Visual Studio Community 2019 for Windows 10 and CMake Guide

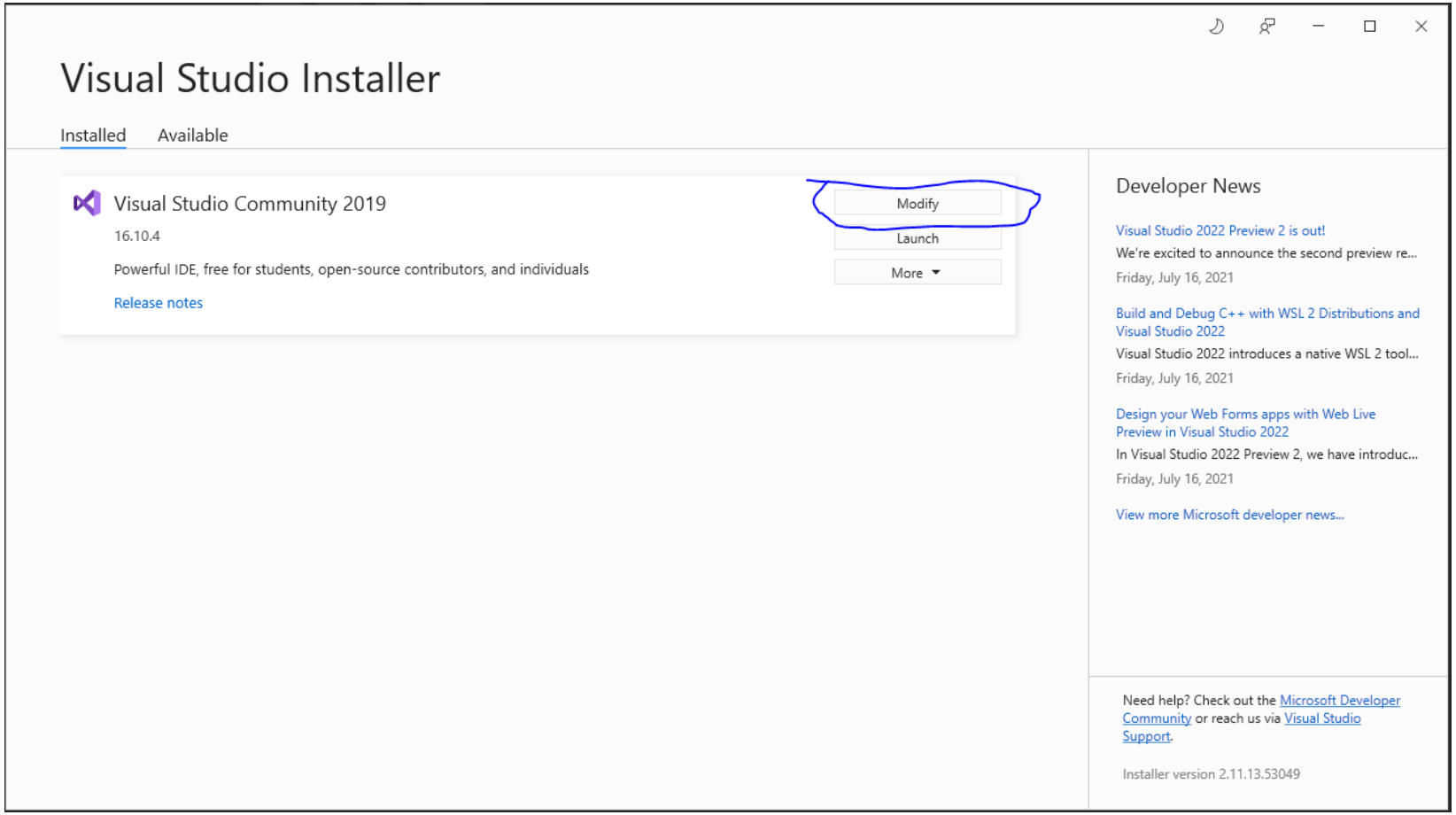
Prepared by Yevhenii Kovryzhenko

Visual Studio Setup

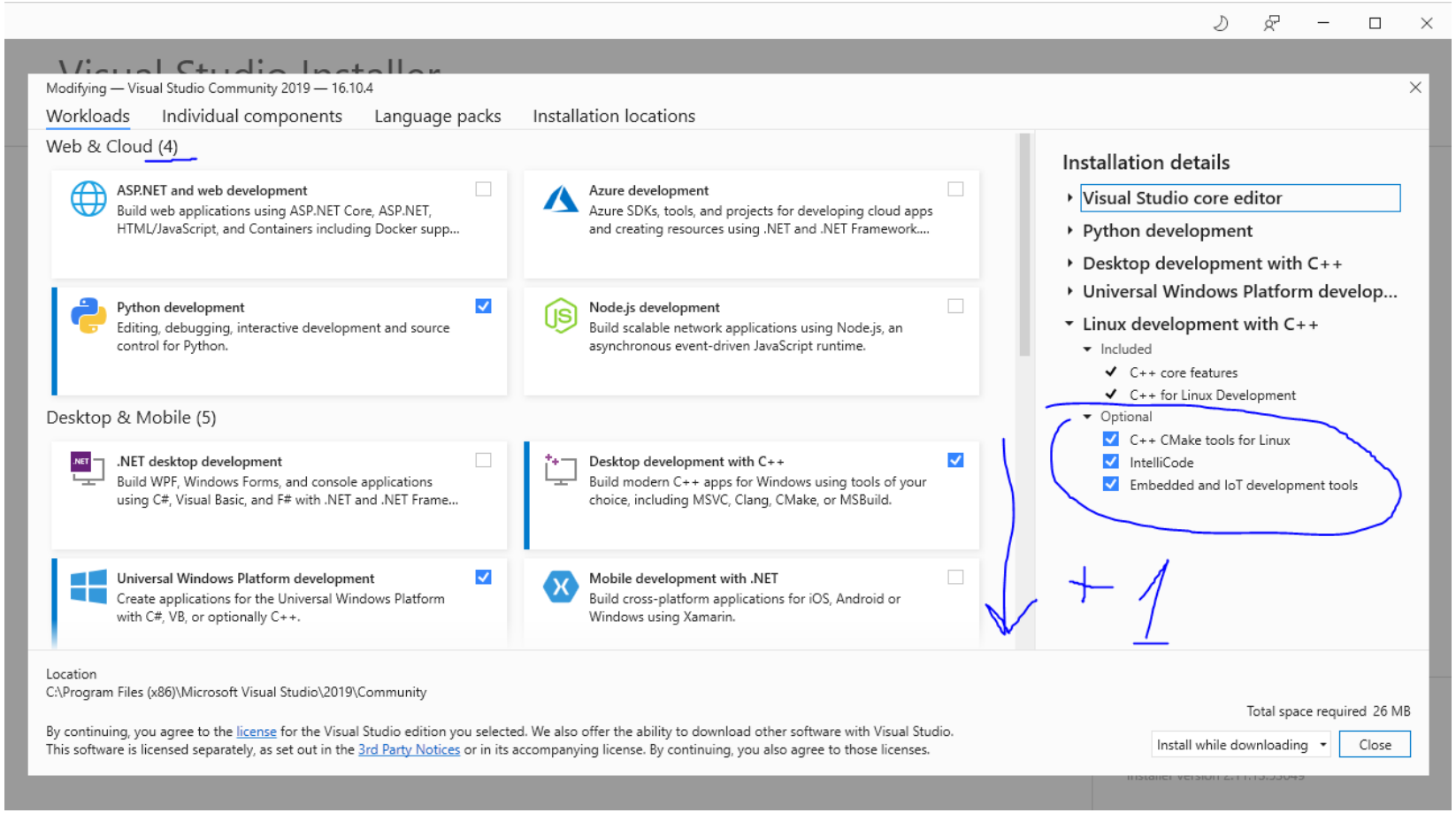
Installing Visual Studio 2019 with all the needed addons

