JohnLamp.net

Search …

Home CMake Tutorial

← CMake Tutorial CMake Tutorial – Chapter 2: IDE Integration →

CMake Tutorial – Chapter 1: Getting Started

Posted on 2013-03-28 by John Lamp

Contents

- Introduction
- Installation
  - Windows
    - Download and Install
    - Cygwin
  - Mac OS X
    - Download and Install
    - Homebrew
  - Linux
    - Ubuntu (Debian)
    - Red Hat/CentOS
    - Fedora
  - Source
- Hands On

Archives

March 2015

July 2013

May 2013

March 2013

Categories

CMake Tutorial

Meta

Log in

Entries RSS

Comments RSS

WordPress.org

Diving In
Simple Improvements

Introduction

In this chapter we start by installing CMake. Like most open source software the best way to do this depends on your platform and how you usually do things. Once we have CMake installed we create a simple project. Perhaps it's a little fancier than "hello world" but not much. We finish up with the test support built into CMake.

I won't cover any particular aspect of CMake in great detail yet. That will be left for future chapters. However, after this chapter you will know enough to build simple programs with CMake and run simple tests with CTest.

Installation

Windows

Download and Install

Download the installer from the CMake website (2012-06-02). Run the installer and follow its steps. Be sure to add CMake to the system `PATH` so that you can use it from the command line. Add it for the current or all users as appropriate.

This provides both the `cmake` command and the CMake GUI (`cmake-gui`) but not the curses interface (`ccmake`).

Cygwin

CMake can, of course, be installed as part of Cygwin. Even if you don't already have Cygwin installed you may want to as it provides a Linux-like environment natively in Windows. This way common

Linux tools and utilities can be available. Also most of this tutorial is done in a Linux-like environment, so with Cygwin installed it will be easier to follow along.

Download Cygwin's `setup.exe` from their website (2012-06-02). Run `setup.exe`. Follow its steps until you can select packages, then either chose to install all packages or just CMake. To install all packages click the word "Default" next to "All" until it reads "Install". If you don't want to install everything click the word "Default" next to "Devel" until it reads "Install"; this will install just the development tools. If you chose to install all packages the install will take a a few hours, but even just installing the development tools will take at least half an hour. After the installer has finished the Cygwin environment can then be accessed via the `Cygwin Terminal` which can be found in the Start Menu.

This provides the `cmake` command and the curses interface (`ccmake`) but not the CMake GUI.

Mac OS X

Download and Install

Download the disk image from the CMake website (2012-06-02). Pick the correct download for whichever version of OS X you are using. Use the installer and follow its directions. It will ask if you want it to make the command line tools available in your path by creating symbolic links, have it do so.

This provides the `cmake` command, the CMake GUI (`CMake.app`), and the curses interface (`ccmake`).

Homebrew

If you already have homebrew installed you can simply install CMake with the command `brew install cmake`.

This provides the `cmake` command and the curses interface (`ccmake`) but **not** the CMake GUI.

Linux

Ubuntu (Debian)

The simplest way to install CMake is via the command line: `sudo apt-get install cmake`. However, searching for CMake in the Ubuntu Software Center or in the Synaptic Package Manager, depending upon your Ubuntu version, will find the `cmake` package. If your Ubuntu install doesn't include X or you primarily use ssh sessions you will also want to install the `cmake-curses-gui` package. Again this is simplest with the command `sudo apt-get install cmake-curses-gui`, but either GUI interface can be used instead.

This provides the `cmake` command and the CMake GUI (`cmake-gui`). The second, optional, package provides the curses interface (`ccmake`).

Red Hat/CentOS

To install CMake via the command line is straightforward. First use `yum search cmake` to find the correct package to install. On a 64 bit install it would be `cmake.x86_64`. Use whichever package your search found when installing: `sudo yum install cmake.x86_64`. If `sudo` is not setup use `su` first and then run `yum install cmake.x86_64`.

This provides the `cmake` command and the curses interface (`ccmake`), but not the CMake GUI.

