

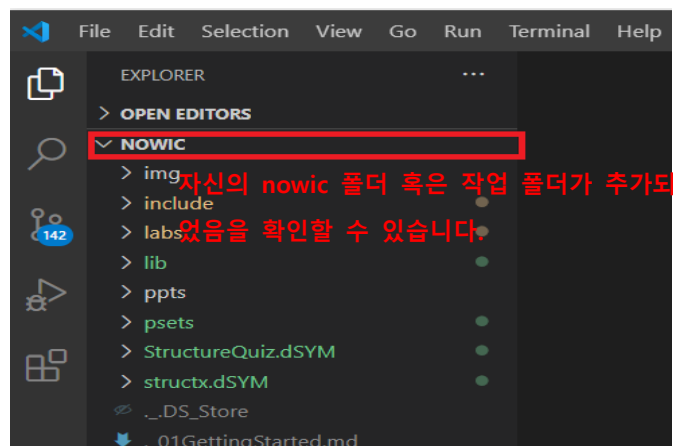
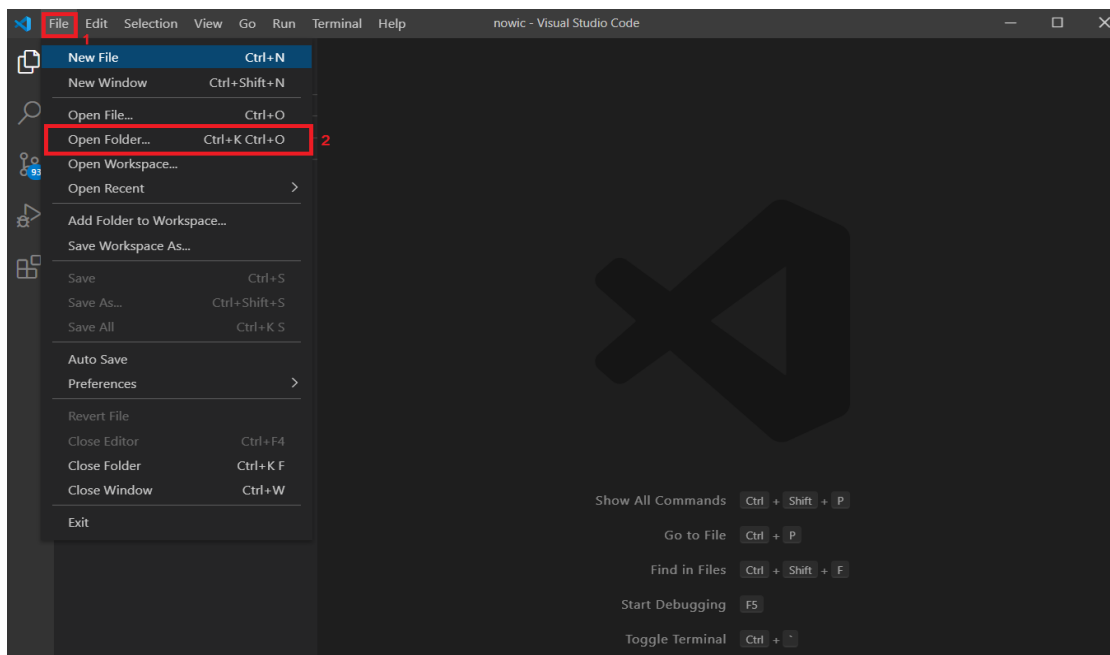
Getting Started with Visual Studio Code

VSCoDe Step4.JSON 파일 설정 및 사용 방법

1. 세 개의 VSCoDe JSON 파일 생성 및 설정

1. 세 개의 VSCode JSON 파일 생성 및 설정

- VSCode에서 C++을 사용하기 위해서는 기본적으로 3개의 JSON 파일이 필요합니다.
JSON 파일을 생성하기 전에 먼저 VSCode에 작업 폴더와 cpp 파일을 생성해야 합니다. VSCode 좌측 상단의 [파일]을 클릭한 후에 [폴더 열기]를 클릭하면 됩니다. 이때에 자신이 업데이트 받아온 nowic의 폴더 혹은 자신이 사용할 작업 폴더를 검색한 후에 선택하여 열면 됩니다.폴더 추가를 완료하였다면 앞서 사용했던 폴더 안의“hello.cpp” 파일 (테스트용)을 열어둡니다.
- JSON 파일은 .vscode라는 폴더 안에 자동 생성됩니다. 이때에 자신이 사용하는 각각의 작업 폴더 안에 .vscode 폴더와 JSON 파일이 있어야 합니다. 예를 들어 Pset01이 과제로 주어졌다면 작업 폴더를 Pset01로 선택한 후에 JSON 파일을 생성 및 설정해주어야 합니다. JSON 파일 생성 및 설정은 각 작업 폴더마다 해주어야 합니다.

