been tested or not.
* Measurement of Coverage can be determined by the following formula.

Coverage= Number of coverage items exercised / Total number of coverage items \*100%.

* It should be kept in mind that 100% code coverage does not mean that the application is 100% tested.  An application with high code coverage means it has been more thoroughly tested and would contain less software bugs than an application with low code coverage. There are numerous ways to calculate code coverage like program subroutines and program statements called during the execution of test suite.
* Benefit of code coverage measurement:
* It creates additional test cases to increase coverage
* It helps in finding areas of a program not exercised by a set of test cases
* It helps in determining a quantitative measure of code coverage, which indirectly measure the quality of the application or product.

## V.3 Technique

* CodeCoverageTools is tools that enable you measure the CodeCoverage of your tests.
* We cover the popular Code Coverage tools to measure Code Coverage: OpenCppCoverage, Gcov, [Testwell CTC++](http://www.testwell.fi/ctcdesc.html), [CoverageMeter](http://www.coveragemeter.com/), BullseyeCoverage, TestCocoon …

## V.4 Tool

## 1. On Windows

* **OpenCppCoverage** is an open source code coverage tool for C++ under Windows. The main usage is for unit testing coverage, but you can also use it to know the executed lines in a program for debugging purpose.
* **OpenCppCoverage** use program database file (.pdb) to know which source lines contain code and for html report.
* InstallOpenCppCoverage:

1. Get the OpenCppCoverage version 10.9.5.3 at [softwareteam\\tools\TestFramework\TestCoverage\Tools\OpenCppCoverage\winx64\OpenCppCoverageSetup-x64-0.9.5.3.exe](file:///Z:\tools\TestFramework\TestCoverage\Tools\TestCocoon\windows\TestCocoonSetup_1_6_13_x86.exe).

Or <http://opencppcoverage.codeplex.com/>

2. Install OpenCppCoverage.

3. Change Debug Information

Right click on project -> Select “Properties” item -> Select “Configuration Properties” item on the left frame:

* Select “C/C++” item -> Select “General” item -> Select “Debug Information Format” item on the right frame -> Select “**Program Database (/Zi)**” item from drop down list.
* Select “Linker” item -> Select “Debugging” item -> Select “Generate Debug Info” item on the right frame -> Select “**Yes(/DEBUG)**” item from drop down list.

4. Build Application

5. Open command line with administrator

|  |
| --- |
| **[Path]OpenCppCoverage.exe --sources=sourceDir --export\_type=html:pathExport --excluded\_sources=pathExcluded [Path]xxx.exe** |

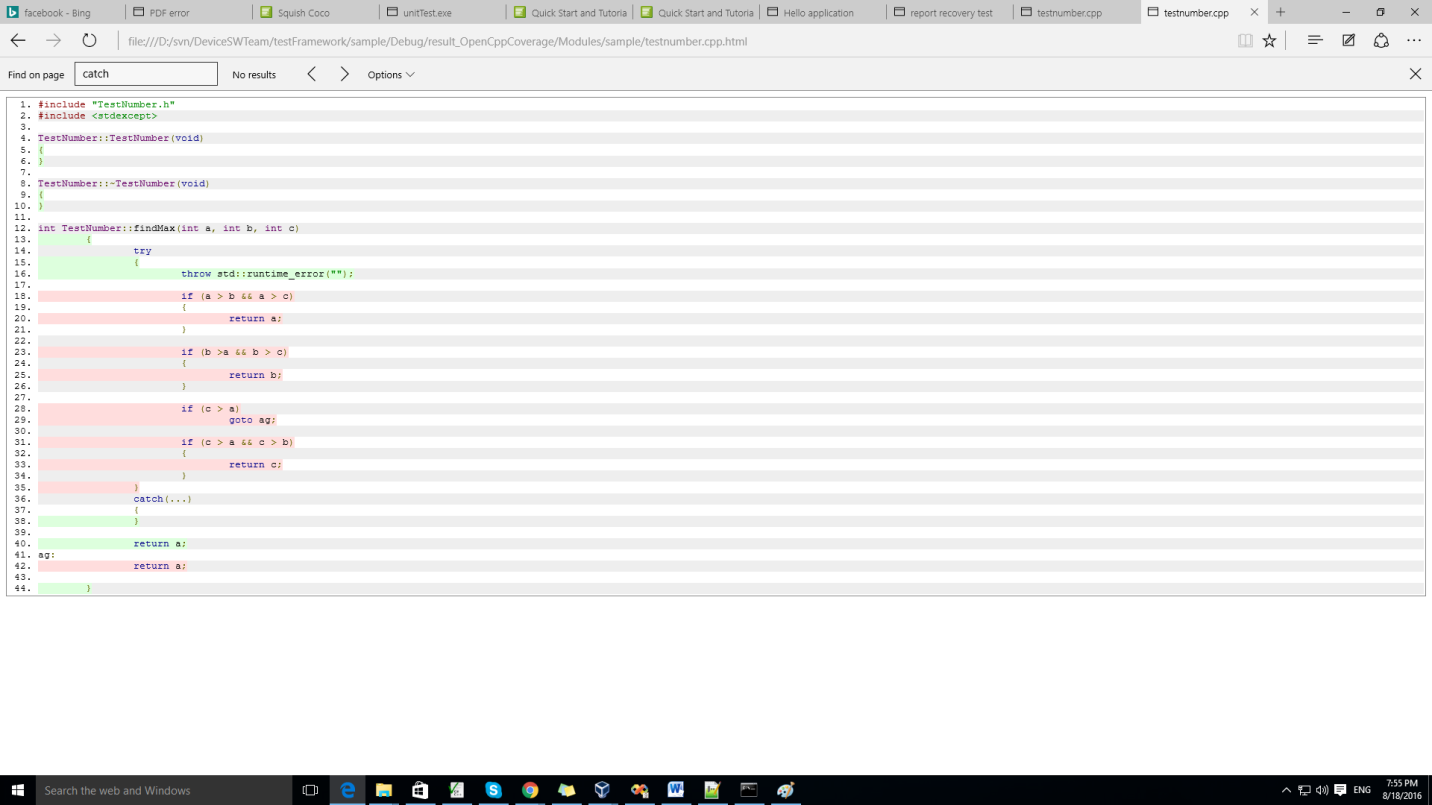
Ex: cd/d D:\svn\DeviceSWTeam\Gemini\trunk\projects\freescale\GeminiDemo\test\Win32\Release "C:\Program Files\OpenCppCoverage\OpenCppCoverage.exe" --source="D:\svn\DeviceSWTeam\Gemini\trunk\projects\freescale\GeminiDemo\test" --excluded\_sources="D:\svn\DeviceSWTeam\Gemini\trunk\projects\freescale\GeminiDemo\test\GeneratedFiles" --export\_type=html:"D:\svn\DeviceSWTeam\Gemini\trunk\projects\freescale\GeminiDemo\Coverage\ResultTest" test.exe.