Fedora

Either the command line or the Add/Remove Software GUI can be used. In the GUI simply search for cmake and install at least the `cmake` module. If you desire the CMake GUI as well install the `cmake-gui` module. From the command line use `sudo yum install cmake` and `sudo yum install cmake-gui`, if you desire the GUI as well.

This provides the `cmake` command and the curses interface (`ccmake`). The second, optional, package provides the CMake GUI (`cmake-gui`).

Source

As CMake is an open source tool you can, of course, download the source code and build it yourself. However, that is outside the scope of this tutorial.

Hands On

For this tutorial we will create a To Do List program. Naturally our focus will be on CMake more than the actual code and its functionality. Most examples will be done using the command line generating Makefiles. CMake can be used with a GUI (chapter 3) and also generate projects for many IDEs (chapter 2).

Diving In

Just as any IDE has project files or Make has Makefiles CMake has `CmakeLists.txt` files. These describe your project to CMake and affect its output. They are fairly simple especially compared to Makefiles. Here's our first `CMakelists.txt`:

CMakeLists.txt

```
project("To Do List")


add_executable(toDo main.cc
                    ToDo.cc)
```

`project(name)`

The `project` command names your project. Optionally you can specify what language the project supports, any of `CXX`, `C`, `JAVA`, or `FORTRAN`. CMake defaults to `C` and `CXX` so if you do not have compilers for C++ installed you may need to specify the language supported so that CMake doesn't search for it.

Note: If your project name contains spaces it must be surrounded by quotes.

project() documentation (2013-03-26)

`add_executable(target sources…)`

This command tells CMake you want to make an executable and adds it as a target. The first argument is the name of the executable and the rest are the source files. You may notice that header files aren't listed. CMake handles dependencies automatically so headers don't need to be listed.

add_executable() documentation (2013-03-26)

Of course we need some source code to build, so we will start with the simplest skeleton possible:

main.cc

```
#include "ToDo.h"

int main(
    int    argc,
    char** argv
)
{
    ToDo list;

    return 0;
}
```

ToDo.h

```
#ifndef TODO_H
#define TODO_H

class ToDo
{
public:
    ToDo();
    ~ToDo();
};

#endif // TODO_H
```

ToDo.cc

```
#include "ToDo.h"


ToDo::ToDo()
{
}

ToDo::~ToDo()
{
}
```

Source

CMake’s documentation strongly suggests that out-of-source builds be done rather than in-source builds. I agree as it makes it much easier to convince yourself that your build has really been cleaned since you can simply delete the build folder and start over. Building with CMake is actually rather simple, so we will charge ahead:

```
 > mkdir build
 > cd build
 > cmake -G "Unix Makefiles" ..
-- The C compiler identification is GNU 4.2.1
-- The CXX compiler identification is GNU 4.2.1
-- Checking whether C compiler has -isysroot
-- Checking whether C compiler has -isysroot - yes
-- Checking whether C compiler supports OSX deployment target flag
-- Checking whether C compiler supports OSX deployment target flag - yes
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Checking whether CXX compiler has -isysroot
-- Checking whether CXX compiler has -isysroot - yes
-- Checking whether CXX compiler supports OSX deployment target flag
-- Checking whether CXX compiler supports OSX deployment target flag - yes
-- 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
-- Configuring done
-- Generating done
-- Build files have been written to: /Volumes/Documents/Programming/C++/CMake Tutorial
 > ls
CMakeCache.txt          Makefile
CMakeFiles              cmake_install.cmake
 > make
Scanning dependencies of target toDo
[ 50%] Building CXX object CMakeFiles/toDo.dir/main.cc.o
[100%] Building CXX object CMakeFiles/toDo.dir/ToDo.cc.o
Linking CXX executable toDo
[100%] Built target toDo
```

Note: If you are using Cygwin you may see a warning. Don't worry about it, we will take care of that shortly.

`mkdir build`

Create the directory in which to build our application. In this example it is a subdirectory of our source directory, but it could be anywhere. With our build happening outside of the source tree we can easily clean up by simply removing the build directory.

