ChatGPT

Setting Up MSYS2 with GCC 15 and Clang/LLVM 21 on Windows (x64) and VS Code

1. Downloading and Installing MSYS2

Download MSYS2: Go to the official MSYS2 website and download the latest installer for 64-bit Windows (e.g. `msys2-x86_64-20250830.exe`) [1]. Run the installer on your Windows 10/11 system (64-bit is required) [2]. When prompted, choose an installation folder **without spaces or special characters** (for example, `C:\msys64`) [3]. This ensures a smooth setup (long or space-containing paths can cause tool issues).

MSYS2 installer – choose a short ASCII-only path like `C:\msys64` *for the installation directory* [3].

Finish installation: Complete the installer. At the end, check the option to launch MSYS2. This will open an MSYS2 terminal window (by default, it opens the *UCRT64* environment, which is recommended for most users) [4]. You should see a terminal prompt like `MSYS UCRT64 ~` indicating the active environment.

MSYS2 UCRT64 terminal launched after installation (ready for package updates and installs).

Update MSYS2 packages: Before installing any toolchain, update MSYS2's package database and core packages. In the MSYS2 terminal, run:

```
pacman -Syu
```

If prompted to close/reopen the terminal during core updates, follow those instructions (run the update again after reopening until no more updates) [5]. This ensures you have the latest package manager and system libraries. *For example:* after initial install, **update all packages** using `pacman -Syu` [6].

2. Installing GCC 15, Clang/LLVM 21, and Clangd in MSYS2

All installations are done via MSYS2's package manager `pacman`. Make sure you are in the **MinGW UCRT64 environment** (the default MSYS2 MinGW 64-bit environment using the Universal C Runtime). You can launch it from the Start Menu (look for "MSYS2 MinGW UCRT64"). The UCRT64 environment is recommended for modern C/C++ development on Windows [4], but if you prefer the older MSVCRT-based environment, substitute `ucrt64` with `mingw64` in the package names below.

In the MSYS2 UCRT64 terminal, run the following commands to install the required toolchains and tools:

```
# Install GCC (GNU Compiler Collection) and related tools:
pacman -S --needed mingw-w64-ucrt-x86_64-toolchain

# Install Clang (LLVM) compiler:
pacman -S --needed mingw-w64-ucrt-x86_64-clang

# Install Clang extra tools (includes **clangd** language server, clang-format,
etc.):
pacman -S --needed mingw-w64-ucrt-x86_64-clang-tools-extra

# (Optional) Install LLVM core tools (lld linker, llvm-ar, etc):
pacman -S --needed mingw-w64-ucrt-x86_64-llvm

# Install CMake build system:
pacman -S --needed mingw-w64-ucrt-x86_64-cmake

# Install Ninja build tool (optional but recommended for CMake):
pacman -S --needed mingw-w64-x86_64-ninja
```

- The first command installs the `mingw-w64-ucrt-x86_64-toolchain` package group, which includes **GCC 15** (g++, gcc), the GNU binutils, make, GDB, and other essentials [7] [8]. Accept the default selection (press Enter) when prompted to install all group members. After this, you'll have GCC 15.2.0 (or latest 15.x) and G++ for C/C++ [9] [10].
- The second command installs **Clang/LLVM 21** (the C/C++ frontend `clang` and `clang++` compiler) [11] [12]. In the UCRT64 environment, this Clang will target the MinGW-w64 environment (producing native Windows executables using MinGW's runtime and libstdc++) [13].
- The third command installs **Clang Tools Extra 21**, which includes tools built on Clang's API [14] – notably `clangd` (the C/C++ language server), as well as `clang-format`, `clang-tidy`, etc. (The package is `clang-tools-extra` in MSYS2 [15]).
- The fourth (optional) installs the rest of **LLVM 21** tools (such as the LLD linker, `llvm-ar`, `opt`, etc.). This is useful if you plan to use LLVM's linker (LLD) or other LLVM utilities, but not strictly required for basic C/C++ development.
- The fifth installs **CMake** (if you don't already have it). This allows the VS Code CMake Tools extension to invoke CMake from MSYS2. (Alternatively, you could use an official CMake installation, but using MSYS2's CMake is convenient [16].)
- The sixth installs **Ninja**, a fast build system. Ninja is optional but **highly recommended**. The VS Code CMake Tools extension will automatically prefer Ninja if it is available [17], which simplifies configuration (no need to configure `make`). By installing Ninja, you can avoid dealing with Makefile compatibility issues – CMake will generate Ninja build files and build with Ninja, which works out-of-the-box on Windows [18].

