cmake 101

John S. Fourkiotis

November 6, 2011

Contents

License

1 Introduction

This article presents a simple introduction to the `cmake`[1] build system. The *audience* of this article should be engineers and scientists that use `C/C++` in their applications and research. This article is not a full-blown tutorial of `cmake`. It just presents the amount of information needed to develop a simple *application* or *project*. This information is presented through building a simple math library that is used from other simple programs. All sources used in this article are hosted in `git` [2]. To grab them, just use `git clone git://github.com/jstark/cmake101` [3]. The tutorial assumes a UNIX or GNU/Linux OS, with `cmake` installed. If you run a Windows box, check the appendix for instructions.

2 The quadratic equation

The first program that I write to learn a new programming language is one that finds the roots of the world-famous (I hope) *quadratic equation*. The quadratic equation is given by:

$$ax^2 + bx + c = 0 \tag{1}$$

where coefficients $a, b, c \in \Re$ and $a \neq 0$. The coefficients a, b and c could also be *complex* numbers, but let's keep things simple.

Whether a quadratic equation has complex or real roots depends on the value of the *discriminant* δ which is equal to:

$$\delta = b^2 - 4ac$$

The discriminant can be 0, negative or positive. If it is positive, then equation (1) has two dinstict real roots, x_1 and x_2 given by:

$$x_{1,2} = \frac{-b \pm \sqrt{\delta}}{2a}$$

If the discriminant is 0, then there are two identical real roots, e.g $x_1 = x_2$ given by:

$$x_1 = x_2 = \frac{-b}{2a}$$

[1] see www.cmake.org

[2] check www.github.com

[3] grab git from http://git-scm.com/download

If the discriminant is negative, then the quadratic equation has two complex roots. The two complex roots are given by:

$$x_{1,2} = \frac{-b \pm i\sqrt{\delta}}{2a}$$

where i is the square root of -1. For those interested in the details, check the wiki article http://en.wikipedia.org/wiki/Quadratic_equation.

3 Starting up the project

Let's say that we want to build a simple application that will ask the user to type the three coefficients, and then present the roots of the quadratic equation. Because our quadratic equation solver will be used by many programs, we want it to be in a library (*shared* or *static*) that will be used from our programs. We also want to include some tests for our library, so that we can be sure that any changes we make to the code, won't create any bugs in our solver.

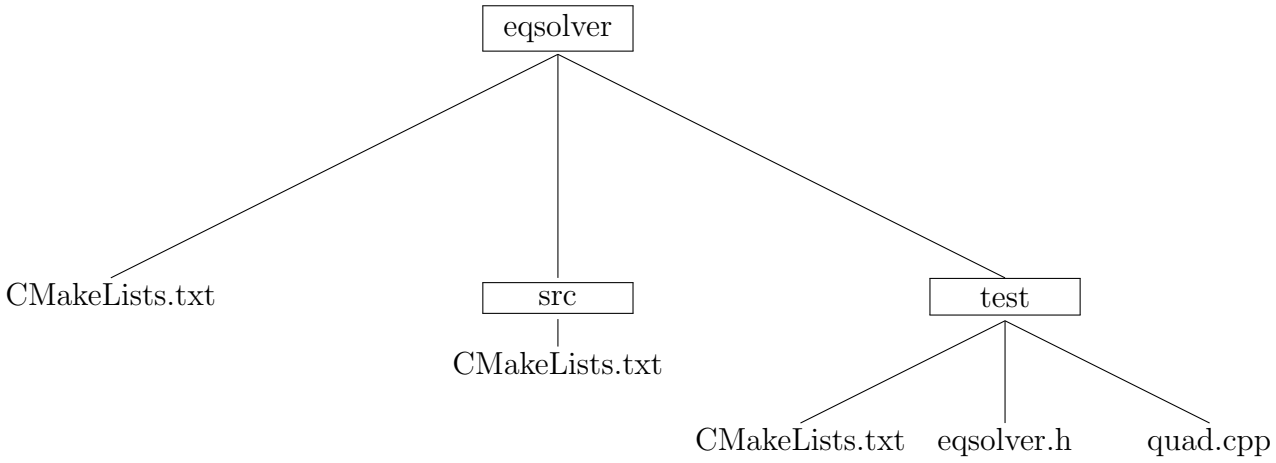
Our quadratic equation solver won't be our only math functionality offered by our library. It will be just the first. We can think of it as the first component of a library. Let's name this library as `eqsolver`, which will also be our project name. To start up our project, we can take the following easy steps:

Create a project skeleton For our little project, we need to create a root directory named `eqsolve` and add to it two more directories, `src`, which will hold our source files, and `test` which will hold simple library tests.

Add `cmake` support To make the project *understandable* for `cmake`, add a file named `CMakeLists.txt` *to each of our directories*. We will add to these files later.

Add the source files Add a header file named `eqsolver.h` and a source file named `quad.cpp` to the source folder `src`. We will add code to these files in the following section.

Our project structure should now look more or less this:
We will add files to the `src` directory later.



4 `cmake` setup

Now that our project structure is ready, we can add some `cmake` *scripting code* to the three `CMakeLists.txt` files. Add the following lines to the first file, in `eqsolver` directory:

file eqsolver/CMakeLists.txt

```
1 cmake_minimum_required (VERSION 2.6)
2
3 project (eqsolver)
4
5 add_subdirectory (src)
6
7 enable_testing()
8 add_subdirectory (test)
```

At the first line, we require a `cmake` version of at least 2.6. Then we name our project (at line 3), and state that there are two more directories (`src` and `test`) in this project, that need processing by `cmake`. We state this by using the `add_subdirectory` command. Each directory that is added by using `add_subdirectory` must contain another `CMakeLists.txt` file, that describes to `cmake` what to do in that directory.

The `CMakeLists.txt` file inside `src` directory contains the following commands:

file eqsolver/src/CMakeLists.txt

```
1 include_directories(.)
2
```

```
3 SET(source_code eqsolver.h quad.cpp)
4
5 add_library (eqsolver SHARED ${source_code})
6
7 install (TARGETS eqsolver DESTINATION "${PROJECT_BINARY_DIR}/lib")
8 install (FILES eqsolver.h DESTINATION "${PROJECT_BINARY_DIR}/include")
```

At the first line, we state that this directory should be searchable for include files, during the project compilation. Then, we create a *set* that we name as `source_code`. This set contains our source files. At line 5, we state that there is a *target* in this directory, which is named `eqsolver`. This target is a *shared* library (`dll` for windows, `dylib` for Mac OS X, and a `so` for unix/linux), and depends on the files contained in the `source_code`. In other words, we ask `cmake`, to build a shared library based on `eqsolver.h` and `quad.cpp`. After the build process is completed, we state that the library executable should be copied to a directory named `lib` and that the header `eqsolver.h` should be copied to a directory named `include`. These directories will be created in the folder that is the *build* folder. A *build* folder can be any directory. The `PROJECT_BINARY_DIR` denotes the build folder. We leave empty our last `CMakeLists.txt` for now.

The `eqsolver.h` file contains the declaration of our function, `solve_quadratic_eq` plus one other stuff that should be familiar to you, regarding header guards, symbol exporting from libraries (`DLL_PUBLIC`) and compilation and linking to `C/C++` applications.

Listing 1: eqsolver.h

```
1  #ifndef eqsolver_h
2  #define eqsolver_h
3
4  #if defined _WIN32 || defined __CYGWIN__
5          #ifdef eqsolver_EXPORTS // define this when generating DLL
6                  #ifdef __GNUC__
7                          #define DLL_PUBLIC __attribute__((dllexport))
8                  #else
9                          #define DLL_PUBLIC __declspec(dllexport)
10                 #endif
11         #else
12                 #ifdef __GNUC__
13                         #define DLL_PUBLIC __attribute__((dllimport))
14                 #else
15                         #define DLL_PUBLIC __declspec(dllimport)
16                 #endif
```

```
17          #endif
18          #define DLL_HIDDEN
19  #else
20          #if __GNUC__ >= 4
21                  #define DLL_PUBLIC __attribute__ ((visibility("default")))
22                  #define DLL_HIDDEN __attribute__ ((visibility("hidden")))
23          #else
24                  #define DLL_PUBLIC
25                  #define DLL_HIDDEN
26          #endif
27  #endif
28
29  #ifdef __cplusplus
30  extern "C" {
31  #endif
32
33  DLL_PUBLIC int solve_quadratic_eq(const double (*coeffs)[3], double (*roots)[4]);
34
35  #ifdef __cplusplus
36  }
37  #endif
38
39  #endif /* eqsolver.h */
```

The file `quad.cpp`[4] contains a simple implementation of our library's function, plus some helper functions that we won't export to the user:

Listing 2: quad.cpp

```
1  #include "eqsolver.h"
2  #include <cmath>
3
4  const double zero_tol = 1.0e-06;
5
6  static double discriminant(const double (*coeffs)[3])
7  {
8          return (*coeffs)[1] * (*coeffs)[1] - 4.0 * (*coeffs)[0] * (*coeffs)[2];
9  }
10
11 int solve_quadratic_eq(const double (*coeffs)[3], double (*roots)[4])
12 {
13         // find discriminant and then calculate roots
```

[4]The currently implemented solution suffers from numerical problems. Check the wikipedia article for more information

```
14          double d = discriminant(coeffs);
15
16          int retCode = 0;
17          if (d > zero_tol) // two real roots
18          {
19                  double d1 = (-(*coeffs)[1]+sqrt(d))/(2.0* (*coeffs )[0]);
20                  double d2 = (-(*coeffs)[1]-sqrt(d))/(2.0* (*coeffs )[0]);
21                  *roots[0] = d1;
22                  *roots[1] = d2;
23          } else if (d < -zero_tol) // complex roots
24          {
25                  double r = (-(*coeffs)[1])/(2.0*(* coeffs )[0]);
26                  double w = (sqrt(d)/(2.0*(*coeffs )[0]));
27                  *roots[0] = r; // first root is x1 = r + iw,
28                  *roots[1] = w; // real part is roots[0], imaginary is roots[1]]
29                  *roots[2] = r; // second root is x2 = r - iw,
30                  *roots[3] = -w; //real part is roots[2], imaginary is roots[2]
31                  retCode = 1;
32          } else // two identical real roots
33          {
34                  double r = (-(*coeffs)[1])/(2.0*(* coeffs )[0]);
35                  *roots[0] = r;
36                  *roots[1] = r;
37          }
38          return retCode;
39  }
```

5 Building our project

Now we can try building our project. To do that, create a new directory, named `build` inside the main directory `eqsolver`. Change to the build directory, and type `cmake ../` to build the project. `cmake` will run a bunch of tests to check what compilers are available. If no error was encountered, you will notice that lots of files and directories have been created inside the build directory. If you are on a GNU/Linux or UNIX box, you will also notice a `Makefile`. Type `make` to build the project. If no errors were encountered (check your source files if errors were encountered), you will notified that the project was built. In my computer, the output from the `make` command is looks like the following:

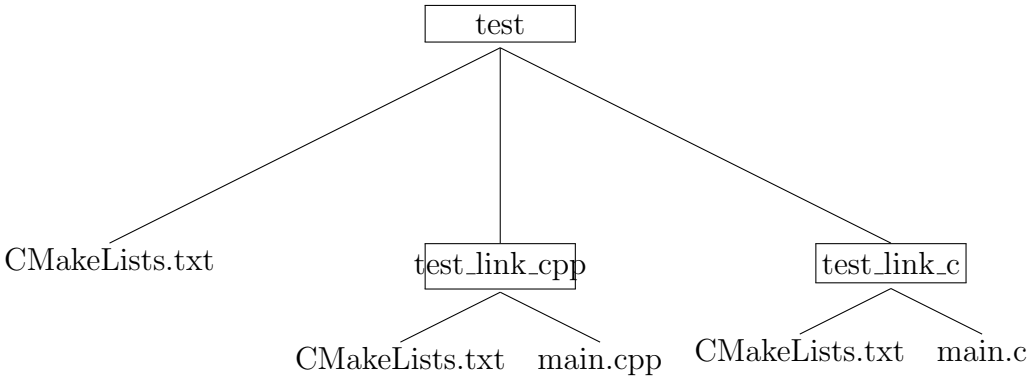
```
Scanning dependencies of target eqsolver
[100%] Building CXX object src/CMakeFiles/eqsolver.dir/quad.cpp.o
```

```
Linking CXX shared library libeqsolver.dylib
[100%] Built target eqsolver
```

After our project is build, we can *install* it. Type `make install`. You will notice that two folders were created inside the *build* directory, one named *include*, and one named *lib*. Check the contents of these directories. They should contain the headed `eqsolver.h` and the our dynamic library. You can now give these two folders with their contents to someone else to use your library.

6 Adding tests

To be sure that our library works as expected, we should create a couple of tests to test the functionality. First of all, we must check that our library can be used from `C` and `C++` code. Create two directories inside the `test` directory, named `test_link_cpp` and `test_link_c`. Add a `CMakeLists.txt` file in each one. Add a source file named `main.cpp` in `test_link_cpp` and a source file named `main.c` in `test_link_c`. The `test` directory should look like this:



Let's add the necessary commands to our new `CMakeLists.txt` files. In the `CMakeLists.txt` inside the `test` directory, add the following commands:

file eqsolver/test/CMakeLists.txt

```
1 include_directories(../src)
2
3 add_subdirectory (test_link_cpp)
4 add_subdirectory (test_link_c)
5
```

```
6 add_test (test1 ${CMAKE_BINARY_DIR}/test/test_link_cpp/test1)
7 add_test (test2 ${CMAKE_BINARY_DIR}/test/test_link_c/test2)
8 # we can add more tests here
9
```

The first three commands should be familiar to you. We are stating that during all test compilation, the compiler should also check our `src` directory for include files. Then, we add our test directories (lines 3 and 4). The interesting lines are 6 and 7, where we *create* two tests, one named `test1` and one named `test2`. The first test, `test1`, will run the executable named `test1` inside the `test/test_link_cpp` directory, and the second test, `test2`, will run the executable `test2` inside the `test/test_link_c` directory.

Edit the `CMakeLists.txt` inside the `test_link_cpp` directory, and add the following commands:

file eqsolver/test/test_link_cpp/CMakeLists.txt

```
1 set (source_code main.cpp)
2
3 add_executable (test1 {source_code})
4
5 target_link_libraries (test1 eqsolver)
6
```

At line 3, we ask the build system to create an executable named `test1` that depends on `main.cpp`. The other interesting command here, is the one at line 5, where we ask the build system to link with our library.

Our last `CMakeLists.txt` is very similar:

file eqsolver/test/test_link_c/CMakeLists.txt

```
1 set (source_code main.c)
2
3 add_executable (test2 ${source_code})
4
5 target_link_libraries (test2 eqsolver)
6
```

Now we must add some code to our test source files, `main.cpp` and `main.c`. Add the following code to `main.cpp`:

Listing 3: main.cpp

```
1 #include <stdio.h>
2 #include "eqsolver.h"
```

```
3
4  int main(int argc, char *argv[])
5  {
6          typedef int (*solve_fun)(const double (*c)[3], double (*r)[4]);
7
8          solve_fun  sf  = solve_quadratic_eq;
9          return 0;
10 }
```