`cd build`

Change into the build directory to work from there.

`cmake -G "Unix Makefiles" ..`

Use CMake to setup a build using Unix Makefiles.

`-G <generator name>`

This allows us to tell CMake what kind of project file it should generate. In this example I wanted to use a Makefile. Which generators are available depends on your platform, use `cmake --help` to list them. Other generators will be covered in the next chapter.

`<path to source>`

The path to the source code. When doing out-of-source builds as is recommended the source code could be anywhere relative to the build directory. This path should be to the directory containing your top level `CMakeLists.txt`. In this example the source is in the parent directory so the path is ‘`..`‘.

`ls`

CMake generates several files which should not be edited by hand. `Makefile` is the most important one to us as we use it to build our project. `CMakeCache.txt` is important to CMake as it stores a variety of information and settings for the project. Again you shouldn’t touch this, however if unexpected problems arise this file probably is the cause; the best option then is to delete your build folder and have CMake regenerate.

`make`

Run `make` to build our target executable. Since we chose “Unix Makefiles” as our generator CMake created a Makefile for us.

CMake does all the hard work of making sure your environment has everything you need and sets up a project file, in this case a Makefile. You will notice that the Makefile created by CMake is quite fancy and has nice color output. If you are used to Make you will notice that this Makefile suppresses the standard output. While this provides a neater and cleaner experience it can make debugging more difficult as you can’t check the flags passed to the compiler, etc. Before you start worrying you can get all of that output by running `make VERBOSE=1`.

```
 > cd build
 > make VERBOSE=1
/usr/local/Cellar/cmake/2.8.8/bin/cmake -H"/Volumes/Documents/Programming/C++/CMake Tu
/usr/local/Cellar/cmake/2.8.8/bin/cmake -E cmake_progress_start "/Volumes/Documents/Pr
make -f CMakeFiles/Makefile2 all
make -f CMakeFiles/toDo.dir/build.make CMakeFiles/toDo.dir/depend
cd "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step1/build" && /u
Dependee "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step1/build/
Dependee "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step1/build/
Scanning dependencies of target toDo
make -f CMakeFiles/toDo.dir/build.make CMakeFiles/toDo.dir/build
/usr/local/Cellar/cmake/2.8.8/bin/cmake -E cmake_progress_report "/Volumes/Documents/P
[ 50%] Building CXX object CMakeFiles/toDo.dir/main.cc.o
/usr/bin/c++     -o CMakeFiles/toDo.dir/main.cc.o -c "/Volumes/Documents/Programming/C
/usr/local/Cellar/cmake/2.8.8/bin/cmake -E cmake_progress_report "/Volumes/Documents/P
[100%] Building CXX object CMakeFiles/toDo.dir/ToDo.cc.o
/usr/bin/c++     -o CMakeFiles/toDo.dir/ToDo.cc.o -c "/Volumes/Documents/Programming/C
Linking CXX executable toDo
/usr/local/Cellar/cmake/2.8.8/bin/cmake -E cmake_link_script CMakeFiles/toDo.dir/link.
/usr/bin/c++    -Wl,-search_paths_first -Wl,-headerpad_max_install_names   CMakeFiles/
/usr/local/Cellar/cmake/2.8.8/bin/cmake -E cmake_progress_report "/Volumes/Documents/P
[100%] Built target toDo
/usr/local/Cellar/cmake/2.8.8/bin/cmake -E cmake_progress_start "/Volumes/Documents/Pr
```

You can see that the makefile created by CMake is very precise and detailed. As such if anything moves you will have to run `cmake` again.

