EEEE2076 -Software Development Group Design Project

Worksheet 4 - Software Development with CMake

P Evans

January 24, 2025

Contents

1 Automating the Build - Build Systems

For a program with many source files and many library dependencies, a large number of long commands must be sent to the compiler and linker. These need to be automated in some way, in the past you will have used Code::Blocks and automatically running the compile and build processes was handled by the Code::Blocks IDE. Visual Studio can work in the same way.
The problem with this approach is that if you create a project for Code::Blocks or Visual Studio on your computer and you send it to your friend's computer, it is very unlikely to work. This is because paths to software, and the versions of software, installed on the two computers are likely to be different.
In this example, Visual Studio or Code::Blocks are what is known as a build system - they are following a list of instructions (the project) to build your software, however the instructions must be written specifically for the computer which is building the software.

2 Software Development in Teams

Git allows you to share code between your group members for team software development. Each team member will need to be able to build your software project. The project will be quite large and complicated by the end, and need to link to many libraries from Qt and VTK. If will be very annoying if you have to either: type out all build commands by hand each time you build the software, or recreate/adapt a Visual Studio project to work with your computer each time a group member updates the Github project.

You need a way of defining a Visual Studio project in a generic way on the Github repository, e.g. describing

the compile and linking steps that must be followed to build the software, but without hardcoding any paths that might change between different computers. This generic project could then be automatically adapted with specific paths for a system just before the software is built. This process is achieved by *Build System Generators* such as *CMake*.
CMake allows you to write a simple, generic project description that goes with your project on Github. The CMake program is able to translate the generic description into a fully functional Visual Studio project for a specific computer, it does this (mostly) automatically by searching for the correct paths to use. It can also translate the same generic description into projects for other build systems or IDEs, e.g. Code::Blocks, or for the Make build system on Linux / MacOS.

CMake works as a three-stage process:

- *Configure* CMake takes the generic build instructions and attempts to find the paths for your specific system that it needs. It might need help to do this.
- *Generate* CMake generates a project for your build system (Visual Studio in our case)
- *Build* You then open the project in Visual Studio and compile using the Visual Studio GUI.