Tip: The above `pacman -S` commands can be combined in one line if desired. For example, MSYS2's docs suggest installing GCC in UCRT64 with: `pacman -S mingw-w64-ucrt-x86_64-gcc` [19]. Using the `toolchain` group (as we did) grabs GCC and related tools in one go. You can also add `--needed` to skip re-installing packages you already have.

After installation, it's good to verify that the compilers are installed correctly:

```
# Check GCC version
gcc --version
g++ --version

# Check Clang/LLVM version
clang --version
clang++ --version

# Check Clangd (language server) version
clangd --version
```

Each of those should report the expected versions (GCC 15.x, Clang/LLVM 21.x, etc.). For example, `g++ --version` might show *gcc (Rev2, Built by MSYS2) 15.2.0* and `clang++ --version` should show *clang version 21.1.1* targeting *x86_64-w64-windows-gnu* (the MinGW target).

3. Configuring Compilers in MSYS2 (Switching Between GCC and Clang)

In the MSYS2 environment, the installed compilers are ready to use. By default, the GNU tools (gcc, g++, make) and LLVM tools (clang, clang++, lld, etc.) are located in the MSYS2 directories (under your installation, e.g. `C:\msys64\ucrt64\bin`). When you open an **MSYS2 UCRT64 shell**, the environment's `PATH` is set so that these tools are on the path [20] [21]. This means in the MSYS2 terminal you can immediately use commands like `gcc`, `g++`, `clang`, `clang++`, etc.

Using GCC vs Clang: To compile with a specific compiler, invoke it by name. For example:

- Use **GCC**: `g++ -o myprog.exe myprog.cpp` (or use `gcc` for C code).
- Use **Clang**: `clang++ -o myprog.exe myprog.cpp` (or `clang` for C code).

Both compilers will link against the same MinGW-w64 runtime and libraries by default in this environment. There's no global "switch" needed – you simply call the one you want to use. They can coexist and you can choose per project or per build command which to use.

Setting defaults with environment variables: Some build systems respect the `CC` and `CXX` environment variables for the C and C++ compiler. If you want a particular project to use Clang by default, you can export these in the MSYS2 shell:

```
export CC=clang
export CXX=clang++
```

This would cause tools like Make or CMake (if not told otherwise) to pick Clang. Conversely, setting `CC=gcc` / `CXX=g++` forces use of GCC. You can add these to your `~/.bashrc` in MSYS2 if you frequently

switch and want an easy toggle, but usually it's fine to specify the compiler in your build tool's configuration instead.

Add compilers to Windows PATH (for VS Code integration): If you plan to use the compilers *outside* of the MSYS2 terminal (for example, directly from VS Code or Windows Command Prompt), you should add the MSYS2 *MinGW* binaries directory to your Windows PATH. This lets VS Code and other processes find `gcc.exe`, `clang.exe`, etc.

For the UCRT64 environment, add the path:

```
C:\msys64\ucrt64\bin
```

to your system's PATH. (If you used the older mingw64 environment, it would be `C:\msys64\mingw64\bin`.) To edit PATH, open the Windows *Environment Variables* settings, and append the above to the User or System PATH. **Do not** add the MSYS2 `usr\bin` to PATH for general use – that contains Unix-like tools which can conflict with Windows commands (e.g., a `find.exe` that isn't the Windows find). Only add the specific `...\ucrt64\bin` where the compiler toolchains reside.

After updating PATH, you can open a **new Command Prompt** (or PowerShell) and test that the tools are accessible. For example, in a regular Windows cmd, run:

```
g++ --version && g++ -dumpmachine
clang++ --version
make --version
cmake --version
```

You should see outputs indicating the versions (and targeting triplet for g++). The `make --version` should report a MinGW make (if you installed the toolchain group, MSYS2 provides a `mingw32-make.exe` which we can use). The `cmake --version` should show the CMake installed in MSYS2. These checks confirm that your PATH is set correctly [22] [23]. If any of these aren't found, double-check that the correct directory is in PATH and restart your terminal.

Switching compilers in VS Code or scripts: When using VS Code's CMake Tools (next section), you will **select a kit** to choose Clang vs GCC. If you are using other build systems or compiling manually in the VS Code terminal, just call the compiler you want. You can also adjust `PATH` order to prefer one (not usually needed). There's no need to *uninstall* or disable one compiler to use the other – they can be used interchangeably.

Advanced note: In MSYS2's UCRT64 environment, Clang is using GCC's libstdc++ by default [13]. This is normally fine. If you ever want Clang to use LLVM's libc++, you would use the *CLANG64* environment and packages, but that's beyond our scope. The default setup we did (Clang in UCRT64) yields a GNU-compatible Clang toolchain, which should work for most cases.

4. Integrating MSYS2 Toolchains with Visual Studio Code

Assuming you have Visual Studio Code installed, you'll now integrate the compilers and tools with it for a smooth development experience.

Install VS Code extensions:

- **C/C++ Extension Pack (Microsoft)** – this bundle includes the **C/C++** extension (for IntelliSense, code navigation, debugging support) and the **CMake Tools** extension (for CMake integration), among others. In VS Code, go to the Extensions view and search for "C/C++ Extension Pack", then install it [24]. (This pack will also include extensions like CTest and CMake, which complement the setup.)
- **clangd extension (optional)** – If you prefer to use Clangd as the language server for C/C++ instead of Microsoft's IntelliSense, install the **clangd** extension as well. This will use the `clangd` we installed to provide code completion and analysis. After installing, the extension should auto-detect `clangd.exe` (ensure `C:\msys64\ucrt64\bin` is in PATH so VS Code can find it). You may then disable IntelliSense in the MS C/C++ extension to avoid conflicts (set `"C_Cpp.intelliSenseEngine": "Disabled"` in settings, and enable the clangd extension).

Configure VS Code's Integrated Terminal (optional): You can add a profile for MSYS2 so that VS Code's built-in terminal can launch an MSYS2 shell. This is useful for running commands or debugging in the proper environment. For UCRT64, add the following to your VS Code settings (press **Ctrl+Shift+P** → "Preferences: Open Settings (JSON)"):

```
"terminal.integrated.profiles.windows": {
    "MSYS2 UCRT64": {
        "path": "C:\\Windows\\System32\\cmd.exe",
        "args": [
            "/C", "C:\\msys64\\msys2_shell.cmd -defterm -here -no-start -ucrt64"
        ]
    }
}
```

This creates a new terminal profile named "MSYS2 UCRT64". Now in VS Code, you can use **Terminal > New Terminal**, and select the **MSYS2 UCRT64** profile to get a shell with the MSYS2 environment loaded [25]. This isn't strictly necessary for CMake integration, but it's handy for running your executables or using Unix tools in VS Code.

Ensure VS Code knows your compilers and tools: VS Code (specifically the CMake Tools extension) will need to find `gcc`, `g++`, `clang`, etc., to configure CMake projects. If you added the MSYS2 `bin` path to the Windows PATH as described earlier, VS Code launched normally will inherit that and find the compilers. The CMake Tools extension will automatically **scan for compilers** on first use or when you run "Scan for Kits". It looks in directories on PATH for common compiler names (like `gcc.exe`, `clang.exe`, etc.) [26].

If for some reason you did not want to modify the global PATH, an alternative is to configure the environment within VS Code. You can set the `cmake.environment` setting to include the MSYS2 paths,

and/or set the `cmake.cmakePath` to the MSYS2 CMake. For example, in your VS Code settings (user or workspace), you could add:

```
"cmake.cmakePath": "C:/msys64/ucrt64/bin/cmake.exe",
"cmake.environment": {
    "PATH": "C:\\msys64\\ucrt64\\bin;C:\\msys64\\usr\\bin;${env:PATH}"
}
```

This explicitly tells CMake Tools to use MSYS2's CMake and sets up PATH for configure tasks [27]. (Including `ucrt64\bin` ensures compilers are found; adding `usr\bin` allows MSYS2's build tools like `make` or other utilities to be found if needed.) However, if you already added to global PATH, this may not be necessary.

