# ClangD

## ClangD vs Clang-Tidy

They're both tools provided by the Clang project, which is part of the LLVM. But they serve **different purposes**. While Clang-Tidy is a static code analysis (or linter), ClangD is a *Language Server Protocol* (LSP).

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Clang-Tidy** | **ClangD** | **Note** |
| Static code analysis | ✔️ | ❌ (partial) | Clang-tidy offers a full suite of checks (e.g., bugprone-, performance-); clangd integrates some via clang-tidy but disables expensive checks for performance. |
| Code checks and customizations based on rules | ✔️ | ❌ |  |
| Automatic issue fixing | ✔️ | ❌(partial) | Clang-tidy supports --fix for automatic fixes; clangd offers fixes through editor actions, often linked to clang-tidy. |
| Code completion | ❌ | ✔️ | Clangd provides real-time suggestions, essential for IDEs. |
| Code navigation | ❌ | ✔️ | Includes go-to-definition, find references, etc., supported by clangd. |
| Semantic highlighting | ❌ | ✔️ |  |
| Refactoring support | ❌ | ✔️ |  |
| Real-time feedback | ❌ | ✔️ | Clangd provides instant diagnostics as you edit; clang-tidy is batch-oriented. |

Once again, ClangD and Clang-Tidy serve **different purposes**. You don't need to choose because you can use both. **Just install LLVM**.

<https://clangd.llvm.org/installation>

## Configuration

### File

Configuration is stored in YAML format. These are:

|  |  |
| --- | --- |
| **Type** | **Description** |
| **Project configuration** | A file named .clangd in the source tree.  clangd searches in all parent directories of the active code file.  Generally this should be used for shared and checked-in settings. |
| **User configuration** | A config.yaml file in an OS-specific directory.   * *Windows*: %LocalAppData%\clangd\config.yaml, typically C:\Users\Bob\AppData\Local\clangd\config.yaml. * *macOS*: ~/Library/Preferences/clangd/config.yaml. * *Linux and others*: $XDG\_CONFIG\_HOME/clangd/config.yaml, typically ~/.config/clangd/config.yaml.   Generally this should be used for private settings. And can be scoped to projects using If conditions |

Each file can contain multiple fragments separated by --- lines. (This is only useful if the fragments have different If conditions).

Changes in config files should take effect immediately as you continue to edit code.

You might not know!

JSON is a subset of YAML, so you can use that syntax if you prefer.

