← CMake Tutorial – Chapter 4: Libraries and Subdirectories

CMake Tutorial – Chapter 6: Realistically Getting a Boost →

CMake Tutorial – Chapter 5: Functionally Improved Testing

Posted on 2013-07-14 by John Lamp

Contents

Archives

March 2015
July 2013
May 2013
March 2013

Categories

CMake Tutorial

Meta

Log in
Entries RSS
Comments RSS
WordPress.org

Introduction

Last time we added a nice unit test and then set up CMake to build it, of course, and add it to the list of tests that CTest will run. This is great, now we can run `cmake` then use `make` and `make test` to test our project. Now it's time to build on our success because we certainly aren't done yet.

The main problem we need to tackle is that there are currently 3 steps to creating a test program:

1. add the executable target
2. link the executable against the "gmock_main" library
3. add the test to CTest's list of tests

That's 3 steps too many. If you are thinking that 3 steps aren't too many remember that any project of a useful size will have a rather large number of unit tests, each of which will require these same 3 steps – that's a lot of repetition. As programmers we should not repeat ourselves, and we shouldn't slack off just because we are ~~merely~~ setting up our build system. What we want is the ability to add a new test in a single step. Writing the test is hard enough, building and running it should be easy.

Lucky for us CMake offers the ability to write functions. So we will start by writing a function that combines these 3 steps so that only one step will be needed. Once we have the function we will improve it further taking advantage of the fact that we will only have to write said improvements once.

A Simple Function

We have 3 simple steps to encapsulate in a function, that should be simple, right?

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∞
```

```
if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
    # force this option to ON so that Google Test will use /MD instead of /MT
    # /MD is now the default for Visual Studio, so it should be our default, too
    option(gtest_force_shared_crt
           "Use shared (DLL) run-time lib even when Google Test is built as static
           ON)
elseif (APPLE)
    add_definitions(-DGTEST_USE_OWN_TR1_TUPLE=1)
endif()
add_subdirectory(${GMOCK_DIR} ${CMAKE_BINARY_DIR}/gmock)
set_property(TARGET gtest APPEND_STRING PROPERTY COMPILE_FLAGS " -w")

include_directories(SYSTEM ${GMOCK_DIR}/gtest/include
                           ${GMOCK_DIR}/include)


#
# add_gmock_test(<target> <sources>...)
#
#  Adds a Google Mock based test executable, <target>, built from <sources> and
#  adds the test so that CTest will run it. Both the executable and the test
#  will be named <target>.
#
function(add_gmock_test target)
    add_executable(${target} ${ARGN})
    target_link_libraries(${target} gmock_main)

    add_test(${target} ${target})
endfunction()


add_gmock_test(ToDoTest ToDoTest.cc)
target_link_libraries(ToDoTest toDoCore)
```

Source

I like to put comments before my functions that show how they should be called and explain what they do.

`function(add_gmock_test target)`

Start the definition of the function `add_gmock_test` with one required parameter `target`.

Inside the function its first argument is available as the variable `target` and the rest of the arguments are available in a list stored in the variable `ARGN`. CMake will allow you to pass more arguments to a function than the number of parameters it defined. It is up to the writer of the function to handle all of them, validate them and produce an error if they aren't correct, or merely ignore them. In this case we are just passing them all on to the command `add_executable()`.

Also available is the variable `ARGC` which holds the count of all arguments passed to the function, both ones matching parameters and any extras. Additionally each argument can be accessed via the variables `ARGV0`, `ARGV1`, … `ARGVN`. As if that weren't enough ways to access function arguments all arguments are also available as a list stored in the variable `ARGV`. This affords a lot of flexibility but can make argument validation and handling difficult.

`function()` documentation (2013-06-01)

`endfunction()`

Ends the definition of a function. As I've said before CMake's syntax is a bit strange. You can pass the name of the function as an argument to this command, but it is not required. If you do it should match otherwise CMake will print a warning when configuring. I think it's easier to read if no arguments are passed to `endfunction()` and functions shouldn't be long enough that a reminder of what function is being ended is needed.

`endfunction()` documentation (2013-06-01)

`add_gmock_test(ToDoTest ToDoTest.cc)`

Now we use the function we just wrote to add our Google Mock based test. With the function written it is now much simpler as we don't need to write out the three separate commands every time.

