# C/C++ Extension

<https://code.visualstudio.com/docs/languages/cpp>

Adds language support for C/C++, including:

Language services:

* Syntax highlight
* Code formatting (clang-format)
* Snippet
* Go to definition/declaration
* Peek definition/declaration
* Class/method/symbol navigation
* Signature help
* Quick info (hover)
* Error squiggles
* IntelliSense

Debugging:

* Windows (PDB, MinGW/Cygwin), Linux and macOS applications
* Line by line code stepping
* Breakpoints (including conditional and function breakpoints)
* Variable inspection
* Multi-threaded debugging support
* Core dump debugging support
* Executing GDB or MI commands directly when using 'C++ (GDB/LLDB)' debugging environment

<https://code.visualstudio.com/docs/cpp/cpp-ide>

## IntelliSense

Without any configuration, extension will attempt to locate headers by searching your workspace folder and by emulating a compiler it finds on your computer. But if if this is insufficient, you can self-configure IntelliSense.

There are two popular options for self-configuring IntelliSense for C++. Here is a comparison between them:

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Default IntelliSense** | **Clangd** |
| Ease of setup | *Winner*: Beginner-friendly. Auto-detects compilers, uses c\_cpp\_properties.json with UI support (C/C++: Edit Configurations). Ideal for quick setup or small projects. | Requires compile\_commands.json or compile\_flags.txt. More complex for non-CMake projects, challenging for beginners. |
| Performance | Adequate for small/medium projects. Slower in large codebases, with laggy completion and high memory usage. Database may need resets. | *Winner*: Fast indexing, near-instant completion. Scales well to large projects. Lower memory usage, but depends on correct configuration. |
| Accuracy | Decent for C++11/14 or simple code. Struggles with modern C++ (C++20/23), templates, or complex macros. May miss symbols. | *Winner*: Highly accurate, excels with modern C++, templates, and macros due to Clang’s parser. |
| Build system compatibility | *Winner*: Flexible with any build system or compiler (GCC, Clang, MSVC). Manual c\_cpp\_properties.json configuration is versatile. | Best with CMake/Bazel (compile\_commands.json). Non-CMake projects need tools like Bear. Less flexible for custom builds. |
| Modern c++ support | Limited support for C++20/23 features (e.g., concepts, modules). Less reliable for heavy template usage. | *Winner*: Strong support for C++20/23, templates, and complex constructs. Precise parsing for modern standards. |
| Refactoring | Limited; basic rename symbol support. Lacks advanced refactoring features like extract function. | *Winner*: Supports local refactorings (e.g., rename, extract function) via code actions, enhancing productivity in modern workflows. |
| Additional features | Basic clang-tidy, clang-format integration. Lacks inlay hints, advanced refactoring, or semantic highlighting. | *Winner*: Inlay hints, advanced refactoring (rename, extract), semantic highlighting, seamless clang-tidy via .clang-tidy. |
| Cross-platform support | Excellent support for Windows, macOS, Linux. Strong MSVC integration on Windows. | Excellent support for all platforms. Auto-downloads Clangd binary, but MSVC setups may need extra configuration. |
| Community & updates | Large user base, Microsoft-backed. Partially open-source. Active GitHub issue tracking. | *Winner*: Fully open-source, LLVM-backed. Rapid updates, strong community adoption in open-source projects. |
| Use case | Best for beginners, small projects, non-CMake builds. | Best for large projects, modern C++, CMake/Bazel workflows, or cross-editor compatibility. |

### Using Default IntelliSense

Running the *C/C++: Edit Configurations (UI)* command will create a c\_cpp\_properties.json file. Now you can modify it.

Read more:

<https://code.visualstudio.com/docs/cpp/configure-intellisense>

<https://code.visualstudio.com/docs/cpp/faq-cpp#_how-do-i-get-intellisense-to-work-correctly>

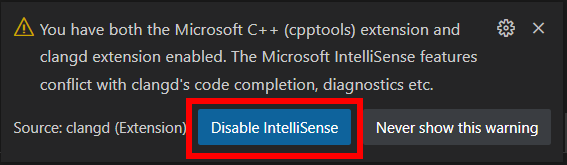
<https://code.visualstudio.com/docs/cpp/customize-cpp-settings>

### Using ClangD

The default intelliSense and Clang will fight over each other as they both try to do the same task at the same time. You need to disable the default intelliSense by adding the following line to your settings.json file:

"C\_Cpp.intelliSenseEngine": "disabled",

Another way is clicking on "Disable IntelliSense" button:



## Code Compilation

In tasks.json:

{

    "tasks": [

        {

            "type": "shell",

            "label": "C/C++: g++.exe build active file", // name whatever you like

            "command": "C:/Program Files/mingw-w64/bin/g++.exe", // specify the program to run, in this case, g++