Troubleshooting PATH issues: Ensure no conflicting tools shadow your compilers. For instance, **Git for Windows** ships a `gcc.exe` in its user-friendly MinGW, which could confuse VS Code. If you encounter unexpected compiler picks, check your PATH order. It may help to remove or reorder entries so that `C:\msys64\ucrt64\bin` comes before any other MinGW distributions [4].

Debugging setup: The Microsoft C/C++ extension allows debugging using GDB. We installed `mingw-w64-ucrt-x86_64-gdb` as part of the toolchain group, so **GNU GDB** is available. The CMake Tools extension can generate a debug configuration for you: after a successful build, click the **Debug** icon in the status bar (or use **CMake: Debug** from the command palette) and it will create a launch configuration that uses GDB (cpptools). You might need to approve the prompt to configure debugging. Ensure that `"miDebuggerPath"` in the launch config points to the MSYS2 gdb (e.g., `C:\\msys64\\ucrt64\\bin\\gdb.exe`). If using clang and you prefer LLDB, you could install `mingw-w64-ucrt-x86_64-lldb` and adjust your configuration, but using GDB is simpler in this MinGW context.

5. Using CMake Tools in VS Code with GCC and Clang/LLVM

With everything installed, you can now use the **CMake Tools** extension to configure, build, and run C++ projects in VS Code using either GCC or Clang.

Selecting a Compiler (Kit): Open your CMake project folder in VS Code (a folder with a `CMakeLists.txt`). The CMake Tools extension will activate. Look at the VS Code status bar – you should see a **[No Kit Selected]** or a kit name there. Click that label or run **"CMake: Select a Kit"** from the Command Palette. VS Code will present a list of detected kits (compilers). If you followed the setup above, you should see entries like:

- *GCC 15.2.0 x86_64-w64-mingw32 (C:\msys64\ucrt64\bin\g++.exe)*
- *Clang 21.1.1 x86_64-w64-windows-gnu (C:\msys64\ucrt64\bin\clang++.exe)*

(The exact wording may vary, but it will include the version and path). Choose the one you want as your active compiler. For example, select the GCC kit to use GNU g++ or select the Clang kit to use LLVM/Clang. You can always switch kits later to change the compiler – CMake Tools will reconfigure the project with the new compiler [28] [29].

If you do not see the expected options, you may need to click **Scan for Kits** in that selection menu. This will rescan PATH for compilers [26]. (Make sure VS Code was restarted after adding PATH, if you did that.) In rare cases, you might want to define a custom kit. You can do so by opening the **kits file**: press **Ctrl+Shift+P** → "CMake: Edit User-Local CMake Kits". This opens a JSON where you can add entries manually. For example, to add Clang manually:

```
{
    "name": "Clang 21.1.1 MSYS2 UCRT64",
    "compilers": {
        "C": "C:/msys64/ucrt64/bin/clang.exe",
        "CXX": "C:/msys64/ucrt64/bin/clang++.exe"
    }
}
```

Save the file and then select this kit. (Generally, auto-detection should have found it if PATH is set. Manual kits are for advanced scenarios or custom toolchain files.)

Configuring CMake Project: Once a kit (compiler) is selected, the extension may automatically run the CMake **configure** step. If not, trigger it via **"CMake: Configure"**. During configuration, CMake finds your compilers and sets up build files.

- **Generator choice:** By default, on Windows, CMake might choose the *Ninja* generator if Ninja is available [17]. Since we installed Ninja, the extension will prefer it, which is ideal. You'll see messages like `"Configuring using Ninja generator"`. If Ninja wasn't installed, CMake might default to "MSYS Makefiles" or "MinGW Makefiles". In such a case, ensure that a suitable `make` is available. We installed `mingw32-make` as part of the toolchain, but CMake might not find it automatically unless it's named `make`. One solution (already done in our setup) was installing Ninja to avoid this. Another solution is to create a symlink or copy of `mingw32-make.exe` named `make.exe` in the `ucrt64\bin` directory, so that CMake's "MSYS Makefiles" generator finds it [30] [31]. Our earlier installation of the toolchain and base-devel likely put a `make` in the MSYS2 path, but to keep things simple: **use Ninja**. It "just works" without additional config [18]. (If you do use Makefiles and encounter *CMAKE_MAKE_PROGRAM not set* errors, either switch to Ninja or set the path to `make.exe` in your kit or `settings.json` as we showed above.)