6. Open result at “*pathExport”*



7. Click items on the right frame to show detail.

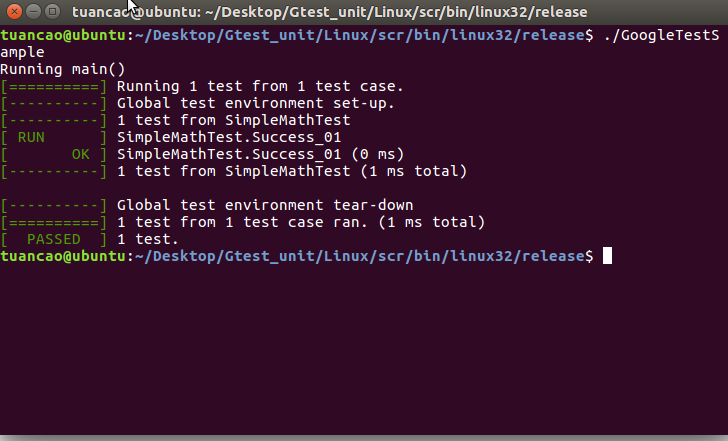
You will see the code coverage of the method, condition, file, line with different colors: red (never executed), green (executed)



## 2. On Linux

* **Gcov** is a tool you can use in conjunction with GCC to test code coverage in your programs.
* When using gcov, you must first compile your program with two special GCC options: ‘-fprofile-arcs -ftest-coverage’.
* Install Gcov:

1. Rebuild and run “GoogleTestSample” file.



1. Running the program will cause profile output to be generated. The .gcno, and .gcda data files will be placed in the object file directory
2. View result:



# GOOGLE C++ TESTING FRAMEWORK

## VI.1 Google C++ overview

* Google C++ Testing Framework helps you write better C++ tests.
* Tests should be independent and repeatable.
* Tests should be well organized and reflect the structure of the tested code.
* Tests should be portable and reusable.
* When tests fail, they should provide as much information about the problem as possible.
* The testing framework should liberate test writers from housekeeping chores and let them focus on the test content.
* Tests should be fast.
* Ex:

|  |
| --- |
| #include <gtest/gtest.h>  TEST(MyTestSuitName, MyTestCaseName)  {  int actual = 1;  EXPECT\_GT(actual, 0);  EXPECT\_EQ(1, actual) << "Should be equal to one";  } |

## VI.2 Features

An [XUnit](https://en.wikipedia.org/wiki/XUnit) test framework.

* Portable
* Test discovery.
* A rich set of assertions.
* User-defined assertions.
* Death tests.
* Fatal and non-fatal failures.
* Value-parameterized tests.
* Type-parameterized tests.
* Various options for running the tests.
* XML test report generation.

## VI.3 Platforms

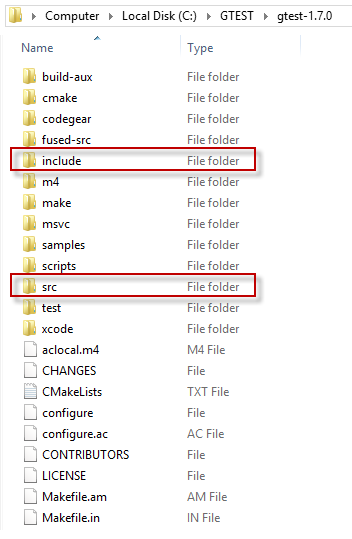
* Google test has been used on a variety of platforms: Linux, Mac OS X, Windows, Cygwin, MinGW, Windows Mobile, Symbian.
* Windows Requirements: Microsoft Visual C++ v7.1 or newer.
* Linux Requirements: These are the base requirements to build and use Google Test from a source package (as described below): GNU-compatible Make or gmake, POSIX-standard shell, POSIX(-2) Regular Expressions (regex.h), A C++98-standard-compliant compiler.