            "args": [ // specify args passed to g++

                "-g",

                "${fileDirname}/\*.cpp", // match all files with extension \*.cpp

// another way: "${workspaceFolder}/\*.cpp"

                "-o",

                "${fileDirname}/${fileBasenameNoExtension}.exe" // bin output file has the same name as active file

            ],

            "options": { // specify additional options for the task execution

                "cwd": "${workspaceFolder}" // set the current working directory to ${workspaceFolder}

            },

            "problemMatcher": [ // selects the output parser to use for finding errors

// and warnings in the compiler output

                "$gcc"

            ],

            "group": {

                "kind": "build", // indicate that the task belongs to a build group

                "isDefault": true // indicate that the task is the default task that

// will run when you initiate a build command in VSCode.

            }

        }

    ],

    "version": "2.0.0"

}

## Code Debugging

**Full guides:**

* https://code.visualstudio.com/docs/cpp/cpp-debug
* <https://code.visualstudio.com/docs/cpp/launch-json-reference>
* Windows: <https://code.visualstudio.com/docs/cpp/config-mingw>
* Linux: <https://code.visualstudio.com/docs/cpp/config-linux>

**Common issues:**

* To fix error *'Unable to launch debugger (gdb) with root permissions'*, run:

$ sudo sysctl -w kernel.yama.ptrace\_scope=0

### Attach Debug

**When to use attach debug?**

1. Multi processes. We can debug multiproesses with breakpoints, but it's very difficult.
2. Skip building source code from VSCode. By default, starting debugging from VSCode triggers building the source code first, but if we want to skip the building process and go straight to the debugging process? One simple way is to use attach debug! (If needed, add a sleep for several seconds at the entry point of the program, so we can prevent our program running and finished so fast).

**How to configure attach debug?**

Step 1: Modify launch.json (only for the first time)

{

  "version": "2.0.0",

  "options": {

    // Environment variables

    "env": {

      // The PATH to system executables (and user executables). Each path is separated by a ':' notation (without spaces)

      "PATH": "/bin/:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin:/other\_paths/to/binary/files",

      // Path(s) to dynamic libs (.so files). Each path is separated by a ':' notation (without spaces)

      "LD\_LIBRARY\_PATH": "/path1/to/dynamic/libs:/path2/to/dynamic/libs"

    }

  },

  "configurations": [

    {

      "name": "myConfig1",                      // Configuration name (can be anything)

      "processId": "${command:pickProcess}",    // Pick a process from drop-down list every time starting debugging

      "program": "/path/to/executable1",        // Specify an executable file's path. Can use ${workspaceFolder}

      "type": "cppdbg",

      "request": "attach",                      // Here we want ATTACH debugging

      "MIMode": "gdb",

      "setupCommands": [

        {

          "description": "Enable pretty-printing for gdb",

          "text": "-enable-pretty-printing",

          "ignoreFailures": false

        }

      ],

      "miDebuggerPath": "/usr/bin/gdb"          // Path to debugger

    },

    {

      "name": "myConfig2",                      // Another configuration name (can be anything)

      "processId": "${command:pickProcess}",    // Pick another process from drop-down list every time starting debugging

      "program": "/path/to/executable2",        // Specify another executable file's path. Can use ${workspaceFolder}

      // ..... Similar settings as above

    }

  ]

}

Step 2: Run the executable

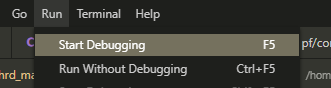
Run the executable which you want to attach debug.

Step 3: Attach the executable

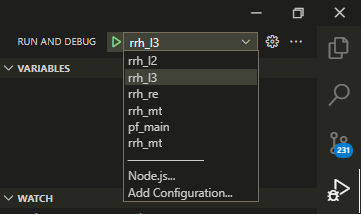
From VSCode, select the executable which you just run at the step 2.

Note:

* Don't start debugging from menu, which cannot pick process:



* But start debugging from sidebar:

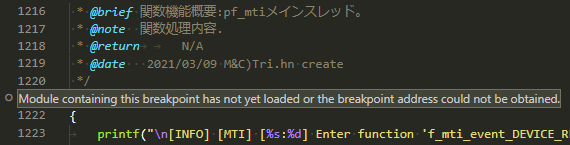


Common issues:

* To fix error *'Superuser access is required to attach to a process. Attaching as superuser can potentially harm your computer. Do you want to continue? [y/N]'*, run:

$ echo 0| sudo tee /proc/sys/kernel/yama/ptrace\_scope

* Following error is caued by setting wrong process in launch.son:



* Unknow:

