Clang-Format Style Options

Clang-Format Style Options describes configurable formatting style options supported by ***LibFormat*** and ***ClangFormat***.

When using **clang-format** command line utility or `clang::format::reformat(...)` functions from code, one can either use one of the predefined styles (LLVM, Google, Chromium, Mozilla, WebKit) or create a custom style by configuring specific style options.

Configuring Style with clang-format

clang-format supports two ways to provide custom style options: directly specify style configuration in the `-style=` command line option or use `-style=file` and put style configuration in the `.clang-format` or `_clang-format` file in the project directory.

When using `-style=file`, **clang-format** for each input file will try to find the `.clang-format` file located in the closest parent directory of the input file. When the standard input is used, the search is started from the current directory.

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

```
key1: value1
key2: value2
# A comment.
...
```

The configuration file can consist of several sections each having different `Language:` parameter denoting the programming language this section of the configuration is targeted at. See the description of the **Language** option below for the list of supported languages. The first section may have no language set, it will set the default style options for all lanugages. Configuration sections for specific language will override options set in the default section.

When **clang-format** formats a file, it auto-detects the language using the file name. When formatting standard input or a file that doesn't have the extension corresponding to its language, `-assume-filename=` option can be used to override the file name **clang-format** uses to detect the language.

An example of a configuration file for multiple languages:

```
---
# We'll use defaults from the LLVM style, but with 4 columns indentation.
BasedOnStyle: LLVM
IndentWidth: 4
---
Language: Cpp
# Force pointers to the type for C++.
DerivePointerAlignment: false
PointerAlignment: Left
---
Language: JavaScript
# Use 100 columns for JS.
ColumnLimit: 100
---
Language: Proto
# Don't format .proto files.
DisableFormat: true
...
```

An easy way to get a valid `.clang-format` file containing all configuration options of a certain predefined style is:

```
clang-format -style=llvm -dump-config > .clang-format
```

When specifying configuration in the `-style=` option, the same configuration is applied for all input files. The format of the configuration is:

```
-style='{key1: value1, key2: value2, ...}'
```

Disabling Formatting on a Piece of Code

Clang-format understands also special comments that switch formatting in a delimited range. The code between a comment `// clang-format off` or `/* clang-format off */` up to a comment `// clang-format on` or `/* clang-format on */` will not be formatted. The comments themselves will be formatted (aligned) normally.

```
int formatted_code;
// clang-format off
    void    unformatted_code  ;
// clang-format on
void formatted_code_again;
```

Configuring Style in Code

When using `clang::format::reformat(...)` functions, the format is specified by supplying the **clang::format::FormatStyle** structure.

Configurable Format Style Options

This section lists the supported style options. Value type is specified for each option. For enumeration types possible values are specified both as a C++ enumeration member (with a prefix, e.g. `LS_Auto`), and as a value usable in the configuration (without a prefix: `Auto`).

BasedOnStyle (`string`)

The style used for all options not specifically set in the configuration.

This option is supported only in the **clang-format** configuration (both within `-style='{...}'` and the `.clang-format` file).

Possible values:

`LLVM` A style complying with the **LLVM coding standards**
`Google` A style complying with **Google's C++ style guide**
`Chromium` A style complying with **Chromium's style guide**
`Mozilla` A style complying with **Mozilla's style guide**
`WebKit` A style complying with **WebKit's style guide**

AccessModifierOffset (`int`)

The extra indent or outdent of access modifiers, e.g. `public:`.

AlignAfterOpenBracket (`bool`)

If `true`, horizontally aligns arguments after an open bracket.

This applies to round brackets (parentheses), angle brackets and square brackets. This will result in formattings like code someLongFunction(argument1, argument2); endcode

AlignEscapedNewlinesLeft (`bool`)

If `true`, aligns escaped newlines as far left as possible. Otherwise puts them into the right-most column.

AlignOperands (`bool`)

If `true`, horizontally align operands of binary and ternary expressions.

AlignTrailingComments (`bool`)

If `true`, aligns trailing comments.