## VI.4 Setup project unit test with Google C++ Framework

In the following example, we used Visual Studio 2008 with 4 steps:

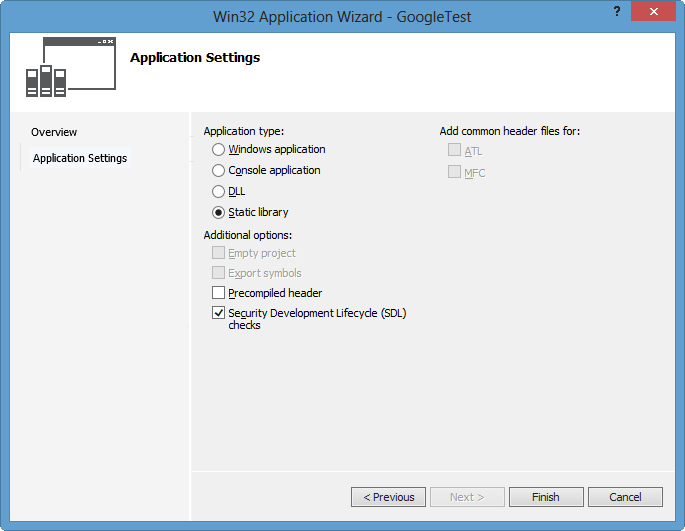
**Step 1.** Download Google test (gtest)

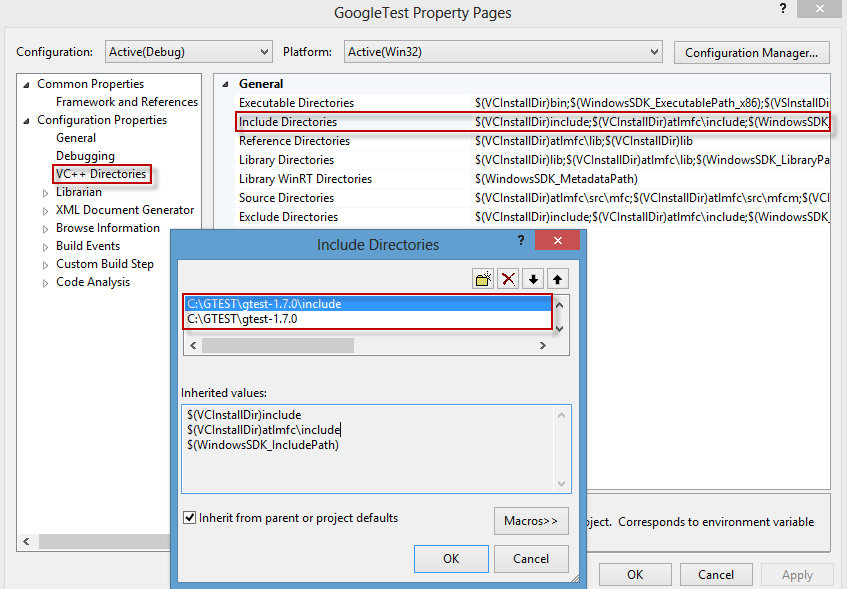
* Download the **gtest-1.7.0-rc1.zip** from [Google C++ Unit Test](http://code.google.com/p/googletest/) or from [gtest-1.7.0-rc1.zip](http://www.bogotobogo.com/cplusplus/files/cpptest/gtest-1.7.0-rc1.zip), then extracts it.
* Let's look at the **C:\GTEST\gtest-1.7.0** directory to see what files are there.



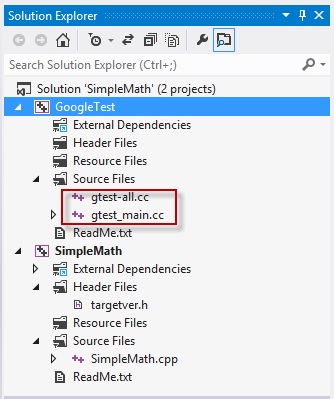
* The **src** folder has all the gtest source files and later we need to add the **include** directory to the include path.

**Step 2**. Compile gtest into a static library

* Create a new static library project with a name **GoogleTest**.  
  Add->New Project->Win32 Project->Static Library without precompiled header.
* Right click on our new project, **GoogleTest**.
* On the Properties Pages, add include path: **C:\GTEST\gtest1.7.0** and **C:\GTEST\gtest-1.7.0\include**.



**Step 3.** Add source files by Add->Existing Item...   
**C:\GTEST\gtest-1.7.0\src\gtest\_all.cc**   
and **C:\GTEST\gtest-1.7.0\src\gtest\_main.cc**.



