CMake Guide

Edinburgh Napier University

January 9, 2017

Abstract

CMake (pronounced "See-Make") is a build system, it lets developers set-up the environment for building their software correctly. It can also be used to manage project dependencies and help with cross platform portability. Cmake is used in a few Napier C++ based modules and this guide will get you up to speed with using it correctly.

1 Why use a build system?

Many high-level languages and software stacks have something along the lines of a package manager (NPM, Gradle, NuGet, Maven, Bower). These allow application writers to say "To use my software, you need *these* other libraries", then the package manager will usually go grab the required software from some central repository/github. In a high-level language this is usually all you have to think about when distributing software, you don't have to care about what the users environment or operating system is as the language / runtime will abstract that away.

For C/C++ development, things are still a little stone-aged. There is no commonly accepted package manager, CMake is gaining traction quickly but it's not universal yet. The other major problem is that as C code operates on a relativity low level, so you **do** need to care about what operating system you are being compiled on, also what's installed, what version things are, and where they are located. Furthermore, as C/C++ is a compiled language you have to deal with the differences in compilers for each platform.

It's not all bad news, the modern C++ language spec has many helpful things that make cross platform code easy to write compared to the dark times of before. Porting C++ code usually means you have to modify the code that such nowadays.

> But what if I don't care about portability? I just want C++ code that works on windows, I can load up visual studio, download any libraries I need as prebuilt .libs and .dlls, add them to the Config and be done.

This is true, and possibly arguably the best for small quickly made programs, but even ignoring portability, it has several issues:

- You will need to commit binary files to Version Control.
- The libraries you are using may not permit you to re-distribute them as binary files via their license.
- The .dlls and .libs you use have to have been built using the same compiler version as you are using.
- If the Libs are updated, you have to download the newest version manually
- You have to dig through settings menus in VS to setup the project, this can be tedious to change.
- VS files, although plain-text, don't version nicely.
- VS creates a lot of junk files, you have to know which you can ignore when using version control.

2 How does CMake work?

First of all, CMake has to be installed on the system you intend to develop on. Projects have a config file called "CMakeLists.txt", this contains the instructions that CMake will use to setup your project. The config contains info like: Where is the source files .cpp / .h, wha project they belong to and what libraries each project will need.

CMake can do other cool things like detect information about the platform and use that to customise header files. For example you may have a set of functions that only need to be included in Windows versions of the code. You could rely on compiler definitions to detect tis in the preprocessor, but would it not be better if this code was not even included in the first place?

After processing the CMakeLists file (called the "Configure" step), CMake will then generate a solution/project/make file. For us, this is usually a Visual Studio .sln solution file. At this point you may think this is a lot of effort to begin with considering we haven't even started to build the software, but consider that the .sln is 100% configured and ready to roll, you won't have to and shouldn't alter any of the project settings from within VS. If you need to alter any of the project settings: edit the config, re-configure and CMake will re-generate a fresh solution file.

Important! The Build and the Source Directories All the files that CMake generate will go into a folder that you specify (The Build Folder). You can specify that the build folder is the same as your source folder. This has the benefit that everything is 'close together' so things like file paths and resource loading work nicely. This is called an 'In-Tree Build' however it has the downside that it will clutter up your source directory.

The better method is to specify the build folder completely outside fo the source folder, called an 'out-of-tree build'. This keeps the stuff you don't need separate. You only need to edit and version the stuff in the source folder, you don't need to keep or care about anything in the build folder because CMake and VS can regenerate it any time from the source folder.

3 What does CMake do for us?

Manages Dependancies Cmake can search the filesystem for common Libraries and set them up within your project. It can also download dependencies from the internet and git. In the case were you are downloading extra projects, usually you will want to download the source and build it your self. This is where CMake really shines, as hopefully the projects you are downloading are correctly setup with their own CMakeLists. This will all get rolled into your own project so for Visual studio will have all your dependencies as projects within your solution, this is fantastic for debugging as you can step through the source of all the libraries you are using.

Creates Makefiles/Solutions

Extra Magic

4 CMake walkthrough

4.1 Using CMake via the Gui

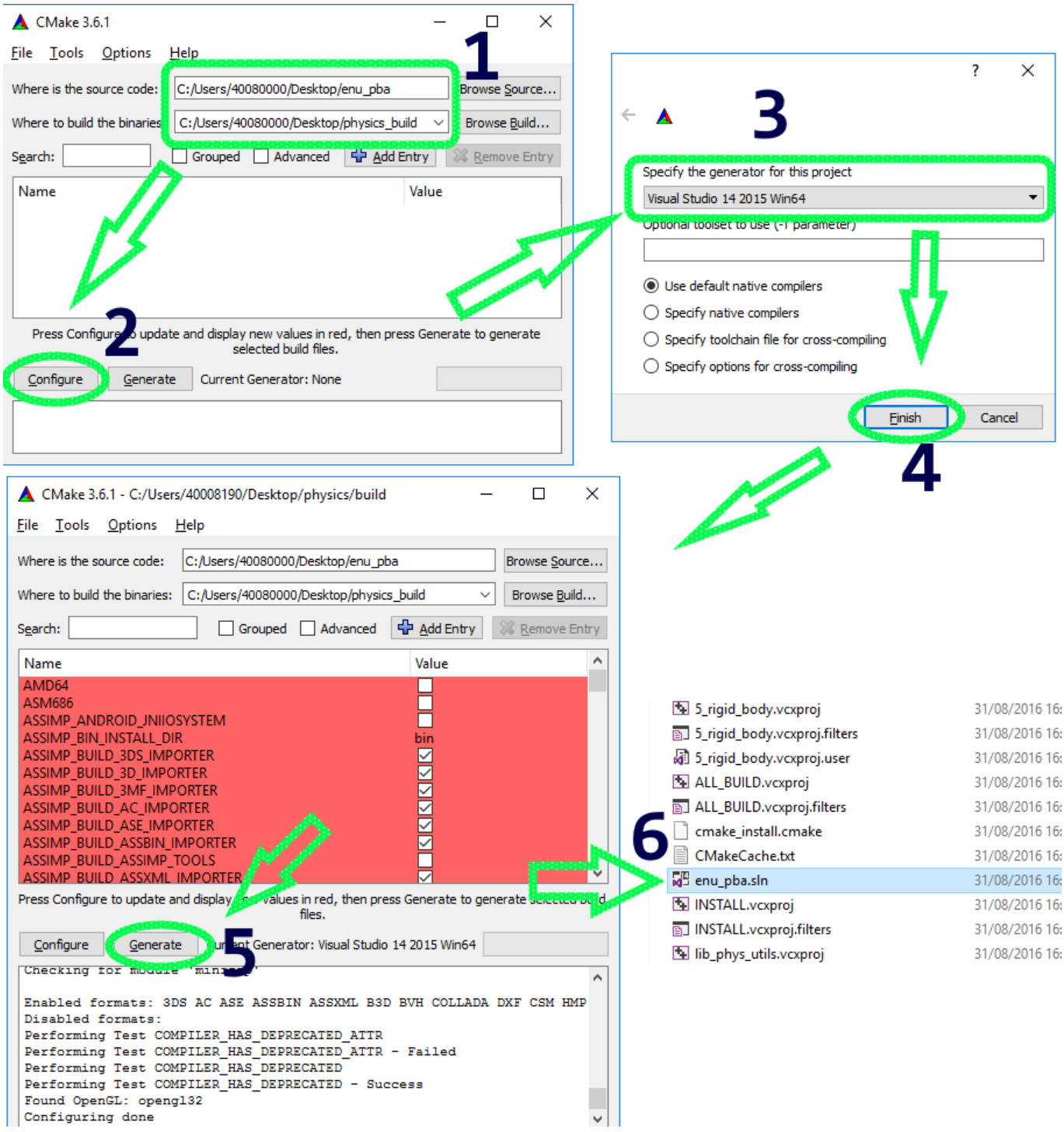

Figure 1: **GUI Process**

4.2 Using CMake via the Command Line

```
1  mkdir projectname_build
2  cd projectname_build
3  cmake -G "Visual Studio 14 2015 Win64" ../projectname/
4
```

5 Common Mistakes

Editing the Build Folder

Committing / Saving the build folder

Building the build folder in the source folder

6 Advice for Using CMake in your future projects