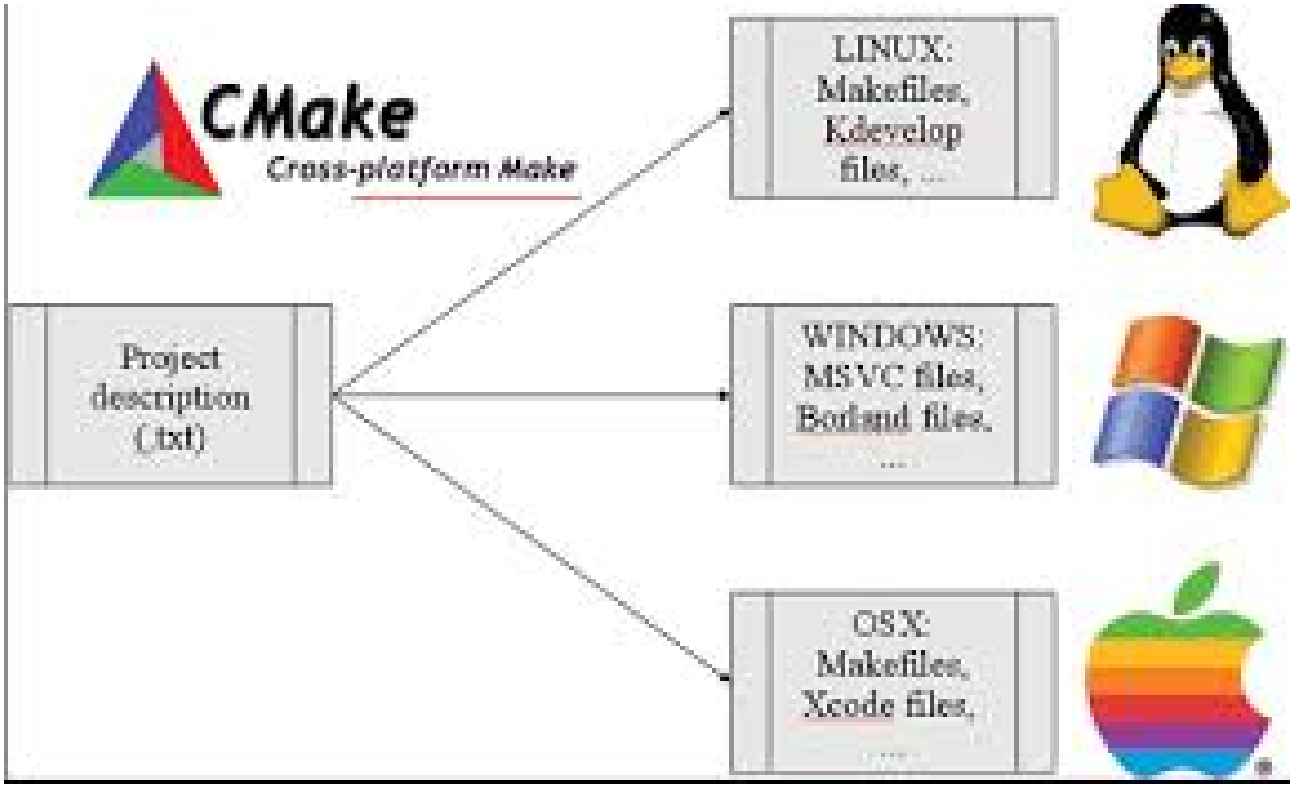

Figure 1: CMake Targets]

3 CMake - Basic Example

3.1 Exercise 1 - CMake from the command line

Download and install CMake, make sure that the path to *cmake.exe* program is present in your system PATH environment variable. Create a new directory in your Individual repo called Worksheet4 and a subdirectory called *hello_cmake* and finally make a copy of *hello.cpp* from the previous worksheet in this directory.

Now create an additional file (in the same directory as the new hello.cpp) called *CMakeLists.txt*, put the following text in it:

```
# hello_cmake/CMakeLists.txt

cmake_minimum_required(VERSION 3.12 FATAL_ERROR)
project( hello )
add_executable( hello hello.cpp )

# /hello_cmake/CMakeLists.txt
```

Note that we have define the name of our program but there's no mention of .exe and there's no mention of compiler or linker program names as both of these things are platform or build system dependent. Now we need to invoke cmake on the project. To do this, cmake needs to know which build system it should generate input for. A wide variety of build systems and other options are supported and you see options by typing:

```
C:\Users\ezzpe>cmake --help
Usage

cmake [options] <path-to-source>
cmake [options] <path-to-existing-build>
cmake [options] -S <path-to-source> -B <path-to-build>

...

Generators

The following generators are available on this platform (* marks default):
* Visual Studio 17 2022        = Generates Visual Studio 2022 project files.
Use -A option to specify architecture.
Visual Studio 16 2019         = Generates Visual Studio 2019 project files.
...

C:\Users\ezzpe>
```

You can see a range of build systems that CMake can generate projects for, on my system the default is Visual Studio 2022 and so to get VS22 project I can just type:

```
C:\Users\ezzpe>cmake .
```

If you wanted to generate a project for a different version of Visual Studio, or a different IDE like Code::Blocks you would use a command similar to:

```
C:\Users\ezzpe>cmake -G "CodeBlocks - MinGW Makefiles" .
```

This would work for Codeblocks but you can change the build system name to anything in the list that CMake gave you in the previous step, names are case-sensitive.

These commands will generate the Visual Studio project in the current directory which is not a good idea because now the source code (which you do want to add to git) and the project (which you don't want to add) are mixed up. The solution is to generate the project in its own "build" directory. Delete the project that was just created (delete all the VS files, and the directory that was created) and type:

```
C:\Users\ezzpe\2022_ezzpe\Worksheet4\hello_cmake>mkdir build
C:\Users\ezzpe\2022_ezzpe\Worksheet4\hello_cmake>cd build
C:\Users\ezzpe\2022_ezzpe\Worksheet4\hello_cmake\build>cmake ..
```