**Step 4.** Build **GoogleTest** into static library.

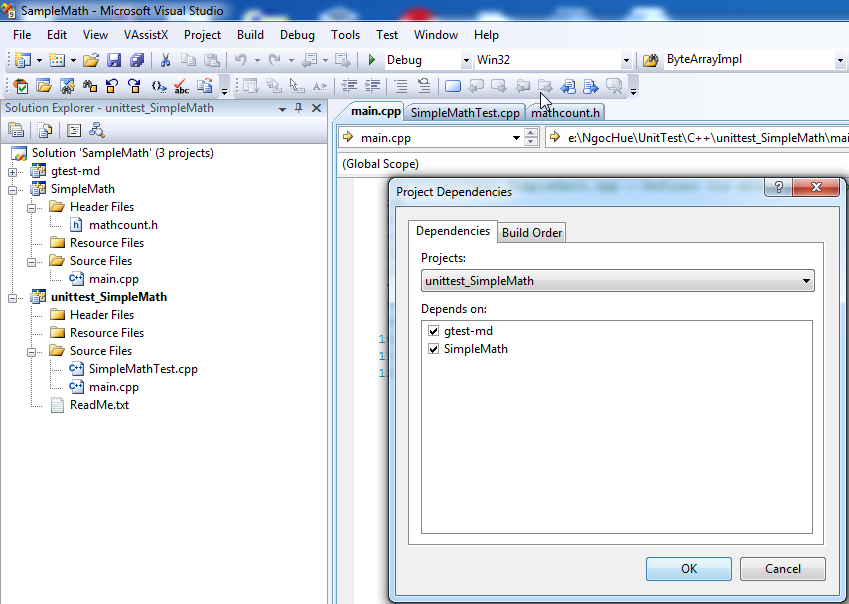
# SUMMARY

## VII.1 Create new project using GoogleTest (gtest)

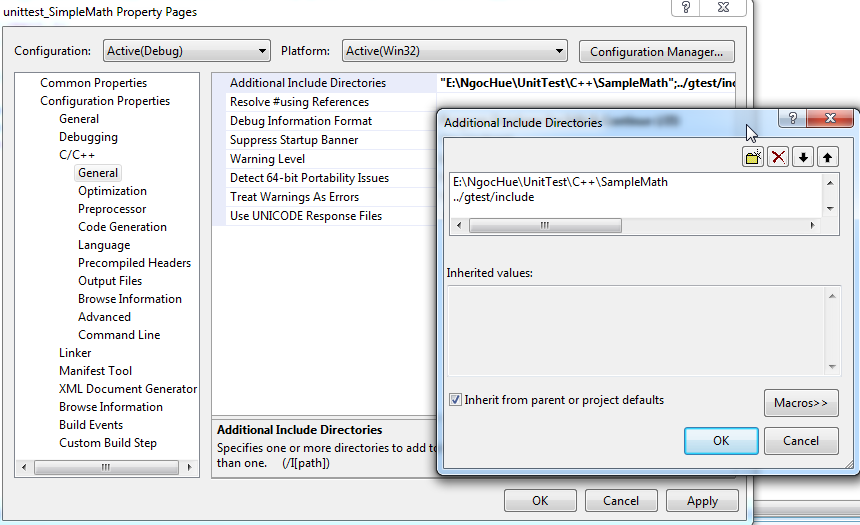
## On windows

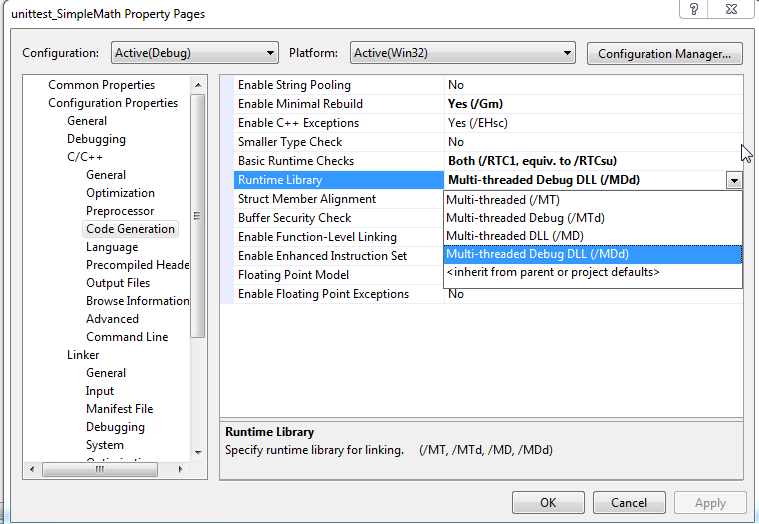
## New project

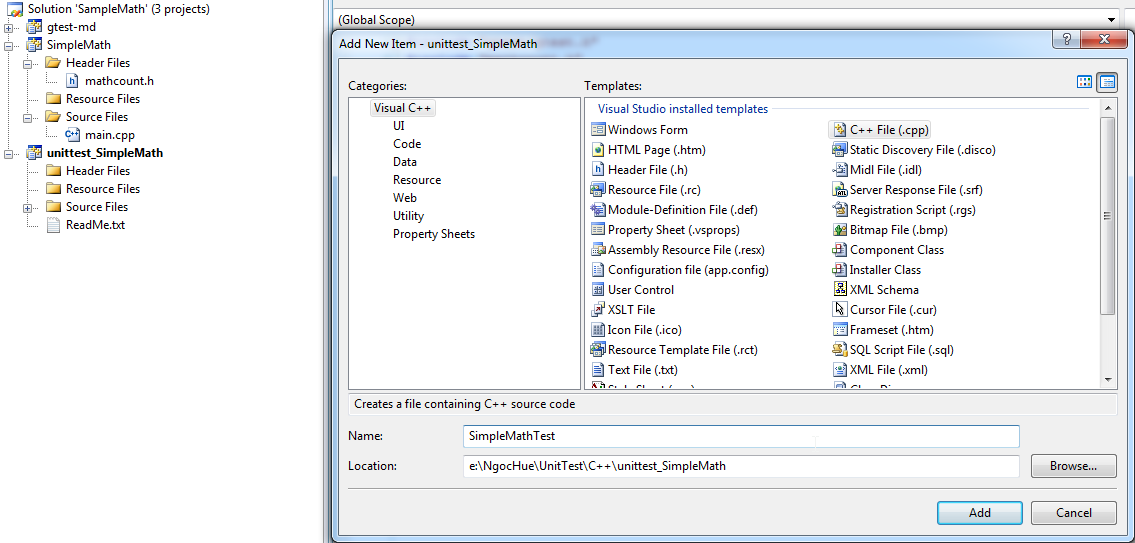
1. Select “Empty Project” -> Input Name, Location -> Click OK
2. Right click “inittest\_SimpleMath” project -> select “Project dependencies…



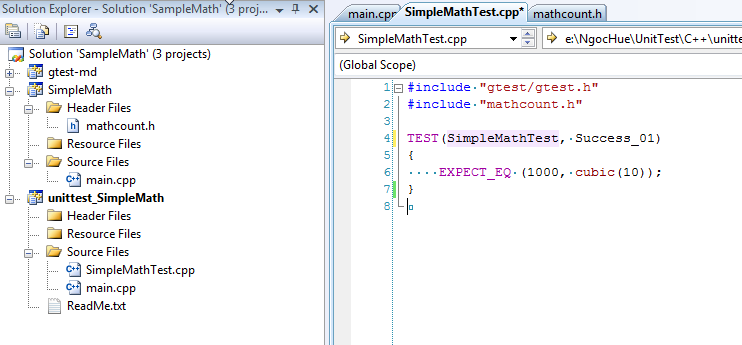
1. Right click “inittest\_SimpleMath” project -> Select “Properties’ item.
2. Select “Configuration Properties” item -> Select “C/C++” item -> Input “Additional include Directories” field.



1. Select “Configuration Properties” item -> Select “Code Generation” item -> Select “Runtime library” is “Multi-threaded Debug DLL(/MTd)” (the same with dependency project).
2. Right click “inittest\_SimpleMath” project -> Select “Set as StartUp Project” item.
3. Right click “inittest\_SimpleMath” project -> Select “Add” item -> Select “New Item” item.



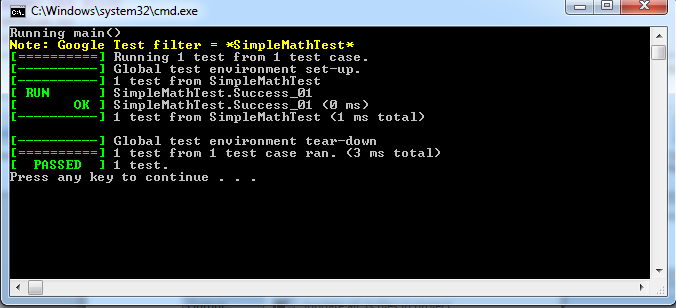
1. Click to open “SimpleMathTest.cpp” to write test case



1. Create main.cpp: Right click “inittest\_SimpleMath” -> Select “Add” item-> Select “New Item” -> Select “Visual C++” item on the left frame -> Select “C++ File(.cpp)” item on the right frame -> Input “Name” -> Select “Location” -> Click “Add” button.
2. File main.cpp



1. Right click “inittest\_SimpleMath project -> Select “Build” item.
2. Press Ctrl+F5 -> Show Result Tests.



## On Linux

1. Download GoogleTest: [gtest-1.7.0.zip](https://code.google.com/p/googletest/downloads/detail?name=gtest-1.7.0.zip)

|  |
| --- |
| unzip gtest-1.7.0.zip  cd gtest-1.7.0  sudo ../ gtest/ build  run: cmake ../  to generate “libgtest.a”, “ libgtest main.a”  sudo mkdir ../inc folder  sudo mkdir ../lib folder  sudo mkdir ../src folder  sudo mkdir ../test folder  copy “libgtest.a”, “ libgtest main.a” to “lib” folder  copy gtest from /gtest/include to “inc” folder |

1. GoogleTest Example:

* MathCount project:

File src/mathcount.h

File src/mathcount.cpp

* Unit test code:

File test/SampleMathTest.cpp

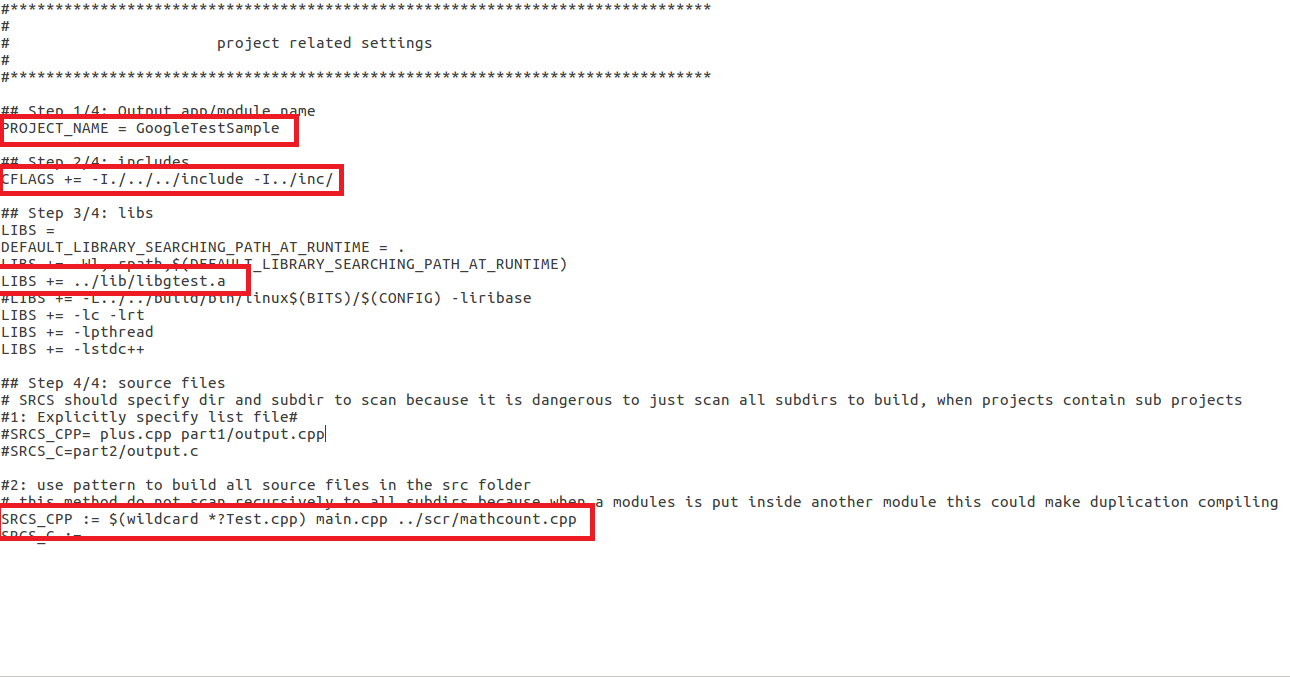
File test/main.cpp

* Create Make file for project

File test/Makefile

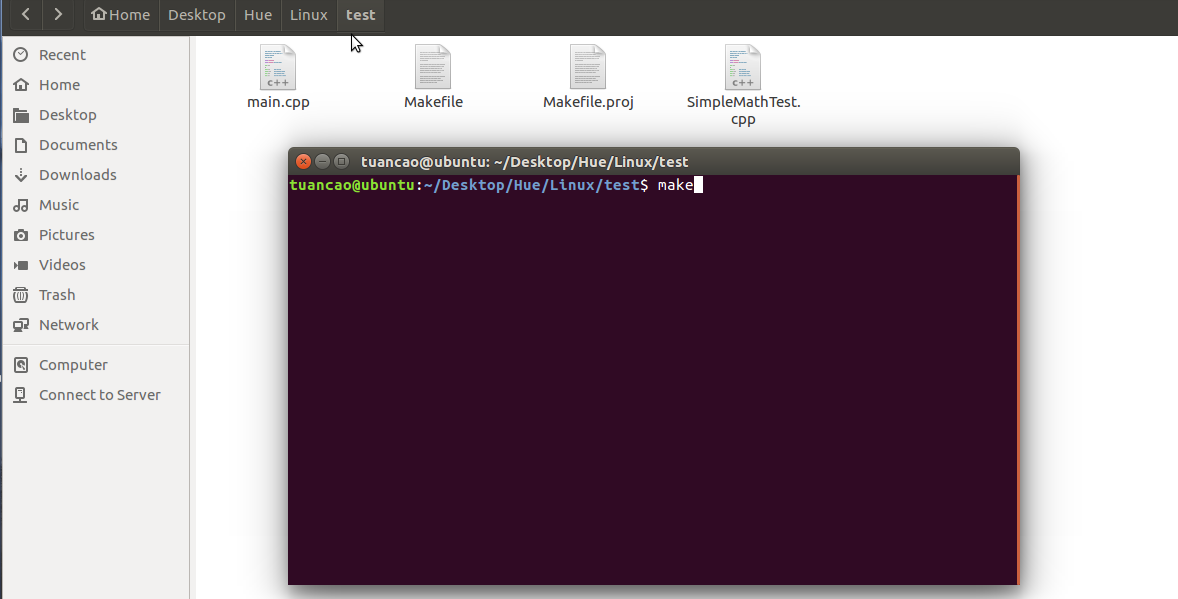
File test/Makefile.proj

1. Configuration: test/Makefile.proj

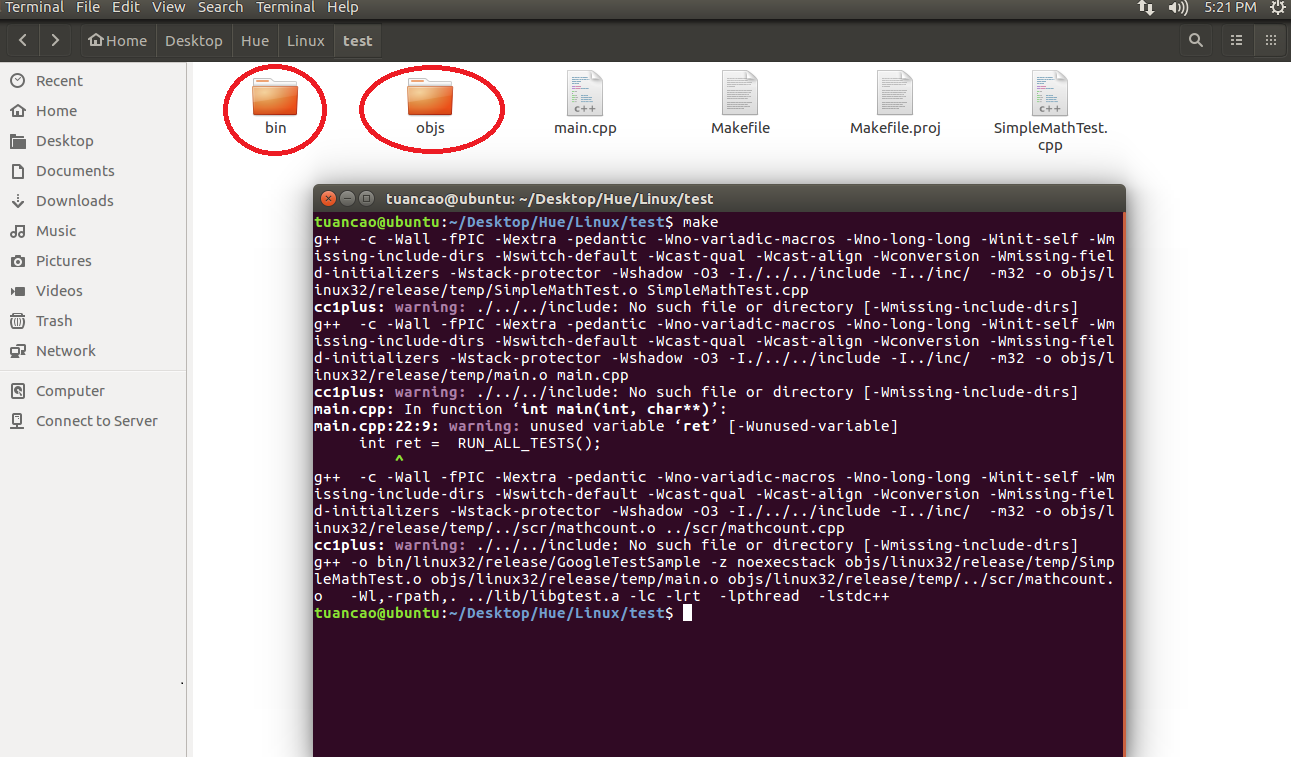


* Step1: Name of project: GoogleTestSample
* Step2: Path of includes file in gtest
* Step3: Path of lib file
* Step4: Build all test file in the “test” folder

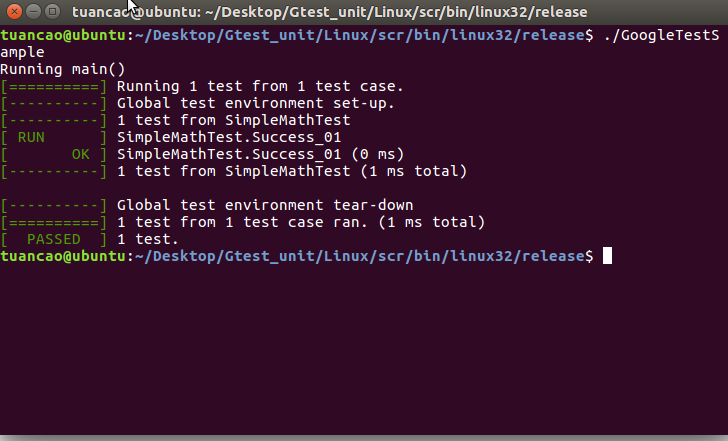
1. Compile: make



Compile successful



1. Final, Go to “OUTDIR” project in “bin” folder to run “GoogleTestSample.exe” and show result test.



## VII.2 Check code coverage with OpenCppCoverage tool

## On windows:

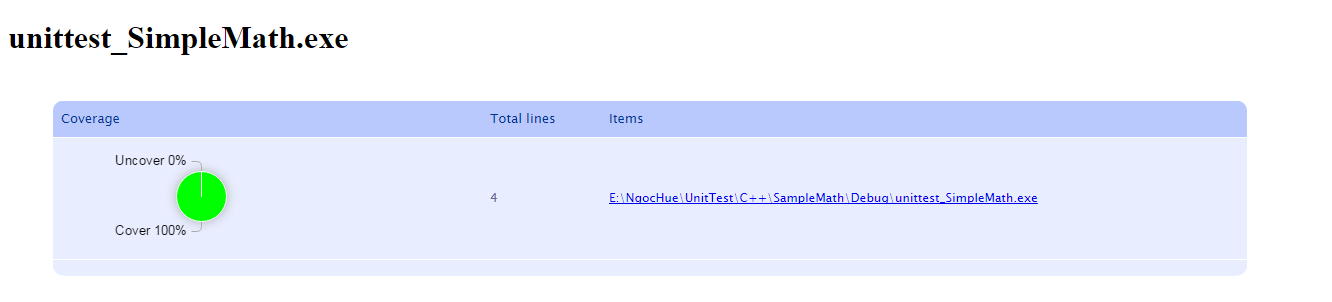
1. [Install OpenCppCoverage](#_V.4_Tool).
2. [Change Debug Information](#_V.4_Tool)
3. [Build project test](#_V.4_Tool)
4. Open command line with administrator

E:\NgocHue\UnitTest\C++\SampleMath\Debug>"C:\Program Files\OpenCppCoverage\OpenCppCoverage.exe" --source="E:\NgocHue\UnitTest\C++\SampleMath" --export\_type=html

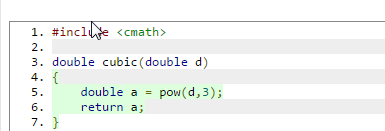
:"E:\NgocHue\UnitTest\C++\SampleMath\ResultTest"

unittest\_SimpleMath.exe

1. Open result at “*pathExport:* "E:\NgocHue\UnitTest\C++\SampleMath\ResultTest"



1. Click items on the right frame to show detail.



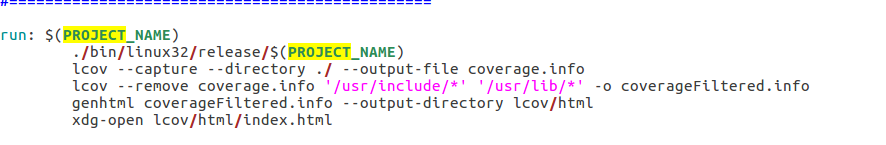
## On Linux

## Install lcov in directory

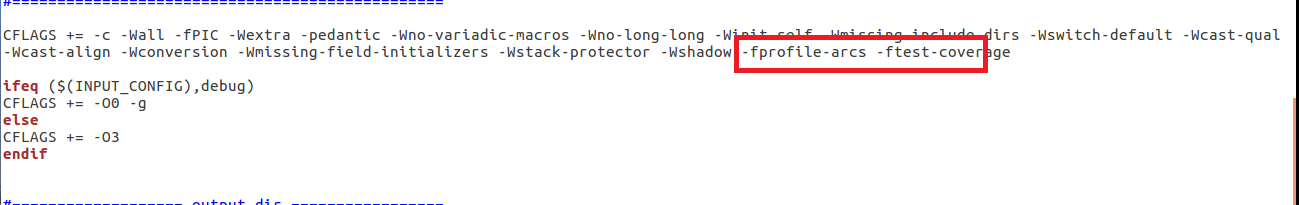
## sudo apt-get install lcov

## Add more lines to configuration in Makefile

|  |
| --- |
| run: $(PROJECT\_NAME)./bin/linux32/release/$(PROJECT\_NAME)lcov --capture --directory ./ --output-file coverage.infolcov --remove coverage.info '/usr/include/\*' '/usr/lib/\*' -o coverageFiltered.infogenhtml coverageFiltered.info --output-directory lcov/htmlxdg-open lcov/html/index.html |



## Add more parameters in “CFLAGS”

****

## E:\111\2017_03_07_13_20_48_Ubuntu_VMware_Workstation_12_Player_Non_commercial_use_only_.pngAdd more parameters in Make.proj file

1. Compile: make run
2. View result shows

* Double click on “mathcount.cpp”
* You will see the code coverage of the method, condition, file, line with different colors: red (never executed), green (executed), orange (partially executed).



# Reference Links

<http://geosoft.no/development/unittesting.html>

<http://softwaretestingfundamentals.com/unit-testing/>

<https://www.slideshare.net/guest45ac48/unit-test>

<https://developer.salesforce.com/page/How_to_Write_Good_Unit_Tests>

<http://www.c-sharpcorner.com/UploadFile/953215/tips-for-writing-test-cases/>

<https://pjcj.net/testing_and_code_coverage/paper.html>

<http://opencppcoverage.codeplex.com/>

<http://open-tube.com/10-code-coverage-tools-c-c/>

<https://github.com/google/googletest/blob/master/googletest/docs/Documentation.md>

<http://www.yolinux.com/TUTORIALS/Cpp-GoogleTest.html>