### Schema

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Sub-field** | **Description** | **Example** |
| If | PathMatch | Files to apply checks.  *Value can be scalar or array.*  *For project config, path is relative. For global config, path is absolute.* | If: # Apply this config  PathMatch: .\*\.h # to all headers  PathExclude: include/llvm-c/.\* # except these |
| PathExclude | Files to exclude from checks.  *Value can be scalar or array.* |
| CompileFlags | Add | Flags to append to the compile command (compile\_commands.json).  *Value can be scalar or array.* | CompileFlags:  Add: [-xc++, -Wall]  Remove: [-W\*]  CompilationDatabase: Ancestors # Search all parent directories  Compiler: clang++  BuiltinHeaders: QueryDriver |
| Remove | Flags to remove from compile command, supports wildcards.  *Value can be scalar or array.* |
| CompilationDatabase | Directory to search for compilation database.  *Can be a path, Ancestors (default), or None.* |
| Compiler | Executable name for compiler. |
| BuiltinHeaders | Controls whether clangd should include its own built-in headers (like stddef.h), or use the system header found from the query driver (via the *--query-driver* command line argument).  *Can be Clangd (default) or QueryDriver.* |
| Index | Background | Whether to build files in background for project index.  This is checked for translation units only, not headers they include.  *Can be Build (default) or Skip.* | Index:  Background: Skip  External:  File: /abs/path/to/an.index.idx  External:  Server: my.index.server.com:50051  External:  MountPoint: /files/  StandardLibrary: Yes |
| External  .File | Path to external index file.  *Must be an absolute path.* |
| External  .Server | Server for external index.  *Must be URL.* |
| External  .MountPoint | Source root for external index.  *Must be absolute path in global config, or relative path in local config. Default: fragment location.* |
| StandardLibrary | Eagerly index standard library for completions.  Can be Yes or No. |
| Style | FullyQualifiedNamespaces | Namespaces always fully qualified, meaning no using declarations, always spell out the whole name (with or without leading ::).  *List of namespace names.* | Style:  FullyQualifiedNamespaces: [std, boost]  QuotedHeaders: ["src/.\*"]  AngledHeaders: ["path/sdk/.\*"] |
| QuotedHeaders | Headers inserted with "" if path matches regex.  *List of regexes.* |
| AngledHeaders | Headers inserted with <> if path matches regex.  *List of regexes.* |
| Diagnostics | Suppress | Diagnostic codes to suppress.  Valid values are:   * '\*', to disable all diagnostics * diagnostic codes exposed by clangd (e.g unknown\_type, -Wunused-result) * clang internal diagnostic codes (e.g. err\_unknown\_type) * warning categories (e.g. unused-result) * clang-tidy check names (e.g. bugprone-narrowing-conversions). | Diagnostics:  Suppress: unknown\_type |
| ClangTidy  .Add | List of clang-tidy checks to enable. Supports globs. | Diagnostics:  ClangTidy:  Add: bugprone-\*  Remove: modernize-use-trailing-return-type  CheckOptions: {  readability-identifier-naming.VariableCase: CamelCase  }  FastCheckFilter: Strict |
| ClangTidy  .Remove | List of clang-tidy checks to disable. Supports globs. |
| ClangTidy  .CheckOptions | Options for clang-tidy checks.  Format is <key>: <value>. |
| ClangTidy  .FastCheckFilter | Run clang-tidy checks based on speed.  Can be Strict (default - Run only checks measured to be fast), Loose (Run checks unless they are known to be slow), None (Run checks regardless of their speed) |
| UnusedIncludes | Enable [unused includes diagnostics](https://clangd.llvm.org/design/include-cleaner).  Can be Strict (default), None. | Diagnostics:  UnusedIncludes: Strict |
| Includes  .IgnoreHeader | Headers to ignore in Include Cleaner diagnostics.  List of regexes. | Diagnostics:  Includes:  IgnoreHeader: ["<.\*>", ".\*\.inc"]  AnalyzeAngledIncludes: true |
| Includes  .AnalyzeAngledIncludes | Enable detection of unused angled includes, excluding std lib.  Boolean. Default: False. |
| MissingIncludes | Enable missing includes diagnostics.  Can be Strict, None (default). | Diagnostics:  MissingIncludes: Strict |
| Completion | AllScopes | Whether code completion should include suggestions from scopes that are not visible.  Can be Yes (default) or No. | Completion:  AllScopes: Yes  ArgumentLists: FullPlaceholders  HeaderInsertion: IWYU |
| ArgumentLists | What to insert in argument list position when completing a call to a functio .  Can be None, OpenDelimiter, Delimiters, FullPlaceholders (default). |
| HeaderInsertion | Add #include directives for completions.  Can be IWYU (default), Never. |
| InlayHints | Enabled | Enable/disable inlay-hints for all kinds.  Can be Yes (default) or No. | InlayHints:  Enabled: Yes  ParameterNames: Yes  DeducedTypes: Yes  Designators: Yes  BlockEnd: No  DefaultArguments: No  TypeNameLimit: 24 |
| ParameterNames | Enable/disable parameter names inlay-hints.  Can be Yes (default) or No. |
| DeducedTypes | Enable/disable deduced types inlay-hints.  Can be Yes (default) or No. |
| Designators | Enable/disable designators inlay-hints.  Can be Yes (default) or No. |
| BlockEnd | Enable/disable block end comment inlay-hints.  Can be Yes or No (default). |
| DefaultArguments | Enable/disable default arguments inlay-hints.  Can be Yes or No (default). |
| TypeNameLimit | Character limit for type hints, 0 means no limit.  Integer. Default: 24. |
| Hover | ShowAKA | Control printing of desugared types in hover cards.  Can be Yes or No (default). | Hover:  ShowAKA: No |
| SemanticTokens | DisabledKinds | Semantic token kinds not sent to client.  *Available kinds could be found*[*here*](https://clangd.llvm.org/features#kinds)*in the Kind column.* | SemanticTokens:  DisabledKinds: []  DisabledModifiers: [] |
| DisabledModifiers | Semantic token modifiers not sent to client.  *Available modifiers could be found*[*here*](https://clangd.llvm.org/features#modifiers)*in the Modifier column.* |

# Clang-Tidy

Clang-Tidy is a Clang-based C++ "linter" tool. It helps diagnose and fix typical programming errors, like style violations, interface misuse, or bugs that can be deduced via static analysis.

