# 1：g++安装

安装支持C++ 17的g++。

下载链接：https://sourceforge.net/projects/mingw-w64/files/Toolchains%20targetting%20Win64/Personal%20Builds/mingw-builds/8.1.0/threads-posix/seh/x86_64-8.1.0-release-posix-seh-rt_v6-rev0.7z/download

我的机器是安装在D:\mingw64。

将安装目录下的bin目录加入PATH环境变量。

验证

```
C:\Users\crackryan>g++ --version
g++ (x86_64-posix-sjlj-rev0, Built by MinGW-W64 project) 8.1.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

# 2：CMake安装

下载地址：https://cmake.org/download/ 下载64位版本

我的机器是安装在C:\CMake。

将CMake安装目录下的bin目录加入PATH环境变量。

验证：

```
C:\Users\crackryan>cmake --version
cmake version 3.24.0

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

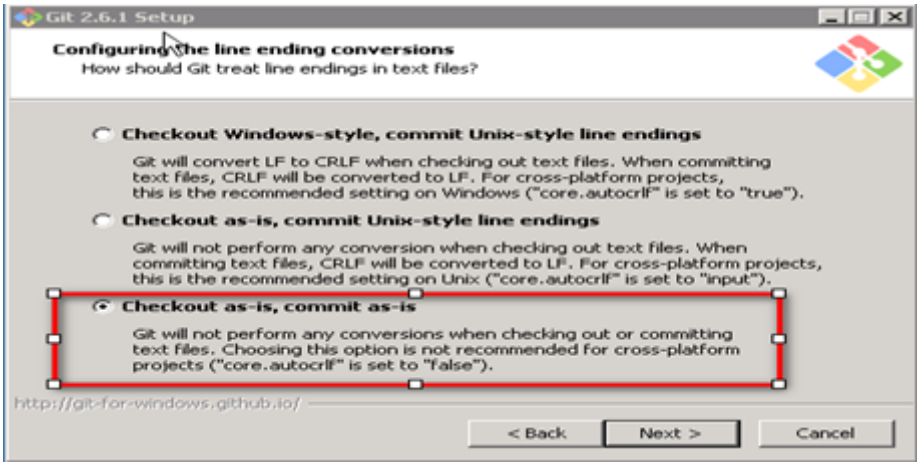
# 3：git安装

下载地址：https://git-scm.com/download](https://link.jianshu.com/?t=https%3A%2F%2Fgit-scm.com%2Fdownload)

**安装过程有两步需要注意，其它步骤按默认安装即可。**

1）修改系统的环境变量


2）配置行尾的结束符




我的机器是安装在"C:\Program Files\Git\git-bash.exe" 。

**Git配置检查及用户设置**

在WINDOWS资源管理器任何一个目录下点击鼠标右键，选择Git Bash Here打开git命令行工具(后面的命令都是在Git Bash环境下执行):


**检查用户名和邮箱是否配置**

$ git config --global --list

**如未配置，则执行以下命令进行配置:**

$ git config –global user.name "这里换上你的用户名"

$ git config --global user.email "这里换上你的邮箱"

# 4：VS Code安装

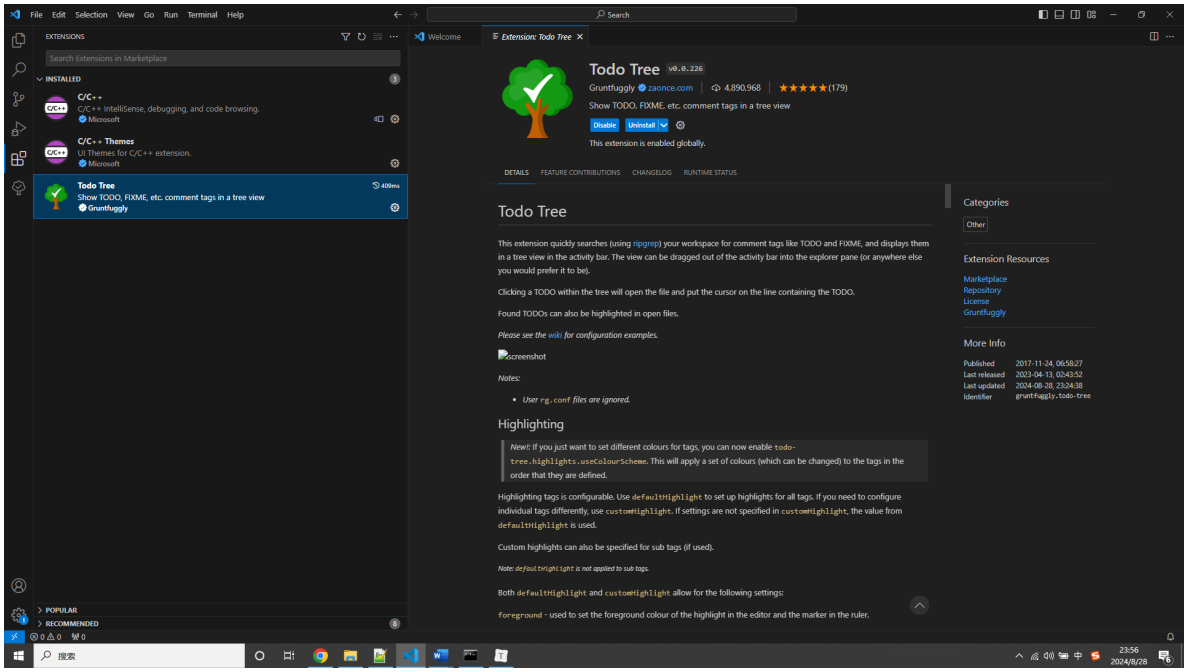
官网：https://code.visualstudio.com/download

默认步骤安装即可。

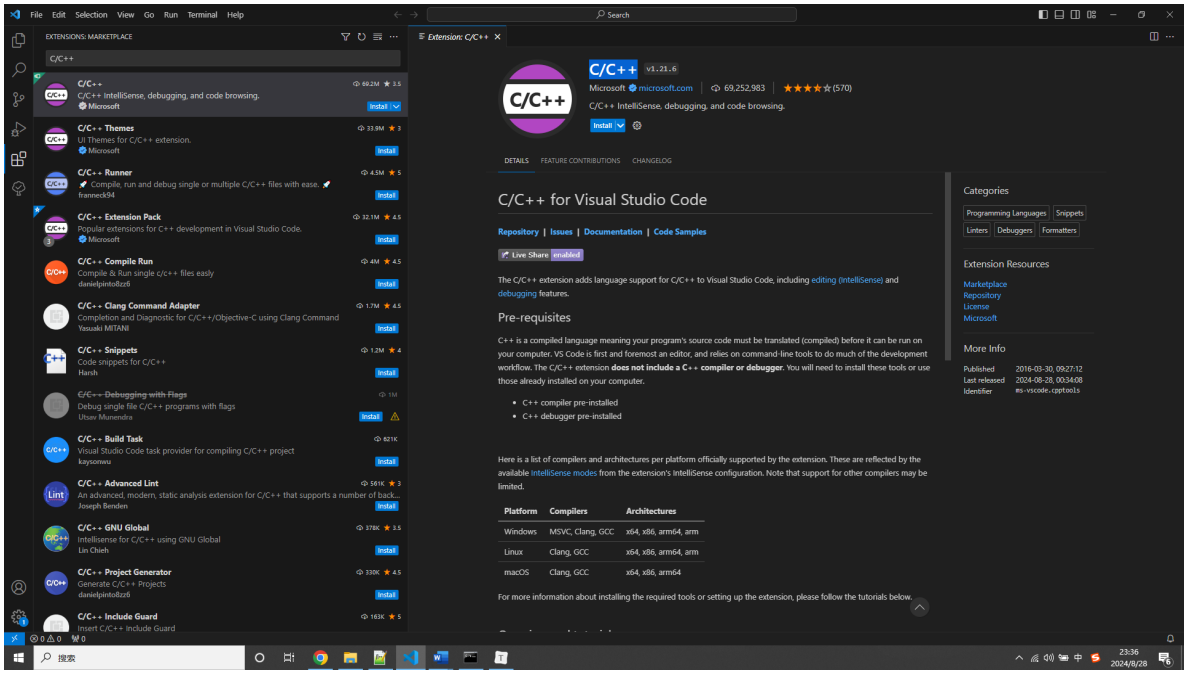
我的机器是安装在D:\Microsoft_VSCode\Code.exe。

# 5：VS Code安装插件

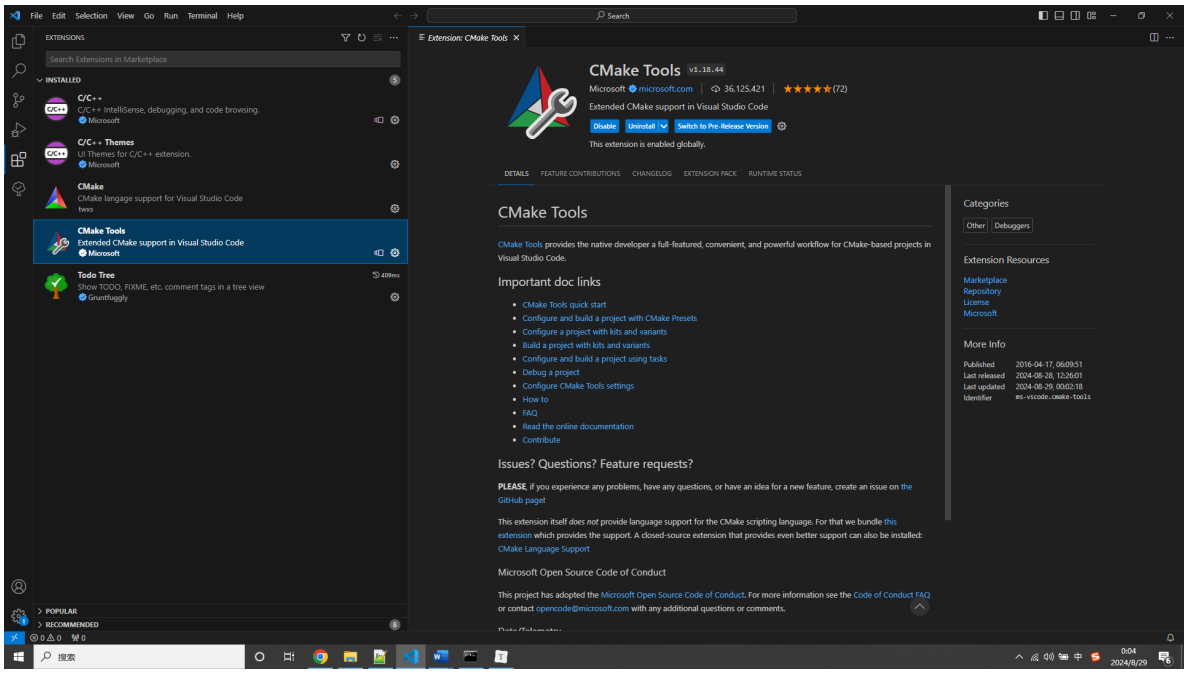
## 5.1 Todo Tree



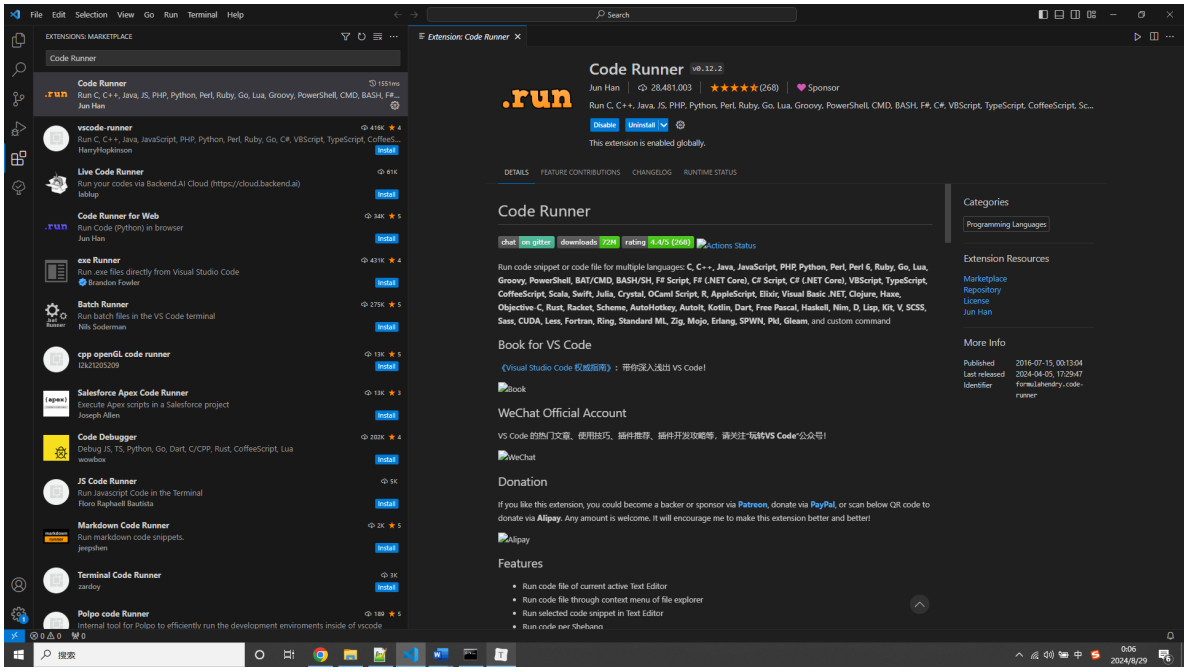

## 5.2 C/C++



## 5.3 CMake Tools

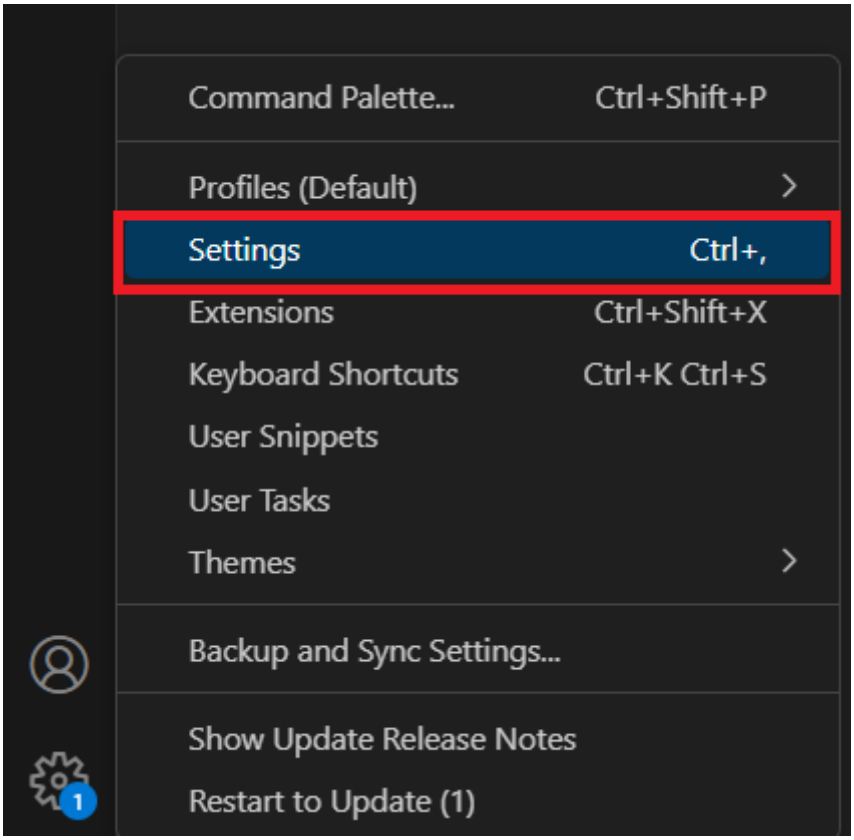


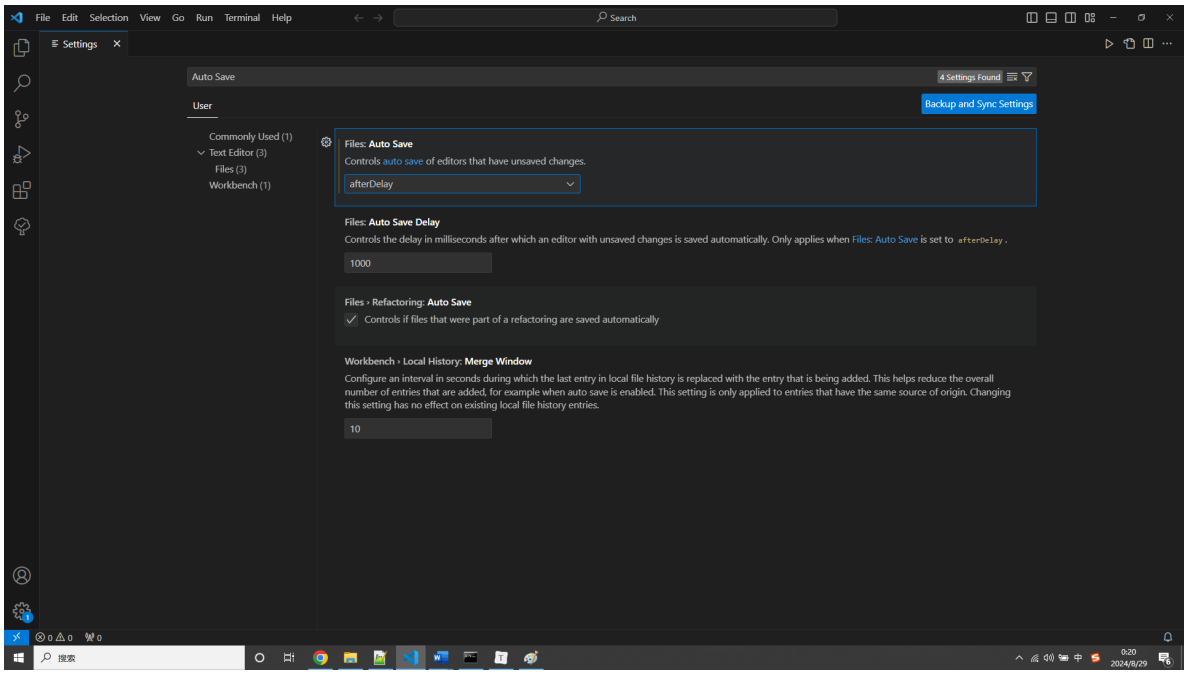

## 5.4 Code Runner



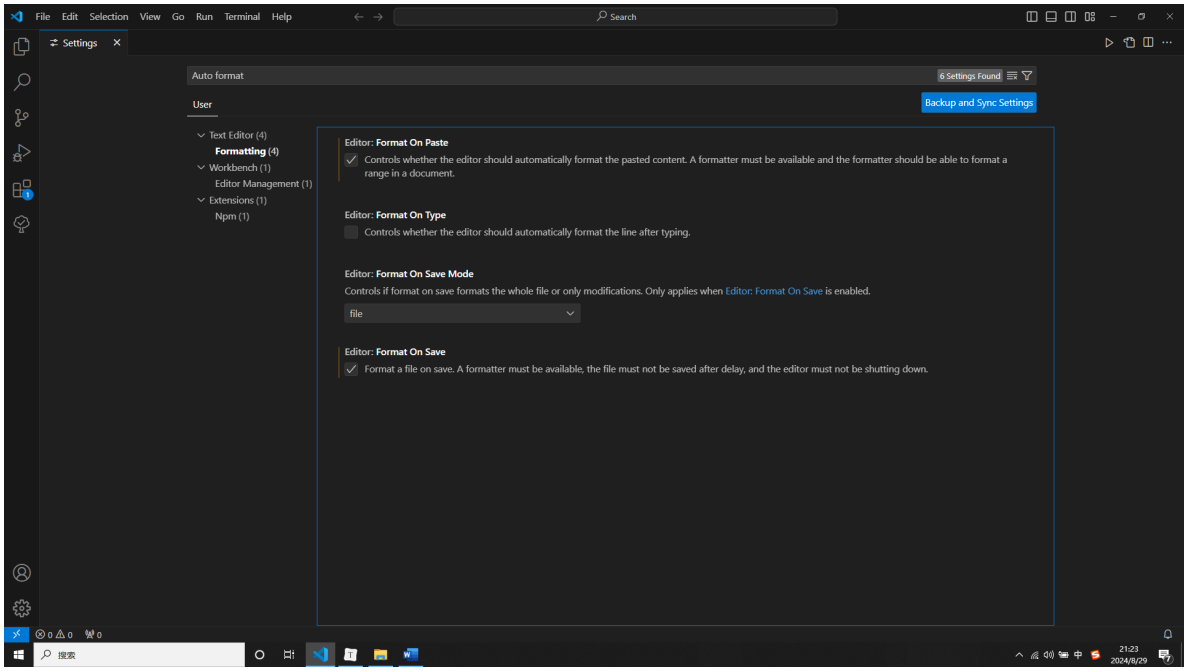

# 6：配置VS Code

## 6.1 Auto Save




# 6.2 Auto format




# 6.3 Clang_format_style

clang 团队创建了一个我们可以利用的出色工具：clang-format。通过使用 clang-format，我们可以创建样式规则列表，使程序员能够快速重新格式化其代码，并创建在我们的构建服务器上运行的格式检查以确保合规性。

当安装好VS Code以后，clang-format工具的安装路径，一般在C盘个人目录下：

```
C:\Users\{你的用户名} \.vscode\extensions\ms-vscode.cpptools-{版本号}\LLVM\bin\
clang-format.exe
```

