



JohnLamp.net

[Home](#) [CMake Tutorial](#)[← CMake Tutorial – Chapter 3: GUI Tool](#)[CMake Tutorial – Chapter 5: Functionally
Improved Testing →](#)

CMake Tutorial – Chapter 4: Libraries and Subdirectories

Posted on [2013-05-05](#) by [John Lamp](#)

Contents

- [Introduction](#)
- [The Library in a Subdirectory](#)
- [Testing – for Real](#)

Introduction

So far our project is rather simple. A real project would be more complicated than the one we've created. Let's add subdirectories, libraries, and proper unit tests to make our project more realistic.

In this chapter we will split up our project to have a library which we can put in a subdirectory. Then we will use [Google Test](#) and [Google Mock](#) to add a more realistic unit test.

The Library in a Subdirectory

Archives

[March 2015](#)[July 2013](#)[May 2013](#)[March 2013](#)

Categories

[CMake Tutorial](#)

Meta

[Log in](#)[Entries RSS](#)[Comments RSS](#)[WordPress.org](#)

We will make the `ToDo` class its own library, and put it in a subdirectory. Even though it is a single source file making it a library actually has one significant advantage. CMake will compile source files once for each target that includes them. So if the `ToDo` class is used by our command line tool, a unit test, and perhaps a GUI App it would be compiled three times. Imagine if we had a collection of classes instead of just one. This results in a lot of unnecessary compilation.

There were some minor changes to the C++, grab the files here:  [Source](#)

(`CMakeLists.txt` listed below)

CMakeLists.txt

New or modified lines in bold.

```
1∞cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
2∞set(CMAKE_LEGACY_CYGWIN_WIN32 0)
3∞
4∞project("To Do List")
5∞
6∞enable_testing()
7∞
8∞
9∞if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "GNU" OR
10∞    "${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang")
11∞    set(warnings "-Wall -Wextra -Werror")
12∞elseif ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
13∞    set(warnings "/W4 /WX /EHsc")
14∞endif()
15∞if (NOT CONFIGURED_ONCE)
16∞    set(CMAKE_CXX_FLAGS "${warnings}")
17∞        CACHE STRING "Flags used by the compiler during all build types." FORCE)
18∞    set(CMAKE_C_FLAGS "${warnings}")
19∞        CACHE STRING "Flags used by the compiler during all build types." FORCE)
20∞endif()
21∞
22∞
```

```
2300include_directories(${CMAKE_CURRENT_SOURCE_DIR})
2400
2500add_subdirectory(ToDoCore)
2600
2700add_executable(toDo main.cc)
2800target_link_libraries(toDo ToDoCore)
2900
3000add_test(toDoTest toDo)
3100
3200
3300set(CONFIGURED_ONCE TRUE CACHE INTERNAL
3400    "A flag showing that CMake has configured at least once.")
```

So now our executable “toDo” only depends on the file “main.cc” and the new library “ToDoCore”. Our project also has a new subdirectory “ToDoCore”.

`include_directories(directories)`

Add directories to the end of this directory’s include paths. We didn’t need this before because all of our files were in the same directory.

[include_directories\(\) documentation](#) (2013-04-20)

`CMAKE_CURRENT_SOURCE_DIR`

The full path to the source directory that CMake is currently processing.

[CMAKE_CURRENT_SOURCE_DIR documentation](#) (2013-04-20)

`add_subdirectory(source_dir)`

Include the directory `source_dir` in your project. This directory *must* contain a `CMakeLists.txt` file.

Note: We’re omitting the optional second parameter. This only works with subdirectories of the current directory. We will see how to add external directories later.

[add_subdirectory documentation](#) (2013-04-20)

`target_link_libraries(target library...)`

Specify that target needs to be linked against one or more libraries. If a library name matches another target dependencies are setup automatically so that the libraries will be built first and target will be updated whenever any of the libraries are.

If the target is an executable then it will be linked against the listed libraries.