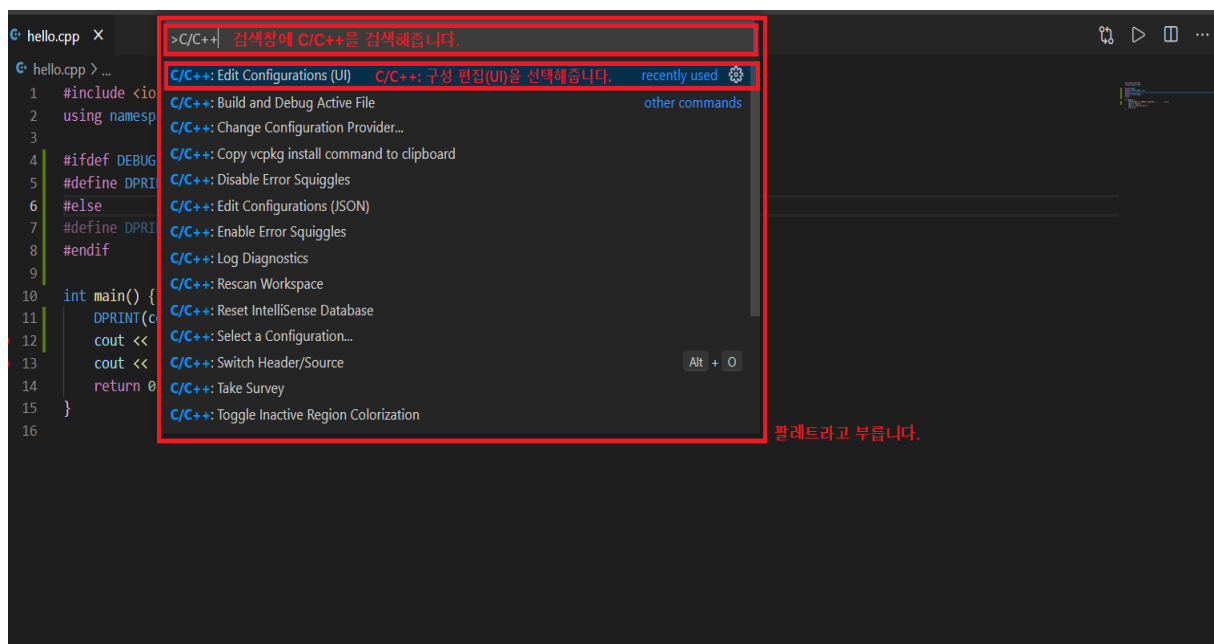


세 개의 JSON 파일 생성 및 설정 순서 중요합니다!!!

1) .vscode/c_cpp_properties.json

c_cpp_properties.json 파일은 VSCode에서 설치한 C/C++ Extension (2단계)의 설정 파일입니다. VSCode에서 어떤 Compiler를 사용할지 설정을 해줘야 합니다. 그래야만 C/C++ Extension이 tasks.json과 launch.json의 기본 템플릿을 만들어줍니다.

1. 먼저 **F1**을 눌러 VSCode의 팔레트를 열어줍니다. 팔레트의 검색 창에 C/C++을 검색하여 **C/C++: 구성 편집(UI)**을 선택하여 줍니다.



2. 설정 화면에서 다음의 설정을 변경해주어야 합니다. 변경과 동시에 자동으로 즉시 저장되며 .vscode 폴더 안에 c_cpp_properties.json 파일이 자동 생성되었음을 확인할 수 있습니다.

A. 구성 이름: Win32

B. 컴파일러 경로: C:/msys64/mingw64/bin/g++.exe

이때에 경로는 w(역슬래시)가 아닌 /(일반 슬래시)가 사용되어야 합니다. 또는, w(역슬래시)를 연속으로 두 번씩 사용해주어야 합니다 (ww).

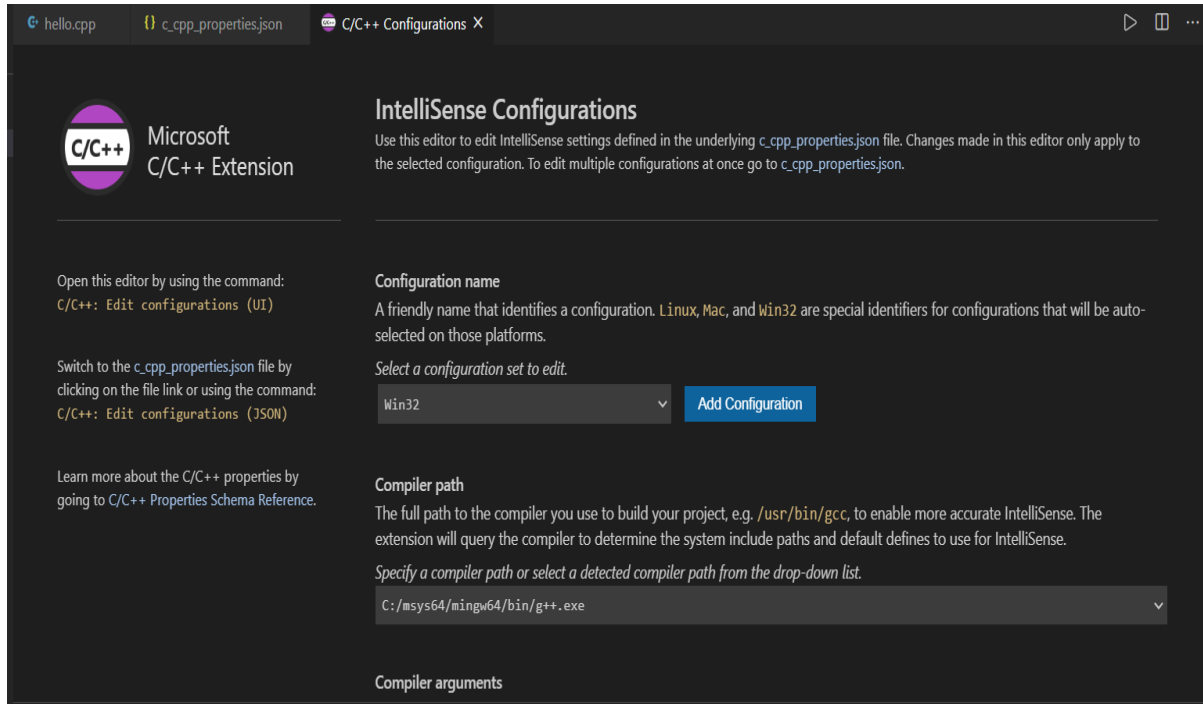
C. Compiler arguments: 빈칸으로 가만히 나둡니다.