例如在我的机器上，位于：

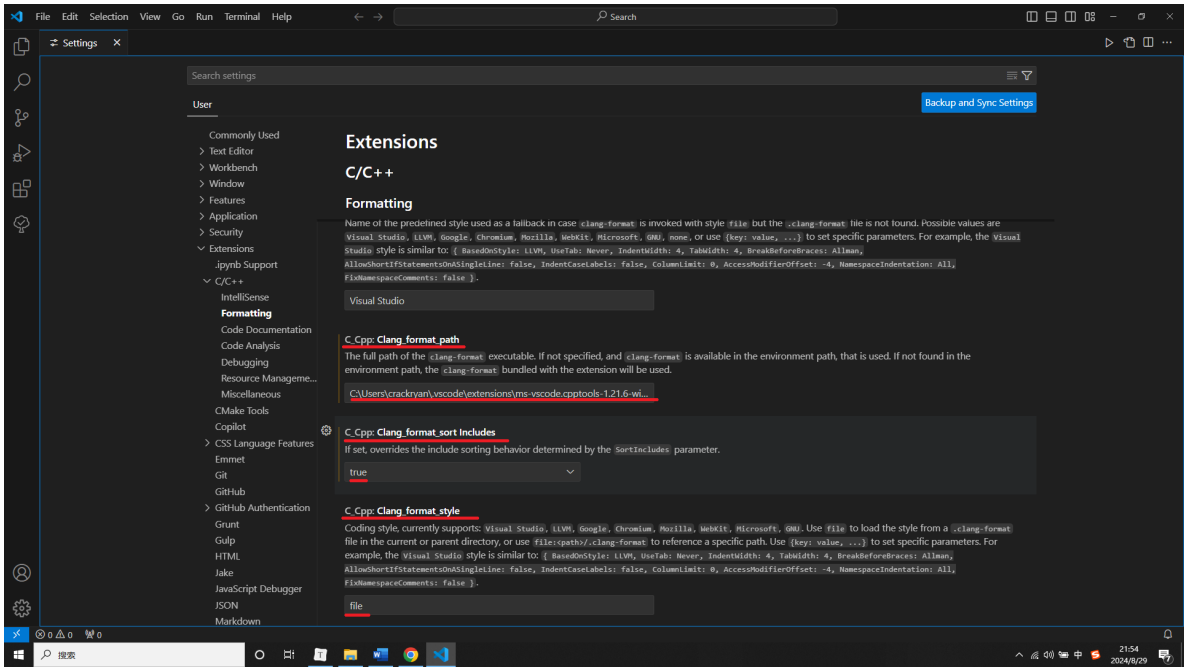
```
C:\Users\crackryan\.vscode\extensions\ms-vscode.cpptools-1.21.6-win32-
x64\LLVM\bin
```

您可以使用以下命令查看为每种默认样式启用的选项：

```
clang-format --style=llvm -dump-config
```

- llvm – complies with the LLVM coding standard
- Google – complies with Google's C++ style guide
- Chromium – complies with Chromium's style guide
- Mozilla – complies with Mozilla's style guide
- WebKit – complies with Webkit's style guide

LLVM也是一个非常有意思的开源项目，参见https://llvm.org/，和https://www.jianshu.com/p/1367dad95445




备注：Clang_format_path中，填入随着VS Code安装的clang-format工具的安装路径，如：

```
C:\Users\crackryan\.vscode\extensions\ms-vscode.cpptools-1.21.6-win32-x64\LLVM\bin\clang-format.exe
```

Clang_format_style选择file。Use `file` to load the style from a `.clang-format` file in the current or parent directory, or use `file:<path>/.clang-format` to reference a specific path.

在工程目录下执行：`clang-format -style=google -dump-config > .clang-format`，会生成google风格的自定义代码风格配置文件`.clang-format`

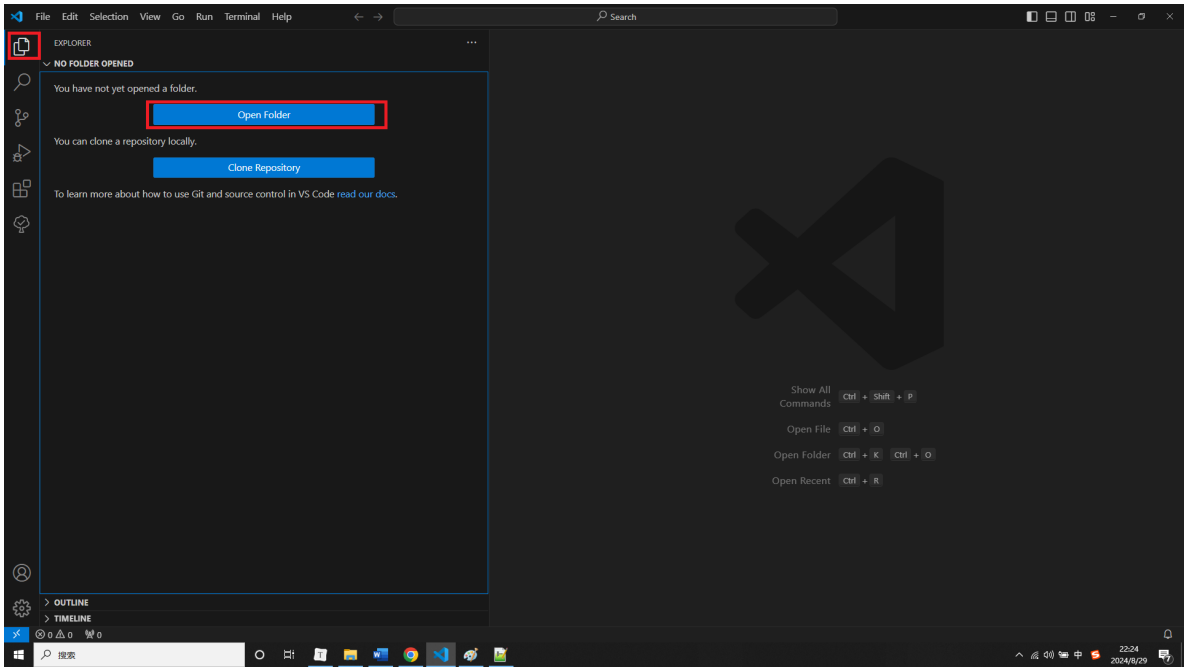
或者在项目根目录下创建一个`.clang-format`文件，内容示例为

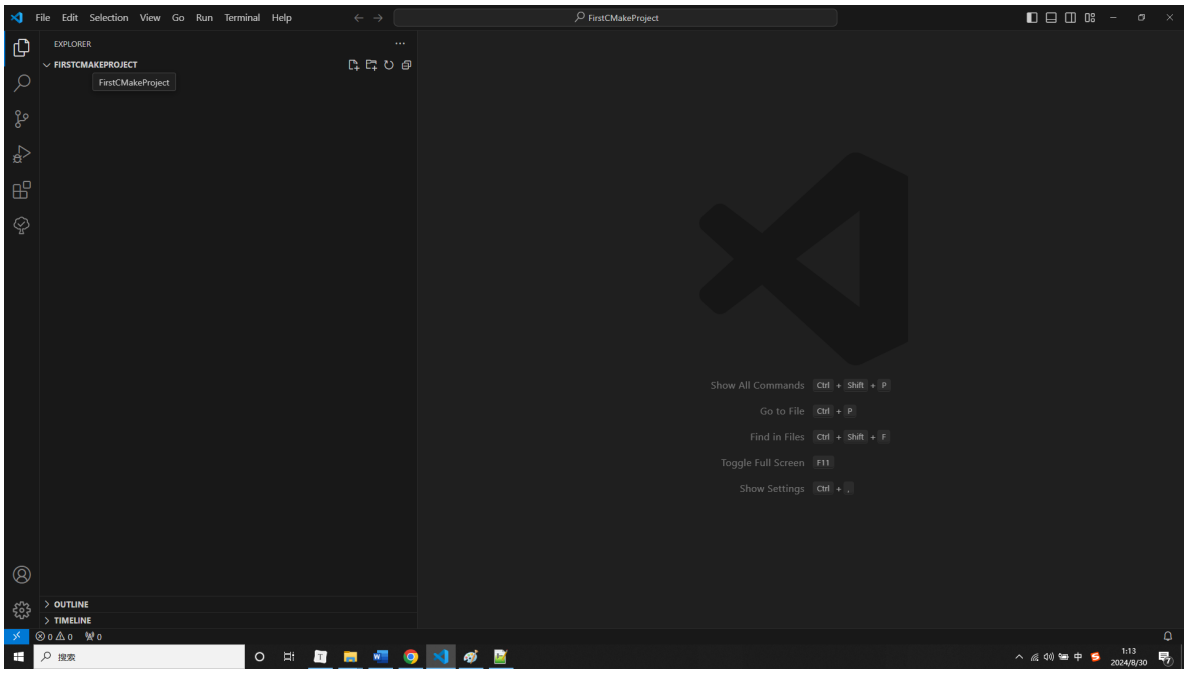
```
# Run manually to reformat a file:
# clang-format -i --style=file <file>
Language: Cpp
BasedOnStyle: Google
UseTab: Never
IndentWidth: 4
TabWidth: 4
BreakBeforeBraces: Linux
AllowShortIfStatementsOnASingleLine: false
AllowShortFunctionsOnASingleLine: false
IndentCaseLabels: false
ColumnLimit: 120
AccessModifierOffset: -4
```

# 7: 创建Quick Start工程

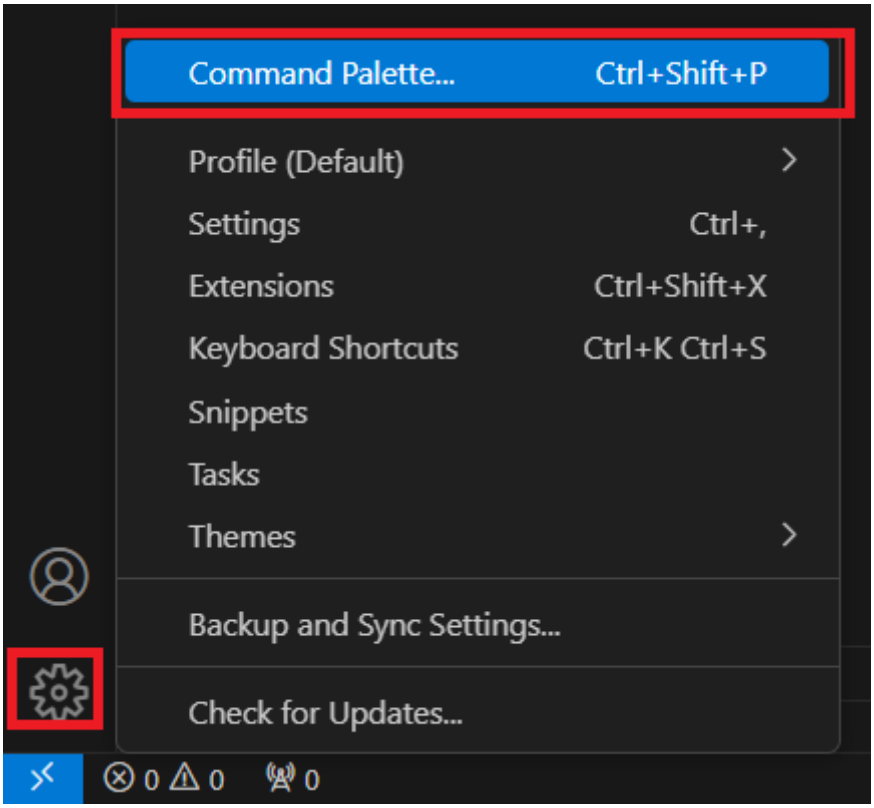
## 7.1 创建工程文件夹

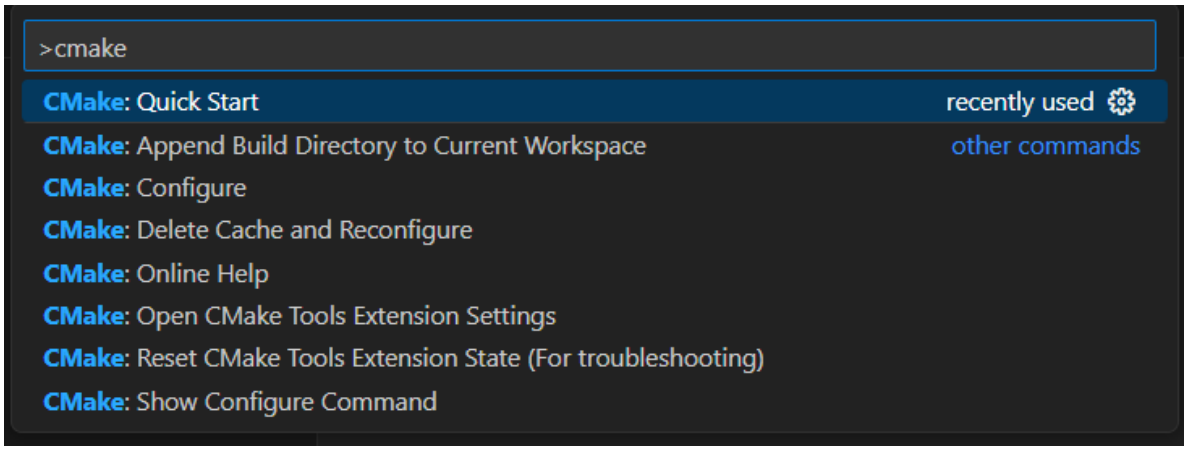
创建文件夹D:\VSCodeProjects\C++Course\FirstCMakeProject，用VSCode打开这个文件夹