- Get the most recent Visual Studio Community 2019 (>50gb of disk space required)
- Go to Visual Studio Installer (search among installed applications) and hit “modify” for Visual Studio Community 2019 block
- Get all the packages for working with Linux, C++ and other environment you might need:
  - **Python Development** (optional, needed if working with python as well)
  - **Desktop development with C++**
  - **Universal Windows Platform development** (optional, needed if compiling directly for Windows)
  - **Linux development with C++**
- On the right panel install optional features (anything you might think you’ll use)
  - **Linux development with C++** install all the optional features (3)

Installing Visual Studio 2019



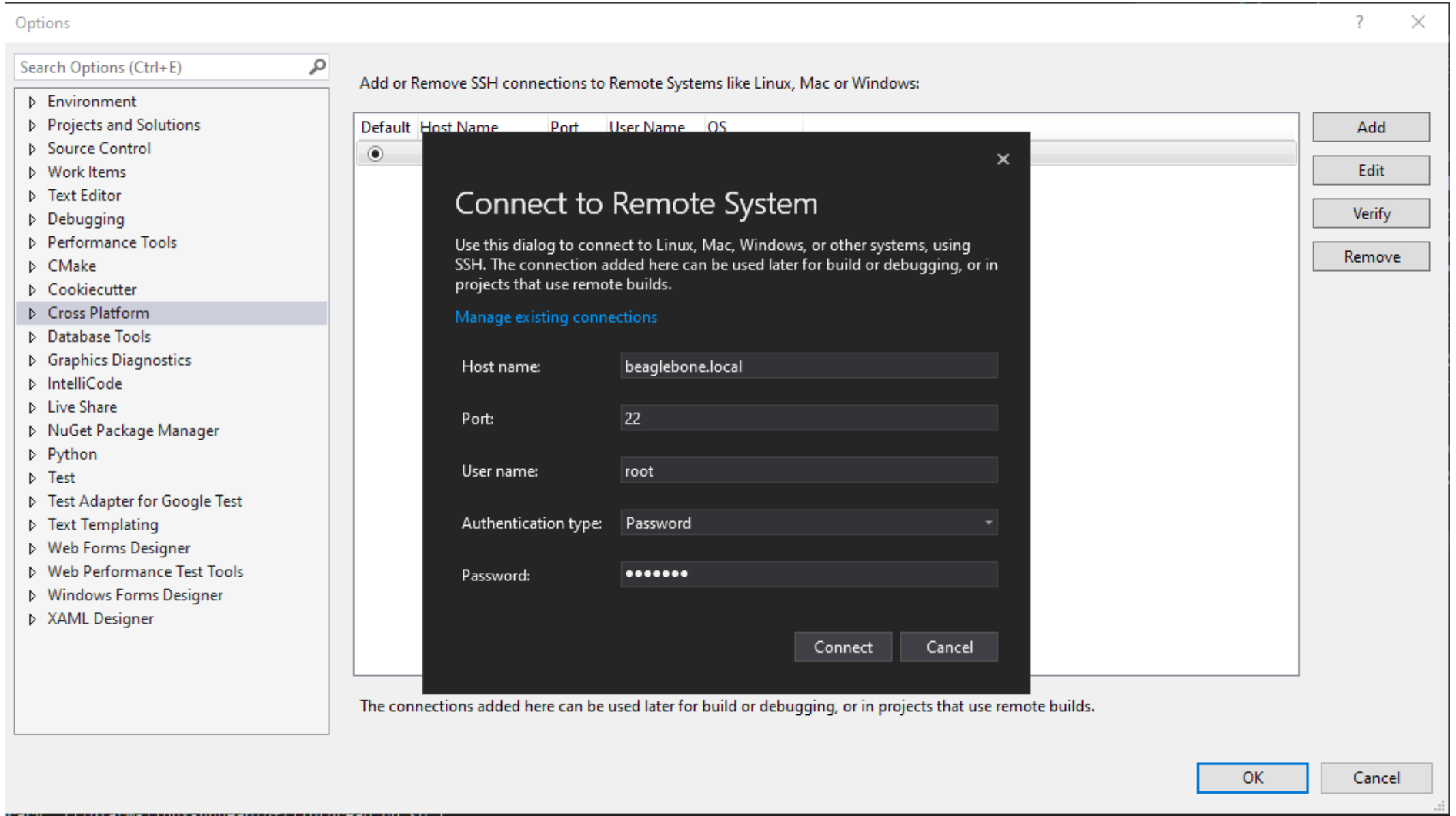
Installing Visual Studio 2019 addons



Setting up build environment within Visual Studio 2019

- Add a remote environment using ssh connection:
  - Go to **Debug-Options-Cross Platform**
    - Click on “Add” and input required parameters. Next slide shows example for BBBlue.
    - If you need to ssh as a root, don’t forget to **nano /etc/ssh/sshd_config** on your remote machine and allow login as root:
      - Change this line: **#PermitRootLogin <whatever is here>**
      - To this: **PermitRootLogin yes** (uncomment it)
- Install Linux as a subsystem on your windows PC to run compile project without external hardware but in native environment:
  - Follow this tutorial for visual studio 2019 up to step 5 of **manual installation**
  - Use **WSL 1** and **not 2** since the later version does not have embedded systems support
    - wsl --set-default-version **1**
  - Proceed with step 6, install Debian
    - this is a completely clean image, install whatever you need

Setting up build environment within Visual Studio 2019: remote system (BeagleBone Blue)



Linux environment, Cmake and Visual Studio required packages

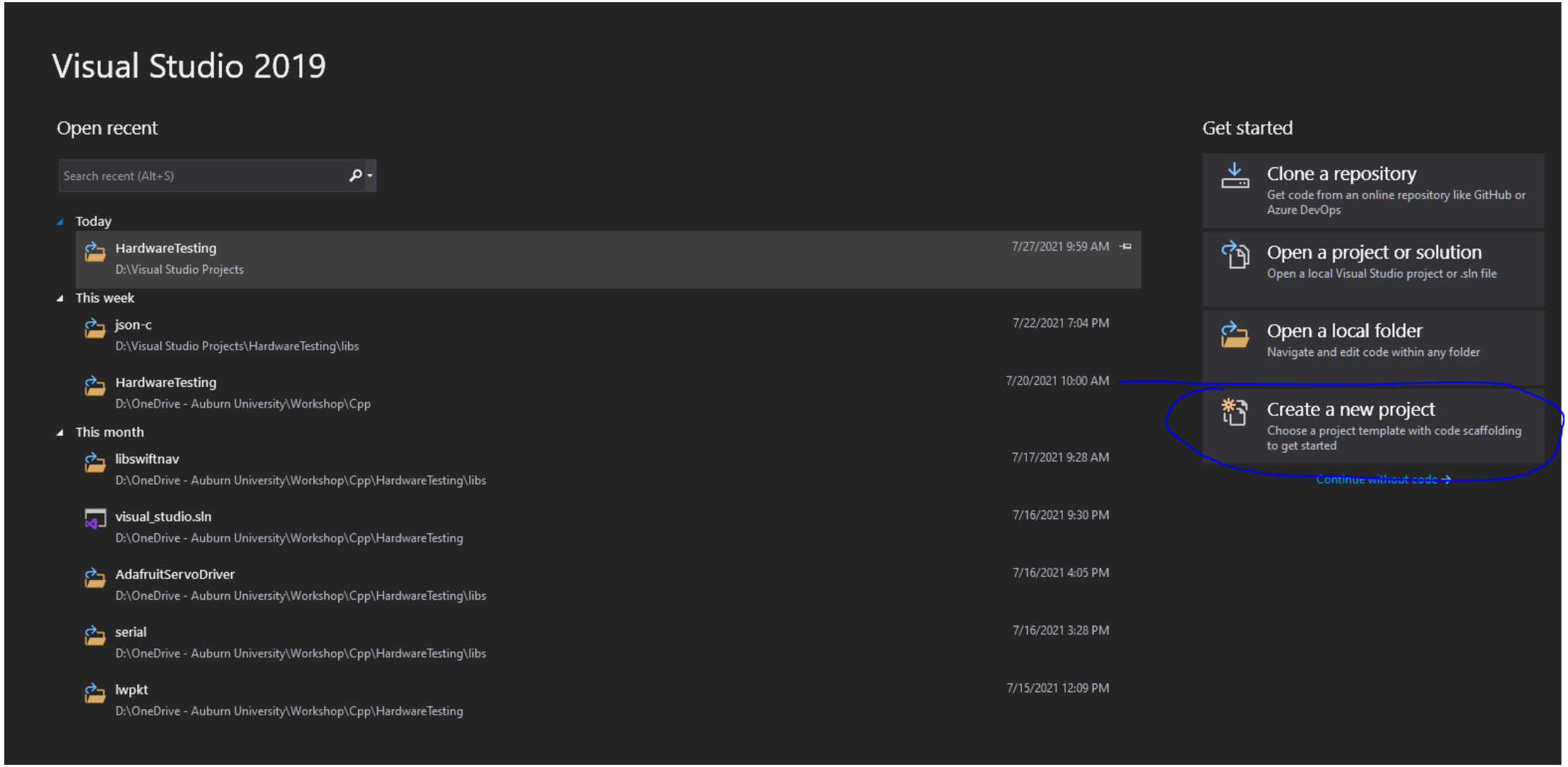
- We need to install the following packages on remote linux system and/or on WSL:
  - gcc – C/C++ compiler
    - **sudo apt-get install gcc -y**
  - gdb
    - **sudo apt-get install gdb -y**
  - rsync – used by visual studio to automatically update cmake build tree on a Linux system
    - **sudo apt-get install rsync -y**
  - Zip – used by Visual Studio, sends build files in zipped format
    - **sudo apt-get install zip -y**
  - Ninja-build – needed for Visual Studio 2019 and above
    - **sudo apt-get install ninja-build -y**
  - Open ssh server – required for remote login
    - **sudo apt-get install openssh-server –y**
  - Make – should already be installed, but just in case:
    - **sudo apt-get install make**
- One-line install:
  - **sudo apt install -y openssh-server build-essential gdb rsync make zip ninja-build**