If the target is a library then its dependency on these libraries will be recorded. Then when something else links against target it will also link against target's dependencies. This makes it much easier to handle a library's dependencies since you only have to define them once when you define library itself.

For the moment we are using the simplest form of this command. For more information see the [documentation](#) (2013-04-20).

When describing `add_subdirectory()` I stated that the subdirectory must contain a `CMakeLists.txt` file. So here's the new file.

ToDoCore/CMakeLists.txt

```
1 add_library(toDoCore ToDo.cc)
```

Conveniently this file is rather simple.

```
add_library(target [STATIC | SHARED | MODULE] sources...)
```

This command creates a new library target built from sources. As you may have noticed this command is very similar to `add_executable`.

With `STATIC`, `SHARED`, and `MODULE` you can specify what kind of library to build. `STATIC` libraries are archives of object files that are linked directly into other targets. `SHARED` libraries are linked dynamically and loaded at runtime. `MODULE` libraries are plug-ins that aren't linked against but can be loaded dynamically at runtime.

If the library type is not specified it will be either `STATIC` or `SHARED`. The default type is controlled by

the [BUILD_SHARED_LIBS](#) variable. By default static libraries are created.

[add_library\(\) documentation](#) (2013-04-20)

Testing – for Real

We have a rudimentary test but if we were really developing software we'd write a real test using a real testing framework. As mentioned earlier we will use [Google Test 1.6.0](#) and [Google Mock 1.6.0](#) . Conveniently they include their own CMakeLists.txt files, which makes them easy for us to use.

First the test:

ToDoCore/unit_test/ToDoTest.cc

```
1  #include "ToDoCore/ToDo.h"
2
3  #include <string>
4  using std::string;
5
6  #include <gmock/gmock.h>
7  using ::testing::Eq;
8  #include <gtest/gtest.h>
9  using ::testing::Test;
10
11
12 namespace ToDoCore
13 {
14     namespace testing
15     {
16         class ToDoTest : public Test
17         {
18         protected:
19             ToDoTest() {}
20             ~ToDoTest() {}
```

```
21∞
22∞     virtual void SetUp(){}
23∞     virtual void TearDown(){}
24∞
25∞
26∞     ToDo list;
27∞
28∞     static const size_t taskCount = 3;
29∞     static const string tasks[taskCount];
30∞ };
31∞
32∞     const string ToDoTest::tasks[taskCount] = {"write code",
33∞                                                 "compile",
34∞                                                 "test"};
35∞
36∞
37∞     TEST_F(ToDoTest, constructor_createsEmptyList)
38∞     {
39∞         EXPECT_THAT(list.size(), Eq(size_t(0)));
40∞     }
41∞
42∞     TEST_F(ToDoTest, addTask_threeTimes_sizeIsThree)
43∞     {
44∞         list.addTask(tasks[0]);
45∞         list.addTask(tasks[1]);
46∞         list.addTask(tasks[2]);
47∞
48∞         EXPECT_THAT(list.size(), Eq(taskCount));
49∞     }
50∞
51∞     TEST_F(ToDoTest, getTask_withOneTask_returnsCorrectString)
52∞     {
53∞         list.addTask(tasks[0]);
54∞
55∞         ASSERT_THAT(list.size(), Eq(size_t(1)));
```

```

56∞     EXPECT_THAT(list.getTask(0), Eq(tasks[0]));
57∞ }
58∞
59∞ TEST_F(ToDoTest, getTask_withThreeTasts_returnsCorrectStringForEachIndex)
60∞ {
61∞     list.addTask(tasks[0]);
62∞     list.addTask(tasks[1]);
63∞     list.addTask(tasks[2]);
64∞
65∞     ASSERT_THAT(list.size(),      Eq(taskCount));
66∞     EXPECT_THAT(list.getTask(0), Eq(tasks[0]));
67∞     EXPECT_THAT(list.getTask(1), Eq(tasks[1]));
68∞     EXPECT_THAT(list.getTask(2), Eq(tasks[2]));
69∞ }
70∞
71∞} // namespace testing
72∞} // namespace ToDoCore

```

This is a rather simple test, but `ToDo` is still a rather simple class. It may look strange if you are unfamiliar with Google Test, taking a look at [Google Test Primer](#) may be helpful. I also use a little functionality from Google Mock so [Google Mock for Dummies](#) may also be useful.

