# 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**.

It’s part of the [**LLVM**](https://clang.llvm.org/) **toolchain** (including a compiler, debugger, and code formatter – ClangFormat). Also, it has a very big community; this makes sure the toolchain will be maintained well.

## 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.

## Configurations

### 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 {} \;

### VS Code 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')

# AStyle

<https://astyle.sourceforge.net/astyle.html>

**Example**:

$ astyle --style=linux --pad-oper --pad-comma --pad-header

--break-blocks --break-closing-braces

--add-braces --keep-one-line-blocks

source1.cpp source2.cpp source3.cpp

**Explaination**:

|  |  |  |
| --- | --- | --- |
| **Item** | **Option** | **Example** |
| Operator | --pad-oper / -p  Insert space padding around operators. This will also pad commas. Any end of line comments will remain in the original column, if possible. Note that there is no option to unpad. Once padded, they stay padded. | if (foo==2)  a=bar((b-c)\*a,d--);  becomes:  if (foo == 2)  a = bar((b - c) \* a, d--); |
| Comma space | --pad-comma / -xg  Insert space padding after commas. This is not needed if pad-oper is used. Any end of line comments will remain in the original column, if possible. Note that there is no option to unpad. Once padded, they stay padded. | if (isFoo(a,b))  bar(a,b);  becomes:  if (isFoo(a, b))  bar(a, b); |
| Keywords  (if, else, while, etc.) | --pad-header / -H  Insert space padding between a header (e.g. 'if', 'for', 'while'...) and the following paren. Any end of line comments will remain in the original column, if possible. This can be used with unpad-paren to remove unwanted spaces. | if(isFoo((a+2), b))  bar(a, b);  becomes:  if (isFoo((a+2), b))  bar(a, b); |
| Linux braces  option | --style=linux / --style=knf / -A8  Linux style uses linux braces. Opening braces are broken from namespace, class, and function definitions. The braces are attached to everything else, including arrays, structs, enums, and statements within a function. The minimum conditional indent is one-half indent. If you want a different minimum conditional indent, use the K&R style instead. This style works best with a large indent. It frequently is used with an indent of 8 spaces.  Also known as Kernel Normal Form (KNF) style, this is the style used in the Linux BSD kernel. | int Foo(bool isBar) {  if (isFoo)  {  bar();  return 1;  }  }  becomes:  int Foo(bool isBar)  {  if (isFoo) {  bar();  return 1;  }  } |
| Pad empty lines | --break-blocks / -f  Pad empty lines around header blocks (e.g. 'if', 'for', 'while'...). | isFoo = true;  if (isFoo) {  bar();  }  isBar = false;  becomes:  isFoo = true;  if (isFoo) {  bar();  }  isBar = false; |
| Broken else | --break-closing-braces / -y  When used with --style=java, --style=kr, --style=stroustrup, --style=linux, or --style=1tbs, this breaks closing headers (e.g. 'else', 'catch', ...) from their immediately preceding closing braces. Closing header braces are always broken with the other styles. | void Foo(bool isFoo) {  if (isFoo) {  bar();  } else {  anotherBar();  }  }  becomes:  void Foo(bool isFoo) {  if (isFoo) {  bar();  }  else {  anotherBar();  }  } |
| Add braces | --add-braces / -j  Add braces to unbraced one line conditional statements (e.g. 'if', 'for', 'while'...). The statement must be on a single line. The braces will be added according to the requested brace style. If no style is requested the braces will be attached.  Braces will NOT be added to a multi-statement line if keep-one-line-statements is requested. Braces will NOT be added to a one line block if keep-one-line-blocks is requested. If used with --add-one-line-braces, the result will be one line braces. | if (isFoo)  isFoo = false;  becomes:  if (isFoo) {  isFoo = false;  } |
| Don't break one-line | --keep-one-line-blocks / -O  Don't break one-line blocks. | if (isFoo)  { isFoo = false; cout << isFoo << endl; }  remains unchanged |