```
target_link_libraries(ToDoTest toDoCore)
```

We still have to link our test with the “toDoCore” library. Since this is specific to this test and not all tests it wouldn’t make sense to include this in our function.

Commands and Functions and Macros! Oh my!

So far we have seen several CMake commands and now even written a function! You may wonder what the difference is between a command and a function. Simply put commands are built into CMake and functions are written using CMake’s language. While some commands behave quite similarly to functions, e.g. `add_executable`, some others behave in ways that cannot be mimicked using functions or macros, e.g. `if()` and `function()`.

Macros, on the other hand, are similar to functions in that they are written the same and offer all of the same ways for accessing arguments. However, macros don’t have their own scope and rather than dereferencing arguments when run arguments are replaced instead. The first difference is what makes macros both useful and dangerous, the second is more subtle and can make working with lists difficult. (Yes, I know. I haven’t talked about lists yet.)

You can’t add commands, but you can create functions and macros. As a rule of thumb do not use a macro unless absolutely necessary, then you will avoid many problems.

Scope

Scope is interesting in CMake and can occasionally be confusing. There’s local scope, directory scope, global scope, and cache scope. As with most languages things are inherited from enclosing scopes. For example if you were to set `someVariable` to “some value” and then call `someFunction()` inside the function dereferencing `someVariable` would yield “some value”.

Local Scope

This refers to the most narrow scope at a given location. So the current function or directory if not inside a function. Note that conditionals, loops, and macros do not create a new scope, which is

important to remember. When you set a variable this is the scope that is affected.

Parent Scope

The scope enclosing the current local scope. For example the scope that called the current function or the directory that executed the most recent `add_subdirectory()` command. This is important because the `set()` command can be used to set variables in the parent scope. In fact this is the only way to return values from a function.

```
set(variable values… PARENT_SCOPE)
```

`set()` documentation (2013-06-01)

Directory Scope

This is the scope of the current directory being processed by CMake which is used by directory properties and macros. The confusing thing is that some commands affect directory properties, such as `add_definitions()` and `remove_definitions()`. Many of these properties affect the targets created within this directory scope but only take effect when generating. So if you create a target and then use the `add_definitions()` command those definitions will apply to the target created previously. It is less confusing if things that affect directory scope are done before creating any targets in that directory. Also do not mix setting directory properties and creating targets inside a function, either use separate functions or set the corresponding target property.

Global Scope

As expected anything defined with global scope is accessible from within any local scope. Targets, functions, and global properties all have global scope. For this reason all targets must have unique names. (Strictly speaking this isn't true, however not all generators can handle multiple targets with the same name. For maximum compatibility it is best to ensure all targets have unique names.) Functions, on the other hand, can be redefined at will, but that is generally not a good idea.

Cache Scope

This is similar to global scope, however only variables can be stored in the cache. In addition the cache persists between CMake configure runs. As we have already seen some cached variables can also be edited using the CMake GUI or the `ccmake` tool.

Let's Include Some Organization

There's two issues with what we have now. First we've combined settings and functions for unit testing as well as an actual target. Second burying the inclusion of Google Mock this deep in our project makes it difficult to use a relative path. If you were to set the path to Google Mock on the command line using `cmake -DGMOCK_DIR=somePath` you would expect the path to be relative to the top project directory rather than two directories deeper. We can fix both of these problems at the same time.

We will refactor the code related to Google Mock into a separate file. Which will resolve problem one. Then we will include our new file from the top `CMakeLists.txt` file, which will address problem two. The question is where to put this new file and what to call it? In CMake files like these are called modules. Cmake comes with many which are stored in a directory called "Modules". Many software projects, on the other hand, store CMake related code in a directory called "cmake", a logical name, sometimes this is done out of necessity (e.g. if using ClearCase). I think we shall put the file in `cmake/Modules`. As for the name since we consistently used `gmock` or `GMOCK` let's go with `gmock.cmake`.

cmake/Modules/gmock.cmake

```
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∞                          ${GMOCK_DIR}/include)
18∞
19∞
20∞#
21∞# add_gmock_test(<target> <sources>...)
22∞#
23∞#  Adds a Google Mock based test executable, <target>, built from <sources> and
24∞#  adds the test so that CTest will run it. Both the executable and the test
25∞#  will be named <target>.
26∞#
27∞function(add_gmock_test target)
28∞    add_executable(${target} ${ARGN})
29∞    target_link_libraries(${target} gmock_main)
30∞
31∞    add_test(${target} ${target})
32∞
33∞endfunction()
```

If you look closely the only change to this code you'll notice is that the default value for `GMOCK_DIR` has two fewer parent directories in it. It is now relative to the top of our project as one would expect.

CMakeLists.txt — **New or modified lines in bold.**

```
1∞cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
2∞set(CMAKE_LEGACY_CYGWIN_WIN32 0)
```