D. IntelliSense: gcc-x64(코딩을 할 때에 큰 도움이 되는 기능)

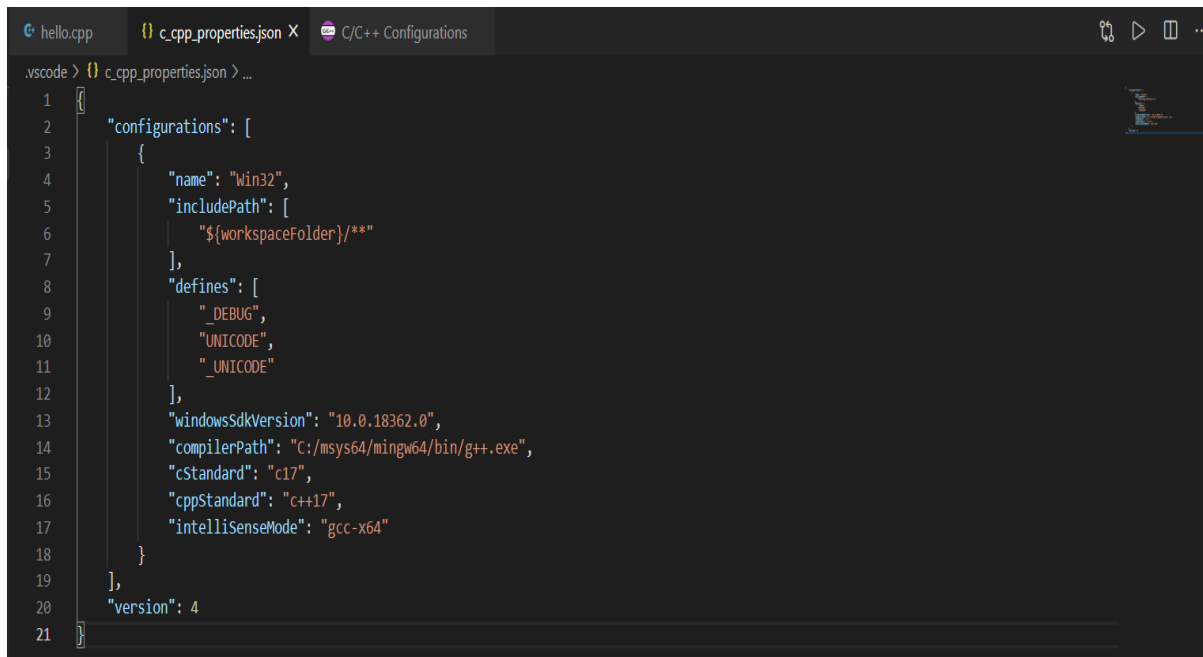
E. 경로 포함: 가만히 나둡니다.

F. 정의: 가만히 나둡니다. (IntelliSense의 추가 옵션 값)

G. C/C++ 표준: c++17



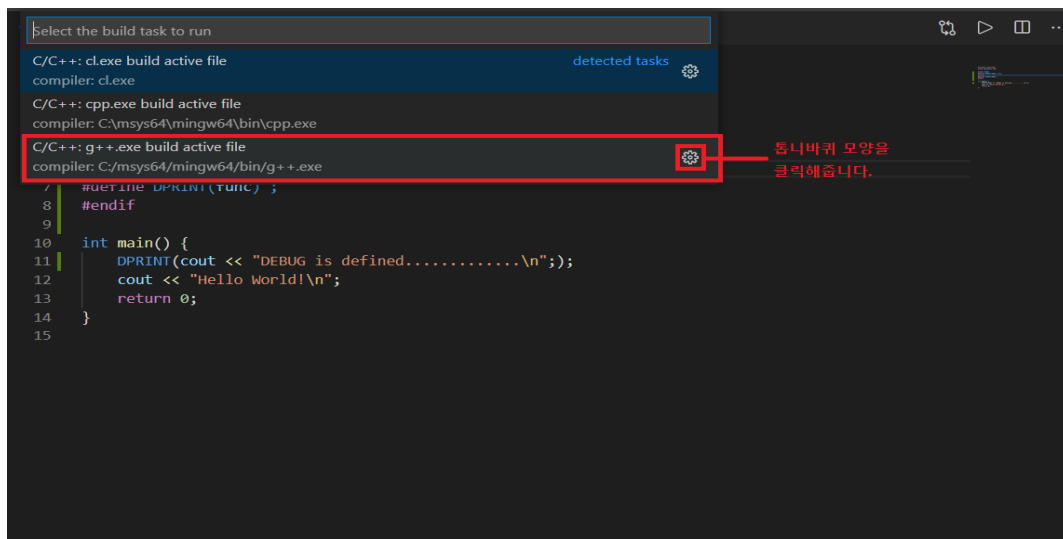
3. UI에서 설정한 값대로 `.vscode` 폴더 안에 `c_cpp_properties.json` 파일이 생성되었는지 확인해줍니다.