AllowAllParametersOfDeclarationOnNextLine (`bool`)

Allow putting all parameters of a function declaration onto the next line even if `BinPackParameters` is `false`.

AllowShortBlocksOnASingleLine (`bool`)

Allows contracting simple braced statements to a single line.

E.g., this allows `if (a) { return; }` to be put on a single line.

AllowShortCaseLabelsOnASingleLine (`bool`)

If `true`, short case labels will be contracted to a single line.

AllowShortFunctionsOnASingleLine (`ShortFunctionStyle`)

Dependent on the value, `int f() { return 0; }` can be put on a single line.

Possible values:

- `SFS_None` (in configuration: `None`) Never merge functions into a single line.
- `SFS_Inline` (in configuration: `Inline`) Only merge functions defined inside a class.
- `SFS_Empty` (in configuration: `Empty`) Only merge empty functions.
- `SFS_All` (in configuration: `All`) Merge all functions fitting on a single line.

AllowShortIfStatementsOnASingleLine (`bool`)

If `true`, `if (a) return;` can be put on a single line.

AllowShortLoopsOnASingleLine (`bool`)

If `true`, `while (true) continue;` can be put on a single line.

AlwaysBreakAfterDefinitionReturnType (`bool`)

If `true`, always break after function definition return types.

More truthfully called 'break before the identifier following the type in a function definition'. PenaltyReturnTypeOnItsOwnLine becomes irrelevant.

AlwaysBreakBeforeMultilineStrings (`bool`)

If `true`, always break before multiline string literals.

AlwaysBreakTemplateDeclarations (`bool`)

If `true`, always break after the `template<...>` of a template declaration.

BinPackArguments (`bool`)

If `false`, a function call's arguments will either be all on the same line or will have one line each.

BinPackParameters (`bool`)

If `false`, a function declaration's or function definition's parameters will either all be on the same line or will have one line each.

BreakBeforeBinaryOperators (`BinaryOperatorStyle`)

The way to wrap binary operators.

Possible values:

- `BOS_None` (in configuration: `None`) Break after operators.
- `BOS_NonAssignment` (in configuration: `NonAssignment`) Break before operators that aren't assignments.
- `BOS_All` (in configuration: `All`) Break before operators.

BreakBeforeBraces (`BraceBreakingStyle`)

The brace breaking style to use.

Possible values:

- `BS_Attach` (in configuration: `Attach`) Always attach braces to surrounding context.
- `BS_Linux` (in configuration: `Linux`) Like `Attach`, but break before braces on function, namespace and class definitions.
- `BS_Stroustrup` (in configuration: `Stroustrup`) Like `Attach`, but break before function definitions, and 'else'.
- `BS_Allman` (in configuration: `Allman`) Always break before braces.
- `BS_GNU` (in configuration: `GNU`) Always break before braces and add an extra level of indentation to braces of control statements, not to those of class, function or other definitions.

BreakBeforeTernaryOperators (`bool`)

If `true`, ternary operators will be placed after line breaks.

BreakConstructorInitializersBeforeComma (`bool`)

Always break constructor initializers before commas and align the commas with the colon.

ColumnLimit (`unsigned`)

The column limit.

A column limit of `0` means that there is no column limit. In this case, clang-format will respect the input's line breaking decisions within statements unless they contradict other rules.

CommentPragmas (`std::string`)

A regular expression that describes comments with special meaning, which should not be split into lines or otherwise changed.

ConstructorInitializerAllOnOneLineOrOnePerLine (`bool`)

If the constructor initializers don't fit on a line, put each initializer on its own line.

ConstructorInitializerIndentWidth (`unsigned`)

The number of characters to use for indentation of constructor initializer lists.

ContinuationIndentWidth (`unsigned`)

Indent width for line continuations.

Cpp11BracedListStyle (`bool`)

If `true`, format braced lists as best suited for C++11 braced lists.

Important differences: - No spaces inside the braced list. - No line break before the closing brace. - Indentation with the continuation indent, not with the block indent.

Fundamentally, C++11 braced lists are formatted exactly like function calls would be formatted in their place. If the braced list follows a name (e.g. a type or variable name), clang-format formats as if the `{}` were the parentheses of a function call with that name. If there is no name, a zero-length name is assumed.

DerivePointerAlignment (`bool`)

If `true`, analyze the formatted file for the most common alignment of & and *. `PointerAlignment` is then used only as fallback.

DisableFormat (`bool`)

Disables formatting at all.

ExperimentalAutoDetectBinPacking (`bool`)

If `true`, clang-format detects whether function calls and definitions are formatted with one parameter per line.

Each call can be bin-packed, one-per-line or inconclusive. If it is inconclusive, e.g. completely on one line, but a decision needs to be made, clang-format analyzes whether there are other bin-packed cases in the input file and act accordingly.

NOTE: This is an experimental flag, that might go away or be renamed. Do not use this in config files, etc. Use at your own risk.

ForEachMacros (`std::vector<std::string>`)

A vector of macros that should be interpreted as foreach loops instead of as function calls.

These are expected to be macros of the form: code FOREACH(<variable-declaration>, ...) <loop-body> endcode

For example: BOOST_FOREACH.

IndentCaseLabels (`bool`)

Indent case labels one level from the switch statement.

When `false`, use the same indentation level as for the switch statement. Switch statement body is always indented one level more than case labels.

IndentWidth (`unsigned`)

The number of columns to use for indentation.

IndentWrappedFunctionNames (`bool`)

Indent if a function definition or declaration is wrapped after the type.

KeepEmptyLinesAtTheStartOfBlocks (`bool`)

If true, empty lines at the start of blocks are kept.

Language (`LanguageKind`)

Language, this format style is targeted at.

Possible values:

- `LK_None` (in configuration: `None`) Do not use.
- `LK_Cpp` (in configuration: `Cpp`) Should be used for C, C++, ObjectiveC, ObjectiveC++.
- `LK_Java` (in configuration: `Java`) Should be used for Java.

`LK_JavaScript` (in configuration: `JavaScript`) Should be used for JavaScript.

`LK_Proto` (in configuration: `Proto`) Should be used for Protocol Buffers (**https://developers.google.com/protocol-buffers/**).

MaxEmptyLinesToKeep (`unsigned`)

The maximum number of consecutive empty lines to keep.

NamespaceIndentation (`NamespaceIndentationKind`)

The indentation used for namespaces.

Possible values:

`NI_None` (in configuration: `None`) Don't indent in namespaces.

`NI_Inner` (in configuration: `Inner`) Indent only in inner namespaces (nested in other namespaces).

`NI_All` (in configuration: `All`) Indent in all namespaces.

ObjCBlockIndentWidth (`unsigned`)

The number of characters to use for indentation of ObjC blocks.

ObjCSpaceAfterProperty (`bool`)

Add a space after `@property` in Objective-C, i.e. use `\@property (readonly)` instead of `\@property(readonly)`.

ObjCSpaceBeforeProtocolList (`bool`)

Add a space in front of an Objective-C protocol list, i.e. use `Foo <Protocol>` instead of `Foo<Protocol>`.

PenaltyBreakBeforeFirstCallParameter (`unsigned`)

The penalty for breaking a function call after "call(".

PenaltyBreakComment (`unsigned`)

The penalty for each line break introduced inside a comment.

PenaltyBreakFirstLessLess (`unsigned`)

The penalty for breaking before the first `<<`.

PenaltyBreakString (`unsigned`)

The penalty for each line break introduced inside a string literal.

PenaltyExcessCharacter (`unsigned`)

The penalty for each character outside of the column limit.

PenaltyReturnTypeOnItsOwnLine (`unsigned`)

Penalty for putting the return type of a function onto its own line.

PointerAlignment (`PointerAlignmentStyle`)

Pointer and reference alignment style.

Possible values:

`PAS_Left` (in configuration: `Left`) Align pointer to the left.

`PAS_Right` (in configuration: `Right`) Align pointer to the right.