Linux Subsystem on Windows: Embedded systems

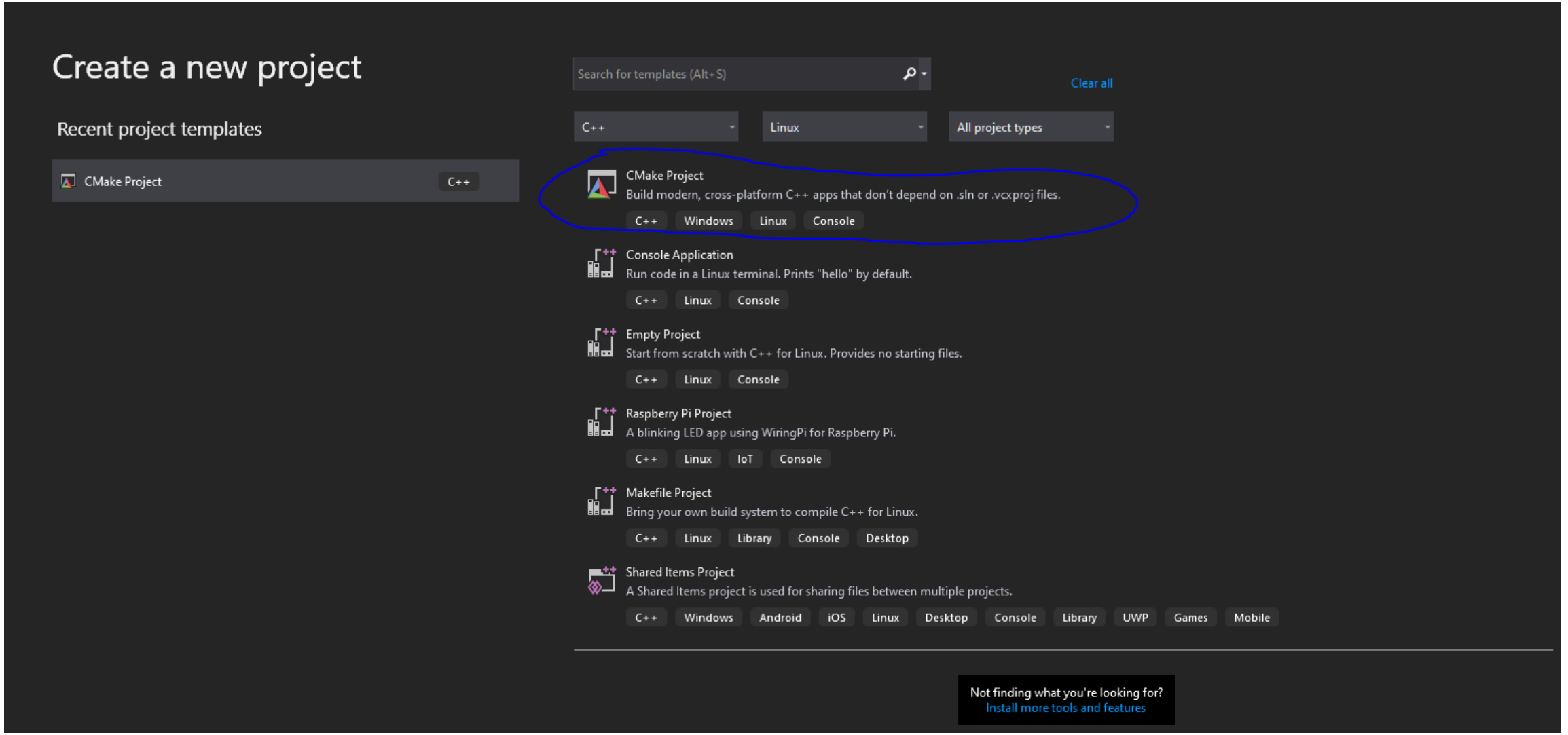
- By default, you are most likely not going to have **permissions** to access serial devices so run this:
  - **adduser $(whoami) dialout**
- For displaying the output of the serial port try accessing one in **/dev/** directory
  - All serial ports in Linux will start with **/dev/ttyS*** were the number instead of * corresponds to the same windows **COM** port
  - Let's say you connected your device to **COM11,** you can display its data using:
    - **cat /dev/ttyS11** (ctrl + c to exit)
    - **screen /dev/ttyS11 96800** (last number is the baudrate)
      - **sudo apt-get install screen** (if not installed)
  - Guide 1 Guide 2 Guide 3