This achieves the same result but separates the VS project and source code. To build this project, you have two options. Firstly you can do it from the command line, you need to run the vcvarsall.bat script to setup the compiler and then use msbuild to process the project that was created. Note that a project for Visual Studio is called a Solution and is described in a file with a .sln file extension.

```
C:\Users\ezzpe\2022_ezzpe\Worksheet4\hello_cmake\build>"C:\Program Files\Microsoft
    Visual Studio\2022\Community\VC\Auxiliary\Build\vcvarsall.bat" x64
C:\Users\ezzpe\2022_ezzpe\Worksheet4\hello_cmake\build>msbuild hello.sln
```

Alternatively, you can open the .sln file by clicking on it to open it in the GUI. When Visual Studio opens, go to the project tree view - there will be a number of targets: "ALL_BUILD", "hello" and "ZERO_CHECK". Right click on hello and select "Set as Startup Project". This target corresponds to the hello.exe file that will be build, and you are just telling Visual Studio that this is what you want it to build and run. Once you have done this you can click the Run / Local Windows Debugger button to build and run.

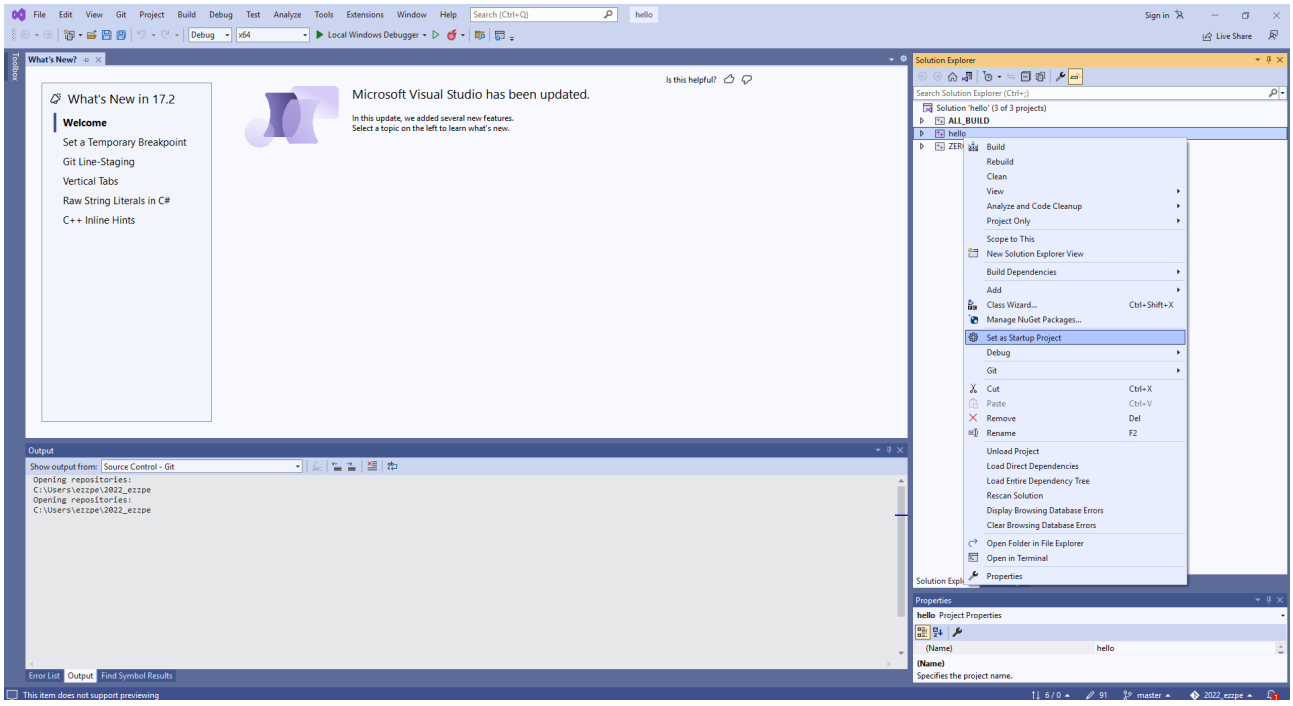

Figure 2: Building in Visual Studio

3.2 Exercise 2 - CMake GUI

There is an easier way to use CMake - it has a GUI. Try to use this now, delete everything in your build folder first. Now open CMake-GUI (Windows start menu / search bar). There are two paths you need to set, "path to source" and "path to build folder" - set these to your source folder (the one that contains the CMakeLists.txt) and the build folder. You then just need to click the three buttons at the bottom in sequence - configure, generate, open project. Note that the initial CMake run is now split into configure and generate steps, the idea is that after you have configured there is the opportunity to review and edit the paths/settings that CMake is going to use when it generates the project. This will be important when we build VTK and your application that uses VTK later.