- **CMake configure done:** After a successful configure, you should see a message like "**CMake project configured**" in VS Code's status bar. A build directory (e.g., `build/`) will be created with generated build files (Ninja files or Makefiles, etc.), and a CMake cache.

Building the project: Click the **Build** button on the status bar (or run **"CMake: Build"** via Command Palette). This will invoke the build system to compile your project. If using Ninja, it will compile with multiple cores efficiently. If using Makefiles, it will call our MSYS2 make. You'll see compiler output in the **CMake/Build** output channel or terminal. The extension will indicate when the build is done. Any compilation errors will be shown in the Problems pane, and you can click them to jump to code.

Running and Debugging:

- For a simple run of the program, you can use the **"CMake: Run Without Debugging"** command or the play button (▷) in the status bar. This will execute the built target (usually your executable) in the integrated terminal. Ensure you've selected the correct target (the target drop-down in the status bar, usually it auto-selects the only executable or you can choose from multiple if present).

- For debugging, click the **Debug** button (the bug icon) in the status bar or use **"CMake: Debug"**. The first time, CMake Tools might ask to configure debug settings; allow it to create a launch configuration. This typically creates a `launch.json` entry using the **C++ (GDB)** debugger, pointing to your program and using GDB as the debugger. You may need to adjust the `miDebuggerPath` to your MSYS2 GDB if it isn't already correct. Once configured, VS Code will launch the debugger: it opens a new terminal (likely the MSYS2 one if we set it, or a regular one) and runs your program under GDB, allowing breakpoints, stepping, etc., via the UI.

Using GCC vs Clang in CMake Tools: The kit mechanism makes it easy to switch. If you want to try the other compiler, repeat the **Select Kit** step and pick the other one. CMake Tools will re-run CMake configure with the new compiler. You might use separate build folders per kit (CMake Tools handles this automatically by appending the kit name to the build directory). For example, one kit's build folder might be `build/GCC-15-Debug` and another `build/Clang-21-Debug`. This keeps the results isolated. After switching kits, build and run again to ensure everything works with the alternate compiler.

CMake presets or toolchain files (if needed): The above method uses VS Code kits, which is straightforward. In more complex scenarios (cross-compiling, custom flags, etc.), you can use CMake Toolchain files or CMakePresets. For example, you might have a `gcc-toolchain.cmake` or a `clang-toolchain.cmake` file that sets up specific flags. You can tell CMake Tools to use it by either selecting a kit that has `"toolchainFile": "path\to\file.cmake"` in its definition, or by specifying in `settings.json` like `"cmake.configureArgs": ["-DCMAKE_TOOLCHAIN_FILE=path/to/toolchain.cmake"]`. However, for a native build like this, a toolchain file isn't required – the kit and environment suffice. Kits internally use a toolchain file on your behalf if you set one [32] [33].

Building and Running C++ Projects: Summary – Once configured, your typical workflow in VS Code will be:

1. **Open project** (with CMakeLists.txt) in VS Code.
2. **Select a kit** (choose GCC or Clang as needed) [34].
3. **Configure** (CMake configure runs, generating build files) – happens automatically on kit selection or when CMakeLists.txt changes.
4. **Build** (compile the code) – via the Build button or command.
5. **Run or Debug** the resulting program – via the Run/Debug button or VS Code's Run view. For debugging, ensure breakpoints are set and the correct debugger is configured (GDB, etc.).

During this process, you might encounter integration issues; here are a few tips to debug them:

- **"No compiler found" or kit empty:** Ensure the MSYS2 `bin` path is in PATH (restart VS Code after adding). Use **Scan for Kits** to refresh [26]. If needed, add a kit manually as described.
- **CMake error "CMAKE_MAKE_PROGRAM not set" or "No suitable generator found":** This means CMake couldn't find a build tool. Easiest fix: install Ninja and retry (as noted, Ninja will be auto-used)

[18]. Or ensure `make` is available: for MSYS Makefiles generator, MSYS2's `make.exe` (from base-devel) should be in `C:\msys64\usr\bin`. For MinGW Makefiles, ensure `mingw32-make.exe` is on PATH (the toolchain provides it). You can also explicitly tell CMake Tools which make to use by setting `CMAKE_MAKE_PROGRAM` in `cmake.configureSettings` [35], but this is rarely needed if the above is done.

