So when you have Visual Studio 2019 and using VC++ compilers like MSVC () you will not create a Makefile when you run cmake. I’m guessing.

How to build Visual Studio on command line and inside Visual Studio 2019 GUI

After

**cmake ..**

type

**cmake –build .**

Or open Visual Studio 2019 , open \*.sln solution file

# Big Question Answered

All the various ways to run the make file after running **cmake ..**

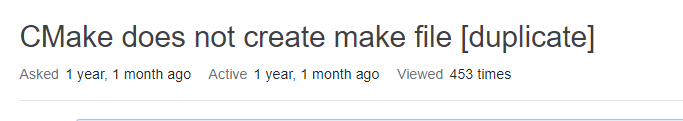
**Classic CMake Build Procedure**

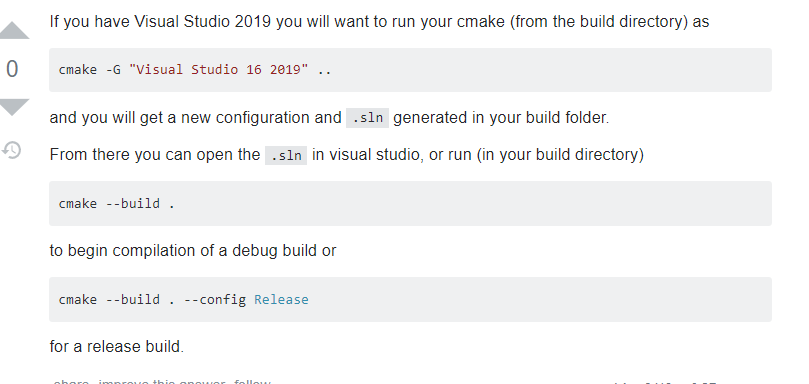
|  |
| --- |
| **mkdir build**  **cd build**  **cmake ..**  **make** |

YOU can **REPLACE** the make line with

|  |
| --- |
| cmake –build . |

Instead, after running cmake (with associated CMakeLists.txt) a \*.sln is created (the \*.vcproj tell you what build flags were set due to cmake…this is what replaces the Makefile)





If you have Visual Studio 2019 you will want to run your cmake (from the build directory) as

cmake -G "Visual Studio 16 2019" ..

and you will get a new configuration and .sln generated in your build folder.

From there you can open the .sln in visual studio, or run (in your build directory)

cmake --build .

to begin compilation of a debug build or

cmake --build . --config Release

for a release build.

## Building a project

Unless otherwise noted, you should always make a build directory and build from there. You can technically do an in-source build, but you'll have to be careful not to overwrite files or add them to git, so just don't.

Here's the Classic CMake Build Procedure (TM):

~/package $ mkdir build

~/package $ cd build

~/package/build $ cmake ..

~/package/build $ make

You can replace the make line with cmake --build . if you'd like, and it will call make or whatever build tool you are using. If you are using a newer version of CMake (which you usually should be, except for checking compatibility with older CMake), you can instead do this:

~/package $ cmake -S . -B build

~/package $ cmake --build build

Any one of these commands will install:

# From the build directory (pick one)

~/package/build $ make install

~/package/build $ cmake --build . --target install

~/package/build $ cmake --install . # CMake 3.15+ only

# From the source directory (pick one)

~/package $ cmake --build build --target install

~/package $ cmake --install build # CMake 3.15+ only

So set of methods should you use? As long as you do not forget to type the build directory as the argument, staying out of the build directory is shorter, and making source changes is easier from the source directory. You should try to get used to using --build, as that will free you from using only make to build. Note that working from the build directory is historically much more common, and some tools and commands (including CTest) still require running from the build directory.

Just to clarify, you can point CMake at either the source directory from the build directory, or at an existing build directory from anywhere.

If you use cmake --build instead of directly calling the underlying build system, you can use -v for verbose builds (CMake 3.14+), -j N for parallel builds on N cores (CMake 3.12+), and --target (any version of CMake) or -t (CMake 3.15+) to pick a target. Otherwise, these commands vary between build systems, such as VERBOSE=1 make and ninja -v.

## Picking a compiler

Selecting a compiler must be done on the first run in an empty directory. It's not CMake syntax per se, but you might not be familiar with it. To pick Clang:

~/package/build $ CC=clang CXX=clang++ cmake ..

That sets the environment variables in bash for CC and CXX, and CMake will respect those variables. This sets it just for that one line, but that's the only time you'll need those; afterwards CMake continues to use the paths it deduces from those values.

## Picking a generator

You can build with a variety of tools; make is usually the default. To see all the tools CMake knows about on your system, run

~/package/build $ cmake --help

And you can pick a tool with -G"My Tool" (quotes only needed if spaces are in the tool name). You should pick a tool on your first CMake call in a directory, just like the compiler. Feel free to have several build directories, like build/ and buildXcode. You can set the environment variable CMAKE\_GENERATOR to control the default generator (CMake 3.15+). Note that makefiles will only run in parallel if you explicilty pass a number of threads, such as make -j2, while Ninja will automatically run in parallel. You can directly pass a parallelization option such as -j2 to the cmake --build . command in recent versions of CMake.

## Setting options

You set options in CMake with -D. You can see a list of options with -L, or a list with human-readable help with -LH. If you don't list the source/build directory, the listing will not rerun CMake (cmake -L instead of cmake -L .).

## Verbose and partial builds

Again, not really CMake, but if you are using a command line build tool like make, you can get verbose builds:

~/package/build $ VERBOSE=1 make

You can actually write make VERBOSE=1, and make will also do the right thing, though that's a feature of make and not the command line in general.

You can also build just a part of a build by specifying a target, such as the name of a library or executable you've defined in CMake, and make will just build that target.

## Options

CMake has support for cached options. A Variable in CMake can be marked as "cached", which means it will be written to the cache (a file called CMakeCache.txt in the build directory) when it is encountered. You can preset (or change) the value of a cached option on the command line with -D. When CMake looks for a cached variable, it will use the existing value and will not overwrite it.

### Standard options

These are common CMake options to most packages:

* -DCMAKE\_BUILD\_TYPE= Pick from Release, RelWithDebInfo, Debug, or sometimes more.
* -DCMAKE\_INSTALL\_PREFIX= The location to install to. System install on UNIX would often be /usr/local (the default), user directories are often ~/.local, or you can pick a folder.
* -DBUILD\_SHARED\_LIBS= You can set this ON or OFF to control the default for shared libraries (the author can pick one vs. the other explicitly instead of using the default, though)
* -DBUILD\_TESTING= This is a common name for enabling tests, not all packages use it, though, sometimes with good reason.

## Debugging your CMake files

We've already mentioned verbose output for the build, but you can also see verbose CMake configure output too. The --trace option will print every line of CMake that is run. Since this is very verbose, CMake 3.7 added --trace-source="filename", which will print out every executed line of just the file you are interested in when it runs. If you select the name of the file you are interested in debugging (usually by selecting the parent directory when debugging a CMakeLists.txt, since all of those have the same name), you can just see the lines that run in that file. Very useful!