```
project("To Do List")

list(APPEND CMAKE_MODULE_PATH ${CMAKE_SOURCE_DIR}/cmake/Modules)

enable_testing()
include(gmock)


if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "GNU" OR
    "${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang")
    set(warnings "-Wall -Wextra -Werror")
elseif ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
    set(warnings "/W4 /WX /EHsc")
endif()
if (NOT CONFIGURED_ONCE)
    set(CMAKE_CXX_FLAGS "${warnings}"
        CACHE STRING "Flags used by the compiler during all build types." FORCE)
    set(CMAKE_C_FLAGS   "${warnings}"
        CACHE STRING "Flags used by the compiler during all build types." FORCE)
endif()


include_directories(${CMAKE_CURRENT_SOURCE_DIR})

add_subdirectory(ToDoCore)

add_executable(toDo main.cc)
target_link_libraries(toDo toDoCore)


set(CONFIGURED_ONCE TRUE CACHE INTERNAL
    "A flag showing that CMake has configured at least once.")
```

```
list(APPEND CMAKE_MODULE_PATH ${CMAKE_SOURCE_DIR}/cmake/Modules)
```

Lists, finally! Okay not quite yet. Here we append the "Modules" directory we created to CMake's module path. This is the path CMake searches when you include a module.

We set the include path because, in the future, we might want to include modules from other `CMakeLists.txt` in other directories. This allows us to include them without having to specify the full path every time.

`include(gmock)`

This includes the new module we created. When used this way CMake searches the module path for the file `gmock.cmake` and when it finds the file it is included. These includes are much like those done by the C preprocessor. The code in the included file executes in the same scope as the file that included it.

`list(APPEND list elements…)`

Appends the elements to the list stored in the variable named `list`. That's correct, you pass in the *name* of the list to be updated, you do not dereference it.

`list()` documentation (2013-06-04)

`CMAKE_MODULE_PATH`

When including modules CMake searches for the requested module in the paths in this list. If this list is exhausted then CMake will look in the directory containing the default modules that come with CMake. Because these paths need to work anywhere in the build tree they must be absolute paths. Since this is a list the `list()` command should be used to manipulate it.

`CMAKE_MODULE_PATH` documentation (2013-06-04)

`include(module | file)`

Include the module or file in the current file being processed. If a *module* name is provided CMake will search for the file `module.cmake` and included it if found. Alternatively if a `file` name is provided CMake will include that file directly; no module path searching is required. If the file cannot be included either because it doesn't exist or wasn't found CMake will issue a warning, but will continue processing.

include() documentation (2013-06-04)

ToDoCore/unit_test/CMakeLists.txt

```
1 add_gmock_test(ToDoTest ToDoTest.cc)
2 target_link_libraries(ToDoTest toDoCore)
```

This file has gone on a serious diet. After moving all general code for unit testing with Google Mock into `gmock.cmake` this file became quite simple.

Source

Lists!

At long last! You've been teased by lists for 2 chapters now, and most of this one too. It is high time we discussed lists.

CMake has two data structures built in: strings and lists. Well, strictly speaking that isn't completely true; lists are semicolon delimited strings. So an empty string is also an empty list and a regular string is a list with only one item. The simplest way to make a list is `set(myList a b c)` which is exactly the same as `set(myList a;b;c)`. However `set(myList "a;b;c")` creates a list with just one item. If a string begins with `"` it is treated as a string literal and any spaces or quotes remain a part of that string rather than causing it to be split into several list items.

Lists are important to understand not just because they are useful but also because all arguments to commands, functions, and macros are processed as a list. So just as `set(myList a b c)` is the same as `set(myList a;b;c)` so too is `set(myList;a;b;c)`. When CMake processes the call to the `set()` command it collects all of the arguments into a single list. This list (`ARGV`) is the separated into the first argument, the variable name (`myList`), and the rest of the items, the values (`a;b;c`). This can cause trouble if you pass a quoted string containing semicolons to a function that then passes it to another function without quoting it as your string will become a list.

While you can create list with `set(myList a b c)` I'd strongly recommend using `list(APPEND myList a b c)`. Using the `list()` command shows that you are using the variable `myList` as a list. Naturally the `list()` command allows you to do other things with lists.

`list()` documentation (2013-06-04)

Auto Play

Well really automatic test running. So far in my experience it takes significantly less time to run unit tests than it does to build them. For this reason I think it is beneficial to run your unit tests every time they are built. This also has the side effect of stopping your build if the unit test fails.

cmake/Modules/gmock.cmake New or modified lines in bold.

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

if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
    # force this option to ON so that Google Test will use /MD instead of /MT
    # /MD is now the default for Visual Studio, so it should be our default, too
    option(gtest_force_shared_crt
           "Use shared (DLL) run-time lib even when Google Test is built as static
           ON)
elseif (APPLE)
    add_definitions(-DGTEST_USE_OWN_TR1_TUPLE=1)
endif()
add_subdirectory(${GMOCK_DIR} ${CMAKE_BINARY_DIR}/gmock)
set_property(TARGET gtest APPEND_STRING PROPERTY COMPILE_FLAGS " -w")

include_directories(SYSTEM ${GMOCK_DIR}/gtest/include
                           ${GMOCK_DIR}/include)


