Preshing on Programming

- Twitter
- RSS

Navigate…

- Blog
- Archives
- About
- Contact
- Tip Jar

May 11, 2017

How to Build a CMake-Based Project

CMake is a versatile tool that helps you build C/C++ projects on just about any platform you can think of. It's used by many popular open source projects including LLVM, Qt, KDE and Blender.

All CMake-based projects contain a script named `CMakeLists.txt`, and this post is meant as a guide for configuring and building such projects. This post won't show you how to *write* a CMake script – that's getting ahead of things, in my opinion.

As an example, I've prepared a CMake-based project that uses SDL2 and OpenGL to render a spinning 3D logo. You can build it on Windows, MacOS or Linux.

The information here applies to any CMake-based project, so feel free to skip ahead to any section. However, I recommend reading the first two sections first.

- The Source and Binary Folders
- The Configure and Generate Steps
- Running CMake from the Command Line
- Running cmake-gui

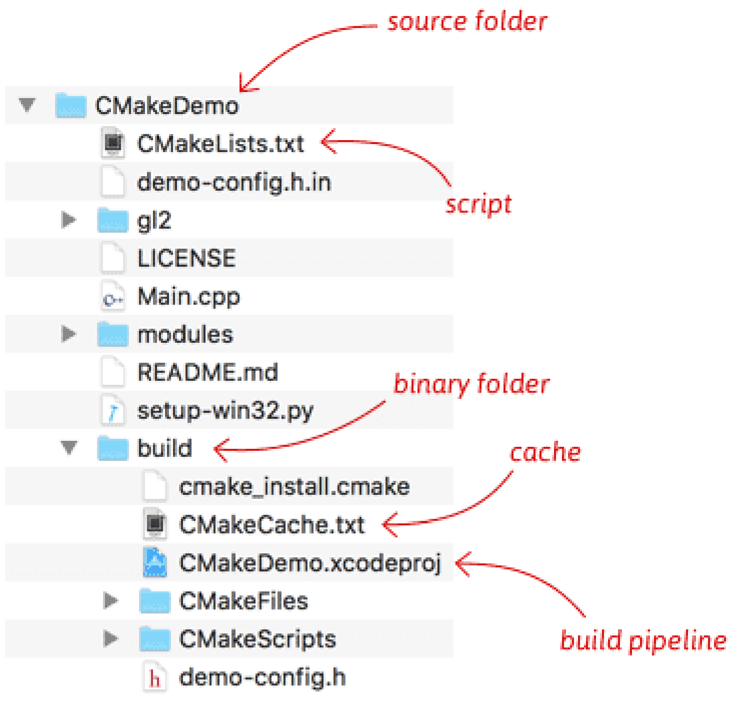
- Running ccmake
- Building with Unix Makefiles
- Building with Visual Studio
- Building with Xcode
- Building with Qt Creator
- Other CMake Features

If you don’t have CMake yet, there are installers and binary distributions on the CMake website. In Unix-like environments, including Linux, it’s usually available through the system package manager. You can also install it through MacPorts, Homebrew, Cygwin or MSYS2.

The Source and Binary Folders

CMake generates **build pipelines**. A build pipeline might be a Visual Studio `.sln` file, an Xcode `.xcodeproj` or a Unix-style `Makefile`. It can also take several other forms.

To generate a build pipeline, CMake needs to know the **source** and **binary** folders. The source folder is the one containing `CMakeLists.txt`. The binary folder is where CMake generates the build pipeline. You can create the binary folder anywhere you want. A common practice is to create a subdirectory `build` beneath `CMakeLists.txt`.



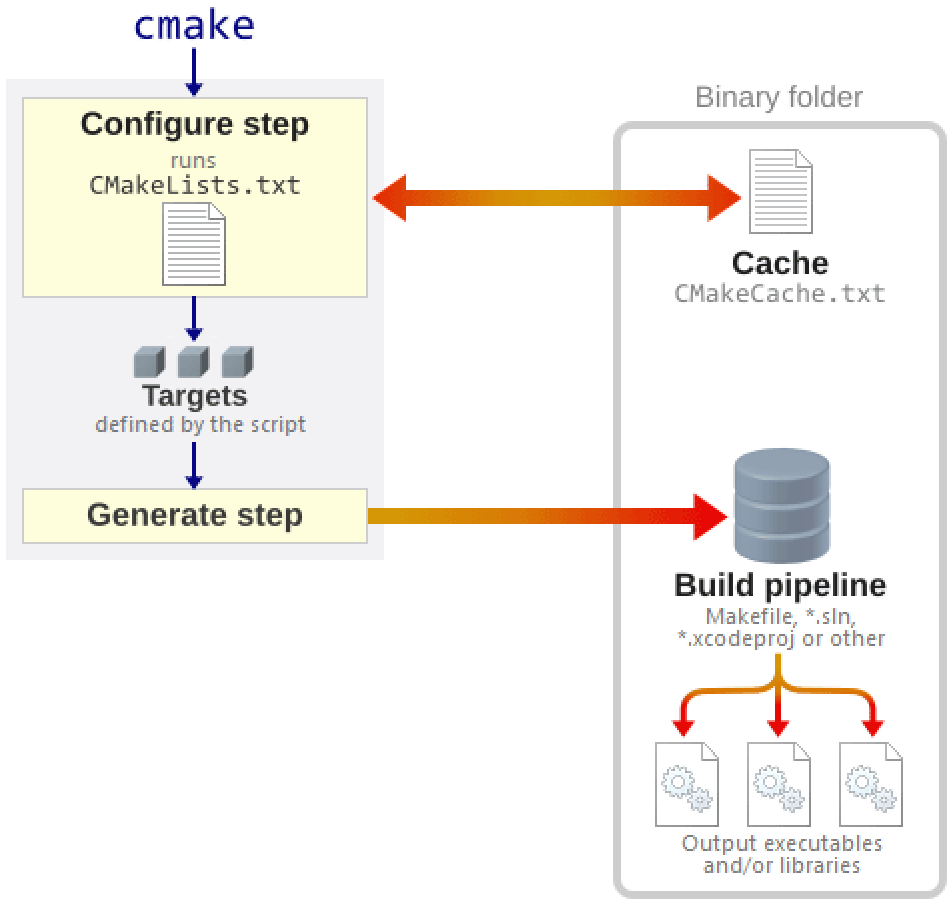
By keeping the binary folder separate from the source, you can delete the binary folder at any time to get back to a clean slate. You can even create several binary folders, side-by-side, that use different build systems or configuration options.

The **cache** is an important concept. It’s a single text file in the binary folder named `CMakeCache.txt`. This is where **cache variables** are stored. Cache variables include user-configurable options defined by the project such as CMakeDemo’s `DEMO_ENABLE_MULTISAMPLE` option (explained later), and precomputed information to help speed up CMake runs. (You can, and will, re-run CMake several times on the same binary folder.)

You aren’t meant to submit the generated build pipeline to source control, as it usually contains paths that are hardcoded to the local filesystem. Instead, simply re-run CMake each time you clone the project to a new folder. I usually add the rule `*build*/` to my `.gitignore` files.

The Configure and Generate Steps

As you'll see in the following sections, there are several ways to run CMake. No matter how you run it, it performs two steps: the **configure** step and the **generate** step.

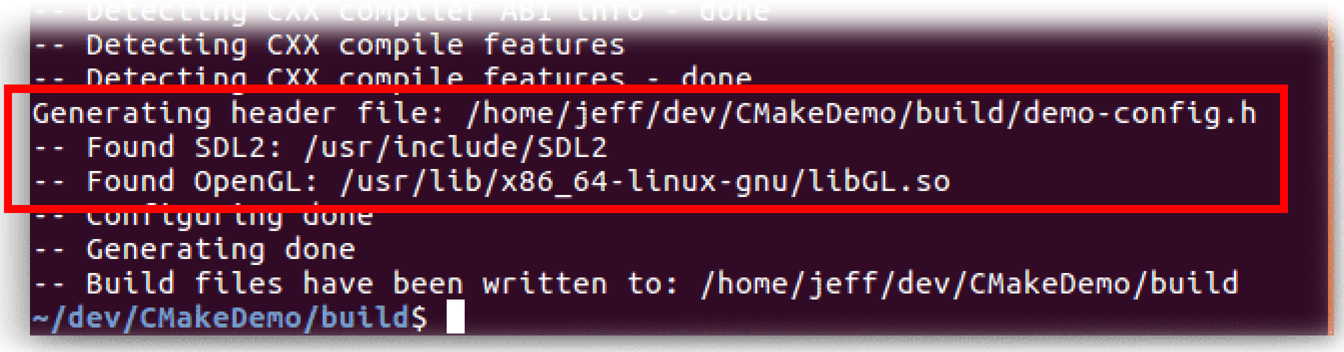


The `CMakeLists.txt` script is executed during the configure step. This script is responsible for defining **targets**. Each target represents an executable, library, or some other output of the build pipeline.

If the configure step succeeds – meaning `CMakeLists.txt` completed without errors – CMake will generate a build pipeline using the targets defined by the script. The type of build pipeline generated depends on the type of **generator** used, as explained in the following sections.

Additional things may happen during the configure step, depending on the contents of `CMakeLists.txt`. For example, in our sample CMakeDemo project, the configure step also:

- Finds the header files and libraries for SDL2 and OpenGL.
- Generates a header file `demo-config.h` in the binary folder, which will be included from C++ code.



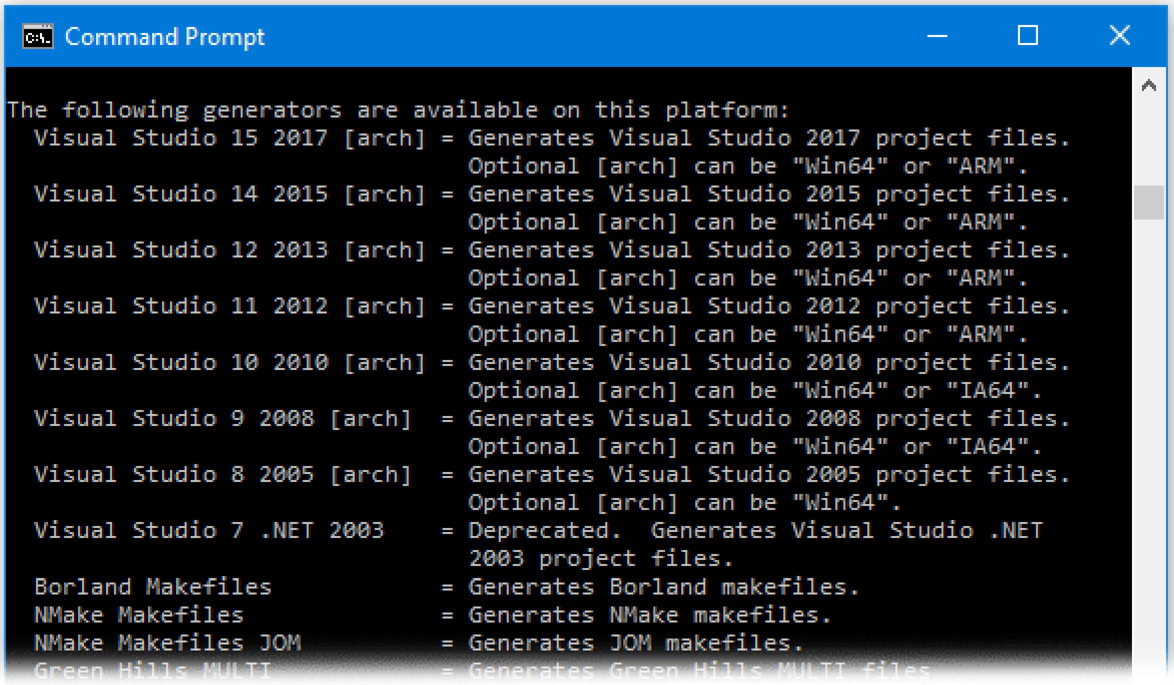
```
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
Generating header file: /home/jeff/dev/CMakeDemo/build/demo-config.h
-- Found SDL2: /usr/include/SDL2
-- Found OpenGL: /usr/lib/x86_64-linux-gnu/libGL.so
-- Configuring done
-- Generating done
-- Build files have been written to: /home/jeff/dev/CMakeDemo/build
~/dev/CMakeDemo/build$
```

In a more sophisticated project, the configure step might also test the availability of system functions (as a traditional Unix `configure` script would), or define a special "install" target (to help create a distributable package). If you re-run CMake on the same binary folder, many of the slow steps are skipped during subsequent runs, thanks to the cache.

Running CMake from the Command Line

Before running CMake, make sure you have the required dependencies for your project and platform. For CMakeDemo on Windows, you can run `setup-win32.py`. For other platforms, check the README.

You'll often want to tell CMake which generator to use. For a list of available generators, run `cmake --help`.



```
The following generators are available on this platform:
  Visual Studio 15 2017 [arch] = Generates Visual Studio 2017 project files.
                                 Optional [arch] can be "Win64" or "ARM".
  Visual Studio 14 2015 [arch] = Generates Visual Studio 2015 project files.
                                 Optional [arch] can be "Win64" or "ARM".
  Visual Studio 12 2013 [arch] = Generates Visual Studio 2013 project files.
                                 Optional [arch] can be "Win64" or "ARM".
  Visual Studio 11 2012 [arch] = Generates Visual Studio 2012 project files.
                                 Optional [arch] can be "Win64" or "ARM".
  Visual Studio 10 2010 [arch] = Generates Visual Studio 2010 project files.
                                 Optional [arch] can be "Win64" or "IA64".
  Visual Studio 9 2008 [arch]  = Generates Visual Studio 2008 project files.
                                 Optional [arch] can be "Win64" or "IA64".
  Visual Studio 8 2005 [arch]  = Generates Visual Studio 2005 project files.
                                 Optional [arch] can be "Win64".
  Visual Studio 7 .NET 2003    = Deprecated.  Generates Visual Studio .NET
                                 2003 project files.
  Borland Makefiles            = Generates Borland makefiles.
  NMake Makefiles              = Generates NMake makefiles.
  NMake Makefiles JOM          = Generates JOM makefiles.
  Green Hills MULTI            = Generates Green Hills MULTI files
```

Create the binary folder, `cd` to that folder, then run `cmake`, specifying the path to the source folder on the command line. Specify the desired generator using the `-G` option. If you omit the `-G` option, `cmake` will choose one for you. (If you don't like its choice, you can always delete the binary folder and start over.)

```
mkdir build
cd build
cmake -G "Visual Studio 15 2017" ..
```

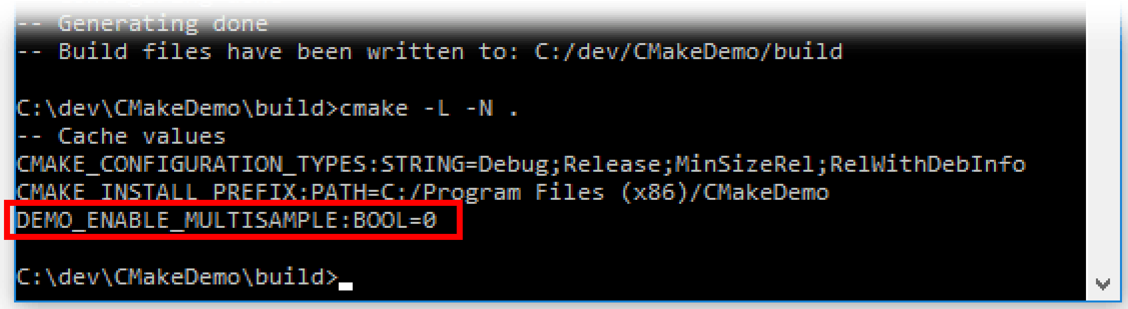
If there are project-specific configuration options, you can specify those on the command line as well. For example, the CMakeDemo project has a configuration option `DEMO_ENABLE_MULTISAMPLE` that defaults to 0. You can enable this configuration option by specifying `-DDEMO_ENABLE_MULTISAMPLE=1` on the `cmake` command line. Changing the value of `DEMO_ENABLE_MULTISAMPLE` will change the contents of `demo-config.h`, a header file that's generated by `CMakeLists.txt` during the configure step. The value of this variable is also stored in the cache so that it persists during subsequent runs. Other projects have different configuration options.

```
cmake -G "Visual Studio 15 2017" -DDEMO_ENABLE_MULTISAMPLE=1 ..
```

If you change your mind about the value of `DEMO_ENABLE_MULTISAMPLE`, you can re-run CMake at any time. On subsequent runs, instead of passing the source folder path to the `cmake` command line, you can simply specify the path to the existing binary folder. CMake will find all previous settings in the cache, such as the choice of generator, and re-use them.

```
cmake -DDEMO_ENABLE_MULTISAMPLE=0 .
```

You can view project-defined cache variables by running `cmake -L -N .`. Here you can see CMakeDemo's `DEMO_ENABLE_MULTISAMPLE` option left at its default 0 value:

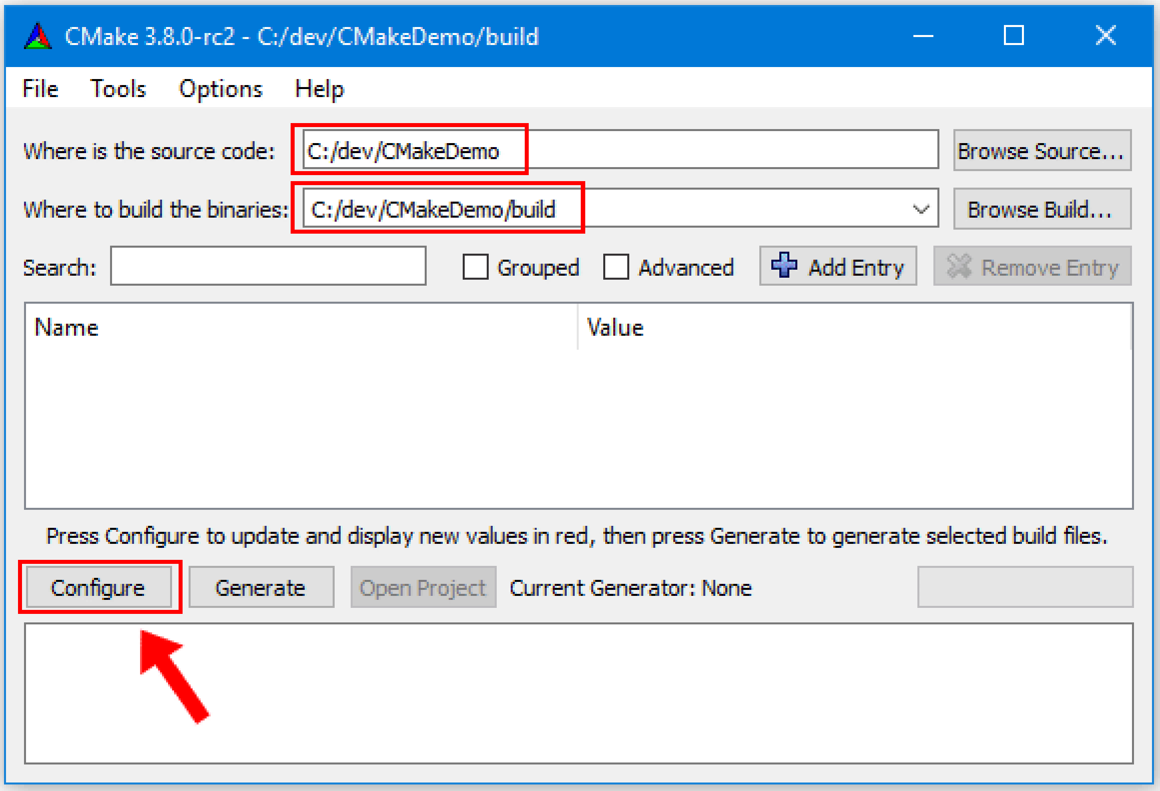


```
-- Generating done
-- Build files have been written to: C:/dev/CMakeDemo/build

C:\dev\CMakeDemo\build>cmake -L -N .
-- Cache values
CMAKE_CONFIGURATION_TYPES:STRING=Debug;Release;MinSizeRel;RelWithDebInfo
CMAKE_INSTALL_PREFIX:PATH=C:/Program Files (x86)/CMakeDemo
DEMO_ENABLE_MULTISAMPLE:BOOL=0

C:\dev\CMakeDemo\build>
```

Running cmake-gui

I prefer the command line, but CMake also has a GUI. The GUI offers an interactive way to set cache variables. Again, make sure to install your project's required dependencies first.

To use it, run `cmake-gui`, fill in the source and binary folder paths, then click Configure.



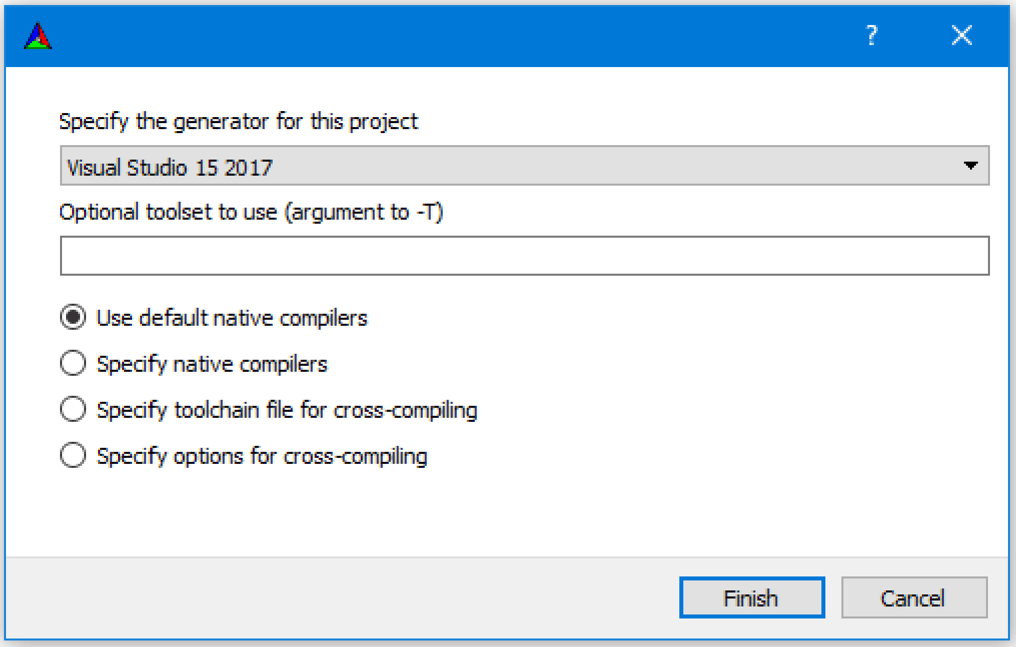
Where is the source code: C:/dev/CMakeDemo

Where to build the binaries: C:/dev/CMakeDemo/build

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: None

If the binary folder doesn't exist, CMake will prompt you to create it. It will then ask you to select a generator.



Specify the generator for this project

Visual Studio 15 2017

Optional toolset to use (argument to -T)

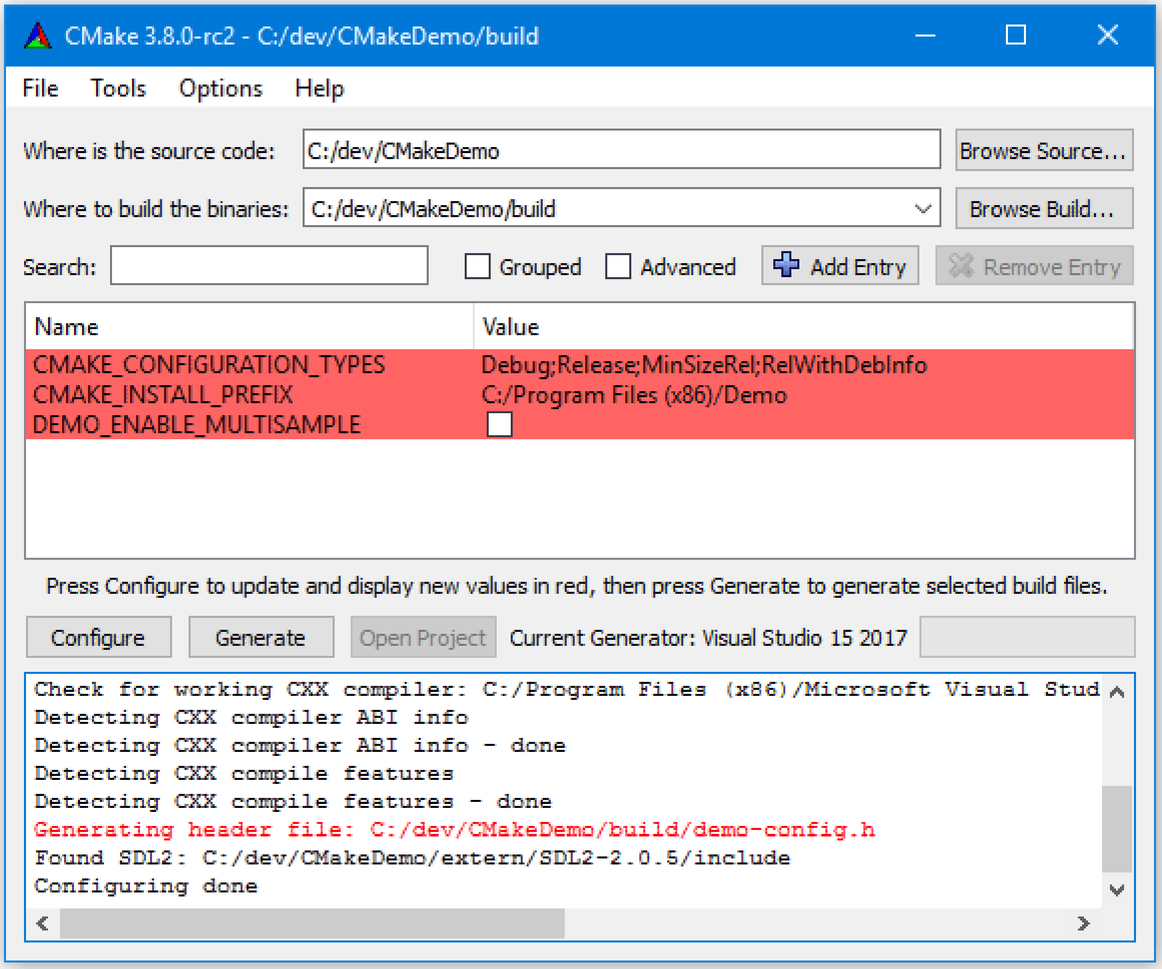
● Use default native compilers

○ Specify native compilers

○ Specify toolchain file for cross-compiling

○ Specify options for cross-compiling

After the initial configure step, the GUI will show you a list of cache variables, similar to the list you see when you run `cmake -L -N .` from the command line. New cache variables are highlighted in red. (In this case, that's all of them.) If you click Configure again, the red highlights will disappear, since the variables are no longer considered new.



Where is the source code: C:/dev/CMakeDemo

Where to build the binaries: C:/dev/CMakeDemo/build

Name	Value
CMAKE_CONFIGURATION_TYPES	Debug;Release;MinSizeRel;RelWithDebInfo
CMAKE_INSTALL_PREFIX	C:/Program Files (x86)/Demo
DEMO_ENABLE_MULTISAMPLE	☐

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 15 2017

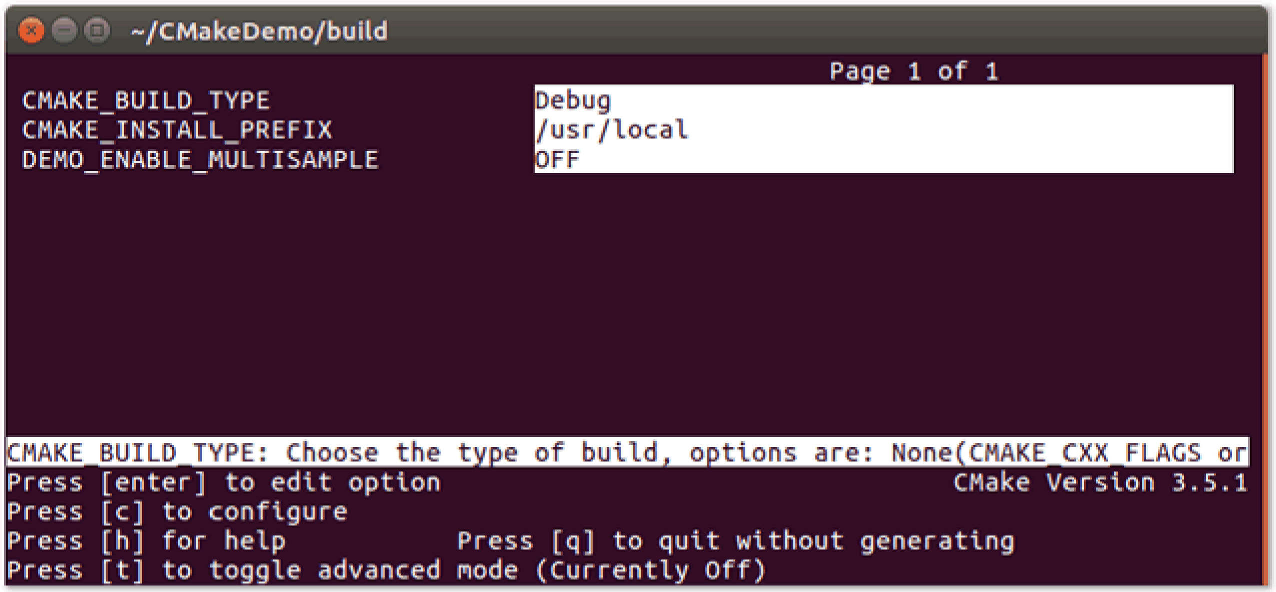
```
Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual Stud
Detecting CXX compiler ABI info
Detecting CXX compiler ABI info - done
Detecting CXX compile features
Detecting CXX compile features - done
Generating header file: C:/dev/CMakeDemo/build/demo-config.h
Found SDL2: C:/dev/CMakeDemo/extern/SDL2-2.0.5/include
Configuring done
```

The idea is that if you change a cache variable, then click Configure, new cache variables might appear as a result of your change. The red highlights are meant to help you see any new variables, customize them, then click Configure again. In practice, changing a value doesn't introduce new cache variables very often. It depends how the project's `CMakeLists.txt` script was written.

Once you've customized the cache variables to your liking, click Generate. This will generate the build pipeline in the binary folder. You can then use it to build your project.

Running ccmake

`ccmake` is the console equivalent to `cmake-gui`. Like the GUI, it lets you set cache variables interactively. It can be handy when running CMake on a remote machine, or if you just like using the console. If you can figure out the CMake GUI, you can figure out `ccmake`.



```
~/CMakeDemo/build
                                                         Page 1 of 1
 CMAKE_BUILD_TYPE                 Debug
 CMAKE_INSTALL_PREFIX             /usr/local
 DEMO_ENABLE_MULTISAMPLE          OFF









CMAKE_BUILD_TYPE: Choose the type of build, options are: None(CMAKE_CXX_FLAGS or
Press [enter] to edit option                                  CMake Version 3.5.1
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Building with Unix Makefiles

CMake generates a Unix makefile by default when run from the command line in a Unix-like environment. Of course, you can generate makefiles explicitly using the `-G` option. When generating a makefile, you should also define the `CMAKE_BUILD_TYPE` variable. Assuming the source folder is the parent:

```
cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Debug ..
```

You should define the `CMAKE_BUILD_TYPE` variable because makefiles generated by CMake are **single-configuration**. Unlike a Visual Studio solution, you can't use the same makefile to build multiple configurations such as Debug and Release. A single makefile is capable of building *exactly one* build type. By default, the available types are Debug, MinSizeRel, RelWithDebInfo and Release. Watch out – if you forget to define `CMAKE_BUILD_TYPE`, you'll probably get an unoptimized build without debug information, which is useless. To change to a different build type, you must re-run CMake and generate a new makefile.

Personally, I also find CMake's default Release configuration useless because it doesn't generate any debug information. If you've ever opened a crash dump or fixed a bug in Release, you'll appreciate the availability of debug information, even in an optimized build. That's why, in my other CMake projects, I usually delete the Release configuration from `CMakeLists.txt` and use RelWithDebInfo instead.

Once the makefile exists, you can actually build your project by running `make`. By default, `make` will build every target that was defined by `CMakeLists.txt`. In CMakeDemo's case, there's only one target. You can also build a specific target by passing its name to `make`:

```
make CMakeDemo
```

The makefile generated by CMake detects header file dependencies automatically, so editing a single header file won't necessarily rebuild the entire project. You can also parallelize the build by passing `-j 4` (or a higher number) to `make`.

CMake also exposes a Ninja generator. Ninja is similar to `make`, but faster. It generates a `build.ninja` file, which is similar to a `Makefile`. The Ninja generator is also single-configuration. Ninja's `-j` option autodetects the number of available CPUs.

Building with Visual Studio

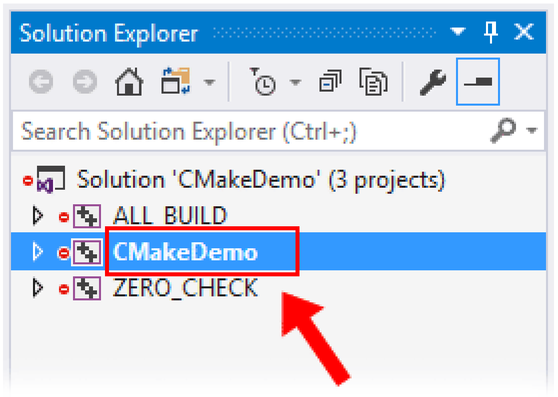
We'll generate a Visual Studio `.sln` file from the CMake command line. If you have several versions of Visual Studio installed, you'll want to tell `cmake` which version to use. Again, assuming that the source folder is the parent:

```
cmake -G "Visual Studio 15 2017" ..
```

The above command line will generate a Visual Studio `.sln` file for a 32-bit build. There are no multiplatform `.sln` files using CMake, so for a 64-bit build, you must specify the 64-bit generator:

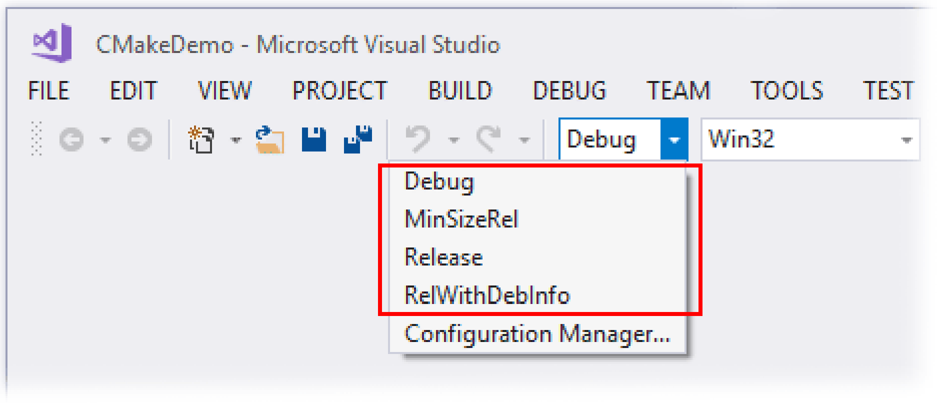
```
cmake -G "Visual Studio 15 2017 Win64" ..
```

Open the resulting `.sln` file in Visual Studio, go to the Solution Explorer panel, right-click the target you want to run, then choose "Set as Startup Project". Build and run as you normally would.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'CMakeDemo' (3 projects)
ALL_BUILD
CMakeDemo
ZERO_CHECK

Note that CMake adds two additional targets to the solution: ALL_BUILD and ZERO_CHECK. ZERO_CHECK automatically re-runs CMake when it detects a change to `CMakeLists.txt`. ALL_BUILD usually builds all other targets, making it somewhat redundant in Visual Studio. If you're used to setting up your solutions a certain way, it might seem annoying to have these extra targets in your `.sln` file, but you get used to it. CMake lets you organize targets and source files into folders, but I didn't demonstrate that in the CMakeDemo sample.

Like any Visual Studio solution, you can change build type at any time from the Solution Configuration drop-down list. The CMakeDemo sample uses CMake's default set of build types, shown below. Again, I find the default Release configuration rather useless as it doesn't produce any debug information. In my other CMake projects, I usually delete the Release configuration from `CMakeLists.txt` and use RelWithDebInfo instead.



Built-In CMake Support in Visual Studio 2017

In Visual Studio 2017, Microsoft introduced another way to use CMake with Visual Studio. You can now open the source folder containing `CMakeLists.txt` from Visual Studio's File → Open → Folder menu. This new method avoids creating intermediate `.sln` and `.vcxproj` files. It also exposes 32-bit and 64-bit builds in the same workspace. It's a nice idea that, in my opinion, falls short for a few reasons:

- If there are any source files *outside* the source folder containing `CMakeLists.txt`, they won't appear in the Solution Explorer.
- The familiar C/C++ Property Pages are no longer available.
- Cache variables can only be set by editing a JSON file, which is pretty unintuitive for a Visual IDE.

I'm not really a fan. For now, I intend to keep generating `.sln` files by hand using CMake.

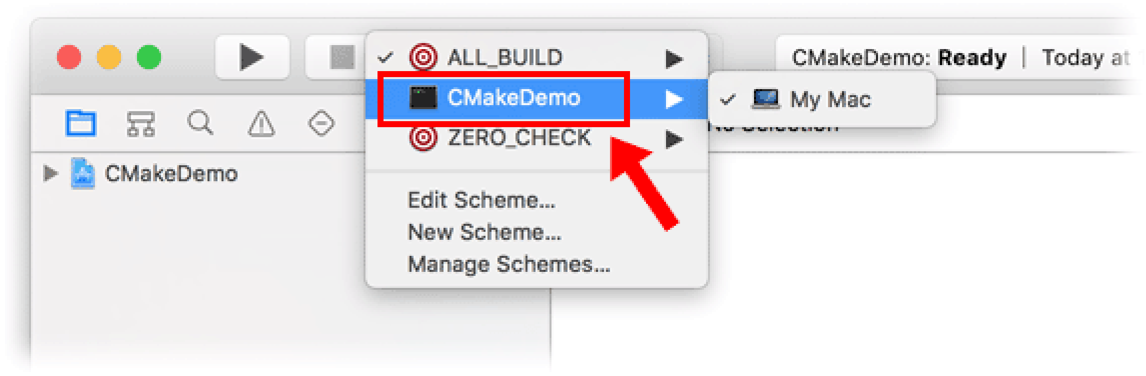
Building with Xcode

The CMake website publishes a binary distribution of CMake for MacOS as a `.dmg` file. The `.dmg` file contains an app that you can drag & drop to your Applications folder. Note that if you install CMake this way, `cmake` won't be available from the command line unless you create a link to `/Applications/CMake.app/Contents/bin/cmake` somewhere. I prefer installing CMake from MacPorts because it sets up the command line for you, and because dependencies like SDL2 can be installed the same way.

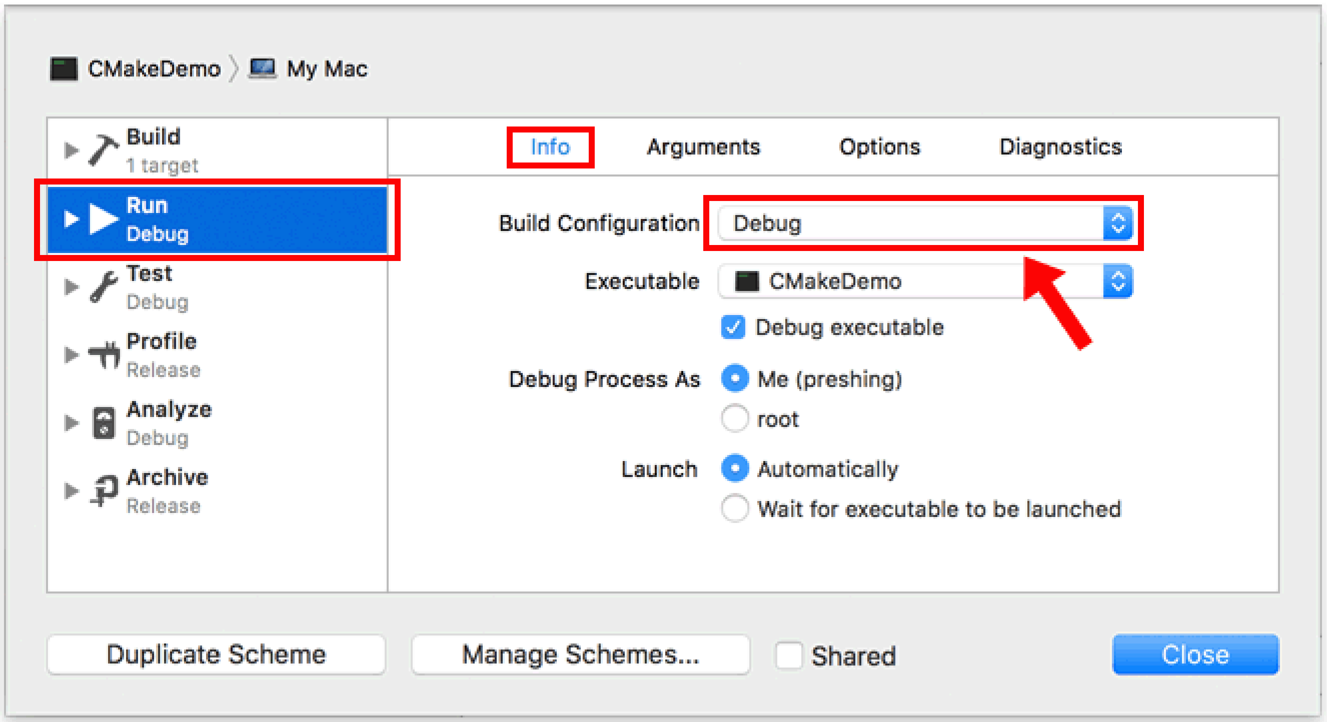
Specify the Xcode generator from the CMake command line. Again, assuming that the source folder is the parent:

```
cmake -G "Xcode" ..
```

This will create an `.xcodeproj` folder. Open it in Xcode. (I tested in Xcode 8.3.1.) In the Xcode toolbar, click the "active scheme" drop-down list and select the target you want to run.



After that, click "Edit Scheme…" from the same drop-down list, then choose a build configuration under Run → Info. Again, I don't recommend CMake's default Release configuration, as the lack of debug information limits its usefulness.



Build Configuration: Debug
Executable: CMakeDemo
☑ Debug executable
Debug Process As: [✓ Me (preshing)] root
Launch: [✓ Automatically] Wait for executable to be launched

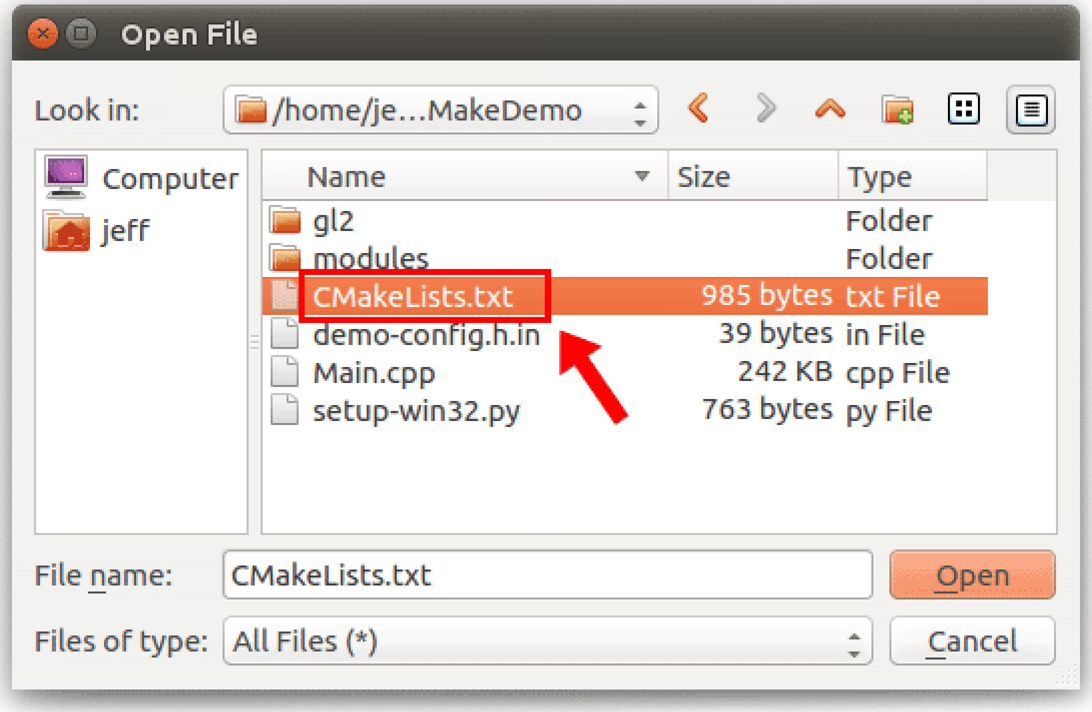
Finally, build from the Product → Build menu (or the ⌘B shortcut), run using Product → Run (or ⌘R), or click the big play button in the toolbar.

It's possible to make CMake generate an Xcode project that builds a MacOS bundle or framework, but I didn't demonstrate that in the CMakeDemo project.

Building with Qt Creator

Qt Creator provides built-in support for CMake using the Makefile or Ninja generator under the hood. I tested the following steps in Qt Creator 3.5.1.

In Qt Creator, go to File → Open File or Project… and choose `CMakeLists.txt` from the source folder you want to build.

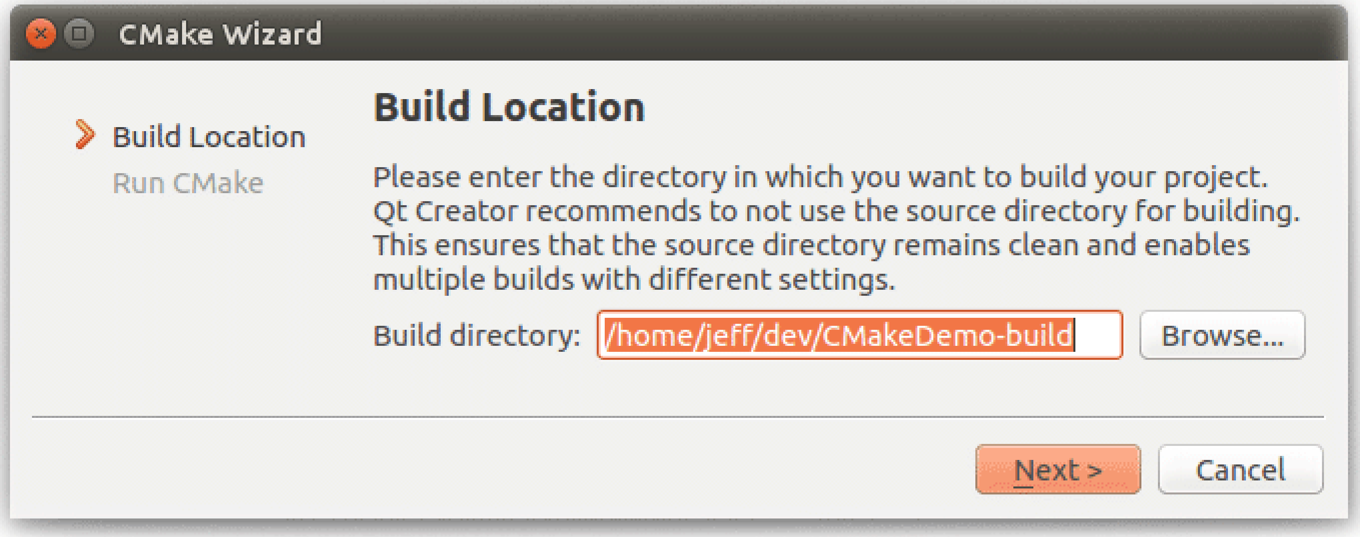


Name	Size	Type
gl2		Folder
modules		Folder
CMakeLists.txt	985 bytes	txt File
demo-config.h.in	39 bytes	in File
Main.cpp	242 KB	cpp File
setup-win32.py	763 bytes	py File

File name: CMakeLists.txt

Files of type: All Files (*)

Qt Creator will prompt you for the location of the binary folder, calling it the “build directory”. By default, it suggests a path adjacent to the source folder. You can change this location if you want.

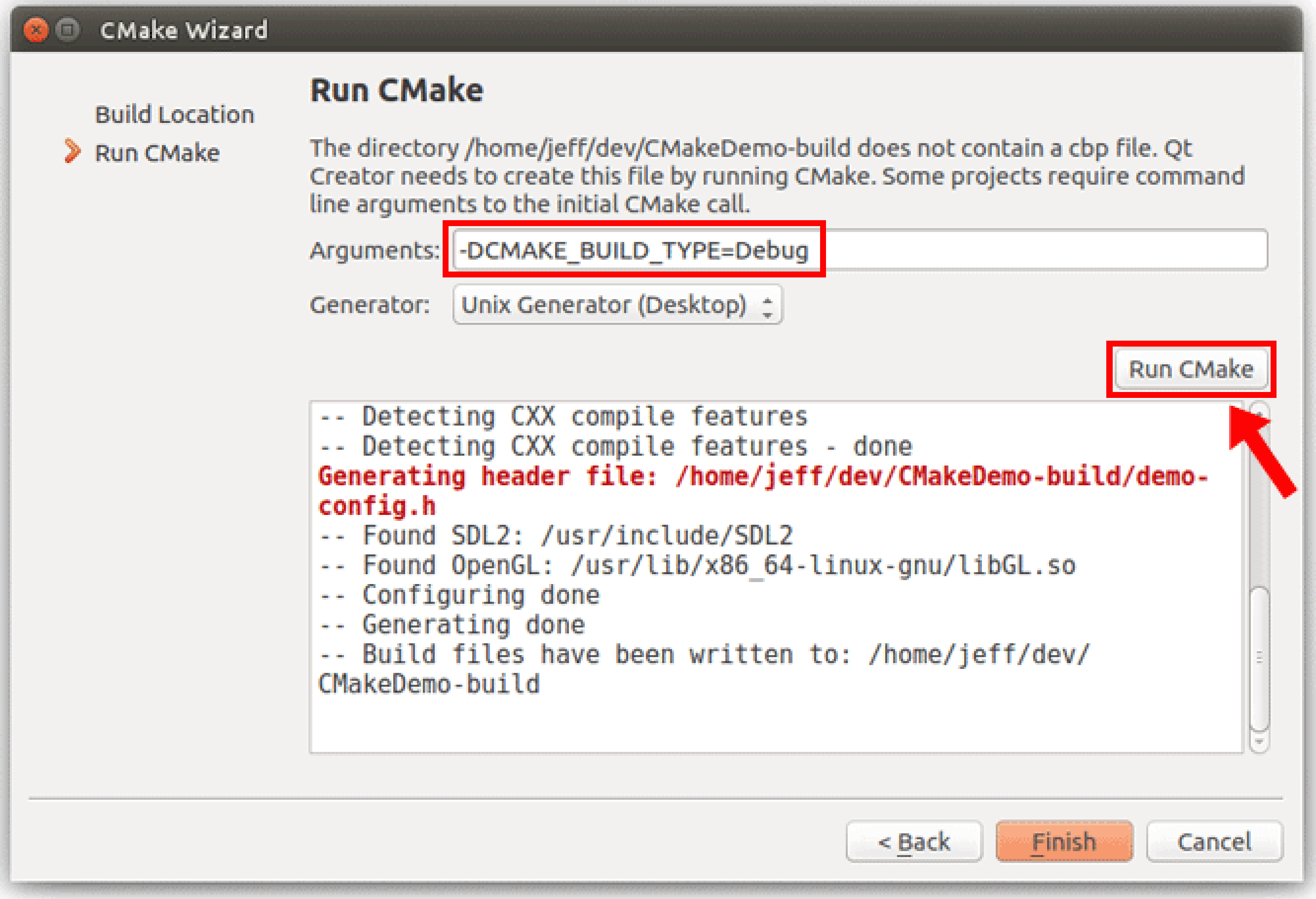


Build Location

Please enter the directory in which you want to build your project. Qt Creator recommends to not use the source directory for building. This ensures that the source directory remains clean and enables multiple builds with different settings.

Build directory: /home/jeff/dev/CMakeDemo-build

When prompted to run CMake, make sure to define the `CMAKE_BUILD_TYPE` variable since the Makefile generator is single-configuration. You can also specify project-specific variables here, such as CMakeDemo’s `DEMO_ENABLE_MULTISAMPLE` option.



Run CMake

The directory /home/jeff/dev/CMakeDemo-build does not contain a cbp file. Qt Creator needs to create this file by running CMake. Some projects require command line arguments to the initial CMake call.

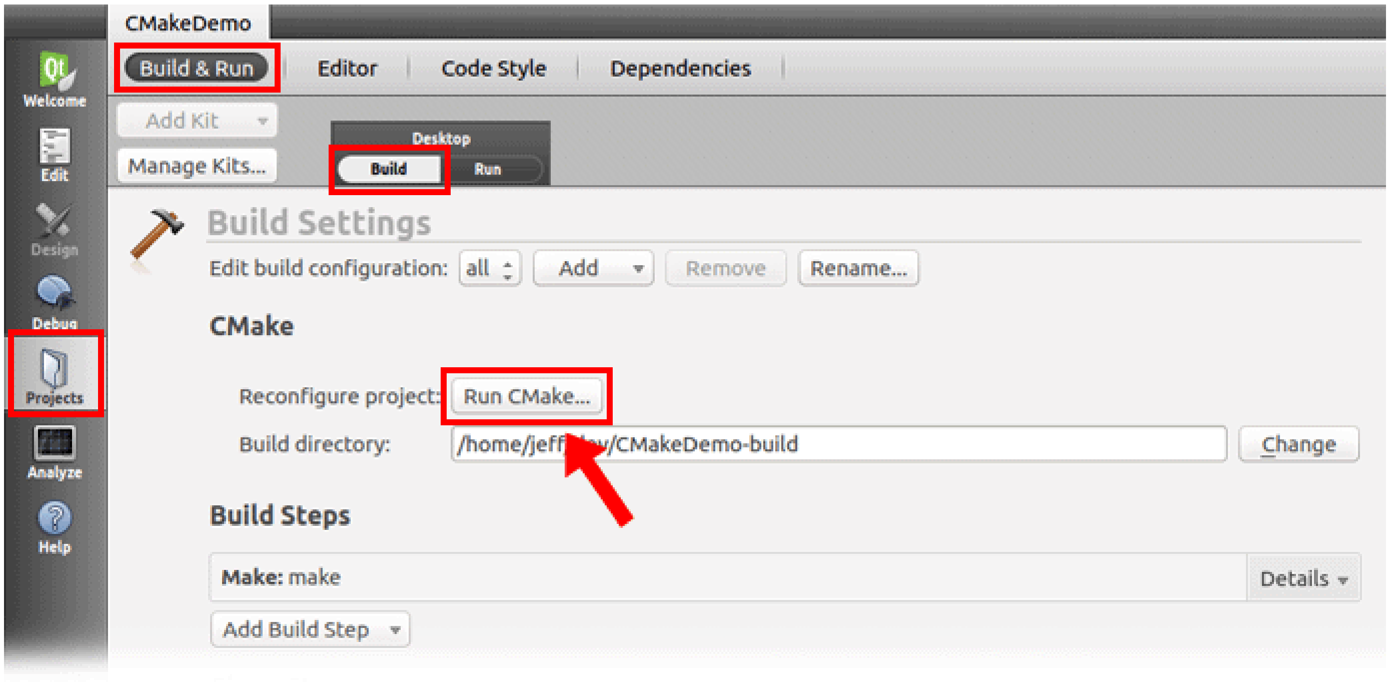
Arguments: -DCMAKE_BUILD_TYPE=Debug

Generator: Unix Generator (Desktop)

```
-- Detecting CXX compile features
-- Detecting CXX compile features - done
Generating header file: /home/jeff/dev/CMakeDemo-build/demo-
config.h
-- Found SDL2: /usr/include/SDL2
-- Found OpenGL: /usr/lib/x86_64-linux-gnu/libGL.so
-- Configuring done
-- Generating done
-- Build files have been written to: /home/jeff/dev/
CMakeDemo-build
```

After that, you can build and run the project from Qt Creator's menus or using the Shift+Ctrl+B or F5 shortcuts.

If you want to re-run CMake, for example to change the build type from Debug to RelWithDebInfo, navigate to Projects → Build & Run → Build, then click "Run CMake".



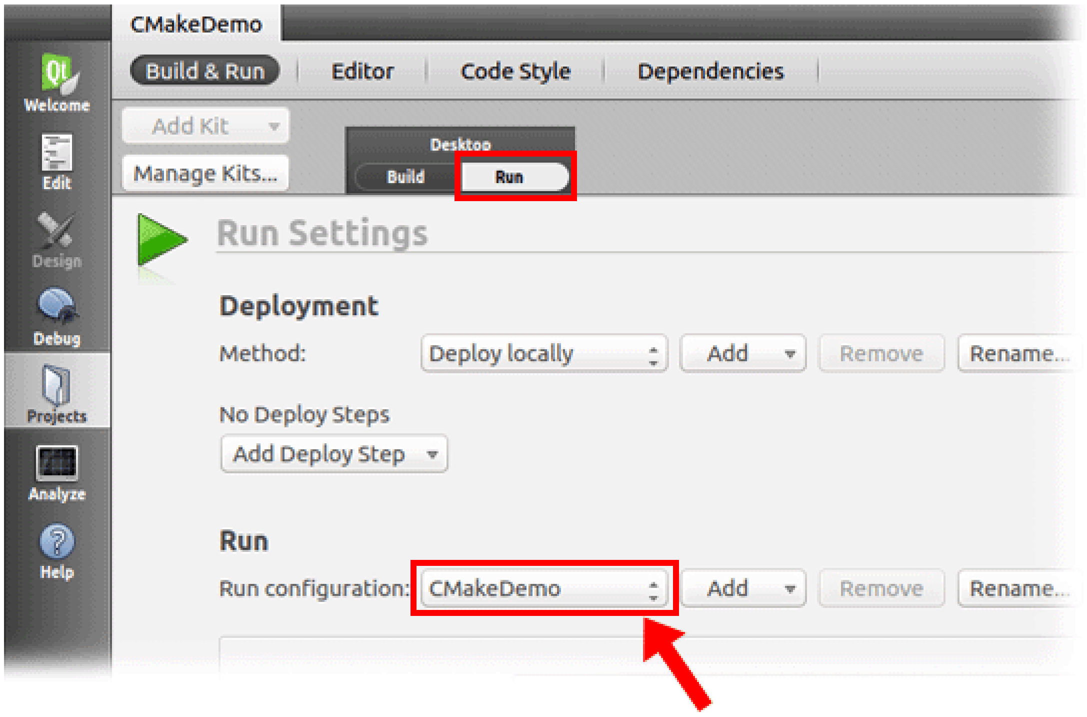
Edit build configuration: all

Reconfigure project: Run CMake...

Build directory: /home/jeff/dev/CMakeDemo-build

Make: make

The CMakeDemo project contains a single executable target, but if your project contains multiple executable targets, you can tell Qt Creator which one to run by navigating to Projects → Build & Run → Run and changing the "Run configuration" to something else. The drop-down list is automatically populated with a list of executable targets created by the build pipeline.



Run Settings

Deployment

Method: Deploy locally

No Deploy Steps

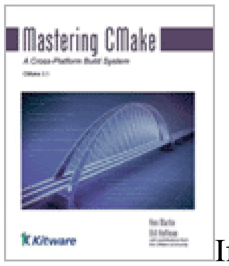
Run

Run configuration: CMakeDemo

Other CMake Features

- You can perform a build from the command line, regardless of the generator used: `cmake --build . --target CMakeDemo --config Debug`
- You can create build pipelines that cross-compile for other environments with the help of the `CMAKE_TOOLCHAIN_FILE` variable.
- You can generate a `compile_commands.json` file that can be fed to Clang's LibTooling library.

I really appreciate how CMake helps integrate all kinds of C/C++ components and build them in all kinds of environments. It's not without its flaws, but once you're proficient with it, the open source world is your oyster, even when integrating non-CMake projects. My next post will be a crash course in CMake's scripting language.


If you wish to become a power user, and don't mind forking over a few bucks, the authors' book Mastering CMake offers a big leap forward. Their article in The Architecture of Open Source Applications is also an interesting read.

« Using Quiescent States to Reclaim Memory Learn CMake's Scripting Language in 15 Minutes »

Comments (16)

Commenting Disabled
Further commenting on this page has been disabled by the blog admin.

Yury · *111 weeks ago*

There's an option -H that's not described in the help screen. It allows you to replace

```
mkdir build
cd build
cmake -G "Visual Studio 15 2017" ..
```

with just this

```
mkdir build
cmake -H. -Bbuild
```

It makes writing build scripts for CI much easier.

Reply

Tobias · *110 weeks ago*

The Qt Creator instructions are seriously outdated: It's cmake support has improved a lot since 3.5.1 -- especially in the upcoming 4.3 version.

Reply **1 reply** · *active 110 weeks ago*

Jeff Preshing · *110 weeks ago*

So I've heard! I wish I used a more recent version for this post, but now that it's done, I'll probably just leave it as is.

Reply

Peter Feichtinger · *110 weeks ago*

Just wanted to let you know that I don't have this post in the RSS feed (on feedly).

Reply **1 reply** · *active 110 weeks ago*

Jeff Preshing · *110 weeks ago*

I know. That's how Feedly treats blogs that have been dormant for a while, even if you click their own Refresh button, which makes no sense. I think we just have to keep waiting.

Reply

Dave · *110 weeks ago*

This apparently crashes when I compile with Visual Studio 2017 Enterprise x64 at the following location. It is issuing an abort(); I haven't done any GL programming therefore something may be missing here on my end.

DrawArraysInstancedNV

Reply **5 replies** · *active 107 weeks ago*

· *110 weeks ago*

Jeff Preshing

I probably need to make the demo more compatible. Could you open an issue on GitHub and copy/paste the last few lines of your Output window starting with "INFO: Vendor"? Also the full callstack. Thanks!

Reply

Dave · *110 weeks ago*

I did a release build and it worked. I evidently just got an assertion.

Assertion failed: DrawArraysInstancedNV, file c:usersdjryadocumentsgithubcmakedemogl2Funcs.h, line 22

Reply

Jeff Preshing · *110 weeks ago*

OK. I just pushed a commit that should remove the assert in Debug. If you could test it, that would be great.

Reply

Dave · *110 weeks ago*

Thanks; I will.

Dave

Reply

Dave · *110 weeks ago*

Works great; thank you very much.

Dave

Reply

Russel F. · *82 weeks ago*

AH! Whoo Hoo!!! HOLY CRAP! I drew a f***ing graph!!!!! (If you knew how many days I've spent on this foolishness...)
Jeff, thanx again. Your tutorial was enough to let me figure out how to run the configure for libzmq! And I was able to compile it. And then, I was able to use the same tricks to compile pyzmq. (this is critical stuff I cloned from github..) ANyway, it bloody worked! After I had installed both the libzmq and the pyzmq, I was finally able to compile Jupyter using pip install jupyter, and IT FINALLY WORKDED. I was within minutes of throwing in the towel here - but once I had Jupyter going, I was able to start interactive Python with "jupyter notebook" and (since I have finally gotten a modern Firefox to work), I saw the webserver had started, and I was looking at http://localhost:8888, and the iPython/Jupyter dashboard! Press the "new" button to gen a new notebook, and you get the familiar interactive ipython prompt - but inside a webpage on the browser. I already had matplotlib installed - and a few lines of python code, and OMG a GRAPHIC! I am amazed. (I am using a custom-built early Fedora kernel - very, very homebuilt..). But the key is - it works! (I am using Python 2.7.14 and gcc 4.3.0. ...) It all works. And it is thanx to your excellent tutorial here. The cmake thing is just bizarre - but I suppose it is not as awful now that I can use it. Oh, please do me a favour, and delete my awful rant in the previous post... (All day - a whole day from dawn to dark - solving the problems on this Linux laptop - but rewarded with success. (I have a TensorFlow thing I have to do...)) Your tutorial really helped. How much is your book?

Reply

Pierric · *81 weeks ago*

Great post! Typo: -DDEMO_ENABLE_MULTISAMPLING= should be -DDEMO_ENABLE_MULTISAMPLE=

Reply

William P. · *63 weeks ago*

Thank you for your great article ! It helped me alot in getting to know Cmake better.

One thing though: Could you elaborate on why you find the Release configuration useless and even delete it in favor of RelWithDebInfo? Or could you link to a blog entry where you explain this?

I see why "Release" does not make much sense during development but once you want to actually release your software you do not want any Debug information bloating your executable. Doesn't enabling Debugging in a build also slow down your executable by some degree (e.g. by preventing certain optimizations) ?

I always thought that in a Release build you do not want to have any Debugging enabled (for performance reasons, ignoring reverse engineering etc. right now). Please correct me if im wrong.

Reply **1 reply** · *active 62 weeks ago*

Jeff Preshing · *63 weeks ago*

The best practice is to always build your executable with debug information, even optimized executables intended for release. On Windows, the debug information is contained in a separate file (.pdb) and on Unix-like OSes, you can strip the debug information out of the executable using "strip". When it's time to release your software, release the stripped executable but keep a copy with debug information in your archives.

Debug information doesn't slow down the executable. It's just information. You can still build a fully optimized executable. When the executable is optimized, it's (much) harder to debug, but debug information still helps greatly, because it lets the debugger show you the callstack, global variables, and you can cast any address to a C/C++ type to inspect its contents.

This is common practice in the game industry and important for any application that relies on crash reporting, for example using Google Breakpad.

Reply

Brett Woodard · *58 weeks ago*

Jeff,

Thank you so much for writing this article and the article Learn CMake's Scripting Language in 15 Minutes. They both helped a lot in teaching me the basics of CMake.

Cheers,
-Brett

Reply

Recent Posts

- A Flexible Reflection System in C++: Part 2
- A Flexible Reflection System in C++: Part 1
- How to Write Your Own C++ Game Engine
- Can Reordering of Release/Acquire Operations Introduce Deadlock?
- Here's a Standalone Cairo DLL for Windows
- Learn CMake's Scripting Language in 15 Minutes

- How to Build a CMake-Based Project
- Using Quiescent States to Reclaim Memory
- Leapfrog Probing
- A Resizable Concurrent Map
- New Concurrent Hash Maps for C++
- You Can Do Any Kind of Atomic Read-Modify-Write Operation
- Safe Bitfields in C++
- Semaphores are Surprisingly Versatile
- C++ Has Become More Pythonic
- Fixing GCC's Implementation of memory_order_consume
- How to Build a GCC Cross-Compiler
- How to Install the Latest GCC on Windows
- My Multicore Talk at CppCon 2014
- The Purpose of memory_order_consume in C++11

Tip Jar

If you like this blog, leave a tip!