Simple Improvements

CMakeLists.txt New or modified lines in bold.

```cmake
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
set(CMAKE_LEGACY_CYGWIN_WIN32 0)

project("To Do List")

enable_testing()


add_executable(toDo main.cc
                    ToDo.cc)

add_test(toDoTest toDo)
```

`cmake_minimum_required(VERSION version [FATAL_ERROR])`

This command specifies the minimum version of CMake that can be used with `CMakeLists.txt` file. The first argument must be `VERSION` verbatim. The next is the minimum version of CMake that can be used. The last is optional, but should be included, it must be `FATAL_ERROR` verbatim. It is recommended that this command be used in all top level `CMakeLists.txt`. If you aren't sure what version to set use the version of CMake you have installed.

cmake_minimum_required() documentation (2013-03-26)

`set(CMAKE_LEGACY_CYGWIN_WIN32 0)`

This gets rid of the warning you would have seen earlier if you were using Cygwin. If you aren't using Cygwin then it has no effect at all.

This tells CMake not to define `WIN32` when building with Cygwin. This is the preferred option and for us it doesn't make a difference either way so we will use the recommended setting.

`enable_testing()`

Enables testing for this CMake project. This should only be used in top level `CMakeLists.txt`. The main thing this does is enable the `add_test()` command.

enable_testing() documentation (2013-03-26)

`add_test(testname executable [arg1 …])`

This command only does something if the `enable_testing()` has already been run, otherwise it does nothing. This adds a test to the current directory that will be run by CTest. The executable can be anything, so it could be a test program, e.g. a unit test created with something like Google Test, a script, or any other test imaginable. *Note:* Tests are not run automatically and if your test program is built as part of your project the test target will not ensure it is up to date. It is best to build all other targets before running the test target.

add_test() documentation (2013-03-26)

Perhaps I lied. One can easily argue that introducing the `add_test()` command is not a simple improvement. And they would probably be right, however, it is an important improvement. Testing will be explored further later in this tutorial.

Naturally we need some more code to go with this, so here goes:

main.cc New or modified lines in bold.

```
#include <iostream>
  using std::cerr;
  using std::cout;
  using std::endl;

#include "ToDo.h"

#define EXPECT_EQUAL(test, expect) equalityTest( test,  expect, \
                                                #test, #expect, \
                                                __FILE__, __LINE__)

template < typename T1, typename T2 >
int equalityTest(const T1    testValue,
                 const T2    expectedValue,
                    const char* testName,
                    const char* expectedName,
                    const char* fileName,
                    const int   lineNumber);


int main(
    int    argc,
    char** argv
)
{
    int result = 0;

    ToDo list;

    list.addTask("write code");
    list.addTask("compile");
    list.addTask("test");

    result |= EXPECT_EQUAL(list.size(),     3);
    result |= EXPECT_EQUAL(list.getTask(0), "write code");
    result |= EXPECT_EQUAL(list.getTask(1), "compile");
    result |= EXPECT_EQUAL(list.getTask(2), "test");

    if (result == 0)
    {
        cout << "Test passed" << endl;
    }

    return result;
}


template < typename T1, typename T2 >
int equalityTest(
    const T1    testValue,
    const T2    expectedValue,
    const char* testName,
    const char* expectedName,
    const char* fileName,
    const int   lineNumber
)
{
    if (testValue != expectedValue)
    {
        cerr << fileName << ":" << lineNumber << ": "
             << "Expected " << testName << " "
             << "to equal " << expectedName << " (" << expectedValue << ") "
             << "but it was (" << testValue << ")" << endl;

        return 1;
    }
    else
    {
        return 0;
    }
}
```

ToDo.h New or modified lines in bold.

```
#ifndef TODO_H
#define TODO_H

#include <string>
#include <vector>


class ToDo
{
```

```
public:
    ToDo();
    ~ToDo();

    size_t size() const;

    void addTask(const std::string& task);
    std::string getTask(size_t index) const;

private:
    std::vector< std::string > this_tasks;
};

#endif // TODO_H
```

ToDo.cc New or modified lines in bold.

```
#include "ToDo.h"


ToDo::ToDo()
{
}

ToDo::~ToDo()
{
}


size_t ToDo::size() const
{
    return this_tasks.size();
}

```

```

void ToDo::addTask(
    const std::string& task
)
{
    this_tasks.push_back(task);
}

std::string ToDo::getTask(
    size_t index
) const
{
    if (index < this_tasks.size())
    {
        return this_tasks[index];
    }
    else
    {
        return "";
    }
}
```

Source

Whew! That was not simple at all. Hopefully some of you are wondering why I didn’t use a test framework. Later we will, but had we done so now we would have gotten further ahead of ourselves than we already have.

Building is exactly the same as before. In fact if you modified the files you had used before you simply need to run `make` again. The Makefile created by CMake will automatically run `cmake` again if you modify your `CMakeLists.txt`. So let’s run our test:

```
> mkdir build
> cd build
```

```
 > cmake -G "Unix Makefiles" ..
-- The C compiler identification is GNU 4.2.1
-- The CXX compiler identification is GNU 4.2.1
-- Checking whether C compiler has -isysroot
-- Checking whether C compiler has -isysroot - yes
-- Checking whether C compiler supports OSX deployment target flag
-- Checking whether C compiler supports OSX deployment target flag - yes
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Checking whether CXX compiler has -isysroot
-- Checking whether CXX compiler has -isysroot - yes
-- Checking whether CXX compiler supports OSX deployment target flag
-- Checking whether CXX compiler supports OSX deployment target flag - yes
-- 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
-- Configuring done
-- Generating done
-- Build files have been written to: /Volumes/Documents/Programming/C++/CMake Tutorial
 > make
Scanning dependencies of target toDo
[ 50%] Building CXX object CMakeFiles/toDo.dir/main.cc.o
[100%] Building CXX object CMakeFiles/toDo.dir/ToDo.cc.o
Linking CXX executable toDo
[100%] Built target toDo
 > make test
Running tests...
Test project /Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/bui
    Start 1: toDoTest
1/1 Test #1: toDoTest .........................   Passed    0.01 sec
100% tests passed, 0 tests failed out of 1
Total Test time (real) =   0.03 sec
```

```
 > ls Testing
Temporary
 > ls Testing/Temporary
CTestCostData.txt       LastTest.log
 > cat Testing/Temporary/LastTest.log
Start testing: Jul 16 22:00 EDT
----------------------------------------------------------
1/1 Testing: toDoTest
1/1 Test: toDoTest
Command: "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/build/
Directory: /Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/build
"toDoTest" start time: Jul 16 22:00 EDT
Output:
----------------------------------------------------------
Test passed
<end of output>
Test time =   0.01 sec
----------------------------------------------------------
Test Passed.
"toDoTest" end time: Jul 16 22:00 EDT
"toDoTest" time elapsed: 00:00:00
----------------------------------------------------------
End testing: Jul 16 22:00 EDT
 > cat Testing/Temporary/CTestCostData.txt
toDoTest 1 0.00976491
---
```

As mentioned earlier building with CMake is the same as it was before.

```
make test
```

The `enable_testing()` function we added to our `CMakeLists.txt` adds the “test” target to our Makefile. Making the “test” target will run CTest which will, in turn, run all of our tests. In our case just the one.

When CTest runs our tests it prints an abbreviated output that just provides the status of each of our tests. It then finishes up with a summary of all tests.

`Testing/Temporary/LastTest.log`
This file is created by CTest whenever it is run. It contains much more detail than the terminal output of CTest shows. Most importantly it contains the output of the tests. This is where you will want to look whenever a test fails.

`Testing/Temporary/CTestCostData.txt`
This file contains the time, in seconds, taken to run each test.

CMake along with CTest makes it easy to run our tests. CTest has many other features which will be presented later in this tutorial. There are, however, a few drawbacks to running our tests this way but we will leave those for later, too.

Revision History

Version	Date	Comment
1	2013-03-28	Original version.
2	2013-07-14	Added line numbers and indication of changes to code sample.

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

← CMake Tutorial   CMake Tutorial – Chapter 2: IDE Integration →

11 thoughts on “CMake Tutorial – Chapter 1: Getting Started”

2014-09-02 at 02:02:33
Steve Price said: