Raymii.org

Quis custodiet ipsos custodes?

Home | About | All pages | Cluster Status | RSS Feed | Gopher

C++ project setup with CMake & unit tests (google test)

Published: 01-10-2019 | Last update: 06-11-2019 | Author: Remy van Elst | Text only version of this article

❗ **This post is over two years old. It may no longer be up to date. Opinions may have changed.**

Table of Contents

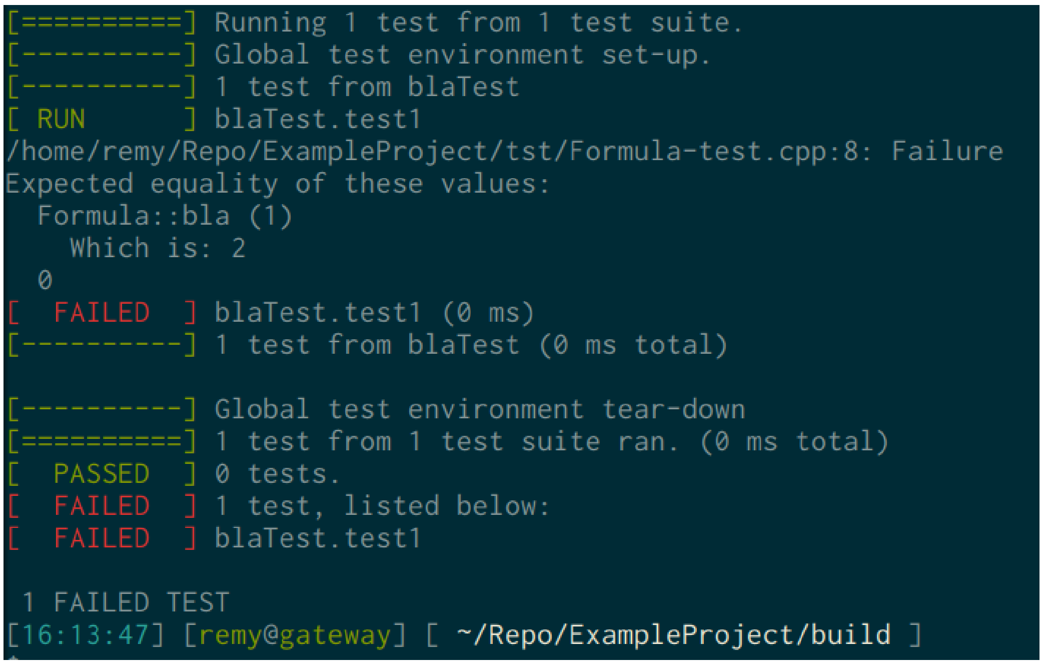
This guide will show you how to setup a new C++ project with CMake and unit tests via Google's test framework. With this setup you can get started right away with test-driven-development in C++. It is also simple enough to look and figure out how to add gtest to your existing project and start doing TDD on your legacy (existing) codebase.

I'm developing a desktop monitoring app, Leaf Node Monitoring, open source, but paid. For Windows, Linux & Android, go check it out.

Consider sponsoring me on Github. It means the world to me if you show your appreciation and you'll help pay the server costs.

You can also sponsor me by getting a Digital Ocean VPS. With this referral link you'll get $100 credit for 60 days.

The picture below shows the end result, a running unit test:



```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from blaTest
[ RUN      ] blaTest.test1
/home/remy/Repo/ExampleProject/tst/Formula-test.cpp:8: Failure
Expected equality of these values:
  Formula::bla (1)
    Which is: 2
  0
[  FAILED  ] blaTest.test1 (0 ms)
[----------] 1 test from blaTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] blaTest.test1

 1 FAILED TEST
[16:13:47] [remy@gateway] [ ~/Repo/ExampleProject/build ]
```

There are a million different ways to "do" C++ projects, but using `CMake` and the google testing framework has my preference. That's not to say that using a `Makefile` or Boost Unit Test is bad, use whatever suits your needs. This guide however will focus on just `CMake` and `gtest`.

It assumes a system running Ubuntu (18.04). It also works on Windows with `mingw`, but I haven't tested in with MSVC.

My preferred code editor is CLion from Jetbrains, which has most of this built in. This guide however focusses on the manual / command line way since CLion is nonfree (and paid) software.

The process is not that complicated:

- Install software (cmake and googletest)
- Create folder structure
- Create the `CMakeLists.txt` files
- Create some sample code and sample tests
- Compile everything
- Run the tests

Install cmake & googletest

I assume you already have your compiler installed and working. Installing cmake can be done with the package manager on Ubuntu:

```
apt-get install cmake
```

On Windows, you can use MinGW or `cygwin` to install your development tools including CMake. Clion offers a nice GUI for that.