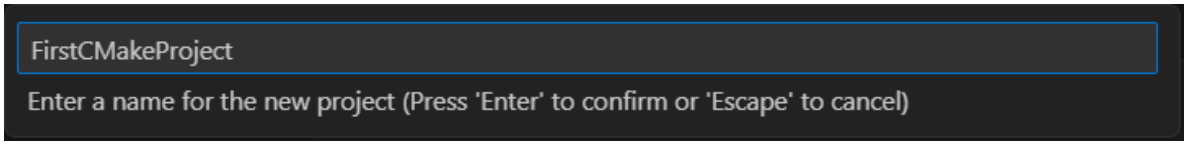




# 7.2 创建CMake工程

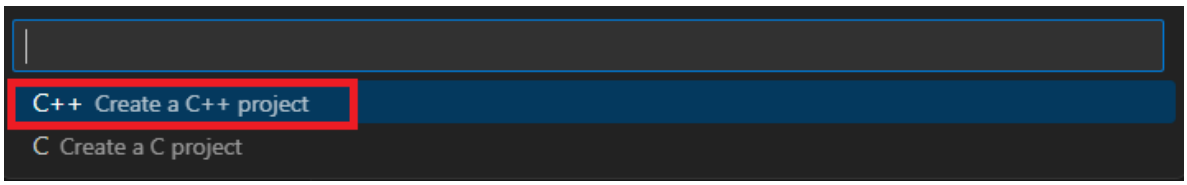


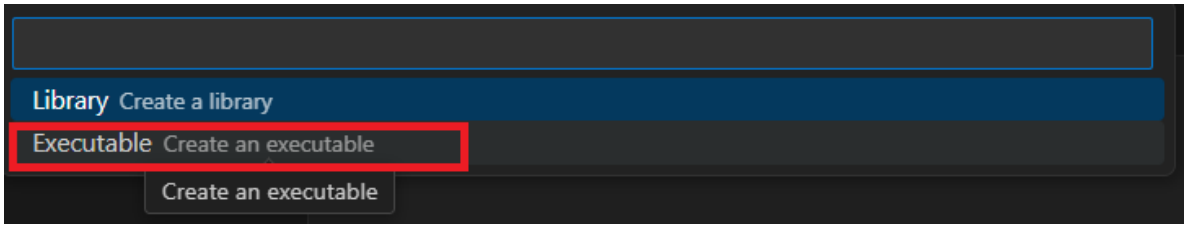



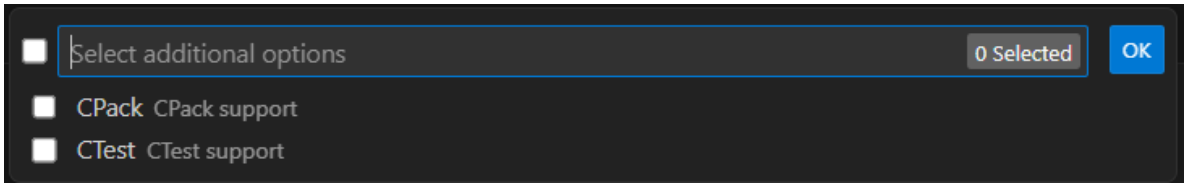

在搜索框里输入cmake，选择CMake:Quick Start




输入项目名称：FirstCMakeProject

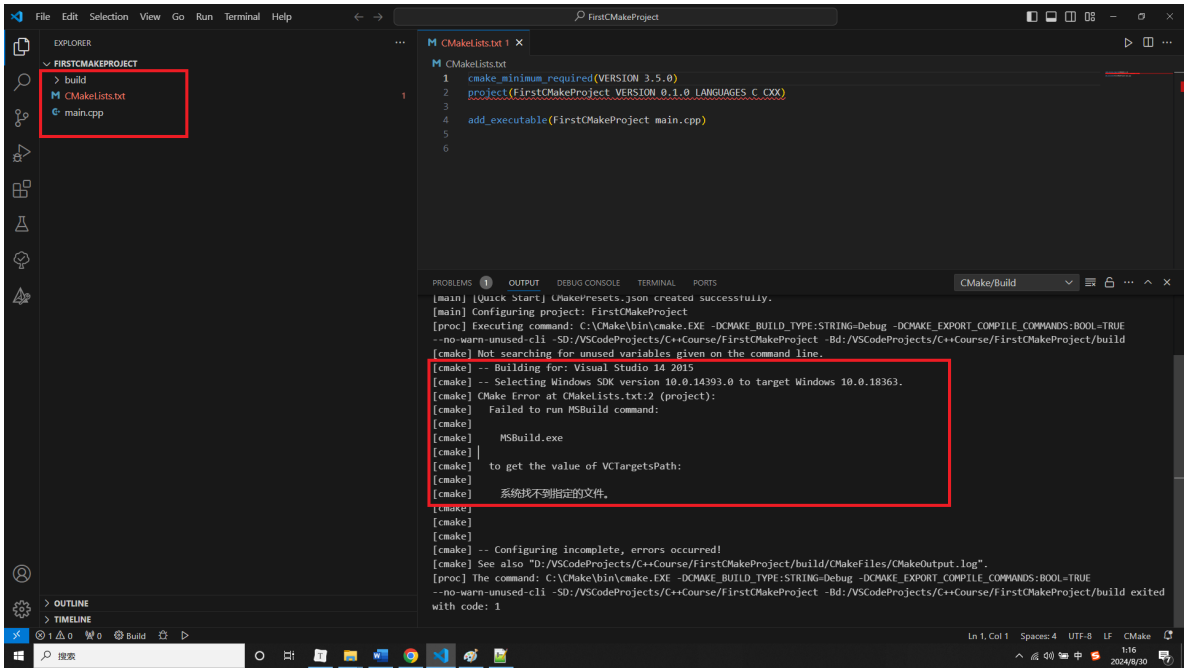





注意：0 selected。

会自动创建如下目录

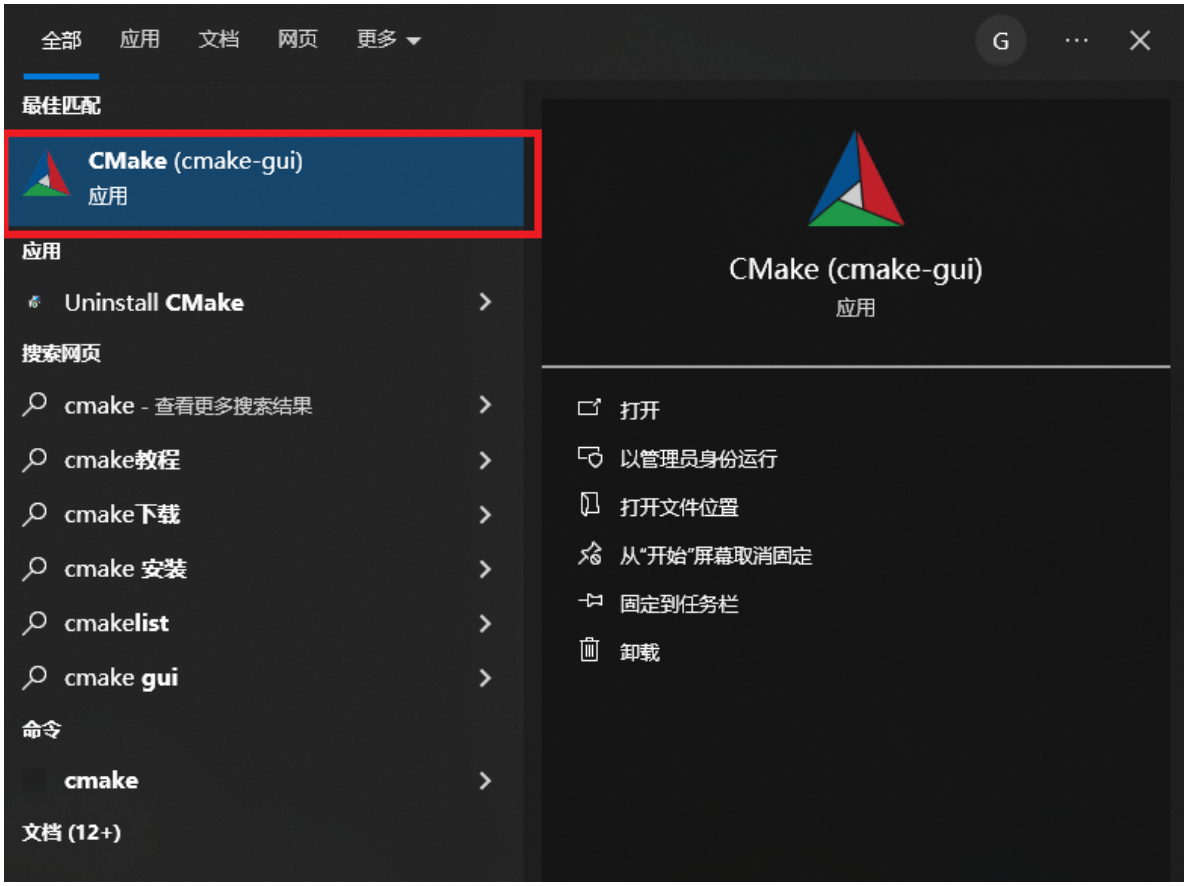


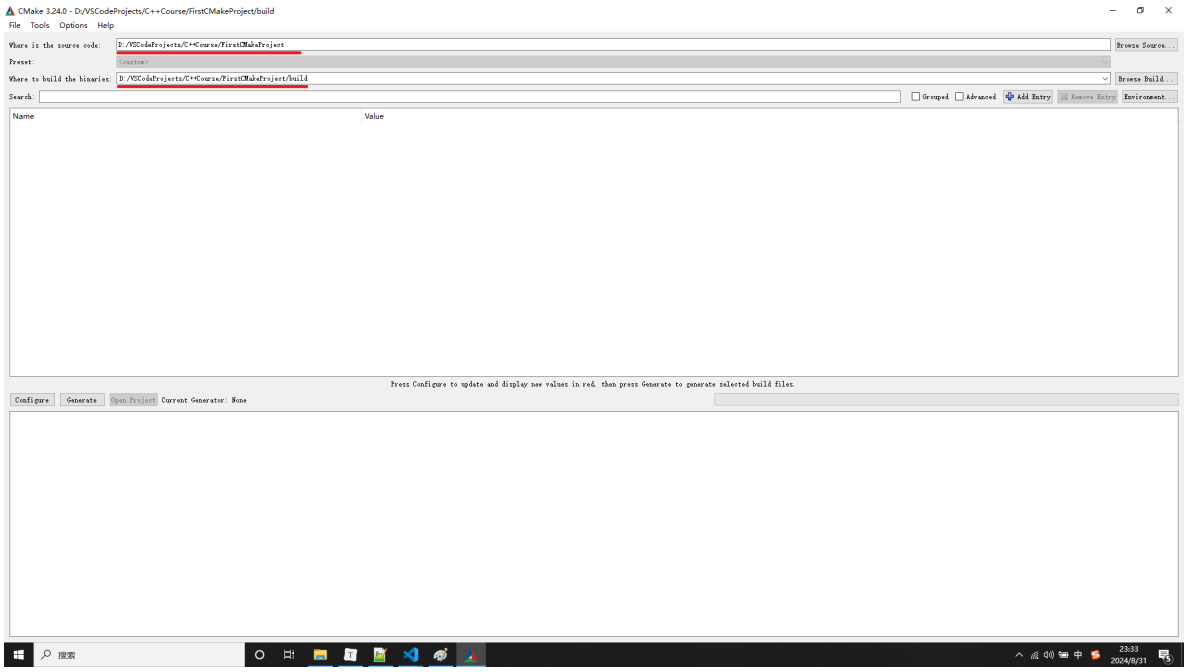

但是可以看到报错，原因：默认VSCode的编译器是微软的MSBuild.exe，找不到指定的文件。**如果没有看到这个报错，恭喜你没有踩坑，7.2节剩下部分忽略。**

原因：观察工程目录\build\CMakeCache.txt，其中有一行：

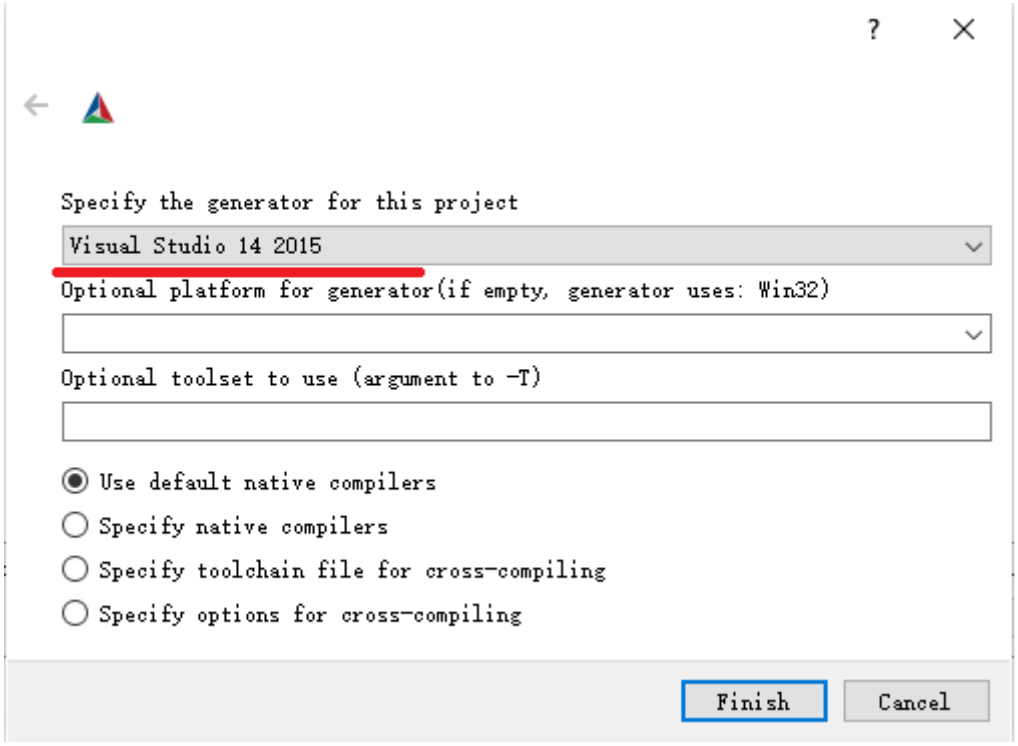
```
//Name of generator.
CMAKE_GENERATORL=Visual Studio 14 2015
```

可以看到默认的CMAKE_GENERATOR是Visual Studio 14 2015。这是因为安装的cmake就是这么配置的。打开安装的cmake-gui

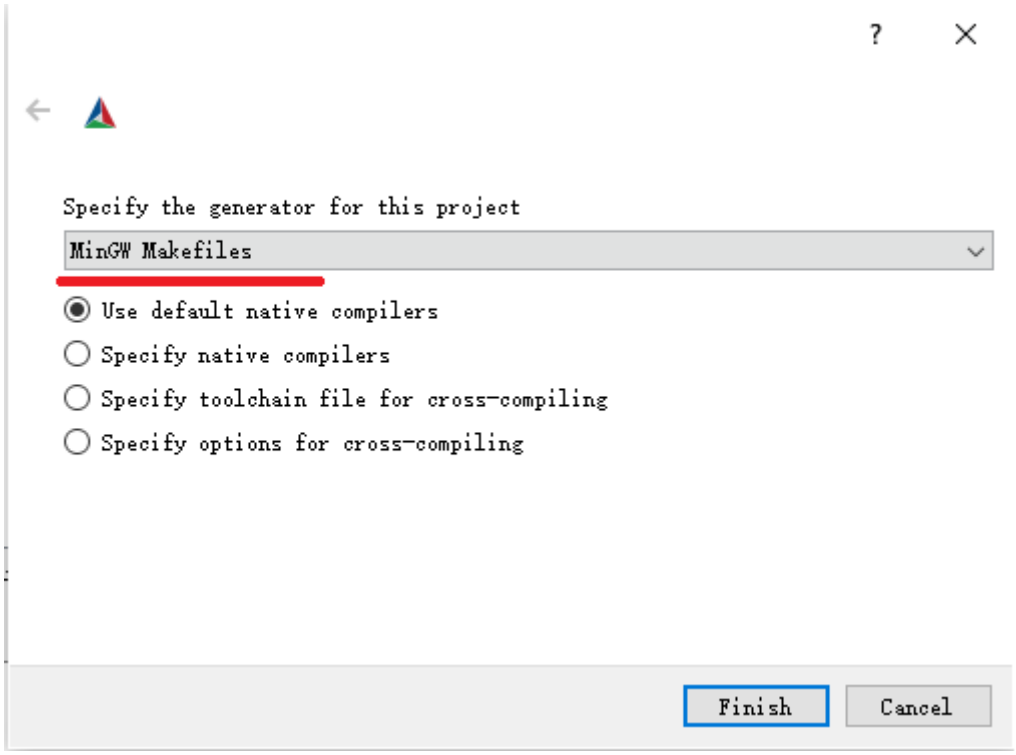


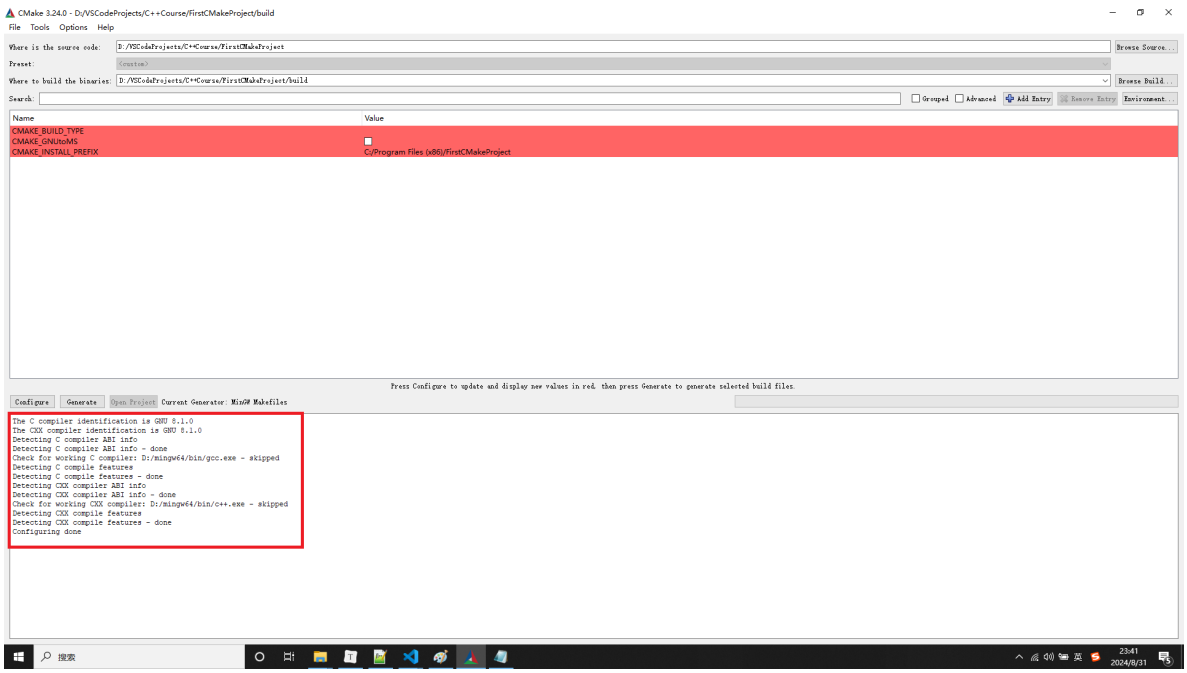




在上面设置好工程目录和工程下面的build目录（红色下划线所示）。再点击Configure按钮

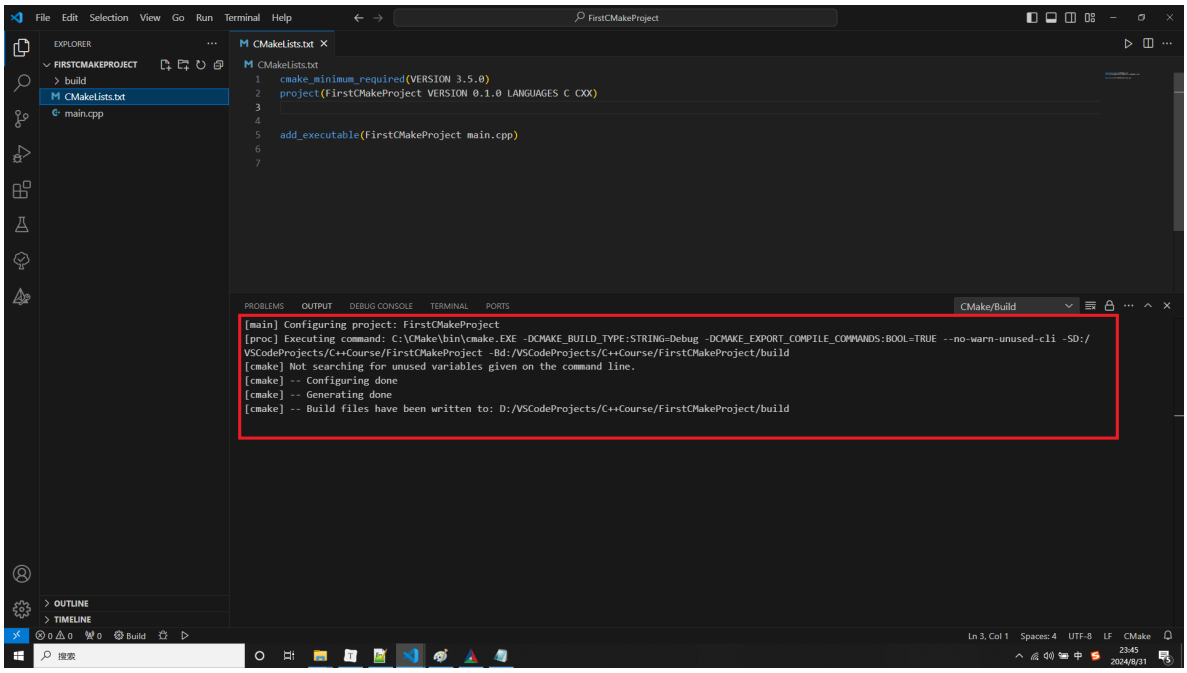



可以看到在CMake里的generator是Visual Studio 14 2015，而本机上没有Visual Studio 14 2015。现在将CMake里的generator设置为`MinGW Makefiles`：






可以看到CMake下面的输出。这个时候再回到VSCode观察（修改工程CMakeLists.txt内容。例如加一个空行，会导致cmake重新生成），这个时候看到错误消失：




再来观察工程目录\build\CMakeCache.txt，其中的CMAKE_GENERATOR变成了

```
//Name of generator.
CMAKE_GENERATOR:INTERNAL=MinGW Makefiles
```

但是观察build目录下自动产生的compile_commands.json，发现其使用的编译器为c++.exe

```
[
{
  "directory": "D:/VSCodeProjects/C++Course/FirstCMakeProject/build",
  "command": "D:\\mingw64\\bin\\c++.exe   -g -o
CMakeFiles\\FirstCMakeProject.dir\\main.cpp.obj -c
D:\\VSCodeProjects\\C++Course\\FirstCMakeProject\\main.cpp",
  "file": "D:/VSCodeProjects/C++Course/FirstCMakeProject/main.cpp"
}
]
```

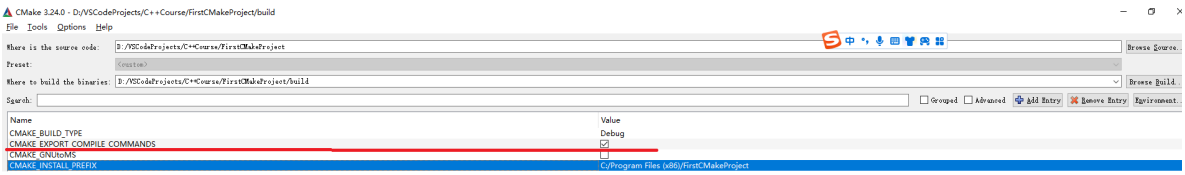
现在在CMakeLists.txt里添加这样一行来设置编译器

```
# 设置C++编译器为g++
set(CMAKE_CXX_COMPILER "D:\\mingw64\\bin\\g++.exe")
```

现在自动产生的compile_commands.json的内容变成了

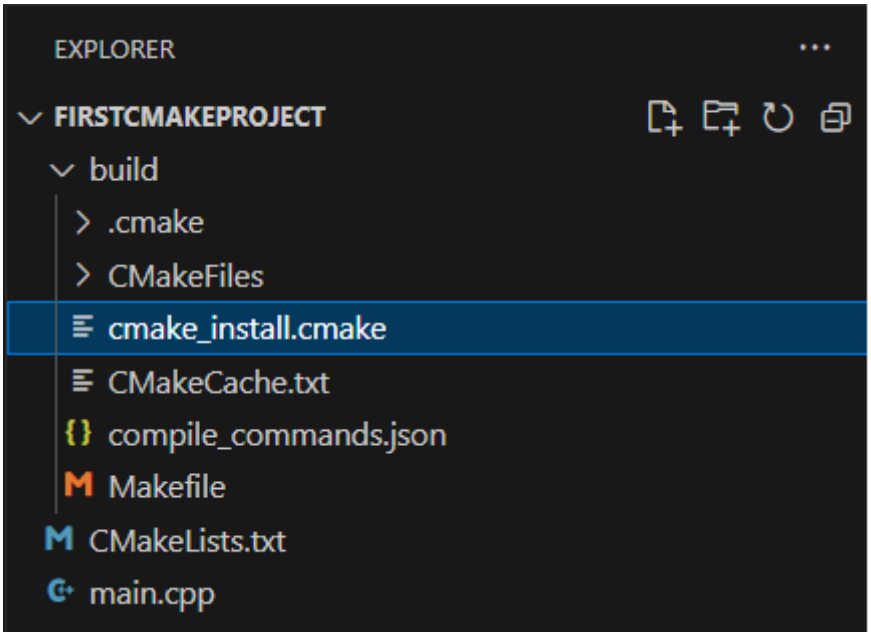
```
[
{
  "directory": "D:/VSCodeProjects/C++Course/FirstCMakeProject/build",
  "command": "D:\\mingw64\\bin\\g++.exe   -g -o
CMakeFiles\\FirstCMakeProject.dir\\main.cpp.obj -c
D:\\VSCodeProjects\\C++Course\\FirstCMakeProject\\main.cpp",
  "file": "D:/VSCodeProjects/C++Course/FirstCMakeProject/main.cpp"
}
]
```

另外，会自动产生compile_commands.json的原因是在CMake里设置如下的Entry：




而且这一项为true（打了√）。

同时可以看到在build下面自动产生了cmake_install.cmake文件



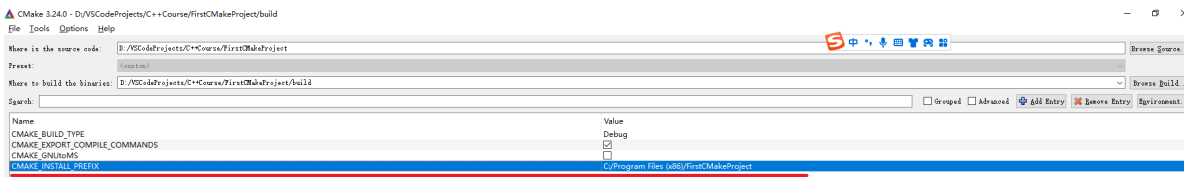
这是一个编译好后的程序安装脚本，其中有几行代码需要注意：

```
# Install script for directory: D:/VSCodeProjects/C++Course/FirstCMakeProject

# Set the install prefix
if(NOT DEFINED CMAKE_INSTALL_PREFIX)
  set(CMAKE_INSTALL_PREFIX "C:/Program Files (x86)/FirstCMakeProject")
endif()
string(REGEX REPLACE "/$" "" CMAKE_INSTALL_PREFIX "${CMAKE_INSTALL_PREFIX}")
```

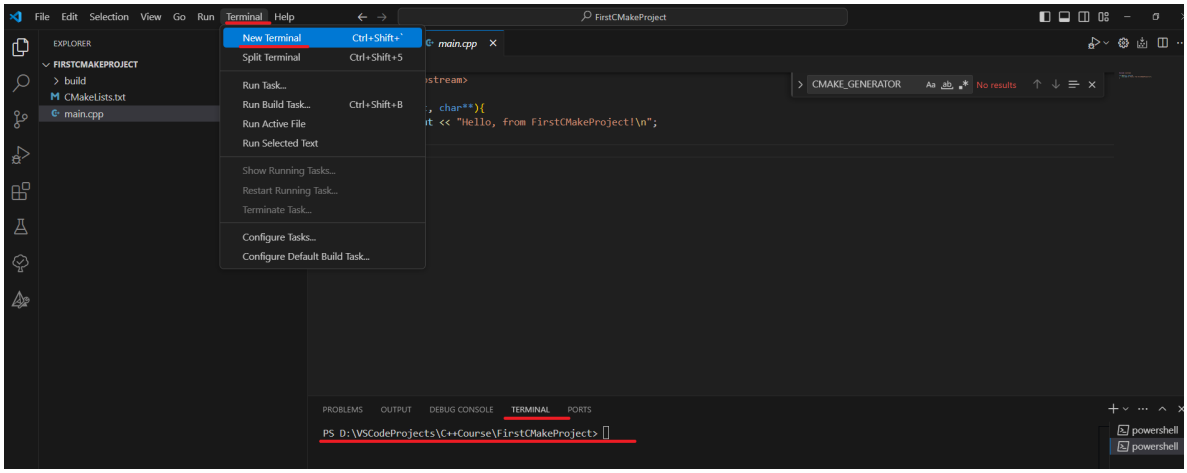
其中的安装目录前缀为


```
# Install script for directory: D:/VSCodeProjects/C++Course/FirstCMakeProject

# Set the install prefix
if(NOT DEFINED CMAKE_INSTALL_PREFIX)
  set(CMAKE_INSTALL_PREFIX "C:/Program Files (x86)/FirstCMakeProject")
endif()
string(REGEX REPLACE "/$" "" CMAKE_INSTALL_PREFIX "${CMAKE_INSTALL_PREFIX}")
```