Cmake Project Setup

Creating **New** CMake project



Creating **New** CMake project



CMakeSettings.json – build settings

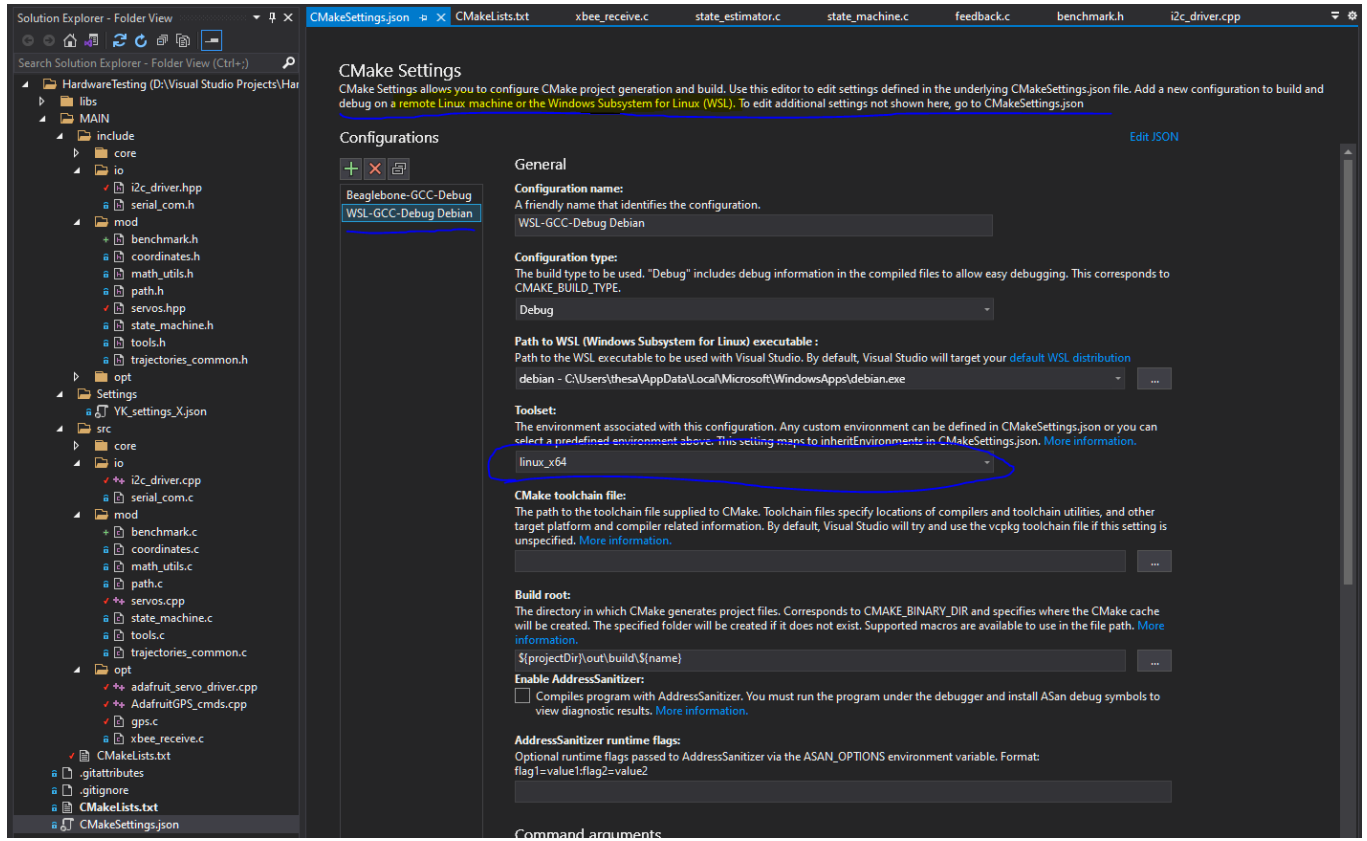
- Select compiler and environment you want your program to compile and debug in. In our case, we have two:
  - Linux-GCC-Debug, Target and build on remote Linux devices with GCC (Debug)
  - WSL-GCC-Debug, target and build on WSL using GCC (Debug)
- If done through Visual Studio, all settings are automatically parsed into a .json file (no need to change it manually).

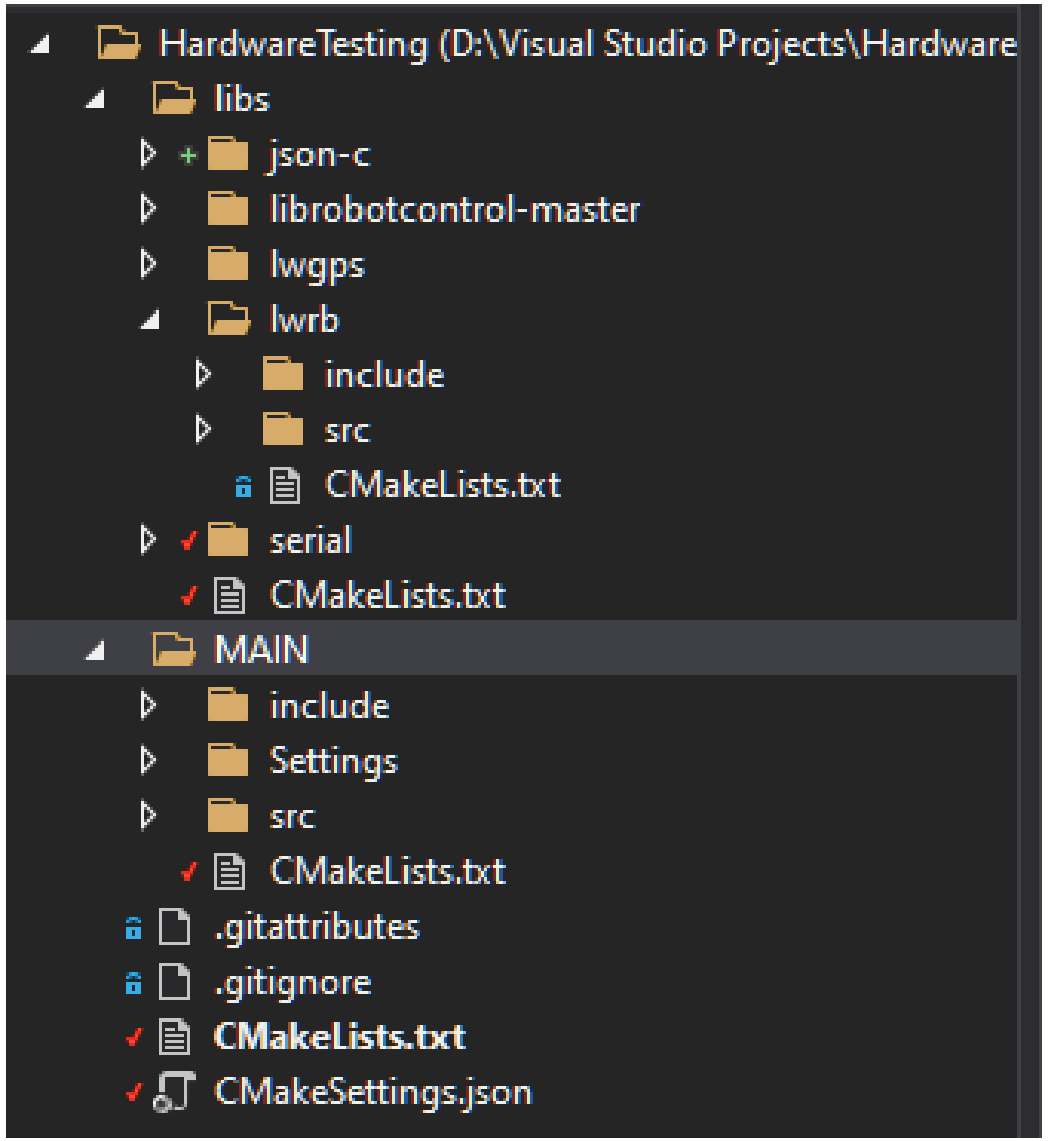
CMakeSettings.json – build settings

```jsonc
{
  "configurations": [
    {
      "name": "Beaglebone-GCC-Debug",
      "generator": "Ninja",
      "configurationType": "Debug",
      "cmakeExecutable": "cmake",
      "remoteCopySourcesExclusionList": [ ".vs", ".git", "out" ],
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "linux_arm" ],
      "remoteMachineName": "${defaultRemoteMachineName}",
      "remoteCMakeListsRoot": "$HOME/.vs/${projectDirName}/${workspaceHash}/src",
      "remoteBuildRoot": "$HOME/${projectDirName}/${workspaceHash}/out/build/${name}",
      "remoteInstallRoot": "$HOME/.vs/${projectDirName}/${workspaceHash}/out/install/${name}",
      "remoteCopySources": true,
      "rsyncCommandArgs": "-t --delete --delete-excluded",
      "remoteCopyBuildOutput": false,
      "remoteCopySourcesMethod": "rsync"
    },
    {
      "name": "WSL-GCC-Debug Debian",
      "generator": "Ninja",
      "configurationType": "Debug",
      "buildRoot": "${projectDir}\\out\\build\\${name}",
      "installRoot": "${projectDir}\\out\\install\\${name}",
      "cmakeExecutable": "cmake",
      "buildCommandArgs": "",
      "inheritEnvironments": [ "linux_x64" ],
      "wslPath": "C:\\Users\\thesa\\AppData\\Local\\Microsoft\\WindowsApps\\debian.exe",
      "variables": [
        {
          "name": "ENABLE_THREADING",
          "value": "True",
          "type": "BOOL"
        },
        {
          "name": "DISABLE_THREAD_LOCAL_STORAGE",
          "value": "False",
          "type": "BOOL"
        },
        {
          "name": "CMAKE_EXPORT_COMPILE_COMMANDS",
          "value": "True",
          "type": "BOOL"
        }
      ]
    }
  ]
}
```