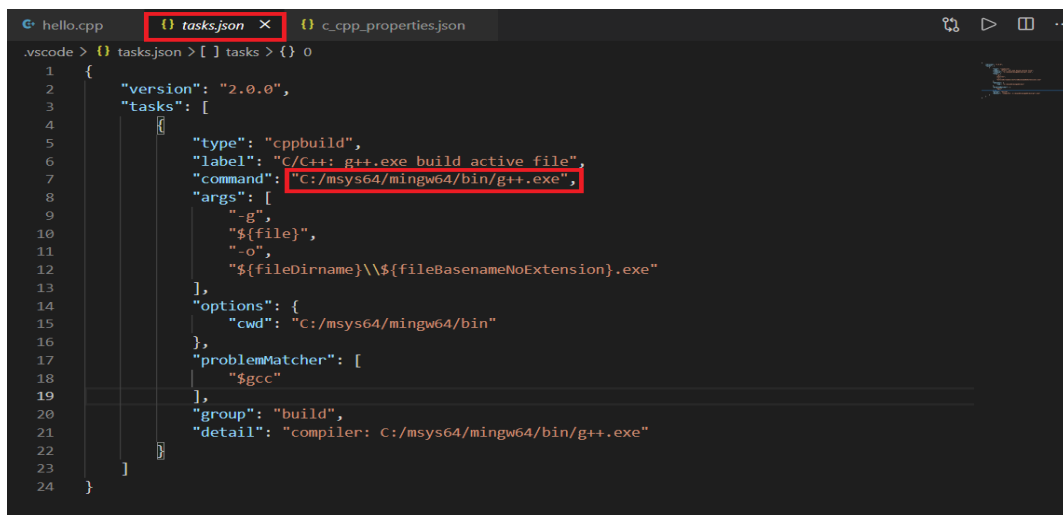
2) .vscode/tasks.json

tasks.json 파일은 배치(Batch)파일과도 같은 역할을 합니다. 이 기능을 통해 빌드, 테스트, 배포 등과 같은 명령어들을 하나의 파일로 지정해서 한번에 실행할 수 있습니다.

1. 먼저 **Ctrl + Shift + B**[Terminal -> Run Build Task...] 단축키를 눌러줍니다. 이때에 반드시 **cpp** 파일(hello.cpp)을 열어둔 상태에서 단축키를 눌러주어야 합니다.
2. 아래 그림과 같이 빌드 작업 선택 창에 나타난 리스트 중에서 **[C/C++: g++.exe build active file]**의 우측 끝에 나타나는 **톱니바퀴 모양의 아이콘(Configuration Task)**을 클릭해줍니다. .vscode 폴더 안에 tasks.json 파일이 자동 생성되었음을 확인할 수 있습니다.



3. .vscode 폴더 안에 tasks.json 파일을 확인해줍니다. 이때에 **"command"**에 지정된 값이 **c_cpp_properties.json** 파일의 **"compilerPath"**와 일치하는지 확인해주어야 합니다.



tasks.json 파일의 속성 값은 아래와 같습니다.

A. **type**: 해당 tasks의 유형을 나타냅니다.

- **"cppbuild"**: 빌드 파일

- **"process"**: 실행 파일

- **"shell"**: bash 또는 PowerShell

B. **label**: 명령 팔레트 UI에 보여지는 이름을 뜻합니다.

C. **command**: 실행할 명령어를 뜻합니다.

D. **args**: 실행할 명령어 뒤에 붙는 arguments를 뜻합니다. **띄어쓰기가 필요할 때마다 ",(쉼표)"를 붙여주어야 합니다.**

E. **options**: 여러 환경 변수 값들을 기록하는 곳입니다. "cwd"는 현재의 작업 폴더 Path를 뜻합니다.

F. **problemMatcher**: 컴파일러들마다 Warning과 Error 메시지를 출력하는 규칙이 모두다릅니다. 정규식 표현 등을 이용하여 나타나는 Warning과 Error 메시지들을 파싱하여 출력창에 출력하도록 설정합니다. 하지만, VSCode에서 다운로드를 받은 C/C++ Extension을 쓰기 때문에 자동으로 처리해 줍니다. problemMatcher는 추가설정없이 "\$gcc"만을 작성하게 되면 자동으로 Extension에 맡겨집니다.

G. **group**: 해당 task 파일이 어느 그룹에 속하는지를 나타냅니다. 그룹으로 여러 tasks를 묶어 놓을 수 있습니다.

H. **detail**: 해당 tasks에 대한 설명을 뜻합니다.

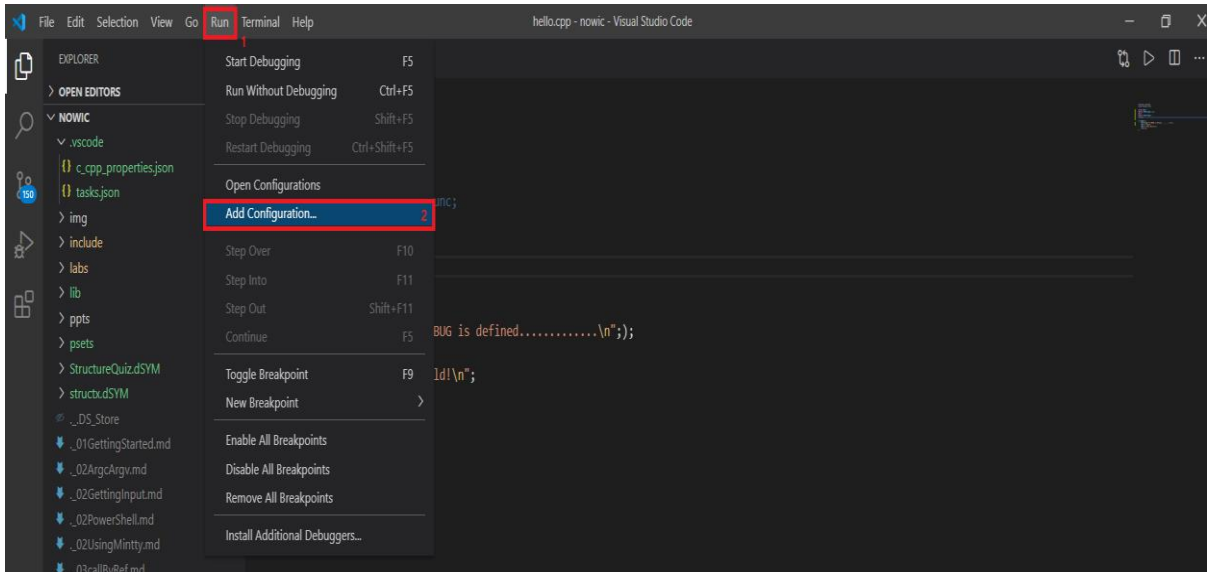
- 추가 속성에 대한 설명은 아래의 URL에서 확인할 수 있습니다.

<<https://cartiertk.tistory.com/52>>

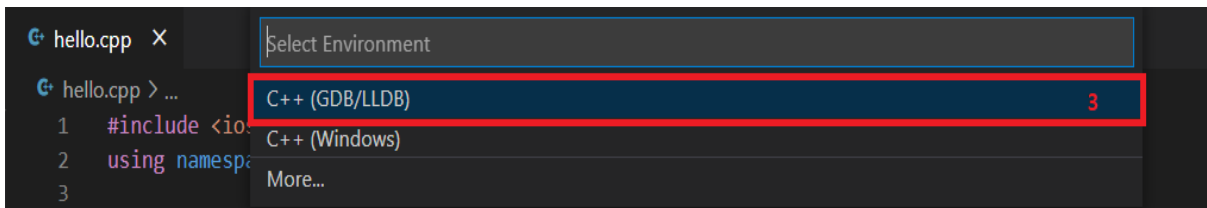
3) .vscode/launch.json

launch.json 파일은 디버깅을 설정하기 위한 파일입니다.

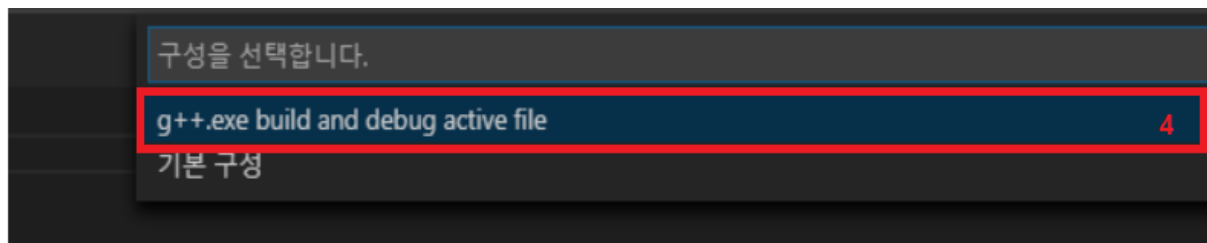
1. VSCode 상단 옵션의 [디버그] 혹은 [실행]을 클릭해줍니다.
2. [구성 추가]를 선택해줍니다.



3. 환경 선택 창에서 "C++ (GDB/LLDB)"을 선택해줍니다.



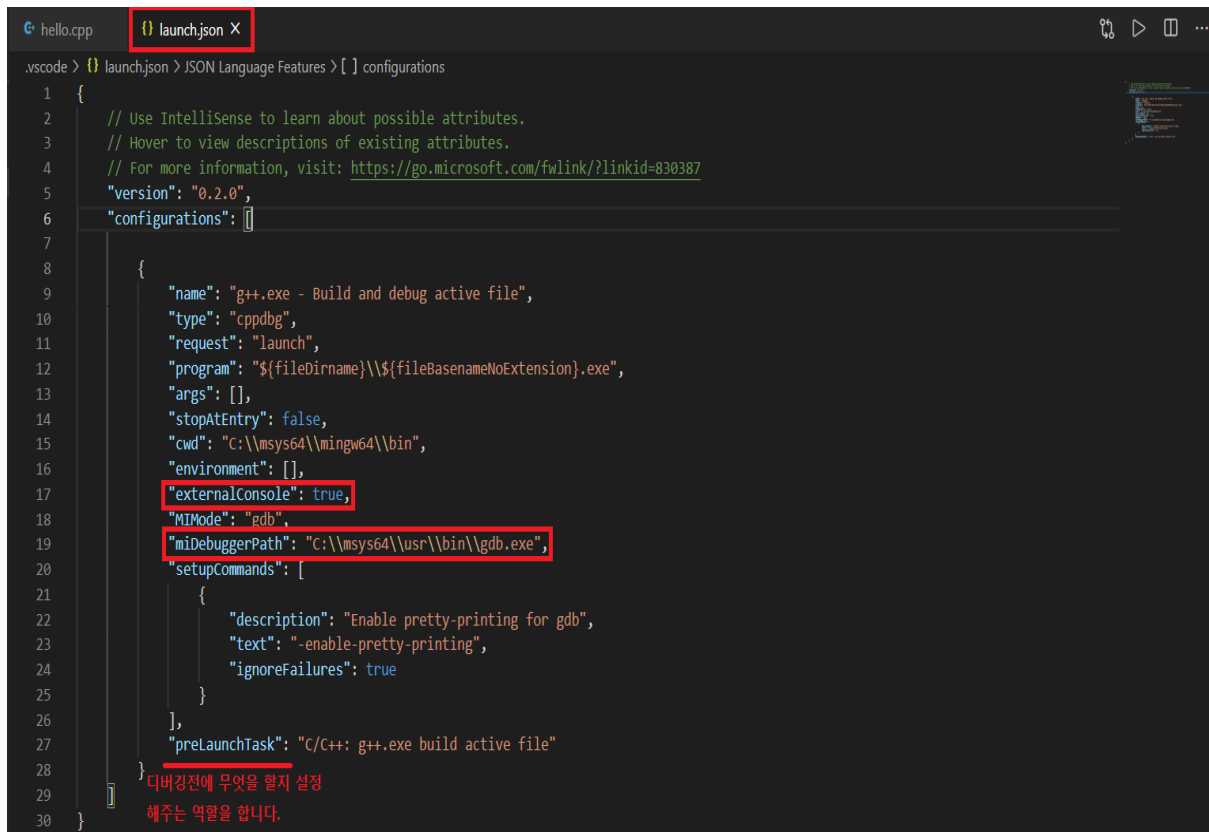
4. 구성 선택 창에서 "g++.exe build and debug active file"을 선택해줍니다. .vscode 폴더 안에 launch.json 파일이 자동 생성되었음을 확인할 수 있습니다.



5. launch.json 파일에서 "miDebuggerPath"를 가장 먼저 확인해주어야 합니다. 자신의 PC에 gdb 툴이 설치되어 있는 Path를 지정해주어야 합니다. Windows 검색 창에 "gdb.exe"을 검색한 후에 해당 Path를 확인하여 값을 지정해주는 것이 안전합니다.

추가적으로 **"externalConsole"**의 값을 **"true"**로 지정해준다면 새로운 콘솔 창이 나타나 디버깅을 눈으로 쉽게 확인할 수 있습니다. (printf, cout 명령어 등)

또한, 알아둬야 할 설정으로는 **preLaunchTask** 설정이 있습니다. 설정내용은 디버깅을 시작하기 전에 **"g++ exe build active file"** 즉, 디버깅을 실행하기 전에 빌드를 먼저 실행하겠다는 것을 의미합니다.



```
.vscode > {} launch.json X
.vscode > {} launch.json > JSON Language Features > [ ] configurations
1 {
2   // Use IntelliSense to learn about possible attributes.
3   // Hover to view descriptions of existing attributes.
4   // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
5   "version": "0.2.0",
6   "configurations": [
7
8     {
9       "name": "g++.exe - Build and debug active file",
10      "type": "cppdbg",
11      "request": "launch",
12      "program": "${fileDirname}\\${fileBasenameNoExtension}.exe",
13      "args": [],
14      "stopAtEntry": false,
15      "cwd": "C:\\msys64\\mingw64\\bin",
16      "environment": [],
17      "externalConsole": true,
18      "MIMode": "gdb",
19      "miDebuggerPath": "C:\\msys64\\usr\\bin\\gdb.exe",
20      "setupCommands": [
21        {
22          "description": "Enable pretty-printing for gdb",
23          "text": "-enable-pretty-printing",
24          "ignoreFailures": true
25        }
26      ],
27      "preLaunchTask": "C/C++: g++.exe build active file"
28    }
29  ]
30 }
```

디버깅전에 무엇을 할지 설정
해주는 역할을 합니다.

- 빌드 혹은 프로그램 실행에 문제가 있을 시에는 자동 생성 및 설정한 JSON 파일을 삭제한 후에 다시 생성 및 설정을 반복하면 됩니다.

4) Multiple Launch

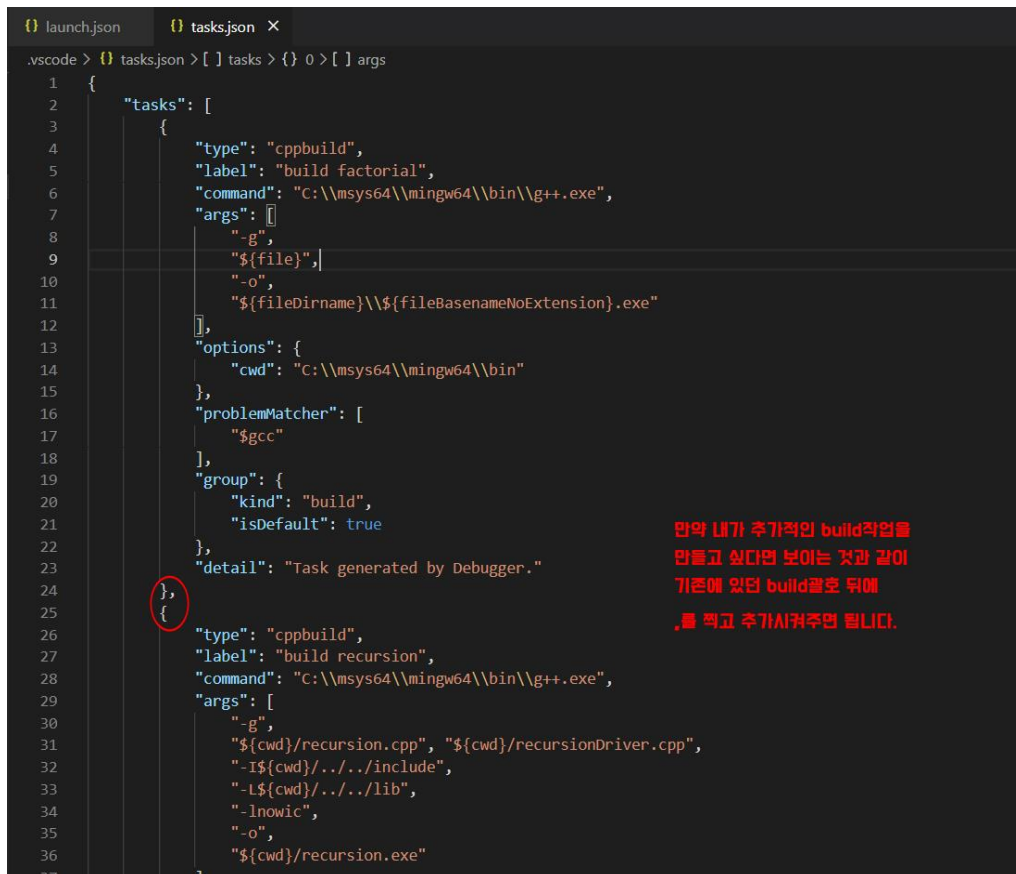
기본적으로 VScode에서 Launch(Run)를 하게 되면 preLaunchTask에서 선언된 build작업을 한 뒤 결과물인 exe파일을 실행시키게 됩니다. 여기서 만약 내가 한 폴더 내에서 여러 개의 build를 수행하고 싶다면 추가로 설정을 해야 합니다.

Ex) 예를 들어 pset02a 폴더 내에서 **factorial.cpp** 과 **recursion.cpp** 파일 2개가 존재한다고 했을 때 각각의 파일을 **factorial.exe** **recursion.exe** 파일로 build하고 따로 실행을 시키고 싶을 때 설정을 하는 것 입니다.

factorial.cpp -> **factorial.exe**

recursion.cpp **recursionDriver.cpp** -> **recursion.exe**

● tasks.json



```
.vscode > {} tasks.json > [ ] tasks > {} 0 > [ ] args
1 {
2   "tasks": [
3     {
4       "type": "cppbuild",
5       "label": "build factorial",
6       "command": "C:\\msys64\\mingw64\\bin\\g++.exe",
7       "args": [
8         "-g",
9         "${file}",
10        "-o",
11        "${fileDirname}\\${fileBasenameNoExtension}.exe"
12      ],
13      "options": {
14        "cwd": "C:\\msys64\\mingw64\\bin"
15      },
16      "problemMatcher": [
17        "$gcc"
18      ],
19      "group": {
20        "kind": "build",
21        "isDefault": true
22      },
23      "detail": "Task generated by Debugger."
24    },
25    {
26      "type": "cppbuild",
27      "label": "build recursion",
28      "command": "C:\\msys64\\mingw64\\bin\\g++.exe",
29      "args": [
30        "-g",
31        "${cwd}/recursion.cpp", "${cwd}/recursionDriver.cpp",
32        "-I${cwd}/../include",
33        "-L${cwd}/../lib",
34        "-lnowic",
35        "-o",
36        "${cwd}/recursion.exe"
37      ],
38    }
39  ]
40 }
```

만약 내가 추가적인 build작업을 만들고 싶다면 보이는 것과 같이 기존에 있던 build괄호 뒤에 ,를 찍고 추가시켜주면 됩니다.

※ 여기서 주의 해야 할 점은 label의 이름을 설정하면서 반드시 cpp파일의 이름과 같게 해줘야 합니다. 예를 들어 recursion.cpp의 파일을 build하는 작업을 추가시키는 것이면 "label" : "build recursion"으로 작성해야 합니다. 여러 개의 cpp파일로 build 할 경우 main{}이 존재하는 파일의 이름으로 작성합니다.

```

24 },
25 {
26   "type": "cppbuild",
27   "label": "build recursion",
28   "command": "C:\\msys64\\mingw64\\bin\\g++.exe",
29   "args": [
30     "-g",
31     "${cwd}/recursion.cpp", "${cwd}/recursionDriver.cpp",
32     "-I${cwd}/../..../include",
33     "-L${cwd}/../..../lib",
34     "-lnowic",
35     "-o",
36     "${cwd}/recursion.exe"
37   ],
38   "options": {
39     "cwd": "C:\\msys64\\mingw64\\bin"
40   },
41   "problemMatcher": [
42     "$gcc"
43   ],
44   "group": {
45     "kind": "build",
46     "isDefault": true
47   },
48   "detail": "Task generated by Debugger."
49 }

```

내가 만드는 파일의 이름을 적어줍니다. (반드시 파일의 이름과 같아야 합니다)

빌드를 하며 필요한 파일들을 작성해줍니다.

exe파일 또한 cpp파일의 이름과 같아야 합니다.