```

```
#
# add_gmock_test(<target> <sources>...)
#
#  Adds a Google Mock based test executable, <target>, built from <sources> and
#  adds the test so that CTest will run it. Both the executable and the test
#  will be named <target>.
#
function(add_gmock_test target)
    add_executable(${target} ${ARGN})
    target_link_libraries(${target} gmock_main)

    add_test(${target} ${target})

    add_custom_command(TARGET ${target}
                       POST_BUILD
                       COMMAND ${target}
                       WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
                       COMMENT "Running ${target}" VERBATIM)
endfunction()
```

Source

```
add_custom_command(TARGET ${target}
                   POST_BUILD
                   COMMAND ./${target}
                   WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
                   COMMENT "Running ${target}" VERBATIM)
```

We use the `add_custom_command()` command to run each test after each time it is built. Here we simply run the test and if it fails the build will stop. However if you were to build again immediately the failed test would **not** be run again and the build will continue. Fixing that will be left for later.

```
add_custom_command(TARGET target
                   PRE_BUILD | PRE_LINK | POST_BUILD
                   COMMAND command [arguments…]
                   [COMMAND command2 [arguments…] …]
```

```
                    [WORKING_DIRECTORY directory]
                    [COMMENT comment] [VERBATIM])
```

`target`

The name of the target to which we are adding the custom command.

`PRE_BUILD | PRE_LINK | POST_BUILD`

When to run the custom command. `PRE_BUILD` will run the command before any of the target's other dependencies. `PRE_LINK` runs the command after all other dependencies. Lastly `POST_BUILD` runs the command after the target has been built.

Note: the `PRE_BUILD` option only works with Visual Studio 7 or newer. For all other generators it is treated as `PRE_LINK` instead.

`COMMAND command [arguments…]`

The command to run and any arguments to be passed to it. If `command` specifies an executable target, i.e. one created with the `add_executable()` command, the location of the actual built executable will replace the name; additionally a target level dependency will be added so that the executable target will be built before this custom command is run.

Note: target level dependencies merely control the order in which targets are build. If a target level dependency is rebuilt this command will not be re-run.

Any number of commands can be listed using this syntax and they will all be run in order each time.

`[ WORKING_DIRECTORY directory ]`

Specify the working directory from which the listed commands will be run.

`[ COMMENT comment ]`

Provide a comment that will be displayed before the listed commands are run.

`[ VERBATIM ]`

This argument tells CMake to ensure that the commands and their arguments are escaped appropriately for whichever build tool is being used. If this argument is omitted the behavior is platform and tool specific. Therefore it is **strongly** recommended that you always provide the

`VERBATIM` argument.

`add_custom_command()` documentation (2013-06-15)

Now it's time to see our hard work in action.

```
 > 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/CMake/CMake Tutorial/flavo
 > 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
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 gtest_main
[ 85%] Building CXX object gmock/gtest/CMakeFiles/gtest_main.dir/src/gtest_main.cc.o
Linking CXX static library libgtest_main.a
[ 85%] Built target gtest_main
Scanning dependencies of target ToDoTest
[100%] Building CXX object ToDoCore/unit_test/CMakeFiles/ToDoTest.dir/ToDoTest.cc.o
Linking CXX executable ToDoTest
Running 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.constructor_createsEmptyList
[       OK ] ToDoTest.constructor_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 (0 ms)
[----------] 4 tests from ToDoTest (0 ms total)
```

```
[----------] Global test environment tear-down
[==========] 4 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 4 tests.
[100%] Built target ToDoTest
```

It still works, just it's more automatic now.

Revision History

Version	Date	Comment
1	2013-07-14	Original version.
2	2014-10-01	Added the work around for a problem with Google Test and newer versions of Mac OS X introduced in Chapter 4

This entry was tagged CMake, long, tutorial. Bookmark the permalink.

← CMake Tutorial – Chapter 4: Libraries and Subdirectories

CMake Tutorial – Chapter 6: Realistically Getting a Boost →

16 thoughts on "CMake Tutorial – Chapter 5: Functionally Improved Testing"

2014-05-24 at 06:17:49

Misgana said:

Thanks, Perhaps this is the best cmake tutorial I've encountered so far.

Reply

2014-06-27 at 21:17:18