1

2

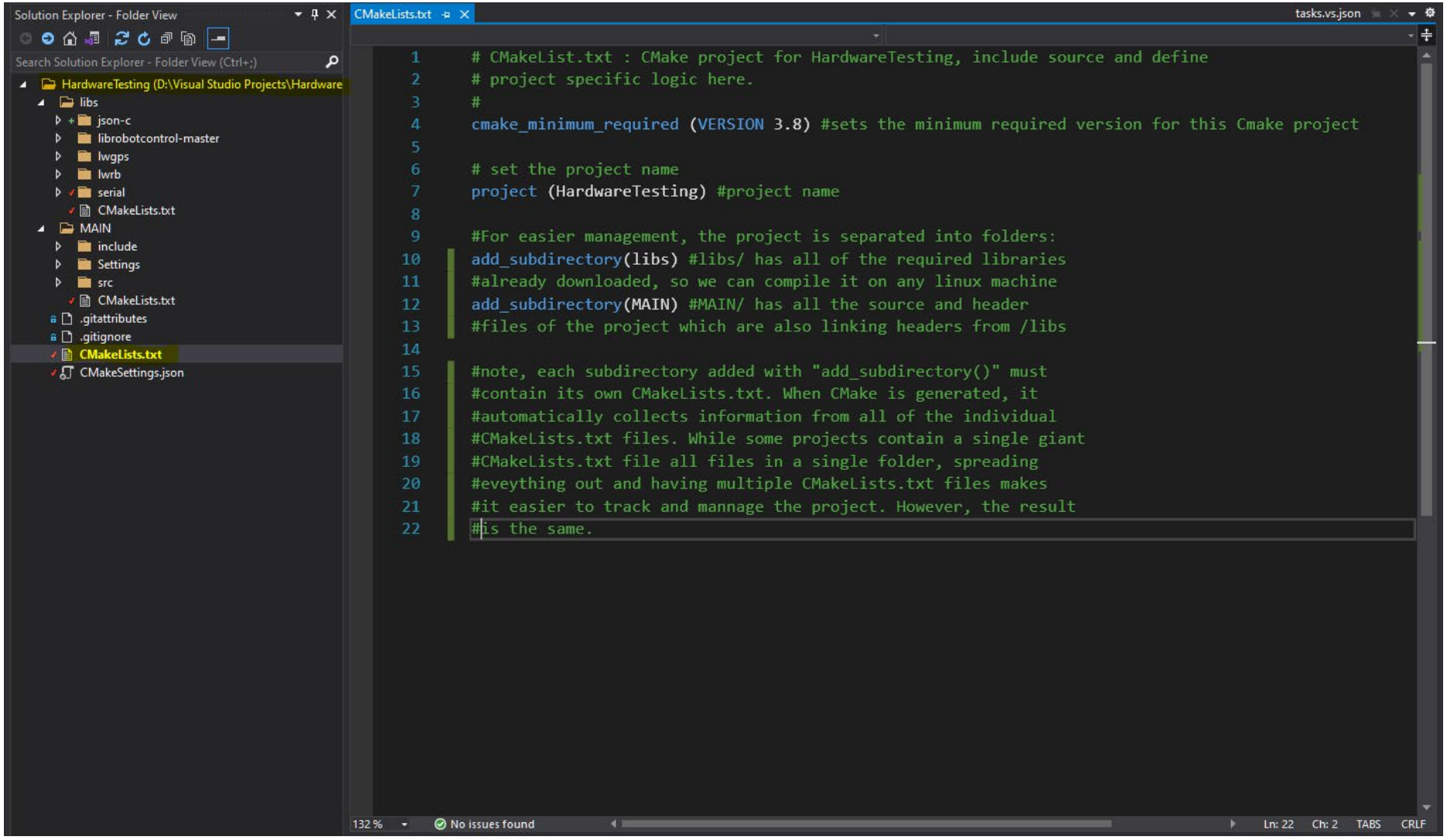




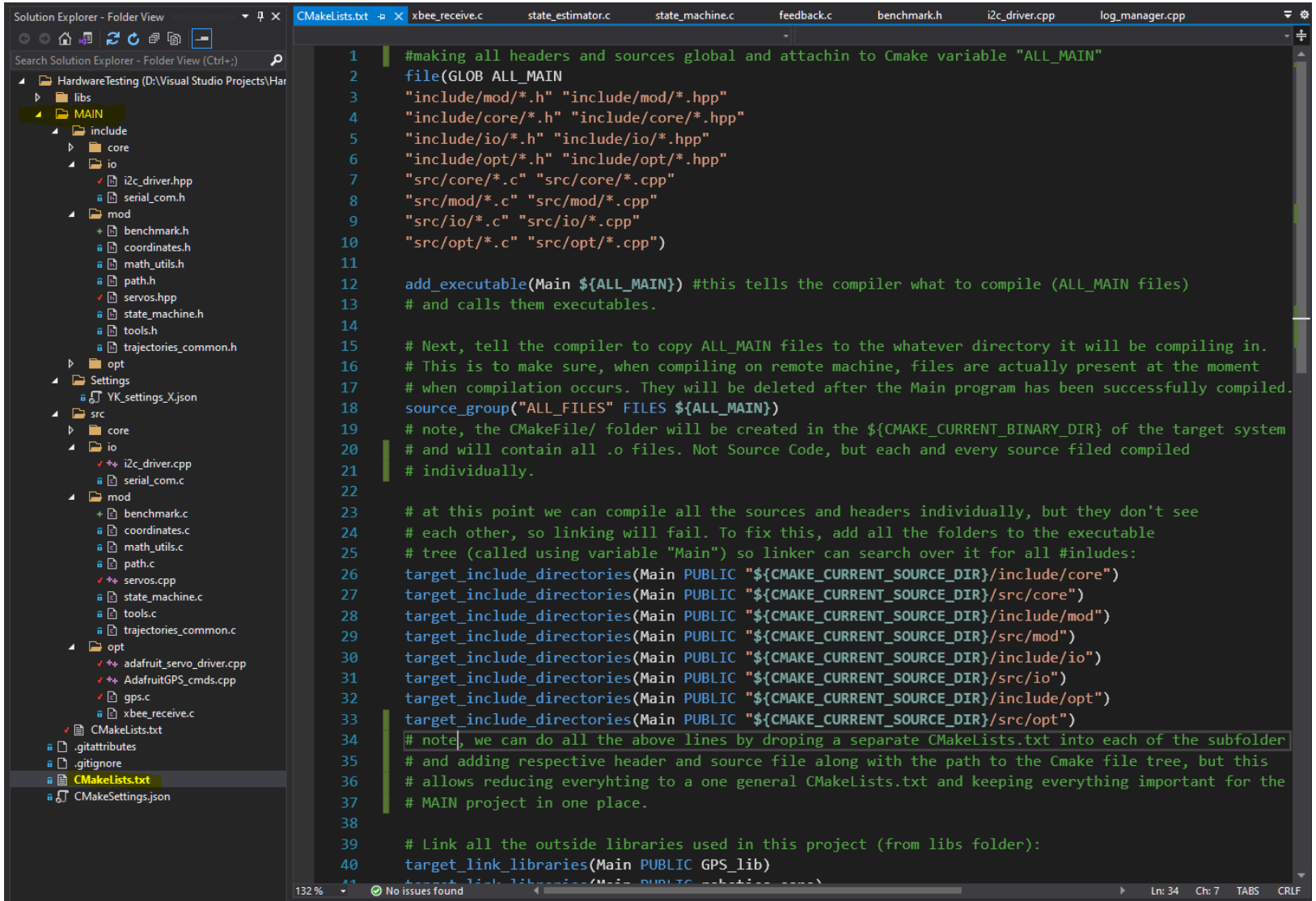
Project Folder Tree

- **Project Name** –Top level directory
  - **Libs** – let's put all external libraries here
    - **Lib1**
      - CMakeLists.txt
    - **Lib2**
      - CMakeLists.txt
    - ……
    - CMakeLists.txt
  - **MAIN** – primary work folder
    - **src** – source folder
    - **Include** – header folder
    - CMakeLists.txt
  - CMakeLists.txt

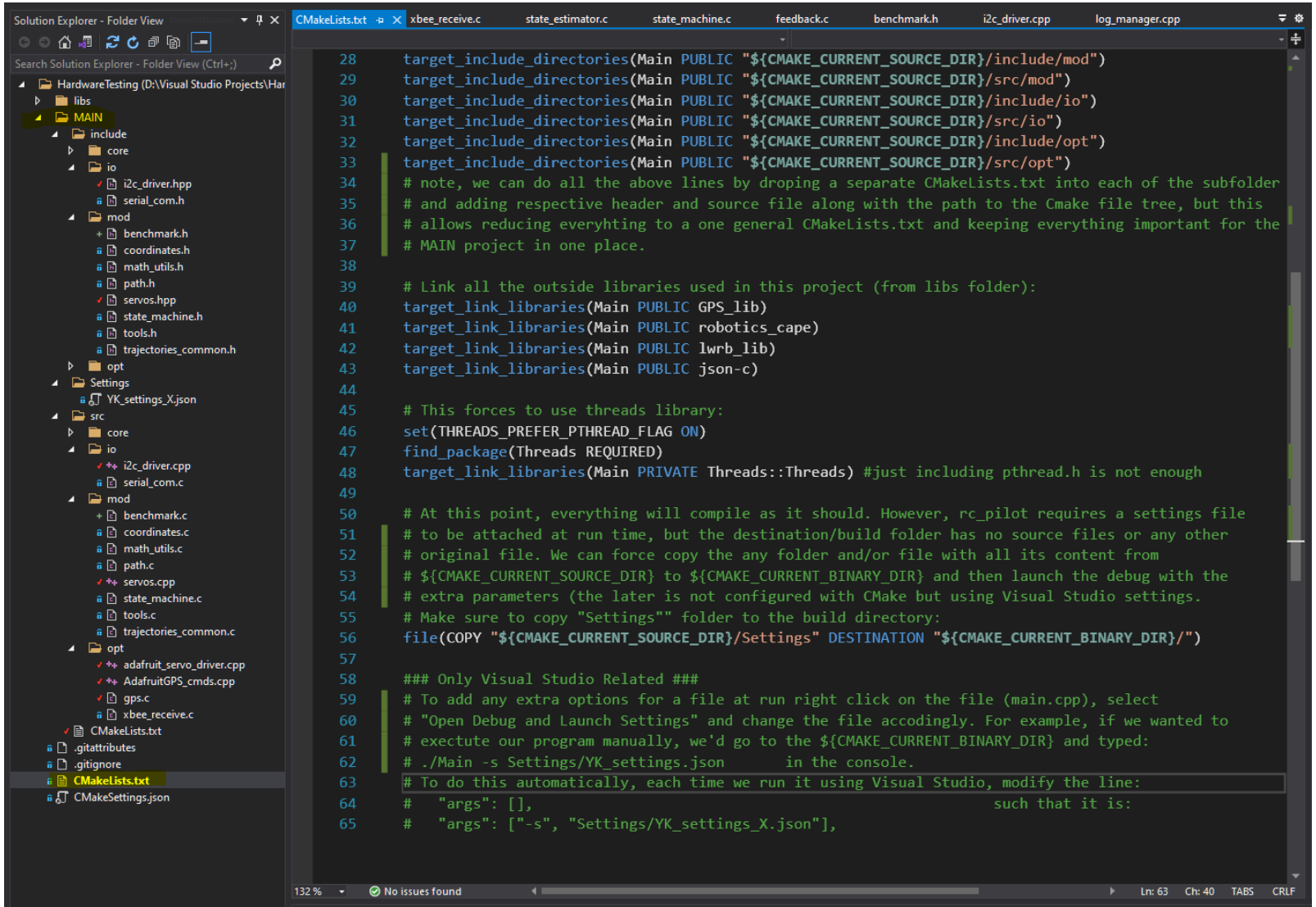
Setting up a CMake file tree and including directories in the top level CMakeLists.txt



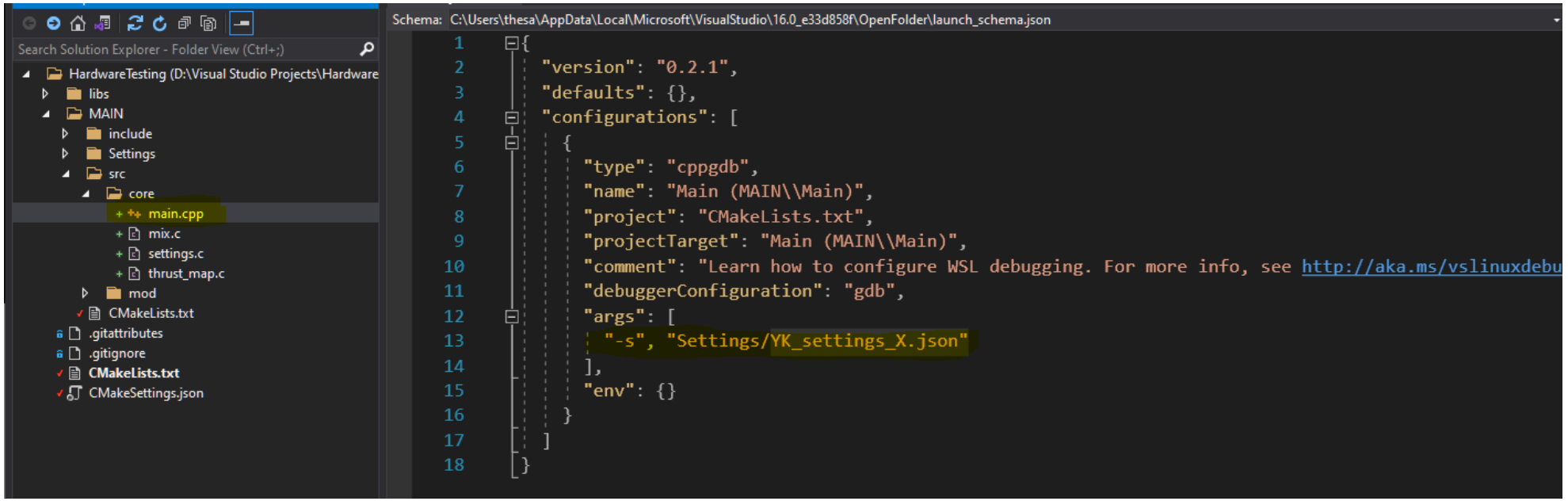
Configuring sources, includes into one executable



Linking with external libraries and additional launch parameters



If your program requires extra parameters on run:



- Go to your main file, right click on it and select “Open Debug and Launch Settings”
- Add your extra parameters into “args”: [ ]
- In this case, it is identical to typing this in the command line:
  ./Main -s Settings/YK_settings_X.json“

(assuming the executable file name is “Main” and I want to pass a string “Settings/YK_settings_X.json”)

Building with Cmake

Using Visual Studio

- Select your target using drop down (WSL or remote build)
- Update Cmake by going to any of the CMakeLists.txt and saving it manually (this will trigger auto-update)
- Go to build tab and select build all

Manual

- Copy the entire source directory with your Cmake project (don’t copy .vs folder)
- Select/create build directory
  - **mkdir build**
- Compile with Cmake, assuming you are in /home/debian/ folder logged as a root and your project is in **Project** folder:
  - **cmake –S Project –B build**
- go to build directory and compile:
  - **MakeIs**