Googletest is available as a git repository which you can clone and then copy into your project. You could go all fancy with CMake scripts to download it if it's not already in your project, but since you can clone once and copy later on I choose not to automate it. Clone the repository:

```
git clone https://github.com/google/googletest/
```

gtest comes with a `CMakeLists.txt` so integrating it in your project is easy.

Folder structure

Create your C++ project folder. I like to keep the following structure for simple projects:

```
$ tree -L 2 ExampleProject/
ExampleProject/
|-- build/
|-- CMakeLists.txt
|-- lib/
|   `-- googletest
|-- src/
|   |-- CMakeLists.txt
|   |-- Formula.cpp
|   |-- Formula.h
|   `-- main.cpp
`-- tst/
    |-- CMakeLists.txt
    |-- Formula-test.cpp
    `-- main.cpp
```

Here is a oneliner to create the folders:

```
mkdir -p ExampleProject/{build,lib,src,tst}
```

Copy the `googletest` repository folder your cloned earlier into the `lib/` folder.

If you have multiple components you can create extra sub folders, but that does require tweaking the `CMakeLists.txt` files to work with multiple libraries.

Most of my personal projects are simple enough to fit into one folder as above.

In the `tst` folder the unit tests reside. I try to keep the tests limited to the same function in seperate files. In the above example I have `Formula.h` and `Formula.cpp`, which house the example `Formula` class. All unit tests related to this class thus should reside in `Formula-test.cpp`.

CMakeLists.txt

The file `CMakeLists.txt` contains a set of directives and instructions describing the project's source files and targets (executable, library, or both). This can get quite complex quite fast, CMake has many options. I try to keep it simple in this guide.

I'm using a non-recommended way to include files. For simple projects with a few files you should use the following:

```
add_executable(ExampleProject main.cpp file1.cpp file1.h)
```

I'm using this:

```
file(GLOB_RECURSE SOURCES LIST_DIRECTORIES true *.h *.cpp)
```

That is a recursive search to include all `*.cpp` and `*.h` in the folder. In my IDE I have auto-reload enabled, that way I can't forget to add a file to `CMakeLists.txt` every time. For proper administration you should not use this since it just includes everything, could have unwanted side-effects.

Update 2019-11-07: If you want Boost in this setup, read this article from me.

Each subdirectory in our case also needs a `CMakeLists.txt` file.

Main folder CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(ExampleProject)

set(CMAKE_CXX_STANDARD 14)

include_directories(src)

add_subdirectory(src)
add_subdirectory(tst)
add_subdirectory(lib/googletest)
```

The name of the project is `ExampleProject`, that variable is used in other files. The rest of the file just includes the different subfolders. If you omit the `include_directories(src)`, your tests will not be able to find the header files.

src folder CMakeLists.txt:

```
set(BINARY ${CMAKE_PROJECT_NAME})

file(GLOB_RECURSE SOURCES LIST_DIRECTORIES true *.h *.cpp)

set(SOURCES ${SOURCES})

add_executable(${BINARY}_run ${SOURCES})

add_library(${BINARY}_lib STATIC ${SOURCES})
```

The name of the compiled program will be `ExampleProject_run`, which is what we defined in `add_executable`. The `add_library` is used to include the code in the unit tests.

tst folder CMakeLists.txt:

```
set(BINARY ${CMAKE_PROJECT_NAME}_tst)

file(GLOB_RECURSE TEST_SOURCES LIST_DIRECTORIES false *.h *.cpp)

set(SOURCES ${TEST_SOURCES})

add_executable(${BINARY} ${TEST_SOURCES})

add_test(NAME ${BINARY} COMMAND ${BINARY})

target_link_libraries(${BINARY} PUBLIC ${CMAKE_PROJECT_NAME}_lib gtest)
```

This list used the `src` defined library and adds the tests as a target. The compiled executable file is named `ExampleProject_tst`.

Add some (example) source code and tests

At this point you start developing. But since this is a example setup, I'll add a simple class file to show you how to do the unit tests.

Source Code

Copy the below code into your project:

`src/main.cpp`:

```
#include <iostream>
#include "Formula.h"

int main() {
    std::cout << "Bla: " << Formula::bla(2) << std::endl;
    return 0;
}
```

`src/Formula.h`:

```
#ifndef EXAMPLEPROJECT_FORMULA_H
#define EXAMPLEPROJECT_FORMULA_H
```

```
class Formula {
public:
    static int bla(int arg1);
};

#endif //EXAMPLEPROJECT_FORMULA_H
```

`src/Formula.cpp`:

```
#include "Formula.h"

int Formula::bla(int arg1) {
    return arg1 * 2;
}
```

This function returns the given `int` multiplied by 2.

Test code

The following code is to setup the unit tests.

`tst/main.cpp`:

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

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

This file will run all the tests and since we recursively included everything with CMake, it will effectively run all tests in all files in this folder.

`tst/Formula-test.cpp`:

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

TEST(blaTest, test1) {
    //arrange
    //act
    //assert
    EXPECT_EQ (Formula::bla (0),  0);
    EXPECT_EQ (Formula::bla (10), 20);
    EXPECT_EQ (Formula::bla (50), 100);
}
```

The Google Test Primer is a great starting point to learn more on the specifics of the testing framework.

Compile all the things

Now that we have sourcecode and testcode in place we can compile everything (both the binary and the tests).

Do note that you should do this in the `build` folder. If you do it in the main folder it will work, but it will litter up the directory.

```
cd build
cmake .. -DCMAKE_BUILD_TYPE=Debug -G "Unix Makefiles"
```

Output:

```
-- The C compiler identification is GNU 7.4.0
-- The CXX compiler identification is GNU 7.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found PythonInterp: /usr/bin/python (found version "2.7.15")
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /home/remy/Repo/ExampleProject/build
```

There are now a bunch of files and folders in the build folder, most important, the `Makefile`. You can now compile the project:

```
make all
```

Output:

```
Scanning dependencies of target ExampleProject_run
[  8%] Building CXX object src/CMakeFiles/ExampleProject_run.dir/Formula.cpp.o
[ 16%] Building CXX object src/CMakeFiles/ExampleProject_run.dir/main.cpp.o
[ 25%] Linking CXX executable ExampleProject_run
[ 25%] Built target ExampleProject_run
Scanning dependencies of target ExampleProject_lib
[ 33%] Building CXX object src/CMakeFiles/ExampleProject_lib.dir/Formula.cpp.o
[ 41%] Building CXX object src/CMakeFiles/ExampleProject_lib.dir/main.cpp.o
[ 50%] Linking CXX static library libExampleProject_lib.a
[ 50%] Built target ExampleProject_lib
Scanning dependencies of target gtest
[ 58%] Building CXX object lib/googletest/CMakeFiles/gtest.dir/src/gtest-all.cc.o
[ 66%] Linking CXX static library ../libgtestd.a
[ 66%] Built target gtest
Scanning dependencies of target ExampleProject_tst
[ 75%] Building CXX object tst/CMakeFiles/ExampleProject_tst.dir/Formula-test.cpp.o
[ 83%] Linking CXX executable ExampleProject_tst
[ 83%] Built target ExampleProject_tst
Scanning dependencies of target gtest_main
[ 91%] Building CXX object lib/googletest/CMakeFiles/gtest_main.dir/src/gtest_main.cc.o
[100%] Linking CXX static library ../libgtest_maind.a
[100%] Built target gtest_main
```

You now have two executable files, as defined in the `CMakeLists.txt`:

```
$ find . -executable -type f
./tst/ExampleProject_tst
./src/ExampleProject_run
```

Run all the things

If all went well, the code should run:

```
./src/ExampleProject_run
```

Output:

```
Bla: 4
```

The tests as well:

```
./tst/ExampleProject_tst
```

Output:

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from blaTest
[ RUN      ] blaTest.test1
[       OK ] blaTest.test1 (0 ms)
[----------] 1 test from blaTest (0 ms total)

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

A quick one-liner to compile and run the tests. You can run this whenever you want to re-run the tests (after changing code for example):

```
make ExampleProject_tst; tst/ExampleProject_tst
```

Output:

```
[ 37%] Built target ExampleProject_lib
[ 62%] Built target gtest
Scanning dependencies of target ExampleProject_tst
[ 75%] Building CXX object tst/CMakeFiles/ExampleProject_tst.dir/Formula-test.cpp.o
[ 87%] Linking CXX executable ExampleProject_tst
[100%] Built target ExampleProject_tst
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from blaTest
[ RUN      ] blaTest.test1
/home/remy/Repo/ExampleProject/tst/Formula-test.cpp:8: Failure
Expected equality of these values:
  Formula::bla (1)
    Which is: 2
  0
[  FAILED  ] blaTest.test1 (0 ms)
[----------] 1 test from blaTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] blaTest.test1

 1 FAILED TEST
```

As you can see I changed a unit test so that it failed.

Tags: boost , c++ , cmake , cpp , development , googletest , linux , software , testing , ttd , tutorials

Search

Home | About | All pages | Cluster Status | Generated by ingsoc.