Add the very same code to `main.c`. Now we are ready to test if our two tests can link with our library. To check this, change to the *build* directory, and rerun the `cmake ../` command. Then build the library by typing `make`. To run the tests, type `make test`. You should see something like the following output:

```
Running tests...
Test project ../eqsolver/build
    Start 1: test1
1/2 Test #1: test1 ............................   Passed    0.00 sec
    Start 2: test2
2/2 Test #2: test2 ............................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   3.01 sec
```

7 Adding more tests

To actually test whether our library works correctly, we can create a *unit* test. In this test, we must check if the function `solve_quadratic_eq` produces the right results for various inputs. The following table contains some possible values for the coefficients a, b and c, and the respective solutions:

Table 1: test values

a	b	c	x_1	x_2
1	2	3	$-1 + 1.4142i$	$-1 - 1.4142i$
1	-2	-3	3	-1
1	-5	1	4.7913	0.20871
1	-2	1	1	1

Create a new directory named `unit_test_1` inside our `test` directory. Add this new directory to the build system, by adding the following commands (shown in red) to `test/CMakeLists.txt`:

file eqsolver/test/CMakeLists.txt

```
include_directories(../src)

add_subdirectory (test_link_cpp)
add_subdirectory (test_link_c)
add_subdirectory (unit_test_1)

add_test (test1 $CMAKE_BINARY_DIR/test/test_link_cpp/test1)
add_test (test2 $CMAKE_BINARY_DIR/test/test_link_c/test2)
add_test (test3 $CMAKE_BINARY_DIR/test/unit_test_1/test3)
set_tests_properties (test3 PROPERTIES PASS_REGULAR_EXPRESSION
"ret 1, roots -1.000000 1.414200 -1.000000 -1.414200
ret 0, roots 3.000000 -1.000000 0.000000 0.000000
ret 0, roots 4.791300 0.208710 0.000000 0.000000
ret 0, roots 1.000000 1.000000 0.000000 0.000000")

```

For the third test, we add a *property* that states that the output of the test should match the string:

```
"ret 1, roots -1.000000 1.414200 -1.000000 -1.414200
ret 0, roots 3.000000 -1.000000 0.000000 0.000000
ret 0, roots 4.791300 0.208710 0.000000 0.000000
ret 0, roots 1.000000 1.000000 0.000000 0.000000"
```

Now, create a new `CMakeLists.txt` inside the `unit_test_1`, and add the following commands:

file eqsolver/test/unit_test_1/CMakeLists.txt

```
set (source_code main.cpp)

add_executable (test3 ${source_code})

target_link_libraries (test3 eqsolver)
```

The `main.cpp` should test the function `solve_quadratic_eq` on the values of table 1. The following listing shows the code:

Listing 4: main.cpp

```
1  #include <stdio.h>
2  #include "eqsolver.h"
3
4  int main(int argc, char *argv[])
5  {
6          const int CASES = 4;
7          const double COEFFS[CASES][7] = {
8                          {1,  2,  3,  -1, 1.4142,  -1, -1.4142},
9                          {1,-2,-3,  3,  -1, 0,  0},
10                         {1,-5,  1,  4.7913,  0.20871,  0,  0},
11                         {1,-2,  1,  1,  1,  0,  0}};
12
13         for (int i = 0; i < CASES; ++i)
14         {
15                         int ret = 0;
16                         const double C[3] = {COEFFS[i][0], COEFFS[i][1], COEFFS[i][2]};
17                         double roots[4] = {0};
18                         ret = solve_quadratic_eq(&C, &roots);
19                         printf("ret %d, roots %f %f %f %f\n", ret,
20                                 COEFFS[i][3], COEFFS[i][4], COEFFS[i][5], COEFFS[i][6]);
21         }
22         return 0;
23 }
```

Now run rebuild the project by typing `cmake ../` inside the build directory, build the project by typing `make`, and finally rerun the tests, by typing `make test`. You should see no errors.

8 Exercises

To further hone your skills, try completing the following exercises:

1. Add the necessary code to test whether coefficient $a = 0$. In this case, make the function `solve_quadratic_eq` return -1. Rebuild the project, and make sure that all tests pass.

2. Expose the function `discriminant` to the library's users (add it to `eqsolver.h`).

3. Create another unit test for the function `discriminant`.

4. Build and test the project under other OSes.

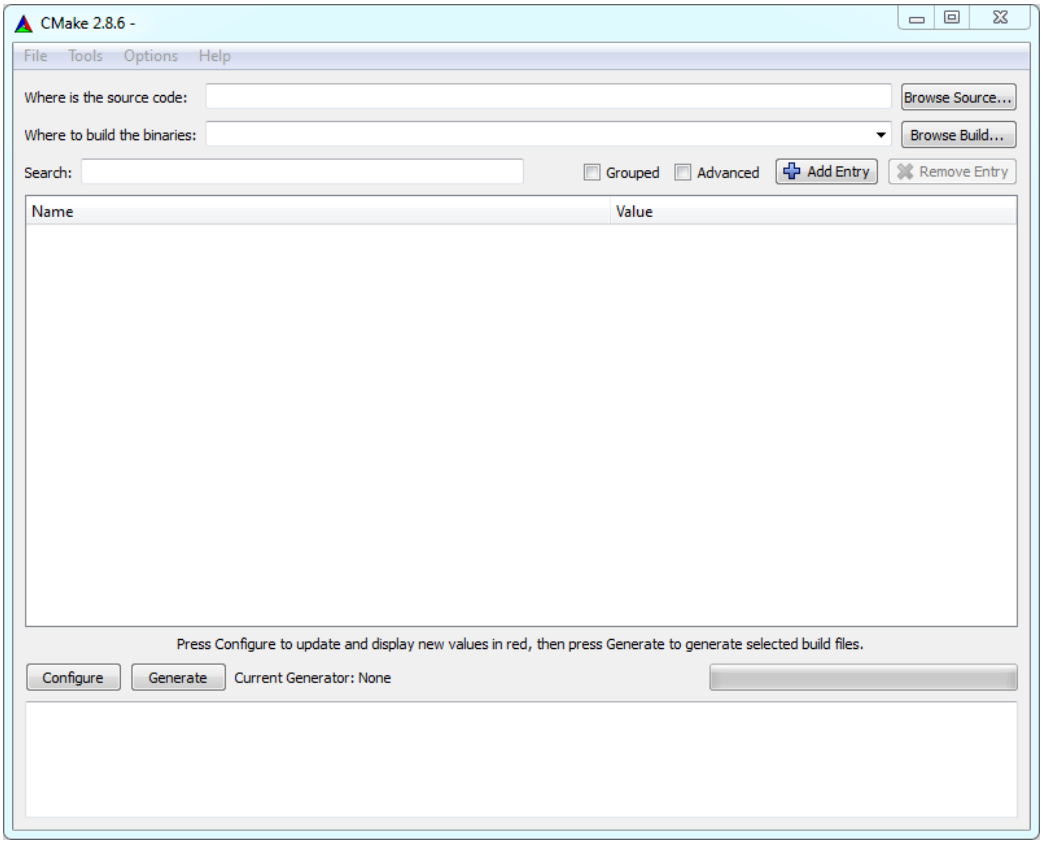


Figure 1: cmake gui

9 `cmake` for Windows

To use `cmake` under Windows, you should download the software bundle from http://cmake.org/cmake/resources/software.html. Under windows, you should use the `GUI` version of `cmake`. The main window of that version is shown on the following picture:

There are some easy steps you must take to build the project. First, add the path to the source project directory in the first line edit, with the label "Where is the source code:". Then create a *build* directory and add its path to the second line edit labeled "Where to build the binaries:". Then press the button labeled "Configure" to have `cmake` test your environment (mainly your compiler tooling). After `cmake` finished, press the button labeled "Generate", to have `cmake` generate a Visual Studio project. You can then open the project solution with Visual Studio.

10 Article revision history

The following table describes the changes to this article.

Table 2: revision history

Date	Notes
7-Nov-2011	version 1.0