Clang-Tidy guide:

<https://clang.llvm.org/extra/clang-tidy/>

<https://docs.platformio.org/en/latest//advanced/static-code-analysis/tools/clang-tidy.html>

https://manpages.ubuntu.com/manpages/jammy/man1/clang-tidy.1.html

## Bug Types

Some of the defects that might be detected include buffer overflow, potential NULL pointer dereferences, use of deallocated memory, out-of-scope memory usage, failure to set a return value from a subroutine, etc.

For a full checklist, see <https://clang.llvm.org/extra/clang-tidy/checks/list.html>

## Configuration

Create a config file called .clang-tidy and then call its path using the --config-file=<path> option. Clang-Tidy will try to find it in the closest parent directory of the source file.

This file is specified in **YAML format**. It has following options:

**CheckOptions** - List of key-value pairs defining check-specific options. Example:

CheckOptions:

some-check.SomeOption: 'some value'

**Checks** - Same as '--checks'. Additionally, the list of

globs can be specified as a list instead of a string.

**ExtraArgs** - Same as '--extra-args'.

**ExtraArgsBefore** - Same as '--extra-args-before'.

**FormatStyle** - Same as '--format-style'.

**HeaderFileExtensions** - File extensions to consider to determine if a

given diagnostic is located in a header file.

**HeaderFilterRegex** - Same as '--header-filter-regex'.

**ImplementationFileExtensions** - File extensions to consider to determine if a

given diagnostic is located in an implementation file.

**InheritParentConfig** - If this option is true in a config file, the

configuration file in the parent directory

(if any exists) will be taken and the current

config file will be applied on top of the parent one.

**SystemHeaders** - Same as '--system-headers'.

**UseColor** - Same as '--use-color'.

**User** - Specifies the name or e-mail of the user

running clang-tidy. This option is used, for

example, to place the correct user name in

TODO() comments in the relevant check.

**WarningsAsErrors** - Same as '--warnings-as-errors'.

The effective configuration can be inspected using --dump-config:

$ clang-tidy --dump-config

---

Checks: '-\*,some-check'

WarningsAsErrors: ''

HeaderFileExtensions: ['', 'h','hh','hpp','hxx']

ImplementationFileExtensions: ['c','cc','cpp','cxx']

HeaderFilterRegex: ''

FormatStyle: none

InheritParentConfig: true

User: user

CheckOptions:

some-check.SomeOption: 'some value'

...

Below are all checks (defined in forms of globs):

|  |  |
| --- | --- |
| **Name prefix** | **Description** |
| abseil-\* | Checks related to Abseil library |
| altera-\* | Checks related to OpenCL programming for FPGAs |
| android-\* | Checks related to Android |
| boost-\* | Checks related to Boost library |
| bugprone-\* | Checks that target **bug-prone code constructs** |
| cert-\* | Checks related to CERT Secure Coding Guidelines |
| clang-analyzer-\* | **Clang Static Analyzer checks**. |
| concurrency-\* | Checks related to **concurrent programming** (including threads, fibers, coroutines, etc.). |
| cppcoreguidelines-\* | Checks related to C++ Core Guidelines |
| darwin-\* | Checks related to Darwin coding conventions |
| fuchsia-\* | Checks related to Fuchsia coding conventions |
| google-\* | Checks related to Google coding conventions |
| hicpp-\* | Checks related to High Integrity C++ Coding Standard |
| linuxkernel-\* | Checks related to the Linux Kernel coding conventions |
| llvm-\* | Checks related to the LLVM coding conventions |
| llvmlibc-\* | Checks related to the LLVM-libc coding standards |
| misc-\* | Checks that we didn’t have a better category for |
| modernize-\* | Checks that advocate **usage of modern (currently "modern" means "C++11") language** constructs. |
| mpi-\* | Checks related to MPI (Message Passing Interface) |
| objc-\* | Checks related to Objective-C coding conventions |
| openmp-\* | Checks related to OpenMP API |
| performance-\* | Checks that target **performance-related issues** |
| portability-\* | Checks that target **portability-related issues** that don’t relate to any particular coding style |
| readability-\* | Checks that target readability-related issues that don’t relate to any particular coding style |
| zircon-\* | Checks related to Zircon kernel coding conventions. |