`PAS_Middle` (in configuration: `Middle`) Align pointer in the middle.

SpaceAfterCStyleCast (`bool`)

If `true`, a space may be inserted after C style casts.

SpaceBeforeAssignmentOperators (`bool`)

If `false`, spaces will be removed before assignment operators.

SpaceBeforeParens (`SpaceBeforeParensOptions`)

Defines in which cases to put a space before opening parentheses.

Possible values:

`SBPO_Never` (in configuration: `Never`) Never put a space before opening parentheses.

`SBPO_ControlStatements` (in configuration: `ControlStatements`) Put a space before opening parentheses only after control statement keywords (`for/if/while...`).

`SBPO_Always` (in configuration: `Always`) Always put a space before opening parentheses, except when it's prohibited by the syntax rules (in function-like macro definitions) or when determined by other style rules (after unary operators, opening parentheses, etc.)

SpaceInEmptyParentheses (`bool`)

If `true`, spaces may be inserted into '()'.

SpacesBeforeTrailingComments (`unsigned`)

The number of spaces before trailing line comments (`//` - comments).

This does not affect trailing block comments (`/**/` - comments) as those commonly have different usage patterns and a number of special cases.

SpacesInAngles (`bool`)

If `true`, spaces will be inserted after '<' and before '>' in template argument lists

SpacesInCStyleCastParentheses (`bool`)

If `true`, spaces may be inserted into C style casts.

SpacesInContainerLiterals (`bool`)

If `true`, spaces are inserted inside container literals (e.g. ObjC and Javascript array and dict literals).

SpacesInParentheses (`bool`)

If `true`, spaces will be inserted after '(' and before ')'.

SpacesInSquareBrackets (`bool`)

If `true`, spaces will be inserted after '[' and before ']'.

Standard (`LanguageStandard`)

Format compatible with this standard, e.g. use `A<A<int> >` instead of `A<A<int>>` for LS_Cpp03.

Possible values:

`LS_Cpp03` (in configuration: `Cpp03`) Use C++03-compatible syntax.
`LS_Cpp11` (in configuration: `Cpp11`) Use features of C++11 (e.g. `A<A<int>>` instead of `A<A<int> >`).
`LS_Auto` (in configuration: `Auto`) Automatic detection based on the input.

TabWidth (`unsigned`)

The number of columns used for tab stops.

UseTab (`UseTabStyle`)

The way to use tab characters in the resulting file.

Possible values:

`UT_Never` (in configuration: `Never`) Never use tab.
`UT_ForIndentation` (in configuration: `ForIndentation`) Use tabs only for indentation.
`UT_Always` (in configuration: `Always`) Use tabs whenever we need to fill whitespace that spans at least from one tab stop to the next one.

Examples

A style similar to the **Linux Kernel style**:

```
BasedOnStyle: LLVM
IndentWidth: 8
UseTab: Always
BreakBeforeBraces: Linux
AllowShortIfStatementsOnASingleLine: false
IndentCaseLabels: false
```

The result is (imagine that tabs are used for indentation here):

```
void test()
{
        switch (x) {
        case 0:
```

```
        case 1:
                do_something();
                break;
        case 2:
                do_something_else();
                break;
        default:
                break;
        }
        if (condition)
                do_something_completely_different();

        if (x == y) {
                q();
        } else if (x > y) {
                w();
        } else {
                r();
        }
}
```

A style similar to the default Visual Studio formatting style:

```
UseTab: Never
IndentWidth: 4
BreakBeforeBraces: Allman
AllowShortIfStatementsOnASingleLine: false
IndentCaseLabels: false
ColumnLimit: 0
```

The result is:

```
void test()
{
    switch (suffix)
    {
    case 0:
    case 1:
        do_something();
        break;
    case 2:
        do_something_else();
        break;
    default:
        break;
    }
    if (condition)
        do_somthing_completely_different();

    if (x == y)
    {
        q();
    }
    else if (x > y)
    {
        w();
    }
    else
    {
        r();
    }
}
```