Now we need to build the test:

ToDoCore/CMakeLists.txt

New or modified lines in bold.

```

1∞add_library(ToDoCore ToDo.cc)
2∞
3∞add_subdirectory(unit_test)

```

ToDoCore/unit_test/CMakeLists.txt

```

1∞set(GMOCK_DIR "../..../gmock/gmock-1.6.0"
2∞    CACHE PATH "The path to the GoogleMock test framework.")

```

```
3∞
4∞if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
5∞    # force this option to ON so that Google Test will use /MD instead of /MT
6∞    # /MD is now the default for Visual Studio, so it should be our default, too
7∞    option(gtest_force_shared_crt
8∞        "Use shared (DLL) run-time lib even when Google Test is built as static
9∞        ON)
10∞elseif (APPLE)
11∞    add_definitions(-DGTEST_USE_OWN_TR1_TUPLE=1)
12∞endif()
13∞add_subdirectory(${GMOCK_DIR} ${CMAKE_BINARY_DIR}/gmock)
14∞
15∞include_directories(SYSTEM ${GMOCK_DIR}/gtest/include
16∞                    ${GMOCK_DIR}/include)
17∞
18∞
19∞add_executable(ToDoTest ToDoTest.cc)
20∞target_link_libraries(ToDoTest toDoCore
21∞                    gmock_main)
22∞
23∞add_test(ToDoTest ToDoTest)
```

First we add the Google Mock directory to our project then we add our test. The path to Google Mock is stored in a cached variable so that you can easily set it to the correct value either from the command line or via one of the GUIs. There are several potential problems with that line but we will worry about those later, for now it's good enough. Okay I oversimplified a little. We don't just add the Google Mock directory, we also work around some OS-specific problems.

When using Visual Studio to build our test we would run into a problem. Even when building static libraries, CMake's default, MSVC defaults to linking against the multi-threaded, DLL-specific version of the standard library. By default Google Test overrides this so that the non-DLL version of the multi-threaded standard library is used. Then when our test links against both `toDoCore` and `gmock_main` the

linker will output a large number of errors since we would be linking against two different copies of the standard library. To avoid this problem we force Google Test to use the DLL-specific version to match Visual Studio's default by setting the `gtest_force_shared_crt` option to ON. See [Microsoft C/C++ Compiler Run-Time Library](#) .

The second problem occurs on newer version of Mac OS X which default to using a different standard library that fully supports C++11. GTest uses the `tuple` class from the draft TR1 standard and therefore looks for it in the `std::tr1` namespace. The `tr1` namespace is not present in the C++11 standard library that Apple uses so GTest cannot find it and won't compile. We fix this by telling GTest to use its own `tuple` implementation.

```
add_subdirectory(source_dir [binary_dir])
```

Add the directory `source_dir` to the current project with `binary_dir` as its corresponding binary output directory. When adding a directory that is a subdirectory of the current directory CMake will automatically determine what the binary output directory should be, making the second argument optional. However if you add a directory that isn't a subdirectory you need to specify the binary output directory.

[add_subdirectory documentation](#) (2013-04-20)

```
CMAKE_BINARY_DIR
```

This variable holds the path to the top level binary output directory, i.e. the directory in which you ran the `cmake` command or the path you chose for "Where to build the binaries" in the GUI.

[CMAKE_BINARY_DIR documentation](#) (2013-04-27)

```
include_directories([AFTER|BEFORE] [SYSTEM] directory...)
```

```
AFTER|BEFORE
```

Specify whether or not these include directories should be appended or prepended to the list of include directories. If omitted then the default behavior is used.