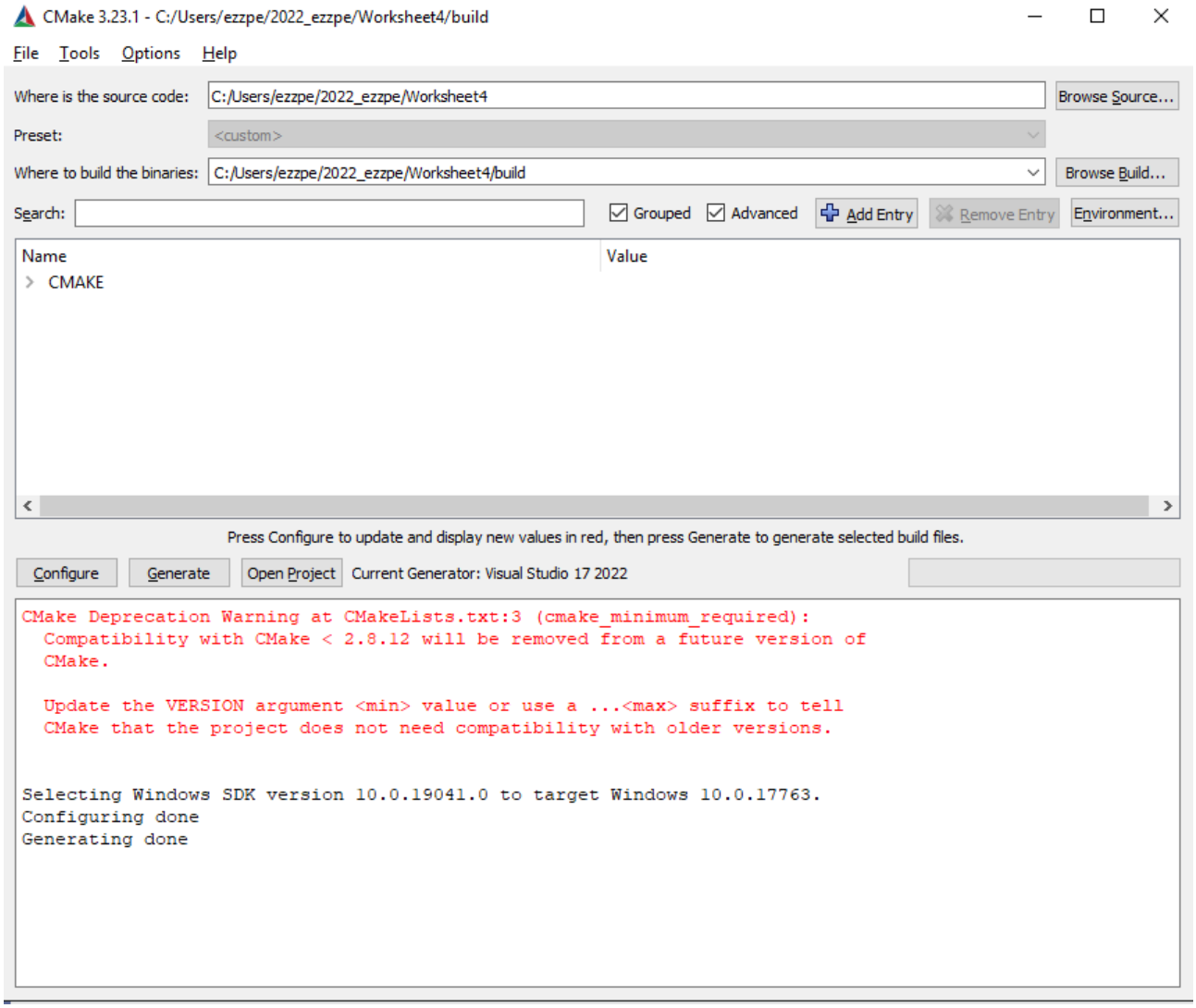

Figure 3: Using CMake GUI

4 CMake - Libraries

4.1 Exercise 3 - Static Calc

Create a new folder called *Calc* and create a copy of your *adder.cpp*, *adder.h*, *calc.cpp*. Add the following file as CMakeLists.txt and build (just use CMake-GUI from now on) unless you particularly want to use the command line.

```
# calc_cmake/CMakeLists.txt
cmake_minimum_required(VERSION 3.12 FATAL_ERROR)
project( calc )

# This project contains a library
add_library( maths adder.cpp )
# Note: you could force CMake to always build a static/shared library:
#add_library( maths STATIC adder.cpp )
#add_library( maths SHARED adder.cpp )

if( NOT BUILD_SHARED_LIBS )
# if static libs are compiled we need
# to somehow communicate that to the
# source code. The definition will be
# used to determine how MATHSLIB_API
# is defined in adder.h
add_definitions( -Dmaths_STATIC )
endif()

# It also contains an executable
add_executable( calc calc.cpp )

# The executable uses the library
target_link_libraries( calc maths )

# /calc_cmake/CMakeLists.txt
```

Go to your build directory and see what has been created - you should see the executable (.exe) and static library (.lib).

Commit your changes and upload the binaries as a release

4.2 Exercise 4 - Dynamic Calc

If you look at the CMakeLists file above, you'll see CMake checks to see if there is a variable called "BUILD_SHARED_LIBS" defined, if there isn't it forces a static build. As a final exercise - use CMake-GUI to define this variable and force a dynamic build. You should be able to work out how to do this yourself! Check to make sure you have the dynamic library (.dll) and export file (.exp) in your build folder.

Commit your changes and upload the binaries as a release

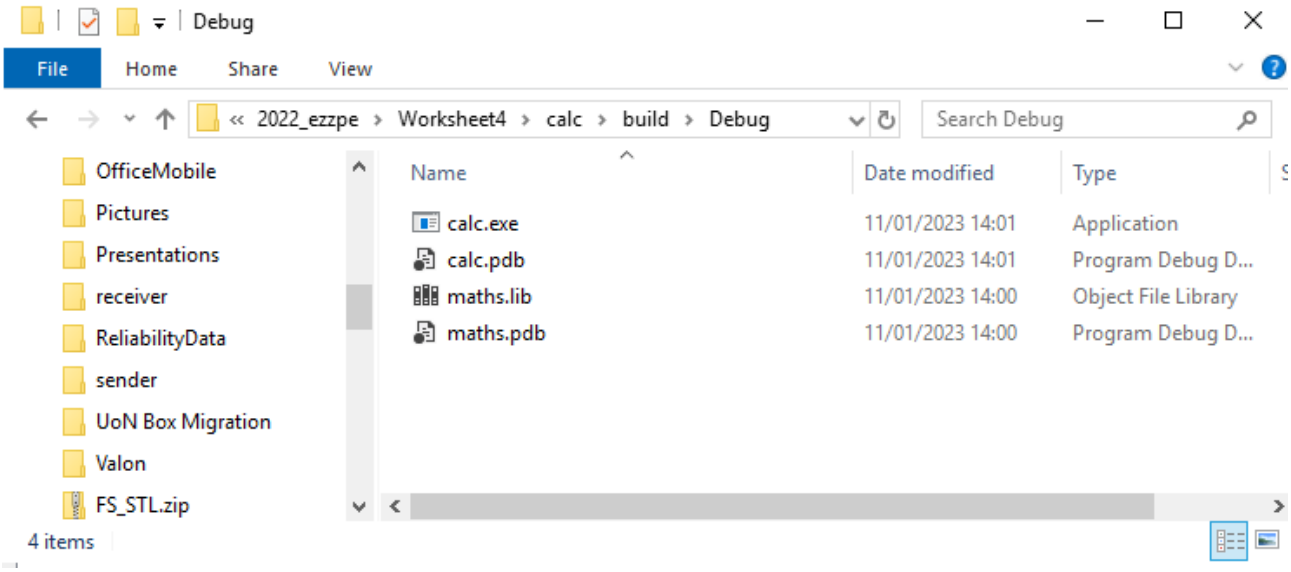

Figure 4: Output from standard CMake build of Calc program and Adder library