**Notes**:

* **Options specified in command-line will overwrite** their counterparts in the config file.

## Usage

### Check One File

In the command line, run:

$ clang-tidy [options] <path-to-file>

Where:

* The <source0> specify the path of the source file.

### Check All Files in Folder

In the command line, run:

$ clang-tidy [options] <path-to-file ... path-to-file>

Where:

* The <source0 ... sourceN> specify the paths of the source files.

### Choose Checks

To enable or disable a specific check, you can either use a configuration file (as mentioned in [this session](#_Configuration)) or use --checks= option (as mentioned here). If you don't specify it, Clang-Tidy uses checks enabled by default.

**Note**: The -checks= option specifies a comma-separated list of positive and negative (prefixed with -) globs. Positive globs add subsets of checks, and negative globs remove them.

For example:

$ clang-tidy test.cpp -checks=-\*,clang-analyzer-\*,-clang-analyzer-cplusplus\*

will disable all default checks (-\*) and enable all clang-analyzer-\* checks except for clang-analyzer-cplusplus\* ones.

**Tips**:

* To list all the enabled checks, use the -list-checks option.
* The -warnings-as-errors= option considers warnings as errors.

### Configure Compilation Options

These options include C/C++ standard, included header files, preprocessors, compiliation flags, etc.

To specify them, use -- following by the options:

$ clang-tidy test.cpp -- -Imy\_project/include -DMY\_DEFINES ...

### Configure Compile Command Database

You can just add compilation options with --, and still be able to run analysis check with Clang-Tidy.

But if you want a more precise (how much???) checking result, you should pass use the Compile Command Database file (called compile\_commands.json) with the -p <build-path> option.

For an example of how to do this, see [How To Setup Tooling For LLVM](https://clang.llvm.org/docs/HowToSetupToolingForLLVM.html). Clang-Tidy need it to figure out specific build options for each file.

For example, it can be in a CMake build directory (use -DCMAKE\_EXPORT\_COMPILE\_COMMANDS=ON CMake option to get this output).

### Suppress Certain Checks

#### Command Line

The -config= option is used to specify suppressions on the command line. The full syntax is:

-config="{Checks: '\*', CheckOptions: [{key1: x, value1: y}, { key2: x, value2: y}]}"

#### In-File

In the above session, we mentioned the config file .clang-tidy. We can adjust (not suppress???) warnings using it via the Checks and CheckOptions fields.

The following example enables all the checks and provides the additional option to cppcoreguidelines-avoid-magic-numbers.IgnoredIntegerValues so that it doesn’t treat 1, 2, 3, 4 as magic integer numbers.

Checks: '\*'

CheckOptions:

- key: cppcoreguidelines-avoid-magic-numbers.IgnoredIntegerValues

value: '1;2;3;4;'

Or:

Checks: '\*'

CheckOptions:

- { key: cppcoreguidelines-avoid-magic-numbers.IgnoredIntegerValues, value: '1;2;3;4;' }

Or:

Checks: '\*'

CheckOptions:

cppcoreguidelines-avoid-magic-numbers.IgnoredIntegerValues: '1;2;3;4;'

The following example completely disables avoid-magic-numbers check. So Clang-Tidy won’t treat magic number as a problem any more.

Checks: '\*,-cppcoreguidelines-avoid-magic-numbers,-readability-magic-numbers'

**Tips**:

* You should generate a full .clang-tidy with all CheckOptions using command:

$ clang-tidy -checks=\* --dump-config > .clang-tidy

#### In-Line

Clang-Tidy has a generic mechanism to suppress diagnostics using NOLINT, NOLINTNEXTLINE, and NOLINTBEGIN … NOLINTEND comments. For details, check [here](https://clang.llvm.org/extra/clang-tidy/#suppressing-undesired-diagnostics).

More usecases to suppress warnings: <https://copyprogramming.com/howto/inline-way-to-disable-clang-tidy-checks>

## Tips

### VSCode Extension

There are two modes for running Clang-Tidy in [C/C++ Official Extension](https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools):

* **C/C++ IntelliSense**: Without Clang-Tidy, we all already use IntelliSense to navigate our code, achieve code completion and many more. All necessary settings for IntelliSense are configured in c\_cpp\_properties.json file. Now, we have nothing more to configure for Clang-Tidy because it'll pick up necessary compilation options (C/C++ standard, included header files, preprocessors, compiliation flags, etc.) from the c\_cpp\_properties.json file. We just let IntelliSense control all stuffs!

To enable/disable this mode, set "C\_Cpp.codeAnalysis.clangTidy.useBuildPath": true / false.

* **Compile Command Database**: This way, we have to create and configure the compile\_commands.json, and (sadly) ignore the c\_cpp\_properties.json file. Check this [session](#_Configure_Compile_Command).

To enable/disable this mode, set "C\_Cpp.codeAnalysis.clangTidy.useBuildPath": true / false.

**Tips:**

* The command to run Clang-Tidy is logged to the Output windows of the C/C++ extension. To be able to see these logs, set C\_Cpp.loggingLevel set to Information.

**Sample of VSCode's settings.json**:

"C\_Cpp.codeAnalysis.clangTidy.enabled": true,           // Enable or disable Clang-Tidy

"C\_Cpp.codeAnalysis.clangTidy.path": "clang-tidy",      // Path to Clang-Tidy executable

"C\_Cpp.codeAnalysis.runAutomatically": true,            // Whether to run checking when a file is opened or saved

"C\_Cpp.codeAnalysis.clangTidy.config": "",              // Path to .clang-tidy config file. Leave empty if you want Clang-Tidy finds .clang-tidy file in its parent directories

"C\_Cpp.codeAnalysis.clangTidy.args": [],                // Additional command line arguments to pass to `clang-tidy`

"C\_Cpp.codeAnalysis.clangTidy.useBuildPath": false,     // If `true` and `compileCommands` is set, the `-p=<build-path>` argument is passed to `clang-tidy` instead of build arguments being passed after `--`. This may not work if environment variables aren't set so that system includes can be found.

"C\_Cpp.loggingLevel": "Information",                    // Set logging level for C/C++ extension. By default, only error logs are printed

# Clang-Format

[Clang-Format](https://clang.llvm.org/docs/ClangFormat.html) describes **a set of tools to** **format code in C/C++/Java/JavaScript/Objective-C/Objective-C++/Protobuf**.

## Installation

### Linux

**With Apt**

In Ubuntu, you can download and install **Clang-Format 17** or other versions with following steps:

$ wget https://apt.llvm.org/llvm.sh

$ chmod +x llvm.sh

$ sudo ./llvm.sh 17

$ sudo apt install clang-format-17

$ rm -r llvm.sh; rm llvm.sh.1

NOTE: By default, if you ignore above steps and just run $ sudo apt install clang-format, then only **Clang-Format 10** is installed.

**With Homebrew**

In Ubuntu, you can download and install **Clang-Format 17** or other versions with following steps:

$ brew install llvm # Same as $ brew install llvm@17

### Windows

You can download and install the LLVM toolchain from the official website.

## Configuration

### Style Configurations

Clang-Format supports **two ways** to provide custom style options:

1. Directly specify style configuration in the **-style=**'{key1: value1, key2: value2, ...}' command line option.
2. Indirectly specify style configuration in the .clang-format file in the project directory:
   1. Use **-style**=file: Clang-Format will try to find the .clang-format file located in the closest parent directory of the input code file.
   2. Use **-style**=file:<format\_file\_path>: Clang-Format will use the file located at <format\_file\_path>. The path may be absolute or relative to the working directory.

The .clang-format file uses **YAML** format:

key1: value1

key2: value2

# A comment

...

### Style Options

Clang-Format 17: <https://releases.llvm.org/17.0.1/tools/clang/docs/ClangFormatStyleOptions.html>

Clang-Format 18 (latest, but in development phase): <https://clang.llvm.org/docs/ClangFormatStyleOptions.html>

For a sample of .clang-format file, check file *Tutorials\Clean Code - Coding Conventions\.clang-format*

## Usage

### Running Clang-Format Alone

Clang-Format could format **a single file** or **all files** with the same file extension.

To format all .cpp files, run:

$ clang-format -i \*.cpp

To format all .h, .c, .hpp, .cpp, .cu files together, run:

$ find . -regex '.\*\.\(cpp\|hpp\|cu\|c\|h\)' -exec clang-format -style=file -i {} \;

### VSCode Extension

VS Code has some extensions to support Clang-Format. They help us format the code right in the editor by either select "*Format Document*" (format the whole file) or "*Format Selection*" (from the selected text) from VSCode’s context menu. Or you can format code each time the file is saved, or even when the file is updated (see below settings).

#### [C/C++ Official Extension](https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools)

The C/C++ extension by MS supports Clang-Format as its default formatter. All we have to do is to install it and set VSCode’s settings.jon file:

"C\_Cpp.clang\_format\_path": "",          // Path to Clang-Format executable.

                                        // If not set, use `clang-format` available in the environment path

"C\_Cpp.clang\_format\_style": "file",     // Choose to use .clang-format file.

                                        // Clang-Format will find it in the closest parent directory

"C\_Cpp.formatting": "default",          // Choose the formatting engine. It can be Clang-Format (by default), vcFormat (Visual C++)

                                        // Set "disabled" if you want to disable code formatting

"editor.formatOnSave": false,           // Automatically format a file on save

"editor.formatOnType": false,           // Automatically format a file on typing

"editor.formatOnPaste": false,          // Automatically format a file on pasting

"editor.formatOnSaveMode": "file",     // Control how format-on-save work.

                                        // Set "file" to format whole file. Or "modifications" to format only modified code (requires source control)

#### [Clang-Format Extension](https://marketplace.visualstudio.com/items?itemName=xaver.clang-format)

It’s not official by it works very well. After installing, add following settings to VSCode’s settings.jon file:

"clang-format.executable": "clang-format-17", // Path to Clang-Format executable

// Here, ver 17 is used. Change it if using a different version

"clang-format.style": "file",                 // Choose to use .clang-format file.

// Clang-Format will find it in the closest parent directory

"editor.formatOnSave": false,                 // Automatically format a file on save

"editor.formatOnType": false,                 // Automatically format a file on typing

"editor.formatOnPaste": false,                // Automatically format a file on pasting

"editor.formatOnSaveMode": "file",       // Control how format-on-save work.

                                        // Set "file" to format whole file. Or "modifications" to format only modified code (requires source control)

"[cpp]": {

    "editor.defaultFormatter": "xaver.clang-format"  // Set Clang-Format as the default formatter.

                                                     // Now, when you click "Format Document" from context menu, VSCode automatically chooses Clang instead of asking you.

},

### Makefile Integration

We can add Clang-Format to Makefile. In below example, when running $ make fmt, code will be formatted:

fmt:

@clang-format -i \

-style="{ BasedOnStyle: Google, \

AlignConsecutiveAssignments: true, \

AlignConsecutiveDeclarations: true, \

ColumnLimit: 0, \

IndentWidth: 4, \

AllowShortFunctionsOnASingleLine: None, \

AllowShortLoopsOnASingleLine: false, \

BreakBeforeBraces: Linux, \

SortIncludes: false, \

DerivePointerAlignment: false, \

PointerAlignment: Left, \

AlignOperands: true, \

}" \

$(shell find . -name '\*.cpp' -o -name '\*.hpp') \

$(shell find . -name '\*.c' -o -name '\*.h') \

$(shell find ../ru\_\* -name '\*.c' -o -name '\*.h')