这个前缀可以在CMake里修改，如下所示：




# 7.3 通过命令行方式编译和运行程序




在打开的终端里依次执行下面的命令进行编译（为了能看到完整过程，清空build目录内容，但保留build目录）

```
cd build
cmake -G"MinGW Makefiles" ..
```

备注：`cmake -G"MinGW Makefiles" ..`命令中`-G"MinGW Makefiles"`是指定CMake生成的项目类型为MinGW的Makefile。在CMake中，-G参数用于指定生成器，即用于生成特定构建系统的Makefiles或项目文件。不同的构建系统有不同的生成器可以选择，例如Unix Makefiles、Ninja、Visual Studio等。总之，-G参数用于指定生成器，帮助CMake生成适用于特定构建系统的Makefiles或项目文件。

`..`是指上级目录。这个命令的意思是：在上级目录（也就是项目根目录）下寻找CMakeLists.txt文件，并根据这个文件生成相应的Makefile，这个Makefile可以被GNU make工具使用，进而编译链接生成最终的可执行文件或者库。

在终端里的输出为

```
PS D:\VSCodeProjects\C++Course\FirstCMakeProject> cd build
PS D:\VSCodeProjects\C++Course\FirstCMakeProject\build> cmake -G"MinGW
Makefiles" ..
-- The C compiler identification is GNU 8.1.0
-- The CXX compiler identification is GNU 8.1.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: D:/mingw64/bin/gcc.exe - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: D:/mingw64/bin/c++.exe - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to:
D:/VSCodeProjects/C++Course/FirstCMakeProject/build
PS D:\VSCodeProjects\C++Course\FirstCMakeProject\build>
```

可以看到在build目录下产生了很多文件，其中有Makefile，这个就是make文件，是真正的用来编译程序的脚本文件。熟悉make的同学可以读该文件的内容。如果熟悉make，是可以直接编写make文件来编译项目的。但是对于一个复杂项目的make文件，人类是不愿意编写的。因此CMake应运而生。CMake是一个强大的跨平台构建工具，它能够自动生成用于构建项目的Makefile，**它可以编译用相对简单的多的语法来定义的CMakeLists.txt文件，产生Makefile文件。**

产生了Makefile文件后，接着再来编译程序

```
cd build
mingw32-make.exe
```

在终端里的输出为

```
PS D:\VSCodeProjects\C++Course\FirstCMakeProject> cd build
PS D:\VSCodeProjects\C++Course\FirstCMakeProject\build> mingw32-make.exe
[ 50%] Building CXX object CMakeFiles/FirstCMakeProject.dir/main.cpp.obj
[100%] Linking CXX executable FirstCMakeProject.exe
[100%] Built target FirstCMakeProject
PS D:\VSCodeProjects\C++Course\FirstCMakeProject\build>
```

可以看到在build目录下产生了FirstCMakeProject.exe文件。

下面来运行FirstCMakeProject.exe：

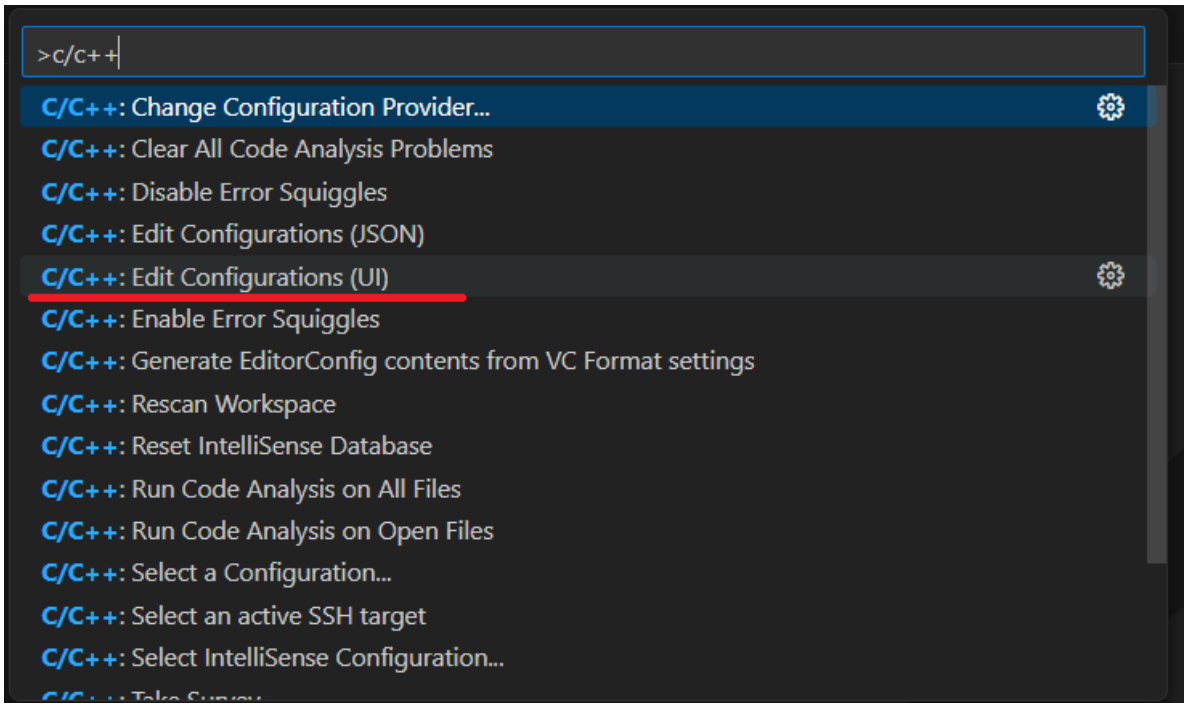
```
cd build
.\FirstCMakeProject
```

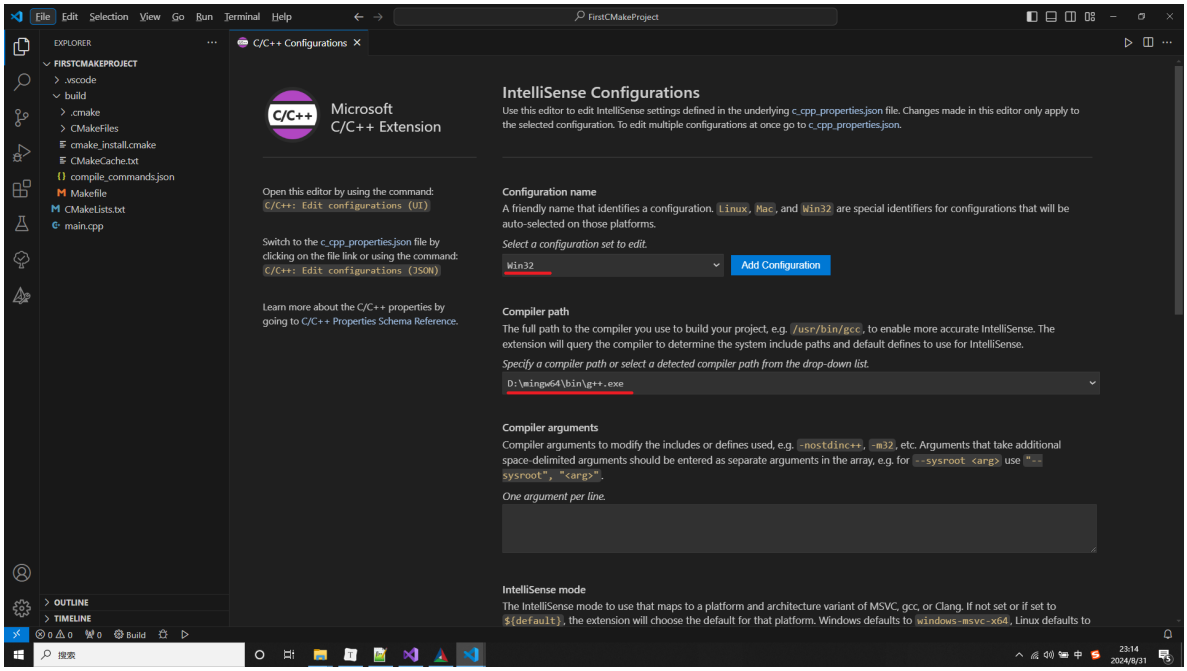
备注：默认情况下，Windows PowerShell 不会从当前位置加载命令。 一定改为键入“.\FirstCMakeProject”。

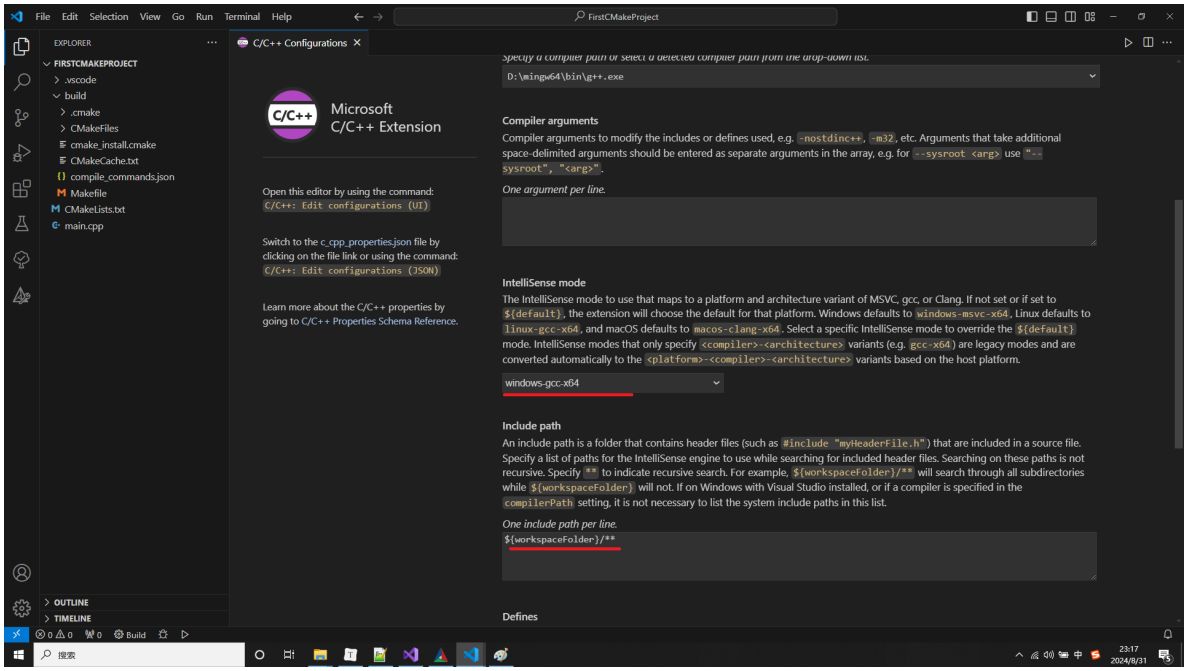
到目前为止，我们已经学会了如何创建一个基于CMake的C++工程，并可以通过命令行方式编译和运行。后面的实验都是基于这种模式进行。但如果不通过命令行方式使用CMake来编译和运行程序，如何在VSCode里编译、调试C++程序呢？下面接着进行介绍。

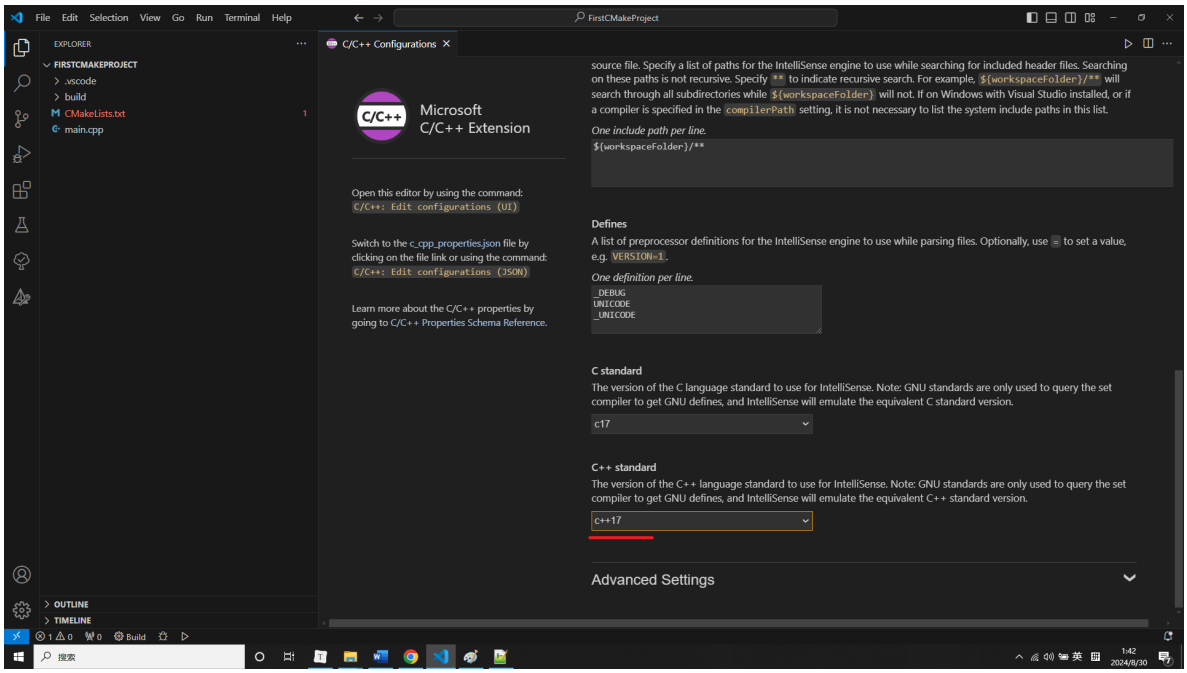
# 7.4 配置VSCode使用g++

按ctrl+shift+p打开命令面板，输入C/C++






关闭这个界面后，可以看到工程根目录多了.vscode文件夹，里面有一个c_cpp_properties.json文件，内容为

```
{
    "configurations": [
        {
            "name": "Win32",
            "includePath": [
                "${workspaceFolder}/**"
            ],
            "defines": [
                "_DEBUG",
                "UNICODE",
                "_UNICODE"
            ],
            "compilerPath": "D:\\mingw64\\bin\\g++.exe",
            "cStandard": "c17",
```

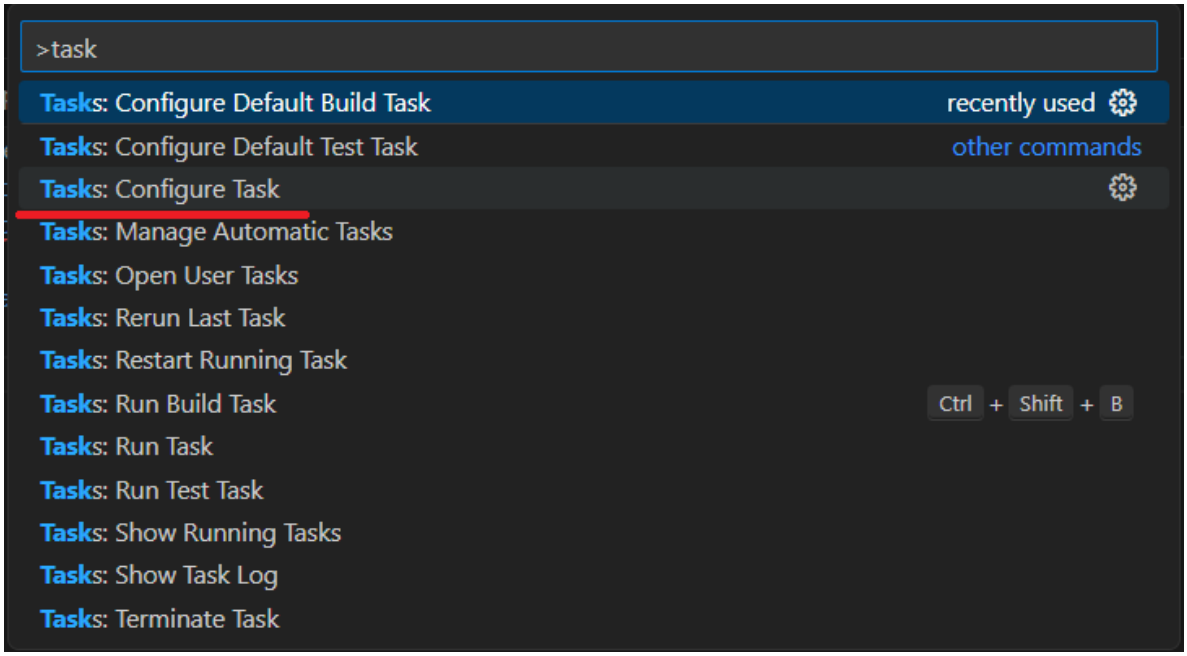
```
            "cppStandard": "c++17",
            "intelliSenseMode": "windows-gcc-x64",
            "configurationProvider": "ms-vscode.cmake-tools"
        }
    ],
    "version": 4
}
```

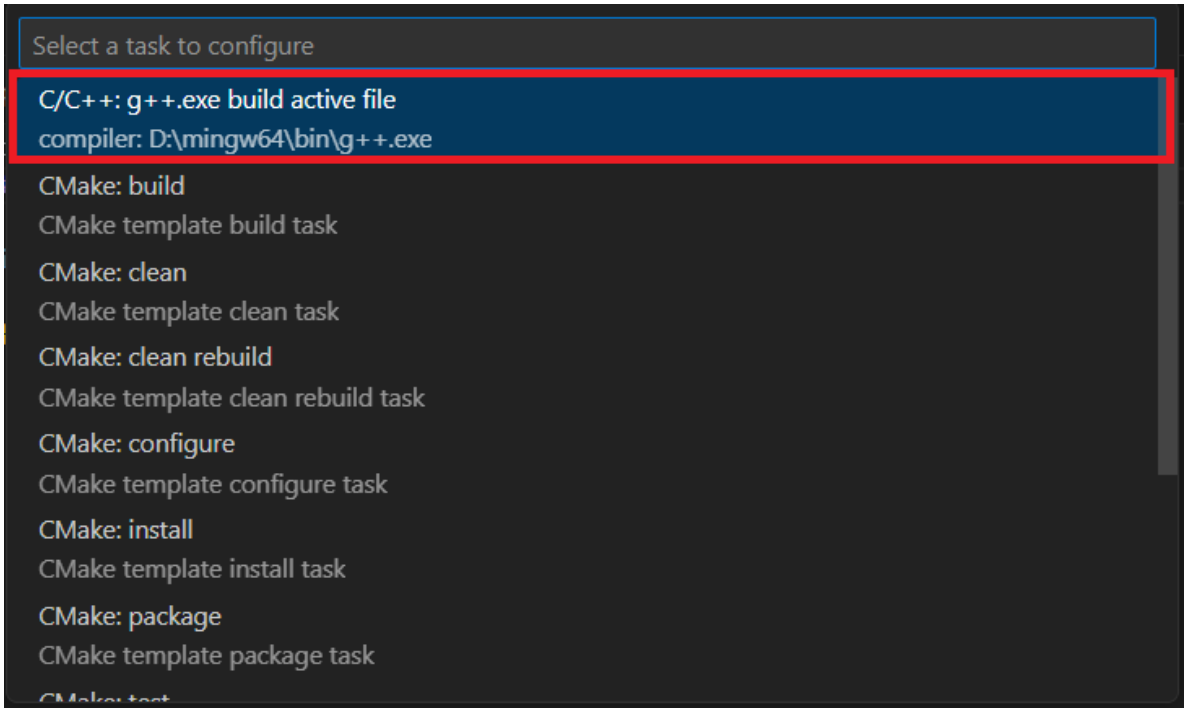
## 7.5 配置并执行编译任务

各种任务，包括编译任务都是在`tasks.json`文件里进行配置（tasks.json文件里可以定义多个任务。熟悉json的同学可以看到，tasks属性的类型是数组[ ]）。

首先打开C++文件main.cpp

按ctrl+shift+p打开命令面板，输入task



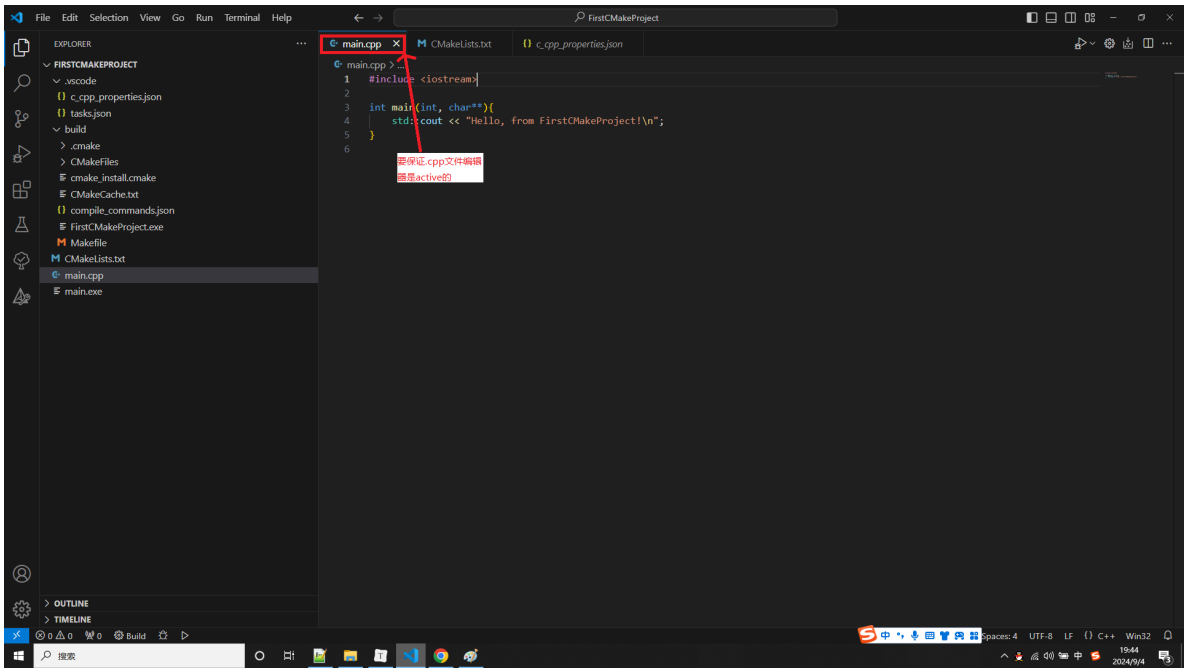




选择红框里的内容。注意必须打开cpp文件才会出现这个选项。这个时候.vscode目录里会出现tasks.json文件，内容为

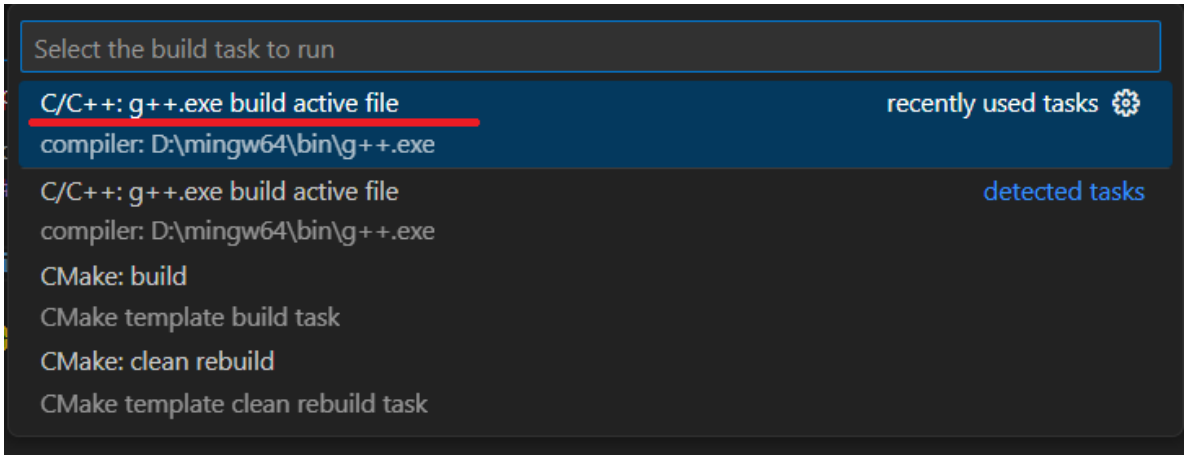
```
{
    "version": "2.0.0",
    "tasks": [
        {
            "type": "cppbuild",
            "label": "C/C++: g++.exe build active file",
            "command": "D:\\mingw64\\bin\\g++.exe",
            "args": [
                "-fdiagnostics-color=always",
                "-g",
                "${file}",
                "-o",
                "${fileDirname}\\${fileBasenameNoExtension}.exe"
            ],
            "options": {
                "cwd": "${fileDirname}"
            },
            "problemMatcher": [
                "$gcc"
            ],
            "group": "build",
            "detail": "compiler: D:\\mingw64\\bin\\g++.exe"
        }
    ]
}
```

注意tasks里只有一个任务，类型为`cppbuild`，任务的label为`C/C++: g++.exe build active file`，记住这个label，因为马上执行Run Task命令时，就要选择这个labe的任务。

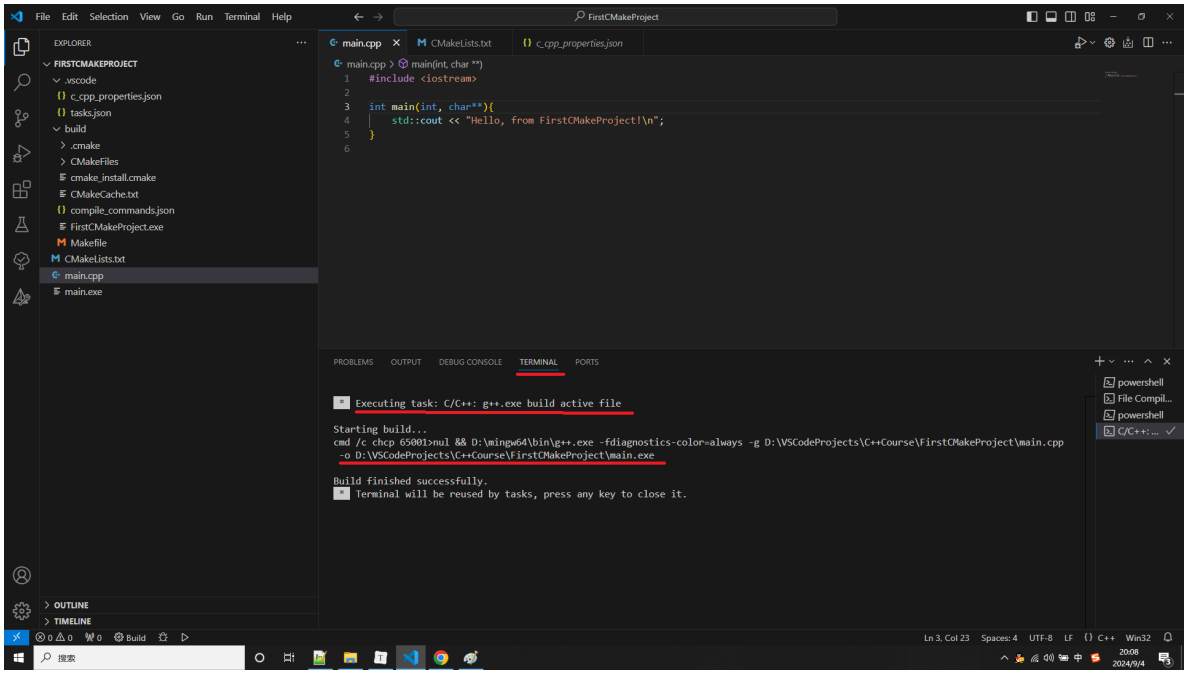
在VSCode里打开main.cpp（而且要保证.cpp文件编辑器是active的）




在点击菜单Terminal->Run Task...，会弹出如下对话框：




选择lebel为`C/C++: g++.exe build active file`的任务，观察下面Terminal的输出




注意是在项目根目录生成的main.exe，因为在tasks.json里有如下配置：

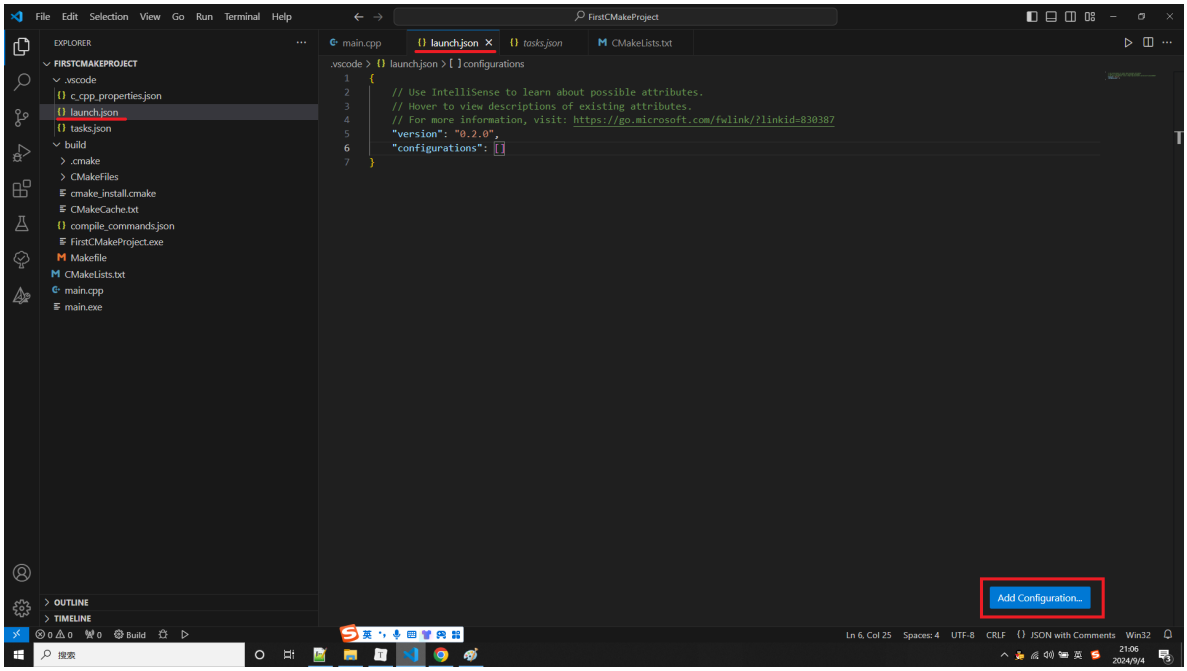
```
            "args": [
                ...
                "-o",
                "${fileDirname}\\${fileBasenameNoExtension}.exe"
            ],
```

`${fileDirname}`就是当前active的cpp文件（main.cpp）所在目录，`${fileBasenameNoExtension}`就是当前active的cpp文件（main.cpp）的文件名（不带后缀名）。`-o`选项指定生成编译好的文件的位置为：`${fileDirname}\\${fileBasenameNoExtension}.exe`。

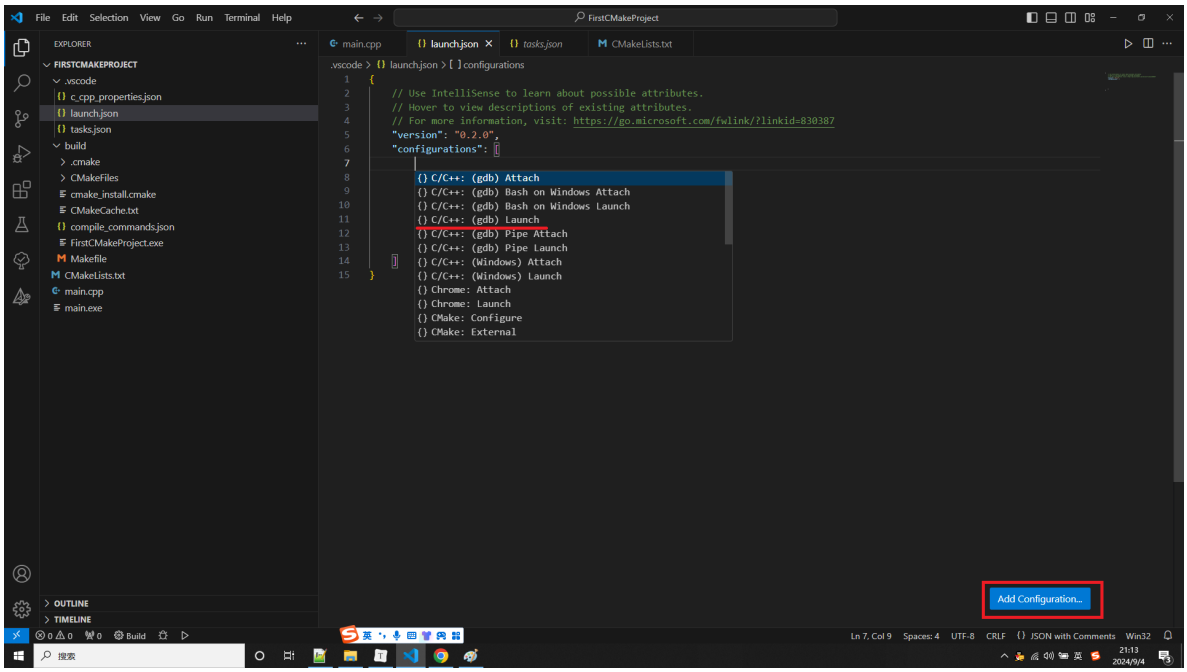
因此上面通过VSCode执行Run Task...来执行编译任务，不是通过CMake。

## 7.6 配置并调试程序

从主菜单中，选择Run > Add Configuration...，然后选择C++ (GDB/LLDB)。然后，将看到各种预定义调试配置的下拉列表。选择**g++.exe build and debug active file**。可以看到会在.vscode目录下创建launch.js目录，但内容只有二行：




点击右下角的Add Configuration...，选择`C/C++(gdb Launch)`：




这时launch.js的内容如下所示：

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?
linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": "(gdb) Launch",
            "type": "cppdbg",
            "request": "launch",
            "program": "enter program name, for example
${workspaceFolder}/a.exe",
            "args": [],
            "stopAtEntry": false,
```

```
            "cwd": "${fileDirname}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "miDebuggerPath": "/path/to/gdb",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                },
                {
                    "description": "Set Disassembly Flavor to Intel",
                    "text": "-gdb-set disassembly-flavor intel",
                    "ignoreFailures": true
                }
            ]
        }

    ]
}
```

更改下面两行，program为可执行文件的位置，miDebuggerPath为自己安装路径

```
"program": "${fileDirname}\\${fileBasenameNoExtension}.exe",

"miDebuggerPath": "D:\\mingw64\\bin\\gdb.exe",
```

现在修改main.cpp，添加一行

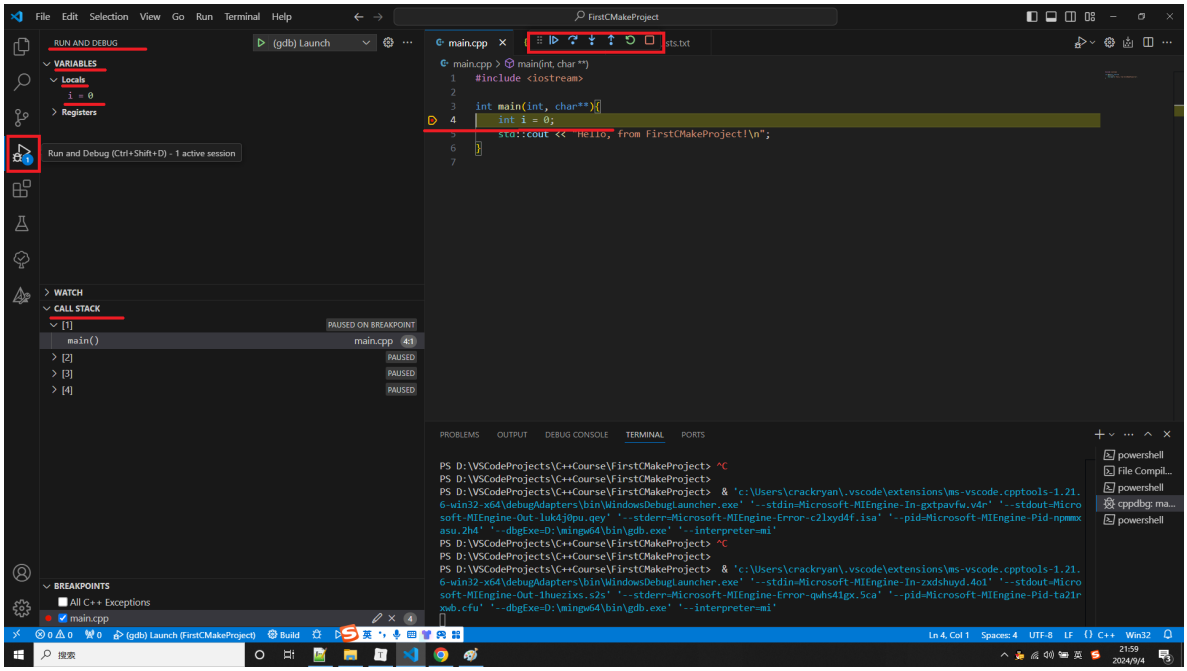
```
#include <iostream>

int main(int, char**){
    int i = 0;
    std::cout << "Hello, from FirstCMakeProject!\n";
}
```

点击菜单Run->Start Debugging（要保证.cpp文件编辑器是active的）启动调试，可以看到Terminal里输出：

```
PS D:\VSCodeProjects\C++Course\FirstCMakeProject>  &
'c:\Users\crackryan\.vscode\extensions\ms-vscode.cpptools-1.21.6-win32-
x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-
sj5rhf14.1a3' '--stdout=Microsoft-MIEngine-Out-es3xh523.sga' '--
stderr=Microsoft-MIEngine-Error-zrdnprx5.kzO' '--pid=Microsoft-MIEngine-Pid-
uiqOvm31.rhp' '--dbgExe=D:\mingw64\bin\gdb.exe' '--interpreter=mi'
Hello, from FirstCMakeProject!
PS D:\VSCodeProjects\C++Course\FirstCMakeProject>
```

从输出可以看到启动了D:\mingw64\bin\gdb.exe作为调试器，但是由于没有设置断点所以并没有中断。

现在在int i= 0这一行设置断点，在点击菜单Run->Start Debugging（快捷键为F5），可以看到VSCode会打开Run and Debug面板，程序进入断点而暂停：




如何单步运行这里就不用多说了。

# 7.7 通过tasks.json运行cmake编译程序

在`tasks.json`文件里添加二个新的任务，分别执行Cmake Configuration和make任务。

Cmake Configuration任务定义如下，**其功能是：编译 CMakeLists.txt，产生Makefile文件。**

```
{
    "version": "2.0.0",
    "tasks": [
        {
            ...
        },
        {
            "type": "shell",
            "label": "cmake configure",
            "command": "cmake",
            "args": [
                "-G",
                "MinGW Makefiles",
                "-S",
                "${workspaceFolder}",
                "-B",
                "${workspaceFolder}\\build"
            ]
        },
        {
            ...
        }
    ]
}
```

任务 label为`cmake configure`，执行命令为`cmake`，命令行参数为: `-G MinGW Makefiles -S ${workspaceFolder} -B ${workspaceFolder}\\build`，其中`${workspaceFolder}`为当前项目的根目录。`-G MinGW Makefiles`的含义7.3节已经有描述；`-S ${workspaceFolder}`指定包含根CMakeLists.txt 文件的目录；`-B ${workspaceFolder}\\build`指定生成的构建目录，即将CMake生

成的Makefile或项目文件保存到指定的目录中，即保存到项目的build目录里。

make任务的定义如下，其功能是：**执行`mingw32-make`命令，编译生成二进制代码。即这个任务才是真正的编译生成二进制文件的任务。**

```
{
    "version": "2.0.0",
    "tasks": [
        {
            ...
        },
        {
            ...
        },
        {
            "label": "make",
            "command": "mingw32-make.exe",
            "args": [
                "-C",
                "${workspaceFolder}\\build"
            ]
        }
    ]
}
```

任务的label为`make`；执行的命令为`mingw32-make.exe`，mingw32-make是一个在Windows系统上使用的make工具，属于MinGW构建系统的一部分。 它主要用于编译和构建项目，支持并行执行多个编译任务；命令行参数为：`-C ${workspaceFolder}\\build`，其含义为进入到`${workspaceFolder}\\build`目录来执行`mingw32-make.exe`命令（因为执行`mingw32-make.exe`命令所需的Makefile文件及其它文件都在`${workspaceFolder}\\build`目录下）。

注意：`mingw32-make.exe`命令会自动在`${workspaceFolder}\\build`下寻找Makefile文件。

最后，完整的tasks.json文件的内容为：

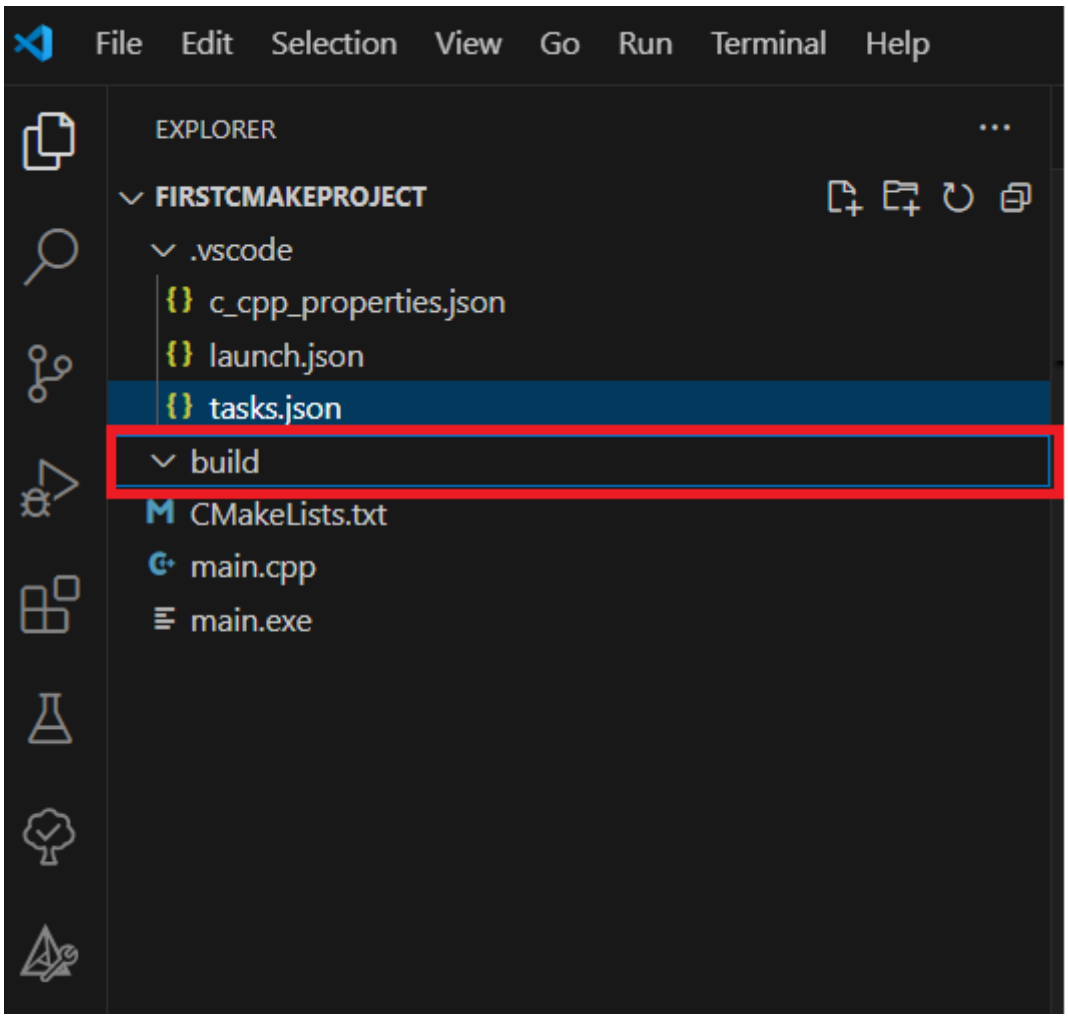
```
{
    "version": "2.0.0",
    "tasks": [
        {
            "type": "cppbuild",
            "label": "C/C++: g++.exe build active file",
            "command": "D:\\mingw64\\bin\\g++.exe",
            "args": [
                "-fdiagnostics-color=always",
                "-g",
                "${file}",
                "-o",
                "${fileDirname}\\${fileBasenameNoExtension}.exe"
            ],
            "options": {
                "cwd": "${fileDirname}"
            },
            "problemMatcher": [
                "$gcc"
            ],
            "group": "build",
            "detail": "compiler: D:\\mingw64\\bin\\g++.exe"
```

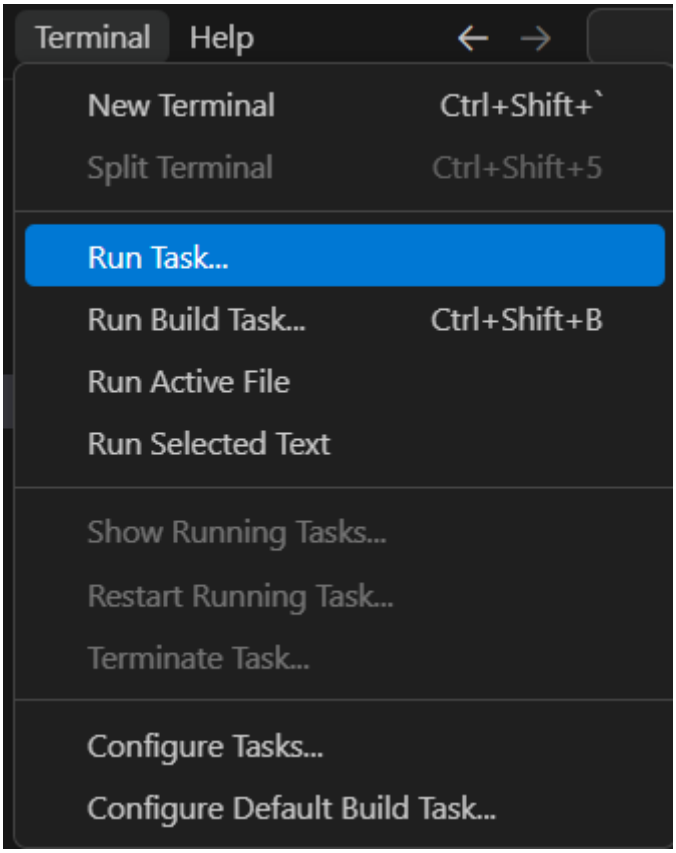
```
        },
        {
            "type": "shell",
            "label": "cmake configure",
            "command": "cmake",
            "args": [
                "-G",
                "MinGW Makefiles",
                "-S",
                "${workspaceFolder}",
                "-B",
                "${workspaceFolder}\\build"
            ]
        },
        {
            "label": "make",
            "command": "mingw32-make.exe",
            "args": [
                "-C",
                "${workspaceFolder}\\build"
            ]
        }
    ]
}
```

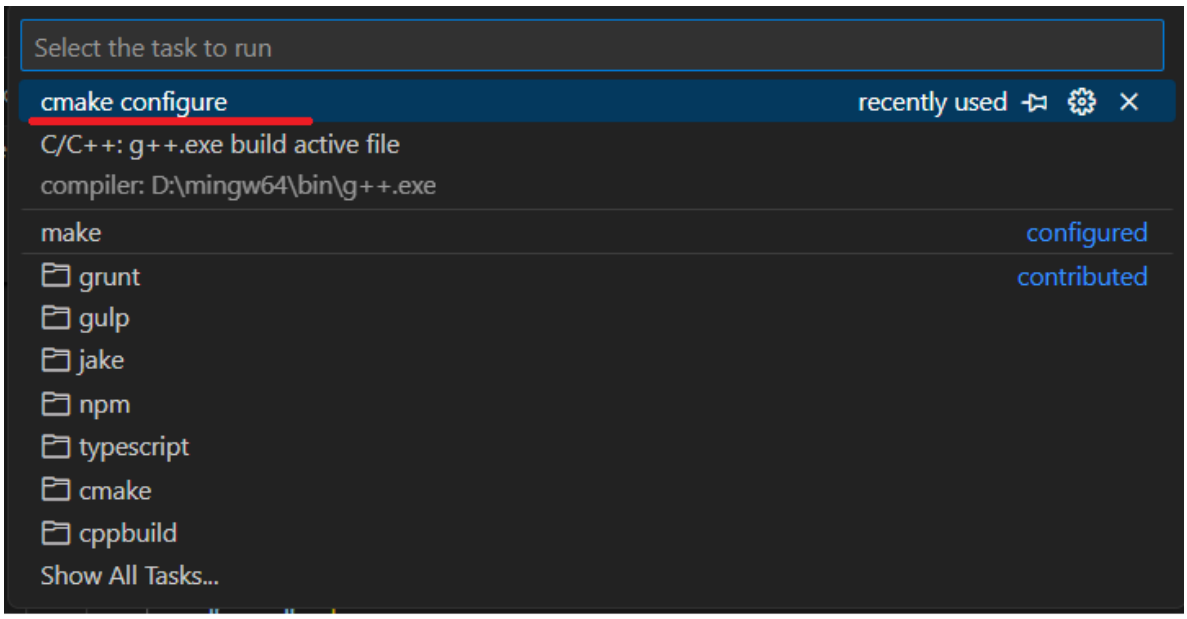
里面一共定义了三个任务，分别为

- C/C++: g++.exe build active file：直接用g++来编译程序（不是使用cmake），7.5节有介绍
- cmake configure：编译 CMakeLists.txt，产生Makefile文件
- make：执行`mingw32-make`命令，编译生成二进制代码

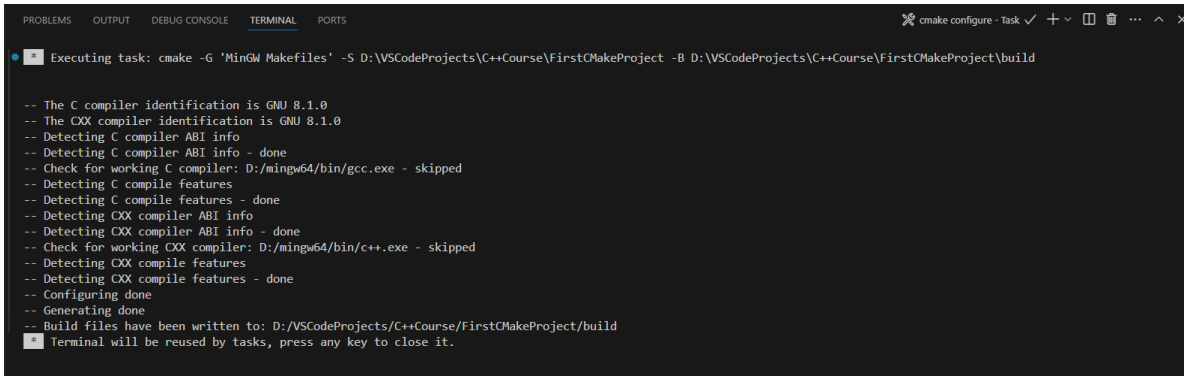
现在来运行cmake configure任务，为了观察效果，首先删除build目录里的所有内容，但是保留build目录

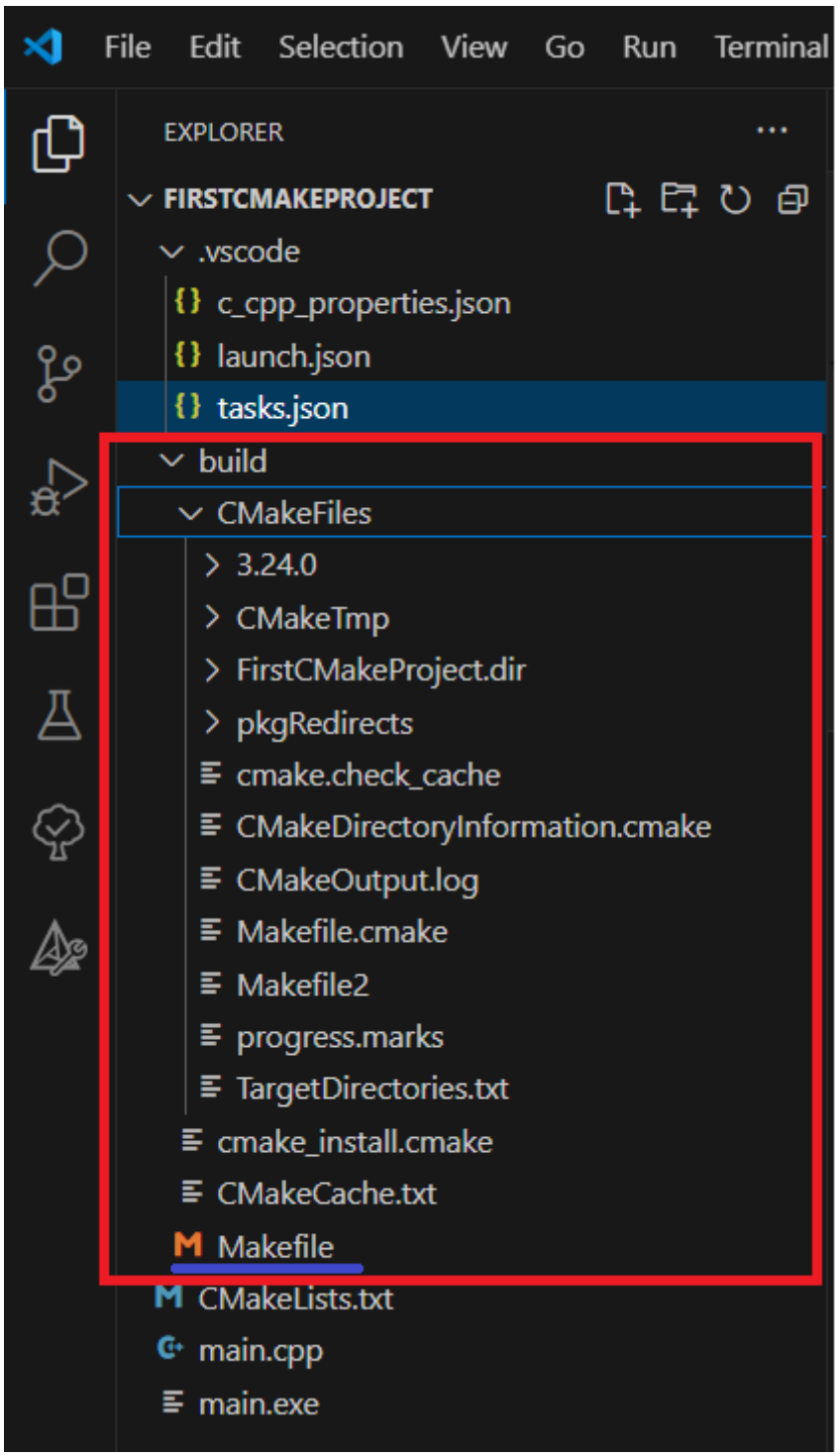

File Edit Selection View Go Run Terminal Help
EXPLORER
FIRSTCMAKEPROJECT
.vscode
c_cpp_properties.json
launch.json
tasks.json
build
CMakeLists.txt
main.cpp
main.exe


Terminal Help
New Terminal Ctrl+Shift+`
Split Terminal Ctrl+Shift+5
Run Task...
Run Build Task... Ctrl+Shift+B
Run Active File
Run Selected Text
Show Running Tasks...
Restart Running Task...
Terminate Task...
Configure Tasks...
Configure Default Build Task...


Select the task to run
cmake configure
recently used
C/C++: g++.exe build active file
compiler: D:\mingw64\bin\g++.exe
make
configured
grunt
contributed
gulp
jake
npm
typescript
cmake
cppbuild
Show All Tasks...

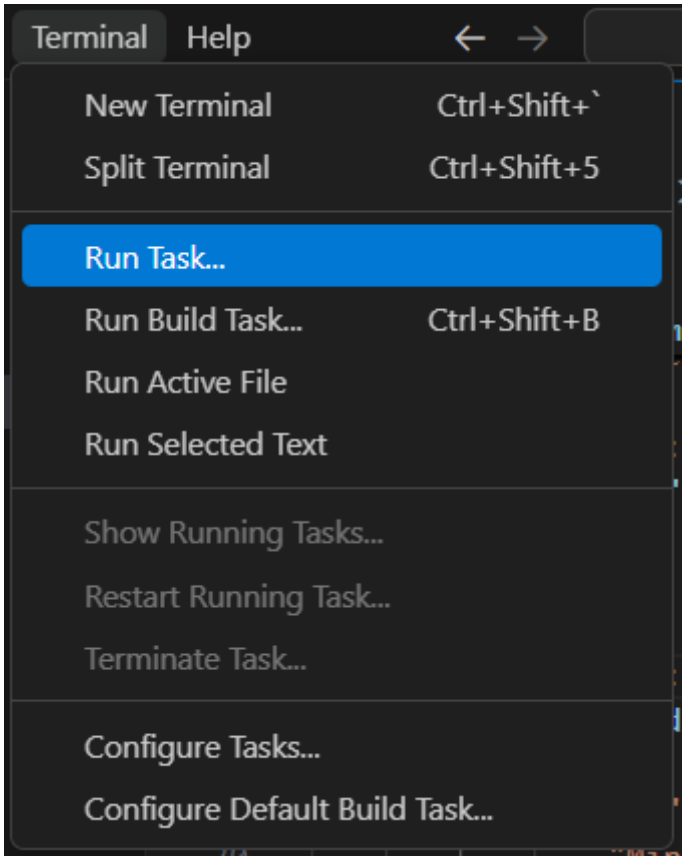
可以看到输出：

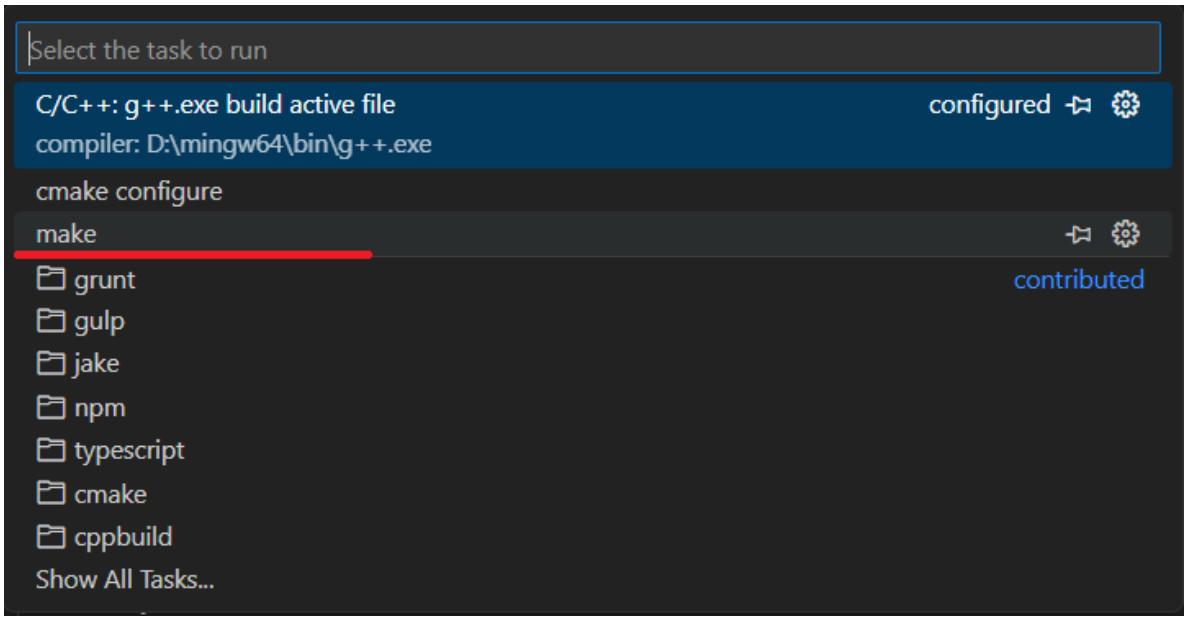

PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
cmake configure - Task
Executing task: cmake -G 'MinGW Makefiles' -S D:\VSCodeProjects\C++Course\FirstCMakeProject -B D:\VSCodeProjects\C++Course\FirstCMakeProject\build
-- The C compiler identification is GNU 8.1.0
-- The CXX compiler identification is GNU 8.1.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: D:/mingw64/bin/gcc.exe - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: D:/mingw64/bin/c++.exe - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: D:/VSCodeProjects/C++Course/FirstCMakeProject/build
Terminal will be reused by tasks, press any key to close it.

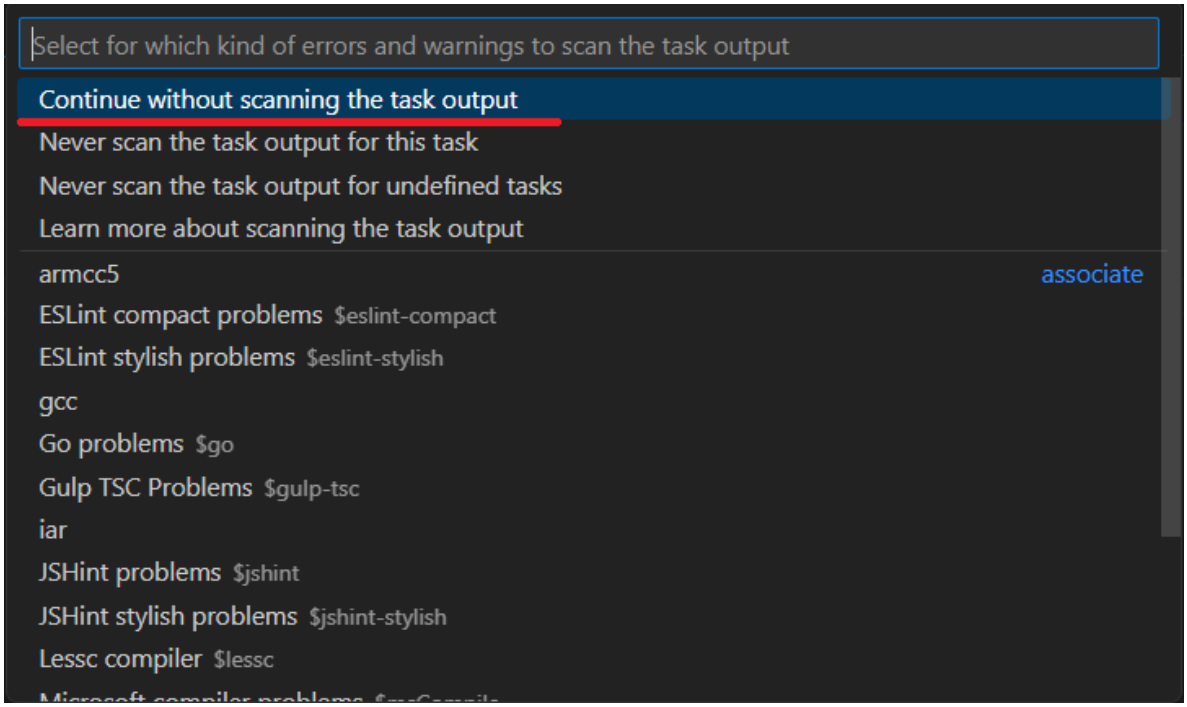



可以看到build目录里产生了一大堆东西，其中最重要的就是Makefile文件（蓝色下划线标出的，可以阅读里面的内容）。下一步就是用`mingw32-make`命令，去执行Makefile文件里面的内容，编译产生二进制执行文件。

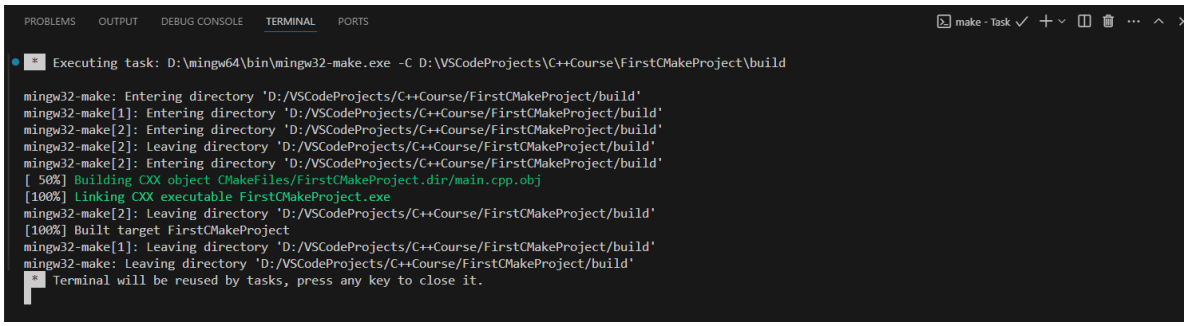
现在来运行make任务，这才是真正的编译：


Terminal
Help
New Terminal Ctrl+Shift+`
Split Terminal Ctrl+Shift+5
Run Task...
Run Build Task... Ctrl+Shift+B
Run Active File
Run Selected Text
Show Running Tasks...
Restart Running Task...
Terminate Task...
Configure Tasks...
Configure Default Build Task...

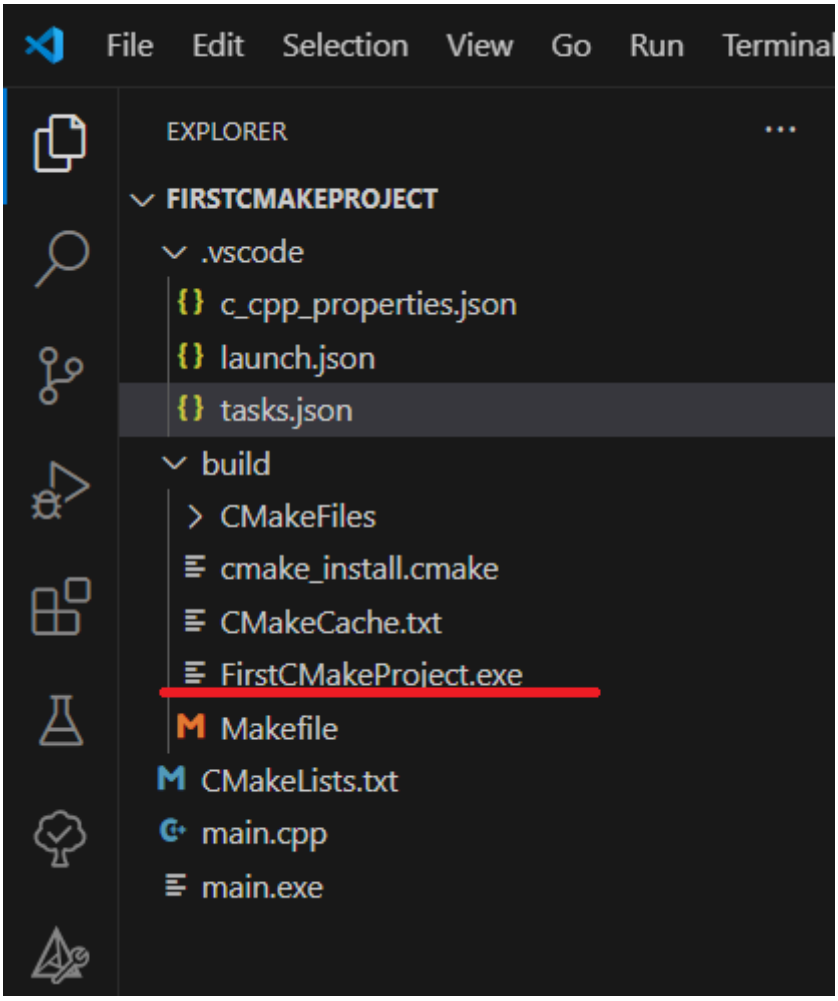

Select the task to run
C/C++: g++.exe build active file
configured
compiler: D:\mingw64\bin\g++.exe
cmake configure
make
grunt
contributed
gulp
jake
npm
typescript
cmake
cppbuild
Show All Tasks...




可以看到输出




在build目录下产生了文件：




**备注1**：只有当CMakeLists.txt文件的内容有变化时，才需要执行cmake configure任务，然后再执行make任务。如果只是C++源代码有变化，只需要执行make任务。

**备注2**：通过CMake编译的代码也可以通过launch.json进行调试，只需要修改launch.json里面的这一行代码

```
"program": "${workspaceFolder}\\build\\FirstCMakeProject.exe",
```

让program指向CMake生成的exe文件就可以了。

# 8: 复杂工程模板

第7章只是通过一个非常简单的C++工程（只有一个main.cpp文件）来说明VSCode里面如何用CMake编译运行调试程序。但是一个复杂的C++工程，包含有很多.hpp和.cpp文件，可能还包含有测试文件、脚本文件、文档、第三方依赖库等，因此本章构建一个复杂工程的模板，方便大家创建复杂工程。

## 8.1 复杂工程的结构

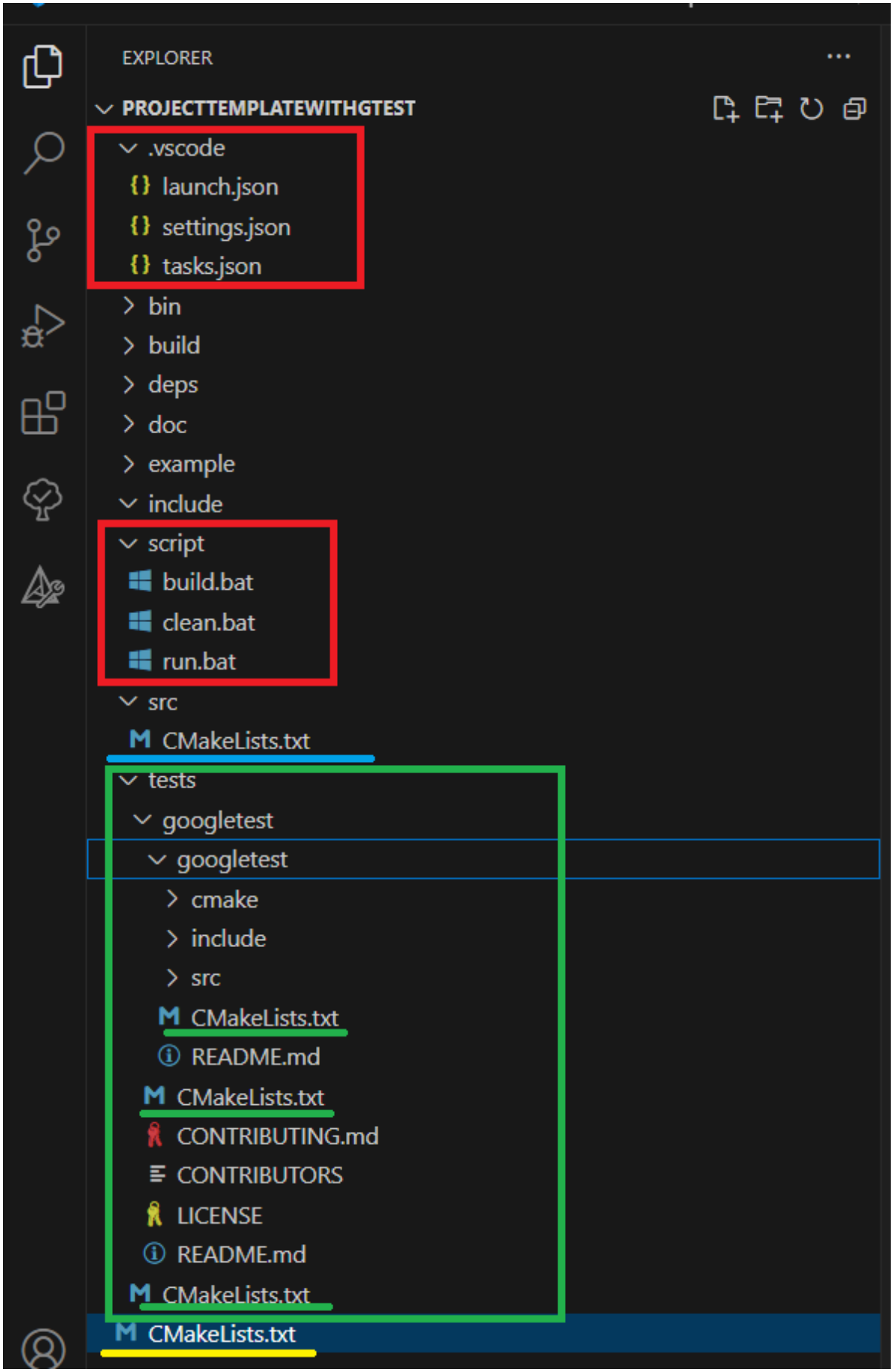
复杂工程的结构如下所示

```
/your-project-name
  /.vscode
  /build              # CMake生成的构建文件和可执行文件将放在这里
  /example            # 一些例子或者demo
  /doc                # API文档或者设计文档
  /tests              # 测试代码，基于Google Test (gtest)
  /deps               # 第三方依赖库(.lib,.dll)
```

```
  /script             # 一些脚本，比如通过命令行方式编译运行代码的脚本文件
  /include            # 头文件，特别注意对于复杂工程，支持include下创建子目录，放置不同模块
下的头文件
    - hello.hpp       # 示例头文件
  /src                # 放置源文件，特别注意对于复杂工程，支持src下创建子目录，放置不同模块下
的头文件
    - main.cpp        # 源文件
    - hello.cpp       # 示例实现文件
  CMakeLists.txt      # 根CMake配置文件
```

## 8.2 模板工程

创建了一个模板工程ProjectTemplateWithGTest，其内容为：




### 8.2.1 .vscode目录下的三个json文件

.vscode目录下有三个json文件，这三个json文件的作用第7章里都有描述。内容分别为：

tasks.json

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "type": "shell",
            "label": "cmake configure",
            "command": "cmake",
            "args": [
                "-DCMAKE_BUILD_TYPE=Debug",
                "-G",
                "MinGW Makefiles",
                "-S",
                "${workspaceFolder}",
                "-B",
                "${workspaceFolder}\\build"
            ]
        },
        {
            "label": "make",
            "command": "mingw32-make.exe",
            "args": [
                "-C",
                "${workspaceFolder}\\build"
            ]
        }
    ]
}
```

**需要特别说明的是：命令行参数`-DCMAKE_BUILD_TYPE=Debug`是必须的，否则debug时程序不会在断点中断。**

launch.json

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?
linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": "(gdb) Launch",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceFolder}\\bin\\${workspaceFolderBasename}-
main.exe",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${workspaceFolder}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "miDebuggerPath": "D:\\mingw64\\bin\\gdb.exe",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
```

```
                    "ignoreFailures": true
                },
                {
                    "description": "Set Disassembly Flavor to Intel",
                    "text": "-gdb-set disassembly-flavor intel",
                    "ignoreFailures": true
                }
            ]
        }
    ]
}
```

**特别特别注意：launch.json里面的`miDebuggerPath`属性要换成你们机器上`gdb.exe`的绝对路径。**

```
"miDebuggerPath": "你机器上的gdb.exe的完整绝对路径",
```

settings.json

```
{
    "files.associations": {
        "any": "cpp",
        "array": "cpp",
        "atomic": "cpp",
        "bit": "cpp",
        "*.tcc": "cpp",
        "cctype": "cpp",
        "chrono": "cpp",
        "clocale": "cpp",
        "cmath": "cpp",
        "compare": "cpp",
        "concepts": "cpp",
        "condition_variable": "cpp",
        "cstdarg": "cpp",
        "cstddef": "cpp",
        "cstdint": "cpp",
        "cstdio": "cpp",
        "cstdlib": "cpp",
        "cstring": "cpp",
        "ctime": "cpp",
        "cwchar": "cpp",
        "cwctype": "cpp",
        "deque": "cpp",
        "list": "cpp",
        "map": "cpp",
        "set": "cpp",
        "unordered_map": "cpp",
        "unordered_set": "cpp",
        "vector": "cpp",
        "exception": "cpp",
        "algorithm": "cpp",
        "functional": "cpp",
        "iterator": "cpp",
        "memory": "cpp",
        "memory_resource": "cpp",
        "numeric": "cpp",
        "optional": "cpp",
```

```
        "random": "cpp",
        "ratio": "cpp",
        "string": "cpp",
        "string_view": "cpp",
        "system_error": "cpp",
        "tuple": "cpp",
        "type_traits": "cpp",
        "utility": "cpp",
        "fstream": "cpp",
        "initializer_list": "cpp",
        "iomanip": "cpp",
        "iosfwd": "cpp",
        "iostream": "cpp",
        "istream": "cpp",
        "limits": "cpp",
        "mutex": "cpp",
        "new": "cpp",
        "ostream": "cpp",
        "ranges": "cpp",
        "sstream": "cpp",
        "stdexcept": "cpp",
        "stop_token": "cpp",
        "streambuf": "cpp",
        "thread": "cpp",
        "typeinfo": "cpp",
        "variant": "cpp"
    },
    "editor.formatOnPaste": true,
    "editor.formatOnSave": true,
    "terminal.integrated.tabs.defaultIcon": "console",
    "terminal.integrated.defaultProfile.windows": "PowerShell",
    "terminal.integrated.profiles.windows": {
        "PowerShell": {
            "source": "PowerShell",
            "icon": "terminal-powershell",
            "args": [
                "-NoExit",
                "/c",
                "chcp 65001"
            ],
        },
        "Command Prompt": {
            "path": [
                "${env:windir}\\System32\\cmd.exe"
            ],
            "args": [
                "/K",
                "chcp 65001"
            ],
            "icon": "terminal-cmd"
        },
        "Git Bash": {
            "source": "Git Bash"
        }
    }
}
```

**需要特别说明的是，settings.json中这几行代码是为了解决在Terminal中中文显示乱码的问题**

```
"terminal.integrated.profiles.windows": {
        "PowerShell": {
            "source": "PowerShell",
            "icon": "terminal-powershell",
            "args": [
                "-NoExit",
                "/c",
                "chcp 65001"
            ],
        },
        ...
    }
```

这几行的作用是：当在VScode里打开Terminal时（缺省时是Powershell），会自动将终端字符集编码设置为UTF-8（通过命令行选项：/c chcp 65001）。因为VScode里面所有文本文件的编码格式我们都统一设为UTF-8。

## 8.2.2 CMakeLists.txt文件

对于一个复杂项目，可能需要按多级目录来管理。每一级目录下面都可以定义该目录的CMakeLists.txt。CMake会递归地进入每个子目录，并根据各自的`CMakeLists.txt`文件来编译相应的目标。

工程模板的CMake的层级结构如8.1节的图所示，现在详细描述如下：

```
/project-root-directory
    /src
        CMakeLists.txt  #src下面的CMake配置文件，用于将src下面的代码构建成.a库文件，最后和tests目录下的测试代码链接，形成exe文件
    /tests
        /googletest     #googletest所有东西在这里
            /googletest #googletest的源码
                CMakeLists.txt  # tests/googletest/googletest下面的CMake配置文件,用于构建googletest的二个静态库
            CMakeLists.txt  #tests/googletest下面的CMake配置文件,用于构建googletest
        CMakeLists.txt  #tests下面的CMake配置文件,用于将tests目录下的测试代码编译成最终的exe文件
    CMakeLists.txt   #根CMake配置文件,位于项目根目录
```

### 1）项目根目录的CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)

PROJECT("ProjectTemplateWithGTest")

SET(CMAKE_CXX_STANDARD 14)
SET(CMAKE_CXX_FLAGS "-Wno-deprecated-declarations -Wall -Werror -Wnon-virtual-dtor -Woverloaded-virtual")

#可执行文件输出目录
SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
#构建的库的输出目录
SET(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
```

```
#构建test 目标
ENABLE_TESTING()
#向当前项目添加一个子目录，并且使得 CMake 在构建过程中进入该子目录继续处理其中的
CMakeLists.txt 文件。
#这个命令通常用于管理项目中的多个模块或子项目。
ADD_SUBDIRECTORY(src)
ADD_SUBDIRECTORY(tests)
```

**需要特别说明的是**，当基于模板工程去创建一个实际的工程时（8.3节会介绍），**唯一需要修改CMakeLists.txt的地方就是**

```
project("ProjectTemplateWithGTest")
```

**需要将`ProjectTemplateWithGTest`换成你创建的实际工程的根目录的目录名称。**

## 2）src目录的CMakeLists.txt

```
#这是位于src下面的CMakeLists.txt，用于将src下面的所有代码构建成静态库

#${CMAKE_SOURCE_DIR}是项目的根目录
message("CMAKE_SOURCE_DIR: " ${CMAKE_SOURCE_DIR})
#将项目根目录下的include和src子目录都加到变量INCLUDE里
SET(INCLUDE ${CMAKE_SOURCE_DIR}/include ${CMAKE_SOURCE_DIR}/src)
message("INCLUDE: " ${INCLUDE})

#查找src下面所有cpp文件（遇到src的子目录也会递归查找），查找结果放在变量SOURCE里
FILE(GLOB_RECURSE SOURCE "*.cpp")
message("SOURCE: " ${SOURCE})

#CMAKE_CURRENT_SOURCE_DIR是当前正在处理的 CMakeLists.txt 所在的源代码目录路径，即:项目根
目录\src
message("CMAKE_CURRENT_SOURCE_DIR: " ${CMAKE_CURRENT_SOURCE_DIR})
#变量SOURCE里面已经是src下面所有cpp文件，现在将 项目根目录\src\Main.cpp和main.cpp从
SOURCE里面删除
#因为src的内容最终不是构建成exe文件，而是库，所以要删除main函数
LIST(REMOVE_ITEM SOURCE ${CMAKE_CURRENT_SOURCE_DIR}/Main.cpp)
LIST(REMOVE_ITEM SOURCE ${CMAKE_CURRENT_SOURCE_DIR}/main.cpp)


#显示所有警告。
ADD_COMPILE_OPTIONS("-Wall")
#将所有的编译器警告视为错误
ADD_COMPILE_OPTIONS("-Werror")
#将${SOURCE}里面包含的所有cpp文件编译构建成一个库（lib），类型是static库
#库的文件名为：lib${PROJECT_NAME}.a，其中${PROJECT_NAME}是项目根目录下的CMakeLists.txt
里面定义的：PROJECT("项目名称")
ADD_LIBRARY(${PROJECT_NAME} ${SOURCE})


#Specify include directories to use when compiling a given target.（为构建目标添加
头文件搜索目录）
#The named <target> must have been created by a command such as add_executable()
or add_library() and must not be an ALIAS target.
TARGET_INCLUDE_DIRECTORIES(${PROJECT_NAME} PUBLIC "${INCLUDE}")
```

**需要说明的是：src目录的CMakeLists.txt用于将src下面的所有代码构建成静态库，用于gtest的测试，见3）tests目录下的CMakeLists.txt的说明。**

### 3）tests目录下的CMakeLists.txt

```
#这个CMakeLists.txt位于test目录下
#用于构建tests目录下的测试代码，最终构建成exe文件

#构建系统时禁用GMock
#设置缓存变量BUILD_GMOCK为OFF，变量类型为BOOL
#FORCE是指如果缓存里已经有该变量BUILD_GMOCK，则强制用新的值覆盖
#""是docstring，用于在cmake-gui里显示该命令的介绍
set(BUILD_GMOCK OFF CACHE BOOL "" FORCE)

##添加googletest子目录，并且使得 CMake 在构建过程中进入该子目录继续处理其中的
CMakeLists.txt 文件。
ADD_SUBDIRECTORY(googletest EXCLUDE_FROM_ALL)


#搜索当前目录下所有源文件，并将它们存储在 SOURCE 变量中
AUX_SOURCE_DIRECTORY(. SOURCE)
#然后，这个 SOURCE 变量被用作 add_executable 命令的参数，以构建名为 ${PROJECT_NAME}-
main 的可执行文件。
ADD_EXECUTABLE(${PROJECT_NAME}-main ${SOURCE})

#构建名为 ${PROJECT_NAME}-main 的可执行文件时，需要依赖的库
#    一个是名为${PROJECT_NAME}的库（在src\CMakeLists.txt里面构建）
#    一个是名为gtest_main的库，这个库是在tests\googletest\googletest\CMakeLists.txt里
构建的（120行）
TARGET_LINK_LIBRARIES(${PROJECT_NAME}-main ${PROJECT_NAME} gtest_main)

ADD_TEST(NAME ${PROJECT_NAME}-main COMMAND ${PROJECT_NAME}-main)
```