By default directories are appended to the list. This behavior can be changed by setting

```
CMAKE_INCLUDE_DIRECTORIES_BEFORE to TRUE.
```

SYSTEM

Specify that these directories are system include directories. This only has an affect on compilers that support the distinction. This can change the order in which the compiler searches include directories or the handling of warnings from headers found in these directories.

directory...

The directories to be added to the list of include directories.

[include_directories\(\) documentation](#) (2013-04-20)

`option(name docstring [initialValue])`

Provide a boolean option to the user. This will be displayed in the GUI as a checkbox. Once created the value of the option can be accessed as the variable `name`. The `docstring` will be displayed in the GUI to tell the user what this option does. If no initial value is provided it defaults to OFF.

While this boolean option is stored in the cache and accessible as a variable you cannot override the `initialValue` by setting a variable of the same name beforehand, not even by passing a `-D` command line option to CMake. Which is why we have to define the option ourselves before Google Test does.

[option\(\) documentation](#) (2013-05-3)

`add_definitions(flags...)`

Add preprocessor definitions to the compiler command line for targets in the current directory and those below it. While this command is intended for adding definitions you still need to precede them with `-D`.

Because this command modifies the `COMPILE_DEFINITIONS` directory property it affects *all* targets in the directory, even those that were defined **before** this command was used. If this is not the desired effect then modifying the `COMPILE_DEFINITIONS` property of particular targets or source files will work better. (Properties are introduced below.)

[add_definitions\(\) documentation](#) (2014-09-28)

[COMPILE_DEFINITIONS directory property documentation](#) (2014-09-28)

[COMPILE DEFINITIONS target property documentation](#) (2014-09-28)

[COMPILE DEFINITIONS source file property documentation](#) (2014-09-28)

Let's go ahead and try out our new test!

 [Source](#)

```
> mkdir build
> cd build
> cmake -G "Unix Makefiles" ..
-- The C compiler identification is Clang 4.2.0
-- The CXX compiler identification is Clang 4.2.0
-- 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
-- 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
-- Found PythonInterp: /usr/local/bin/python (found version "2.7.3")
-- Looking for include file pthread.h
-- Looking for include file pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /Documents/Programming/C++/CMake Tutorial/flavors
> make

Scanning dependencies of target toDoCore
[ 14%] Building CXX object ToDoCore/CMakeFiles/toDoCore.dir/ToDo.cc.o
Linking CXX static library libToDoCore.a
[ 14%] Built target toDoCore
Scanning dependencies of target toDo
```

```
[ 28%] Building CXX object CMakeFiles/ToDo.dir/main.cc.o
Linking CXX executable ToDo
[ 28%] Built target ToDo
Scanning dependencies of target gtest
[ 42%] Building CXX object gmock/gtest/CMakeFiles/gtest.dir/src/gtest-all.cc.o
In file included from /Documents/Programming/C++/gmock/gmock-1.6.0/gtest/src/gtest-all.cc:1:
In file included from /Documents/Programming/C++/gmock/gmock-1.6.0/gtest/src/gtest.cc:1:
/Documents/Programming/C++/gmock/gmock-1.6.0/gtest/src/gtest-internal-inl.h:206:8: error:
    private field 'pretty_' is not used [-Werror,-Wunused-private-field]
    bool pretty_;
        ^
1 error generated.
make[2]: *** [gmock/gtest/CMakeFiles/gtest.dir/src/gtest-all.cc.o] Error 1
make[1]: *** [gmock/gtest/CMakeFiles/gtest.dir/all] Error 2
make: *** [all] Error 2
```

Oh noes! Newer versions of Clang have some pretty strict warnings and we have just run afoul of one. So we have a problem: we want to use strict compiler settings to ensure we write good code but we also don't want to go changing Google Test. As it turns out CMake actually provides us the flexibility we need to disable warnings for just the gtest target.