● launch.json

```

1 // Use IntelliSense to learn about possible attributes.
2 // Hover to view descriptions of existing attributes.
3 // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
4 "version": "0.2.0",
5 "configurations": [
6   {
7     "name": "g++.exe - 활성 파일 빌드 및 디버그",
8     "type": "cppdbg",
9     "request": "launch",
10    "program": "${fileDirname}\\${fileBasenameNoExtension}.exe",
11    "args": [],
12    "stopAtEntry": false,
13    "cwd": "C:\\msys64\\mingw64\\bin",
14    "environment": [],
15    "externalConsole": false,
16    "MIMode": "gdb",
17    "miDebuggerPath": "C:\\msys64\\mingw64\\bin\\gdb.exe",
18    "setupCommands": [
19      {
20        "description": "gdb에 자동 서식 지정 사용",
21        "text": "-enable-pretty-printing",
22        "ignoreFailures": true
23      }
24    ],
25    "preLaunchTask": "build ${fileBasenameNoExtension}"
26  }
27 ]

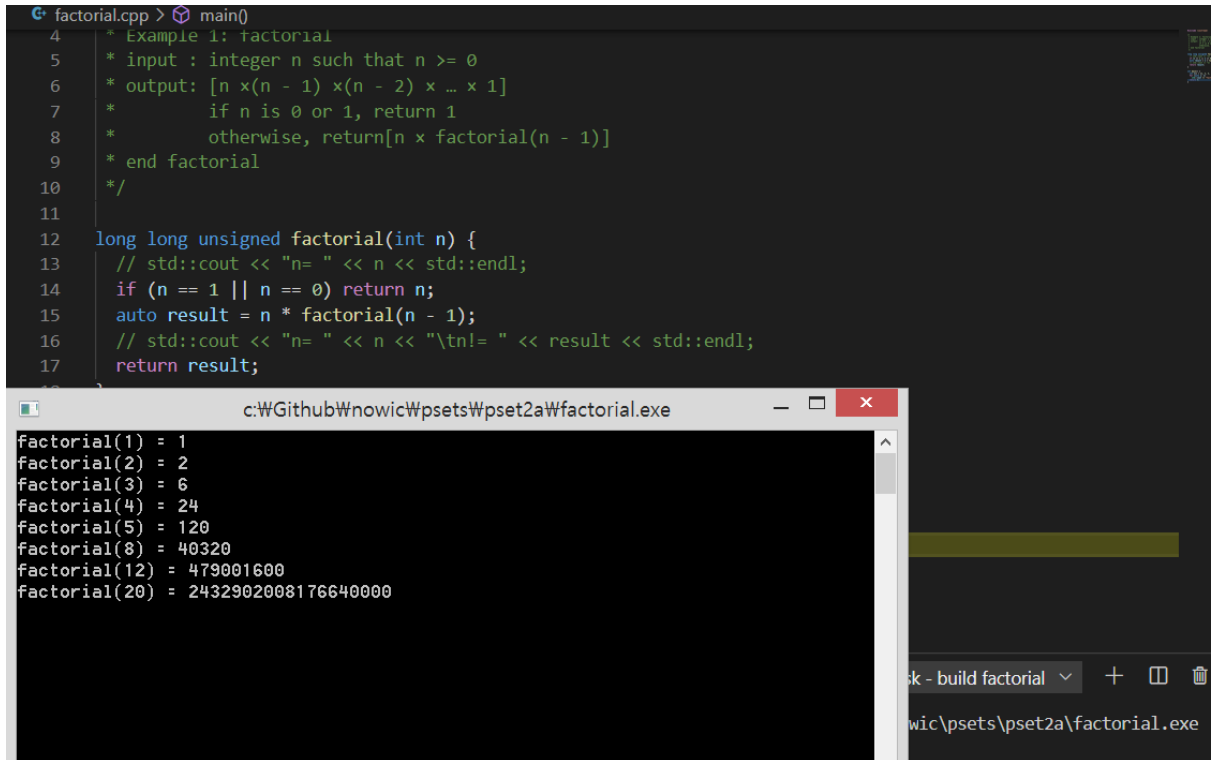
```

preLaunchTask부분을 바꿔줍니다.

\${fileBasenameNoExtension}은 우리가 Launch를 실행시킨 파일의 이름을 지정합니다.

- Test Run

내가 launch시킬 cpp파일에서 **Ctrl + F5 [Terminal ->Run Without Debugging]** 를 눌러 실행시켰을 때 내가 실행시킨 파일과 같은 이름의 exe파일이 실행이 되면 성공입니다.



The image shows a Visual Studio Code editor with a C++ file named `factorial.cpp`. The code defines a recursive factorial function and a `main` function that tests it for values 1, 2, 3, 4, 5, 8, 12, and 20. Below the editor, a terminal window titled `c:\WGithub\Wnowic\Wpsets\Wpset2a\factorial.exe` displays the output of the program, showing the factorial values for each input.

```
factorial.cpp > main()
4  * Example 1: factorial
5  * input : integer n such that n >= 0
6  * output: [n x(n - 1) x(n - 2) x ... x 1]
7  *       if n is 0 or 1, return 1
8  *       otherwise, return[n x factorial(n - 1)]
9  * end factorial
10 */
11
12 long long unsigned factorial(int n) {
13     // std::cout << "n= " << n << std::endl;
14     if (n == 1 || n == 0) return n;
15     auto result = n * factorial(n - 1);
16     // std::cout << "n= " << n << "\tn!= " << result << std::endl;
17     return result;
18 }
19
20 int main() {
21     std::cout << "factorial(1) = 1\n";
22     std::cout << "factorial(2) = 2\n";
23     std::cout << "factorial(3) = 6\n";
24     std::cout << "factorial(4) = 24\n";
25     std::cout << "factorial(5) = 120\n";
26     std::cout << "factorial(8) = 40320\n";
27     std::cout << "factorial(12) = 479001600\n";
28     std::cout << "factorial(20) = 2432902008176640000\n";
29     return 0;
30 }
```

```
Factorial(1) = 1
Factorial(2) = 2
Factorial(3) = 6
Factorial(4) = 24
Factorial(5) = 120
Factorial(8) = 40320
Factorial(12) = 479001600
Factorial(20) = 2432902008176640000
```

※ 만약 아래와 같은 Error가 발생한다면 `tasks.json`파일에서 `label`의 이름이 `cpp`파일의 이름과 다르거나 `launch.json`파일의 `preLaunchTask`의 설정이 잘못된 것입니다.