- **Build errors about missing libraries or includes:** If CMake can't find a package you installed in MSYS2, you might need to set `CMAKE_PREFIX_PATH` to `C:\msys64\ucrt64` (so CMake can find include/libs under that). The UCRT64 environment has its libraries in that prefix. The stack overflow answer recommended this for cases like finding OpenBLAS, etc. [36]. You can set this in `cmake.configureSettings` or as an environment variable.
- **IntelliSense not showing includes correctly:** Make sure the configuration provider for the C/C++ extension is set to CMake Tools (should be automatic when using CMake Tools). If using clangd, ensure a `compile_commands.json` is being generated (CMake Tools turns it on by default with `CMAKE_EXPORT_COMPILE_COMMANDS=ON`). The clangd extension will use that for autocomplete.
- **Runtime issues (program runs in VS Code and immediately exits or can't find DLLs):** Remember that binaries built with MSYS2 MinGW are native Windows programs. They may depend on some MinGW-w64 runtime DLLs (like `libstdc++-6.dll`, etc., located in MSYS2's bin). Since we added MSYS2 bin to PATH, these DLLs are available, so it should be fine. If you deploy the exe outside, you'd need to copy the needed DLLs or statically link. In VS Code, running in the integrated terminal (especially if it's an MSYS2 terminal) will ensure the PATH is set so DLLs are found.
- **Using the MSYS2 environment in tasks:** If you create custom build tasks in VS Code (outside CMake Tools), you can specify the shell to use. For example,

  ```
  "options": { "shell": { "executable": "C:\\msys64\\usr\\bin\\bash.exe" } }
  ```

  to run a task in a bash shell. But for most cases, relying on CMake Tools manages the environment as we set.

Finally, you have a complete C++ development environment: MSYS2 provides up-to-date GCC 15 and Clang/LLVM 21 compilers [9] [11], and VS Code with CMake Tools allows you to easily switch between them, build, and debug your projects. Enjoy coding, and happy debugging!

Sources:

- MSYS2 Installation and Setup – MSYS2 Official Guide [3] [6]
- MSYS2 Package Management – Installing GCC, Clang/LLVM, and Tools [19] [15]
- MSYS2/VSCode Integration Tips – Stack Overflow and MSYS2 Docs [18] [25]
- VS Code CMake Tools Documentation – Kits, Generators, and Usage [26] [34]

---

1 2 3 19 MSYS2
https://www.msys2.org/

4 18 27 35 36 How to configure VS Code's CMake Tools Extension for GCC and MSYS Makefiles on Windows? - Stack Overflow
https://stackoverflow.com/questions/71000530/how-to-configure-vs-codes-cmake-tools-extension-for-gcc-and-msys-makefiles-on-w

5 Updating MSYS2 - MSYS2
https://www.msys2.org/docs/updating/

6 Install gcc compiler on Windows with MSYS2 for C/C++ | DevDungeon
https://www.devdungeon.com/content/install-gcc-compiler-windows-msys2-cc

7 8 Group: mingw-w64-x86_64-toolchain - MSYS2 Packages
https://packages.msys2.org/groups/mingw-w64-x86_64-toolchain

9 10 Package: mingw-w64-x86_64-gcc - MSYS2 Packages
https://packages.msys2.org/packages/mingw-w64-x86_64-gcc

11 12 Package: mingw-w64-x86_64-clang - MSYS2 Packages
https://packages.msys2.org/packages/mingw-w64-x86_64-clang

13 20 21 Environments - MSYS2
https://www.msys2.org/docs/environments/

14 15 Package: mingw-w64-x86_64-clang-tools-extra - MSYS2 Packages
https://packages.msys2.org/packages/mingw-w64-x86_64-clang-tools-extra

16 22 23 24 30 31 34 msys2 vscode integration c++ vcpkg cmake · GitHub
https://gist.github.com/raspberrypisig/0e1f35019b4a930cf9df5bbac13e10ae

17 26 28 29 32 33 CMake Kits — CMake Tools 1.4.0 documentation
https://vector-of-bool.github.io/docs/vscode-cmake-tools/kits.html

25 IDEs and Text Editors - MSYS2
https://www.msys2.org/docs/ides-editors/