This is a capability that can easily be abused. In the case of Google Test we didn't write it and we know, or at least assume, that it works fine. Because of that we don't care about any warnings we might find in Google Test's code. We need to be careful not to use this feature to allow ourselves to write poor code.

ToDoCore/unit_test/CMakeLists.txt

New or modified lines in bold.

```
1 set(GMOCK_DIR ".../.../.../.../gmock/gmock-1.6.0"
2   CACHE PATH "The path to the GoogleMock test framework.")
3
4 if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
5   # force this option to ON so that Google Test will use /MD instead of /MT
6   # /MD is now the default for Visual Studio, so it should be our default, too
```

```

7  option(gtest_force_shared_crt
8      "Use shared (DLL) run-time lib even when Google Test is built as static
9      ON)
10 elseif (APPLE)
11     add_definitions(-DGTEST_USE_OWN_TR1_TUPLE=1)
12 endif()
13 add_subdirectory(${GMOCK_DIR} ${CMAKE_BINARY_DIR}/gmock)
14 set_property(TARGET gtest APPEND_STRING PROPERTY COMPILE_FLAGS " -w")
15
16 include_directories(SYSTEM ${GMOCK_DIR}/gtest/include
17                      ${GMOCK_DIR}/include)
18
19
20 add_executable(ToDoTest ToDoTest.cc)
21 target_link_libraries(ToDoTest toDoCore
22                      gmock_main)
23
24 add_test(ToDoTest ToDoTest)

```

```
set_property(TARGET gtest APPEND_STRING PROPERTY COMPILE_FLAGS " -w")
```

There are a variety of things that have properties in CMake, in this case we are interested in a target's properties. Each target can have it's own compiler flags in addition the ones set in

CMAKE_<LANG>_FLAGS. Here we append " -w" to gtest's COMPILE_FLAGS. The flag "-w" disables all warnings for both GCC and Clang. When compiling with MSVC the "-w" will be automatically converted to "/w" which has the same function. (Although it will warn that "/w" is overriding "/W4")

[COMPILE_FLAGS documentation](#) (2013-04-28)

[GCC Warning Options](#) (2013-04-28), currently these work for Clang too.

[Microsoft C/C++ Compiler Warning Level](#) (2013-04-28)

```
set_property(TARGET target_name... [APPEND|APPEND_STRING] PROPERTY name value...)
TARGET
```

Specify that we want to set the property of a target. Several other types of things have properties you can set. For the moment we are only going to deal with targets, but the concept is the same for the rest.

target_name...

The name of the target whose property you want to set. You can list multiple targets and all will have the property set the same way for each.

[APPEND | APPEND_STRING]

Append to the property's existing value instead of setting it. APPEND appends to the property as a list. APPEND_STRING appends to the property as a string.

Note: Do not provide a multiple values when using APPEND_STRING as the results will not be what you expect.

Don't worry about lists we will cover them in the next [chapter](#).

PROPERTY

name

The name of the property you want to set. See [Properties on Targets](#) .

value...

The value to set for the property. If multiple values are provided they are treated as a list. Only provide one value if also using APPEND_STRING.

Don't worry about [lists](#) yet.

[set_property\(\) documentation](#) (2013-04-28)

Let's give this version a try.

 [Source](#)

```
> mkdir build
> cd build
> cmake -G "Unix Makefiles" ..
-- The C compiler identification is Clang 4.2.0
```

```
-- The CXX compiler identification is Clang 4.2.0
-- 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
-- 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
-- Found PythonInterp: /usr/local/bin/python (found version "2.7.3")
-- Looking for include file pthread.h
-- Looking for include file pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /Documents/Programming/C++/CMake Tutorial/flavors
> make
Scanning dependencies of target toDoCore
[ 14%] Building CXX object CMakeFiles/toDoCore.dir/ToDo.cc.o
Linking CXX static library libtoDoCore.a
[ 14%] Built target toDoCore
Scanning dependencies of target toDo
[ 28%] Building CXX object CMakeFiles/toDo.dir/main.cc.o
Linking CXX executable toDo
[ 28%] Built target toDo
Scanning dependencies of target gtest
[ 42%] Building CXX object gmock/gtest/CMakeFiles/gtest.dir/src/gtest-all.cc.o
Linking CXX static library libgtest.a
[ 42%] Built target gtest
Scanning dependencies of target gmock
[ 57%] Building CXX object gmock/CMakeFiles/gmock.dir/src/gmock-all.cc.o
Linking CXX static library libgmock.a
[ 57%] Built target gmock
```