**需要注意的是：最终是由tests目录下的CMakeLists.txt构建出最终用于代码测试的exe文件，在构建exe文件时，需要link二个库**：

- 一个是名为${PROJECT_NAME}的库（在src\CMakeLists.txt里面构建）
- 一个是名为gtest_main的库，这个库是在tests\googletest\googletest\CMakeLists.txt里构建的（120行）

### 4）tests/googletest下面的CMakeLists.txt

在tests/googletest下面还有二级CMakeLists.txt，用于将googletest构建成静态库，这里不再展开说明。

## 8.2.3 script目录下的三个脚本文件

### 1）build.bat

在Terminal里运行build.bat，可以编译构建我们的项目。即通过命令行的方式来编译我们的程序。注意最后生成的exe文件会放在bin目录下。

```
rem 首先配置cmake，然后make编译程序，生成的exe文件放在bin目录下
@echo off
pushd .

if not exist build (
     mkdir build
)

cd build
cmake -G"MinGW Makefiles" -DCMAKE_BUILD_TYPE=Debug ..
mingw32-make.exe

popd
```

### 2）run.bat

会通过命令行的方式来运行编译好放在bin目录下的exe文件。

```
rem 自动执行bin目录下的exe文件（如果有多个exe文件，会自动执行最后找到的exe文件）
@echo off
pushd .

if not exist bin (
    echo "bin directory not exists!"
    popd
    exit(1)
)

set EXE_FILE=""
for /r ./bin %%f in (*.exe) do (
    set EXE_FILE=%%f
)
echo %EXE_FILE%

if EXE_FILE!=="" (
    echo "exe file not exists in bin directory!"
    popd
    exit(1)
)

%EXE_FILE%

popd
```

### 3）clean.bat

清除build和bin目录里所有内容（会递归删除子目录及其内容）。请确保build和bin子目录里都是自动生成的文件，以免误删除用户自己创建的文件。

```
rem 会清除build和bin目录里所有内容（会递归删除子目录及其内容）
rem 请确保build和bin子目录里都是自动生成的文件，以免误删除用户自己创建的文件
@echo off
pushd .

if not exist build (
    mkdir build
```

```
)

pushd .
cd build
del /f /s /q *.*
for /d %%p in (*) do rmdir "%%p" /s /q
popd


if not exist bin (
    mkdir bin
)
cd bin
del /f /s /q *.*
for /d %%p in (*) do rmdir "%%p" /s /q

popd
```

## 8.3 从模板工程创建一个实际工程的示例

将ProjectTemplateWithGTest复制到另外一个目录下，比如复制到D:\VSCodeProjects\C++Course\HomeworkWithGTest。同时在include下创建子目录ch4_homework，在src下创建子目录ch4_homework。

在include\ch4_homework下加入头文件MyArray.h，内容为

```
#pragma once
/*
    第4章 编程题1
    一维整型数组MyArry的定义如下，请实现相应的函数成员
*/
class MyArray
{
private:
    int size;           // 数组大小
    int *const p ; // 指向动态分配的内存，保存数组的内容

public:
    MyArray(int size = 10);                  // 构造函数，参数size指定数组大小
    MyArray(const MyArray &old);             // 拷贝构造函数，要求实现深拷贝
    // MyArray &operator=(const MyArray &rhs); // 重载=，要求实现深拷贝
    // MyArray(MyArray &&old) noexcept;     // 移动拷贝
    // MyArray &operator=(MyArray &&rhs);       // 移动=
    // ~MyArray();                              // 析构函数，要求能防止反复释放资源
    // int length();                            // 返回数组大小
    // int &get(int index);                 // 返回下标为index的元素，不考虑越界
};
```

在src\ch4_homework下加入cpp文件MyArray.cpp，内容为

```


#include "../../include/ch4_homework/MyArray.h"
#include <iostream>
using namespace std;
```

```
MyArray::MyArray(int size) : size(size), p(new int[size]) { cout << "MyArray
constructor" << endl; }

MyArray::MyArray(const MyArray &old) : size(old.size), p(new int[size])
{
    for (int i = 0; i < size; i++)
    {
        p[i] = old.p[i];
    }
    cout << "MyArray copy constructor" << endl;
}
```

在tests目录下加test.cpp，作为程序的入口

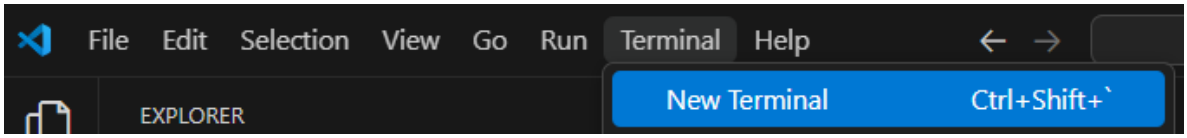
```
#include <gtest/gtest.h>

#include "../include/ch4_homework/MyArray.h"
#include <iostream>
using namespace std;

int myarray_test(){
    cout << "Hello World" << endl;
    MyArray a;
    return 0;
}

// 下面是基于gtest编写的测试
TEST(myarray_test_suit, test_case1)
{
    EXPECT_EQ(0, myarray_test());
}


//也可以不用gtest，编写自己的main函数来启动程序
// int main(){
//     cout << "Hello World" << endl;
//     MyArray a;
//     return 0;
// }
```

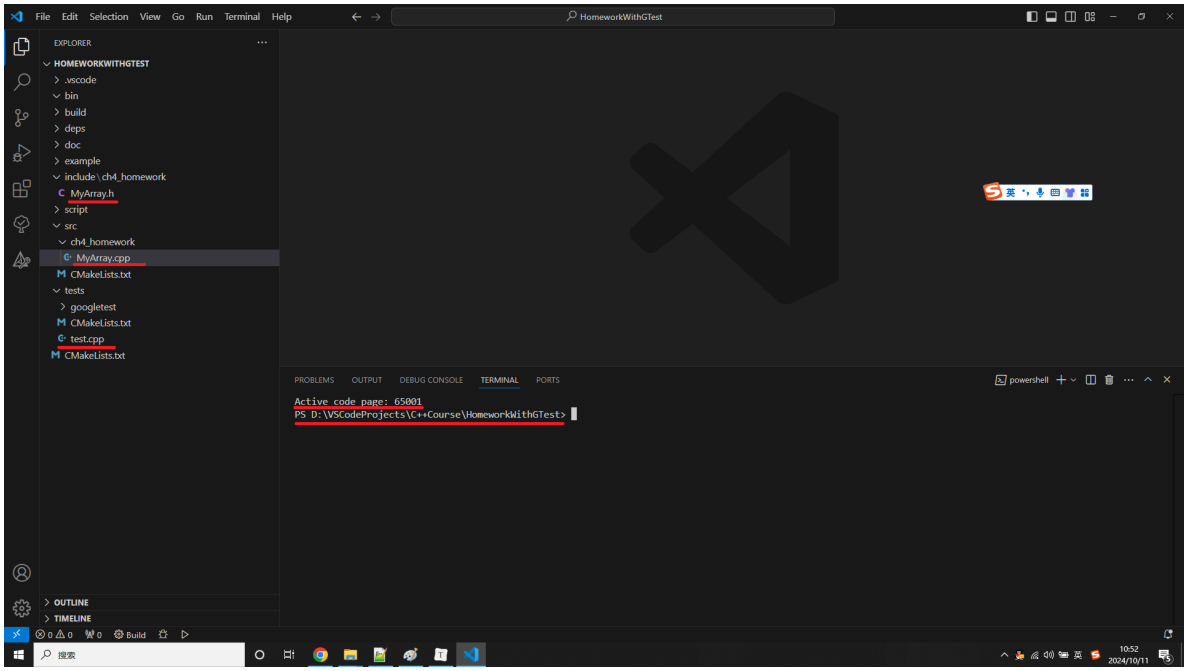
**在HomeworkWithGTest工程里，我们唯一需要修改的配置就是根目录下CMakeLists.txt里面的这一行**

```
project("HomeworkWithGTest")
```

将project的名字换成项目的根目录名`HomeworkWithGTest`。

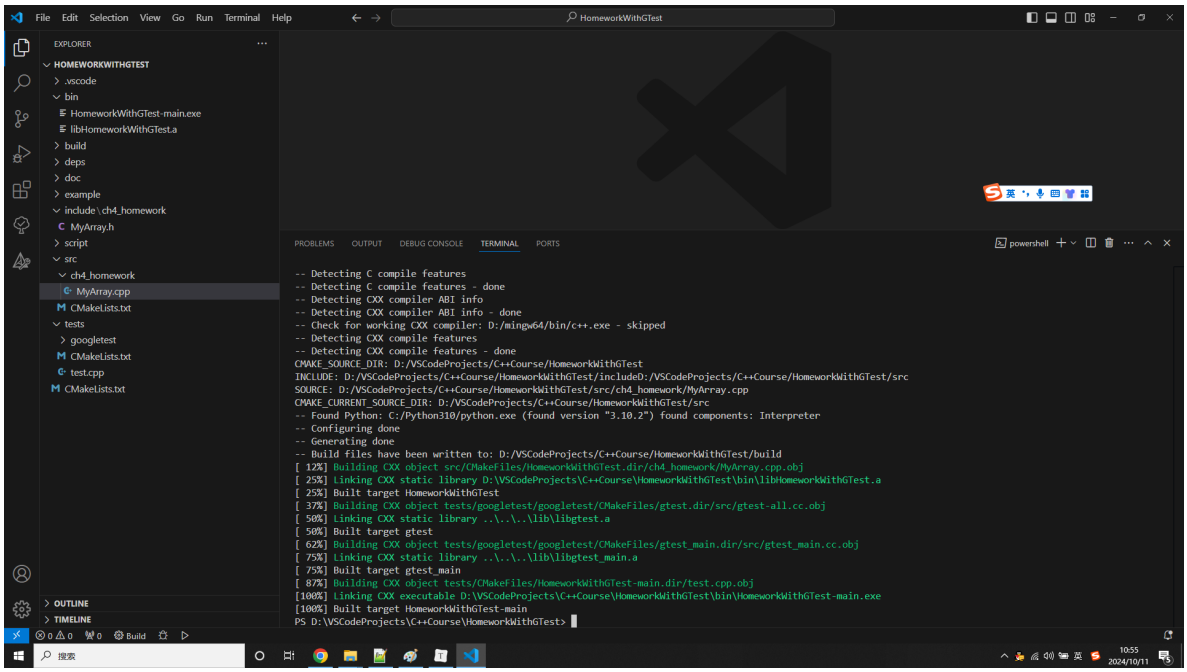
现在开始编译项目，打开Terminal



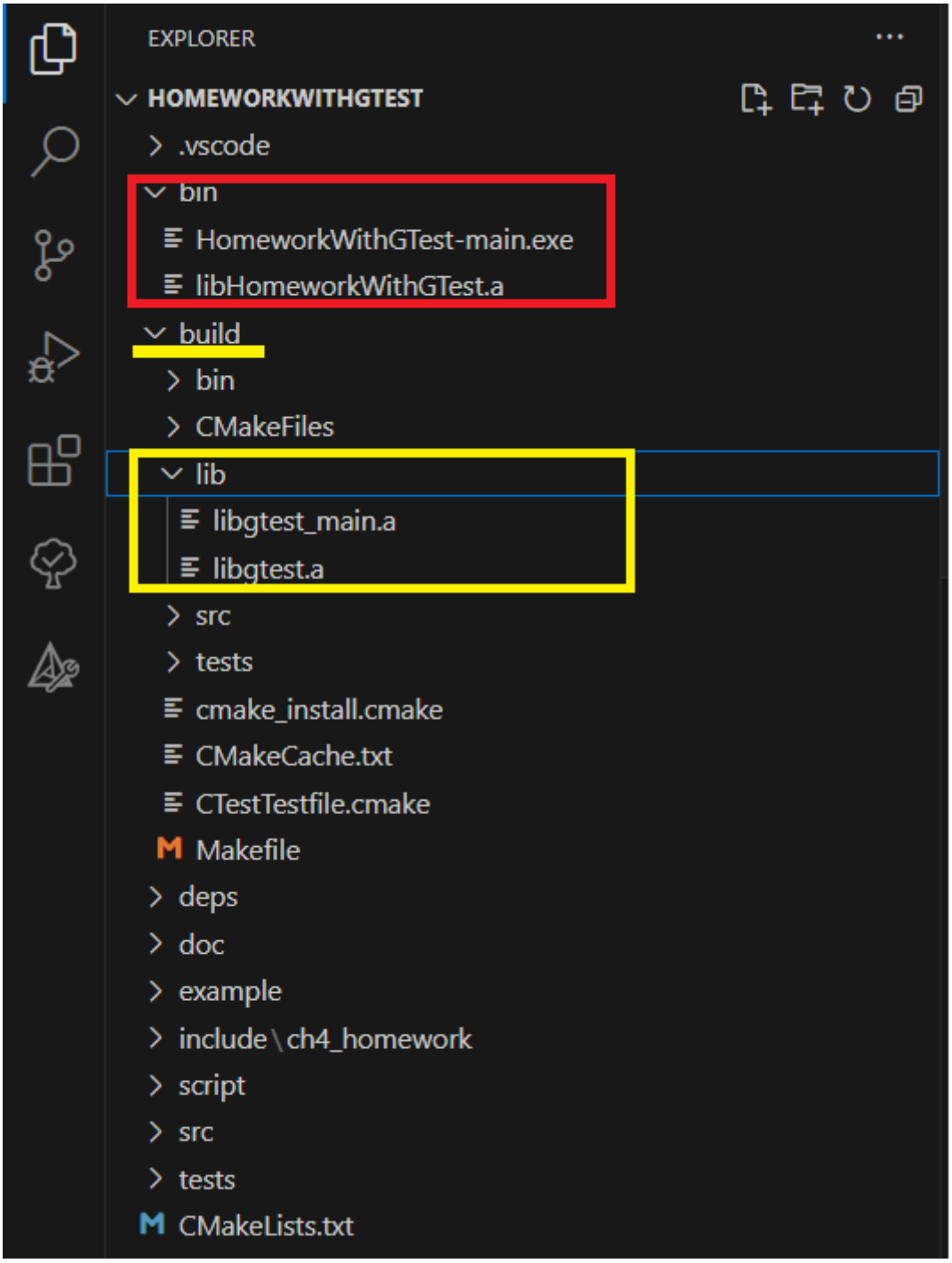



在Terminal窗口里，可以看到编码方式已经切换成65001（UTF-8），当前目录就是项目的根目录，然后在项目的根目录下输入下面的命令

```
.\script\build.bat
```




可以看到，如果编译成功，在bin目录下会产生一个exe文件和一个.a库文件，注意文件名和项目根目录名称的关系。




同时可以看到，在build\lib目录下产生了gtest的二个库文件：libgtest_main.a和libgtest.a。生成bin目录下的HomeworkWithGTest-main.exe时，需要链接如下三个库文件

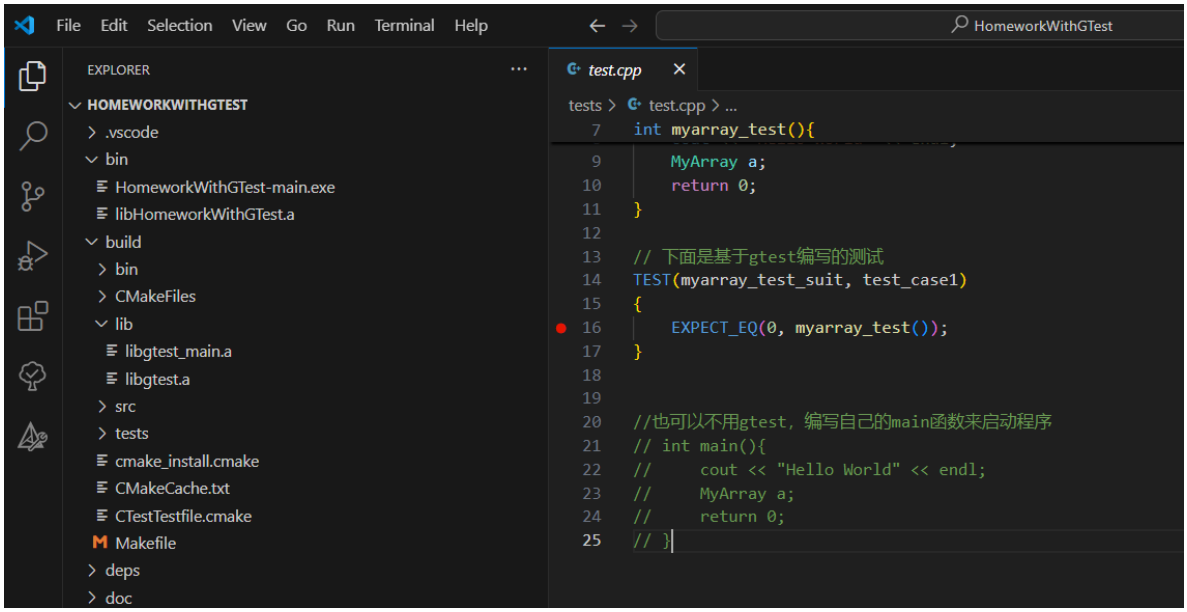
- bin\libHomeworkWithGTest.a
- build\lib\libgtest_main.a
- build\lib\libgtest.a

现在来运行程序。打开Terminal，在项目根目录下执行

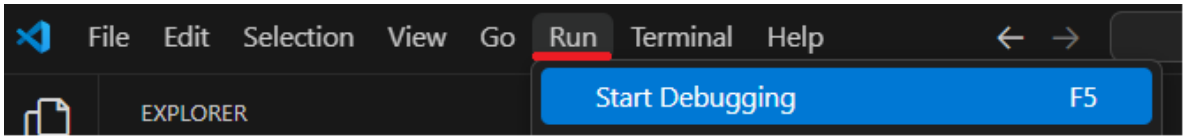
```
.\script\run.bat
```

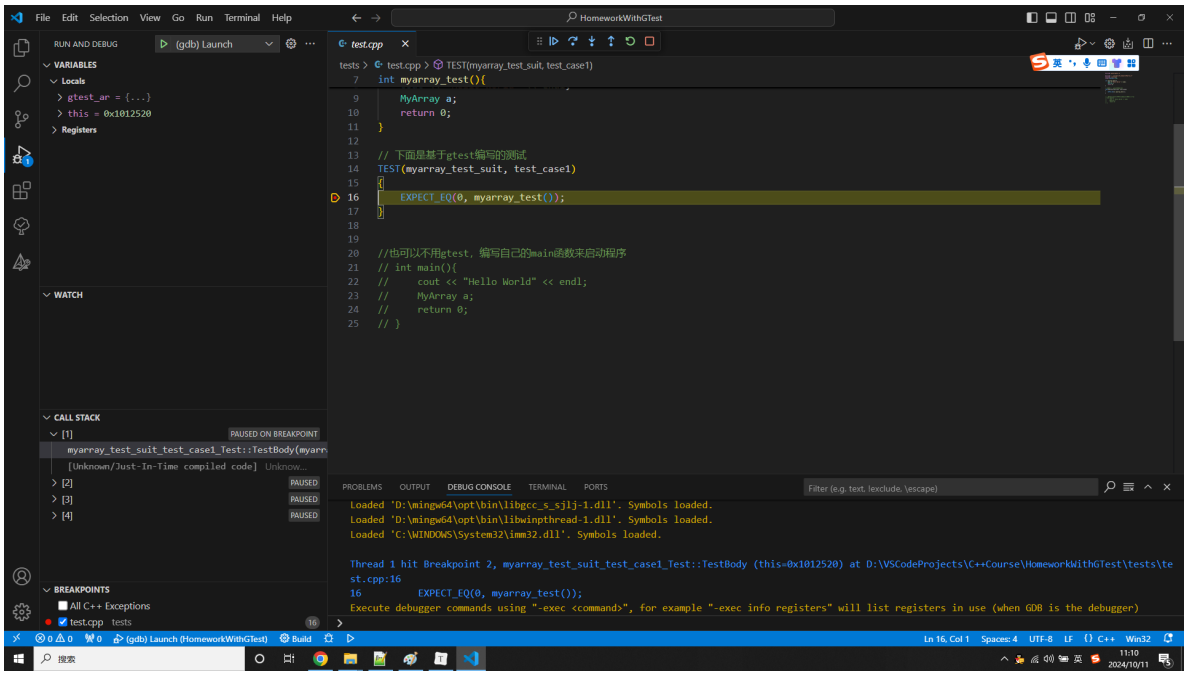
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                    powershell

Active code page: 65001
PS D:\VSCodeProjects\C++Course\HomeworkWithGTest> .\script\run.bat

D:\VSCodeProjects\C+                                    >rem 自动执行bin目录下的exe文件（如果有多个exe文件，会自动执行最后找到的exe文件）
D:\VSCodeProjects\C+ Open file in editor (ctrl + click)  \bin\HomeworkWithGTest-main.exe
Running main() from D:\VSCodeProjects\C++Course\HomeworkWithGTest\tests\googletest\googletest\src\gtest_main.cc
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from myarray_test_suit
[ RUN      ] myarray_test_suit.test_case1
Hello World
MyArray constructor
[       OK ] myarray_test_suit.test_case1 (4 ms)
[----------] 1 test from myarray_test_suit (15 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (43 ms total)
[  PASSED  ] 1 test.
PS D:\VSCodeProjects\C++Course\HomeworkWithGTest>
```

现在来调试程序。VSCode里打开test.cpp，在TEST函数的第1行设置断点




然后保持test.cpp的编辑窗口是Active的，点击菜单




可以看到在断点出程序暂停，我们可以在VSCode里面一步步执行了。