Scanning dependencies of target gmock_main

[71%] Building CXX object gmock/CMakeFiles/gmock_main.dir/src/gmock_main.cc.o

Linking CXX static library libgmock_main.a

[71%] Built target gmock_main

Scanning dependencies of target ToDoTest

[85%] Building CXX object ToDoCore/unit_test/CMakeFiles/ToDoTest.dir/ToDoTest.cc.o

Linking CXX executable ToDoTest

[85%] Built target ToDoTest

Scanning dependencies of target gtest_main

[100%] Building CXX object gmock/gtest/CMakeFiles/gtest_main.dir/src/gtest_main.cc.o

Linking CXX static library libgtest_main.a

[100%] Built target gtest_main

> make test

Running tests...

Test project /Documents/Programming/C++/CMake Tutorial/flavors/part4_step3/build

Start 1: ToDoTest

1/1 Test #1: ToDoTest Passed 0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) = 0.01 sec

> ToDoCore/unit_test/ToDoTest

Running main() from gmock_main.cc

[=====] Running 4 tests from 1 test case.

[-----] Global test environment set-up.

[-----] 4 tests from ToDoTest

[RUN] ToDoTest.construction_createsEmptyList

[OK] ToDoTest.construction_createsEmptyList (0 ms)

[RUN] ToDoTest.addTask_threeTimes_sizeIsThree

[OK] ToDoTest.addTask_threeTimes_sizeIsThree (0 ms)

[RUN] ToDoTest.getTask_withOneTask_returnsCorrectString

[OK] ToDoTest.getTask_withOneTask_returnsCorrectString (0 ms)

[RUN] ToDoTest.getTask_withThreeTasts_returnsCorrectStringForEachIndex

[OK] ToDoTest.getTask_withThreeTasts_returnsCorrectStringForEachIndex (1 ms)

[-----] 4 tests from ToDoTest (1 ms total)

[-----] Global test environment tear-down

[=====] 4 tests from 1 test case ran. (1 ms total)


```
[ PASSED ] 4 tests.
```

Yay! Everything works now and our test passes, too.


Next we will focus on how we could add more unit tests (if we had more units) without duplicating the work we've done here. Also we will make it so that our unit tests are automatically run as needed whenever we build.

Revision History

Version	Date	Comment
1	2013-05-05	Original version.
2	2013-07-14	Added line numbers and indication of changes to code samples. Added a link to the section on lists.
3	2014-10-01	Added the work around for a problem with Google Test and newer versions of Mac OS X along with an explanation and a description of <code>add_definitions()</code>

This entry was tagged [CMake](#), [long](#), [tutorial](#). Bookmark the [permalink](#).

 This entry, "CMake Tutorial – Chapter 4: Libraries and Subdirectories," by John Lamp is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](#).

 To the extent possible under law, John Lamp has waived all copyright and related or neighboring rights to the code samples in this entry, "CMake Tutorial – Chapter 4: Libraries and Subdirectories".

[← CMake Tutorial – Chapter 3: GUI Tool](#)

[CMake Tutorial – Chapter 5: Functionally Improved Testing →](#)

14 thoughts on “CMake Tutorial – Chapter 4: Libraries and Subdirectories”

[2014-09-09 at 05:15:58](#)



[Adam Getchell](#) said:

This is a great tutorial. Unfortunately, gmock 1.6 doesn't build on my Mac.