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, Microsoft) 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
---
Language: CSharp
# Use 100 columns for C#.
ColumnLimit: 100
...
```

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**
- `Microsoft` A style complying with **Microsoft's style guide**

AccessModifierOffset (`int`)

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

AlignAfterOpenBracket (`BracketAlignmentStyle`)

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

This applies to round brackets (parentheses), angle brackets and square brackets.

Possible values:

- `BAS_Align` (in configuration: `Align`) Align parameters on the open bracket, e.g.:

  ```
  someLongFunction(argument1,
                   argument2);
  ```

- `BAS_DontAlign` (in configuration: `DontAlign`) Don't align, instead use `ContinuationIndentWidth`, e.g.:

  ```
  someLongFunction(argument1,
      argument2);
  ```

- `BAS_AlwaysBreak` (in configuration: `AlwaysBreak`) Always break after an open bracket, if the parameters don't fit on a single line, e.g.:

```
someLongFunction(
    argument1, argument2);
```

AlignConsecutiveAssignments (`bool`)

If `true`, aligns consecutive assignments.

This will align the assignment operators of consecutive lines. This will result in formattings like

```
int aaaa = 12;
int b    = 23;
int ccc  = 23;
```

AlignConsecutiveDeclarations (`bool`)

If `true`, aligns consecutive declarations.

This will align the declaration names of consecutive lines. This will result in formattings like

```
int         aaaa = 12;
float       b = 23;
std::string ccc = 23;
```

AlignConsecutiveMacros (`bool`)

If `true`, aligns consecutive C/C++ preprocessor macros.

This will align C/C++ preprocessor macros of consecutive lines. Will result in formattings like

```
#define SHORT_NAME       42
#define LONGER_NAME      0x007f
#define EVEN_LONGER_NAME (2)
#define foo(x)           (x * x)
#define bar(y, z)        (y + z)
```

AlignEscapedNewlines (`EscapedNewlineAlignmentStyle`)

Options for aligning backslashes in escaped newlines.

Possible values:

- `ENAS_DontAlign` (in configuration: `DontAlign`) Don’t align escaped newlines.

```
#define A \
  int aaaa; \
  int b; \
  int ddddddddd;
```

- `ENAS_Left` (in configuration: `Left`) Align escaped newlines as far left as possible.

```
true:
#define A   \
  int aaaa; \
  int b;    \
  int ddddddddd;

false:
```

- `ENAS_Right` (in configuration: `Right`) Align escaped newlines in the right-most column.

```
#define A                                                                      \
  int aaaa;                                                                    \
  int b;                                                                       \
  int ddddddddd;
```

AlignOperands (`bool`)

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

Specifically, this aligns operands of a single expression that needs to be split over multiple lines, e.g.:

```
int aaa = bbbbbbbbbbbbbb +
          ccccccccccccc;
```

AlignTrailingComments (`bool`)

If `true`, aligns trailing comments.

```
true:                                   false:
int a;     // My comment a      vs.     int a; // My comment a
int b = 2; // comment  b                int b = 2; // comment about b
```

AllowAllArgumentsOnNextLine (`bool`)

If a function call or braced initializer list doesn't fit on a line, allow putting all arguments onto the next line, even if `BinPackArguments` is `false`.

```
true:
callFunction(
    a, b, c, d);

false:
callFunction(a,
             b,
             c,
             d);
```

AllowAllConstructorInitializersOnNextLine (`bool`)

If a constructor definition with a member initializer list doesn't fit on a single line, allow putting all member initializers onto the next line, if `` `ConstructorInitializerAllOnOneLineOrOnePerLine` `` is true. Note that this parameter has no effect if `` `ConstructorInitializerAllOnOneLineOrOnePerLine` `` is false.

```
true:
MyClass::MyClass() :
    member0(0), member1(2) {}

false:
MyClass::MyClass() :
    member0(0),
    member1(2) {}
```

AllowAllParametersOfDeclarationOnNextLine (`bool`)

If the function declaration doesn't fit on a line, allow putting all parameters of a function declaration onto the next line even if `BinPackParameters` is `false`.

```
true:
void myFunction(
    int a, int b, int c, int d, int e);

false:
void myFunction(int a,
                int b,
                int c,
                int d,
                int e);
```

AllowShortBlocksOnASingleLine (`ShortBlockStyle`)

Dependent on the value, `while (true) { continue; }` can be put on a single line.

Possible values:

- `SBS_Never` (in configuration: `Never`) Never merge blocks into a single line.

```
while (true) {
}
while (true) {
  continue;
}
```

- `SBS_Empty` (in configuration: `Empty`) Only merge empty blocks.

```
while (true) {}
while (true) {
  continue;
}
```

- `SBS_Always` (in configuration: `Always`) Always merge short blocks into a single line.

```
while (true) {}
while (true) { continue; }
```

AllowShortCaseLabelsOnASingleLine (`bool`)

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

```
true:                                   false:
switch (a) {                    vs.     switch (a) {
case 1: x = 1; break;                   case 1:
case 2: return;                           x = 1;
}                                         break;
                                        case 2:
                                          return;
                                        }
```

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_InlineOnly` (in configuration: `InlineOnly`) Only merge functions defined inside a class. Same as “inline”, except it does not implies “empty”: i.e. top level empty functions are not merged either.

```
class Foo {
  void f() { foo(); }
};
void f() {
  foo();
}
void f() {
}
```

- `SFS_Empty` (in configuration: `Empty`) Only merge empty functions.

```
void f() {}
void f2() {
  bar2();
}
```

- `SFS_Inline` (in configuration: `Inline`) Only merge functions defined inside a class. Implies “empty”.

```
class Foo {
  void f() { foo(); }
};
void f() {
  foo();
}
void f() {}
```

- `SFS_All` (in configuration: `All`) Merge all functions fitting on a single line.

```
class Foo {
  void f() { foo(); }
};
void f() { bar(); }
```

AllowShortIfStatementsOnASingleLine (`ShortIfStyle`)

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

Possible values:

- `SIS_Never` (in configuration: `Never`) Never put short ifs on the same line.

```
if (a)
  return ;
else {
  return;
}
```

- `SIS_WithoutElse` (in configuration: `WithoutElse`) Without else put short ifs on the same line only if the else is not a compound statement.

```
if (a) return;
else
  return;
```

- `SIS_Always` (in configuration: `Always`) Always put short ifs on the same line if the else is not a compound statement or not.

```
if (a) return;
else {
  return;
}
```

AllowShortLambdasOnASingleLine (`ShortLambdaStyle`)

Dependent on the value, `auto lambda []() { return 0; }` can be put on a single line.

Possible values:

- `SLS_None` (in configuration: `None`) Never merge lambdas into a single line.
- `SLS_Empty` (in configuration: `Empty`) Only merge empty lambdas.

```
auto lambda = [](int a) {}
auto lambda2 = [](int a) {
    return a;
};
```

- `SLS_Inline` (in configuration: `Inline`) Merge lambda into a single line if argument of a function.

```
auto lambda = [](int a) {
    return a;
};
sort(a.begin(), a.end(), ()[] { return x < y; })
```

- `SLS_All` (in configuration: `All`) Merge all lambdas fitting on a single line.

```
auto lambda = [](int a) {}
auto lambda2 = [](int a) { return a; };
```

AllowShortLoopsOnASingleLine (`bool`)

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

~~**AlwaysBreakAfterDefinitionReturnType**~~ (`DefinitionReturnTypeBreakingStyle`)

The function definition return type breaking style to use. This option is **deprecated** and is retained for backwards compatibility.

Possible values:

- `DRTBS_None` (in configuration: `None`) Break after return type automatically. `PenaltyReturnTypeOnItsOwnLine` is taken into account.
- `DRTBS_All` (in configuration: `All`) Always break after the return type.
- `DRTBS_TopLevel` (in configuration: `TopLevel`) Always break after the return types of top-level functions.

AlwaysBreakAfterReturnType (`ReturnTypeBreakingStyle`)

The function declaration return type breaking style to use.

Possible values:

- `RTBS_None` (in configuration: `None`) Break after return type automatically. `PenaltyReturnTypeOnItsOwnLine` is taken into account.

```
class A {
  int f() { return 0; };
};
int f();
int f() { return 1; }
```

- `RTBS_All` (in configuration: `All`) Always break after the return type.

```
class A {
  int
  f() {
    return 0;
  };
};
int
f();
int
f() {
  return 1;
}
```

- `RTBS_TopLevel` (in configuration: `TopLevel`) Always break after the return types of top-level functions.

```
class A {
  int f() { return 0; };
};
int
f();
int
f() {
  return 1;
}
```

- `RTBS_AllDefinitions` (in configuration: `AllDefinitions`) Always break after the return type of function definitions.

```
class A {
  int
  f() {
    return 0;
  };
};
int f();
int
f() {
  return 1;
}
```

- `RTBS_TopLevelDefinitions` (in configuration: `TopLevelDefinitions`) Always break after the return type of top-level definitions.

```
class A {
  int f() { return 0; };
};
int f();
int
f() {
  return 1;
}
```

AlwaysBreakBeforeMultilineStrings (`bool`)

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

This flag is mean to make cases where there are multiple multiline strings in a file look more consistent. Thus, it will only take effect if wrapping the string at that point leads to it being indented `ContinuationIndentWidth` spaces from the start of the line.

```
true:                                   false:
aaaa =                          vs.     aaaa = "bbbb"
    "bbbb"                                     "cccc";
    "cccc";
```

AlwaysBreakTemplateDeclarations (`BreakTemplateDeclarationsStyle`)

The template declaration breaking style to use.

Possible values:

- `BTDS_No` (in configuration: `No`) Do not force break before declaration. `PenaltyBreakTemplateDeclaration` is taken into account.

```
template <typename T> T foo() {
}
template <typename T> T foo(int aaaaaaaaaaaaaaaaaaaa,
                            int bbbbbbbbbbbbbbbbbbbb) {
}
```

- `BTDS_MultiLine` (in configuration: `MultiLine`) Force break after template declaration only when the following declaration spans multiple lines.

```
template <typename T> T foo() {
}
template <typename T>
T foo(int aaaaaaaaaaaaaaaaaaaa,
      int bbbbbbbbbbbbbbbbbbbb) {
}
```

- `BTDS_Yes` (in configuration: `Yes`) Always break after template declaration.

```
template <typename T>
T foo() {
}
template <typename T>
T foo(int aaaaaaaaaaaaaaaaaaaa,
      int bbbbbbbbbbbbbbbbbbbb) {
}
```

BinPackArguments (`bool`)

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

```
true:
void f() {
  f(aaaaaaaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaa,
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa);
}

false:
void f() {
  f(aaaaaaaaaaaaaaaaaaaa,
    aaaaaaaaaaaaaaaaaaaa,
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa);
}
```

InsertTrailingCommas (`TrailingCommaStyle`) can be set to `TCS_Wrapped`

to insert trailing commas in container literals (arrays and objects) that wrap across multiple lines. It is currently only available for JavaScript and disabled by default (`TCS_None`).

`InsertTrailingCommas` cannot be used together with `BinPackArguments` as inserting the comma disables bin-packing.

```
TSC_Wrapped:
const someArray = [
  aaaaaaaaaaaaaaaaaaaaaaaaaa,
  aaaaaaaaaaaaaaaaaaaaaaaaaa,
  aaaaaaaaaaaaaaaaaaaaaaaaaa,
  //                        ^ inserted
]
```

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.

```
true:
void f(int aaaaaaaaaaaaaaaaaaa, int aaaaaaaaaaaaaaaaaaa,
       int aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa) {}

false:
void f(int aaaaaaaaaaaaaaaaaaa,
       int aaaaaaaaaaaaaaaaaaa,
       int aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa) {}
```

BraceWrapping (`BraceWrappingFlags`)

Control of individual brace wrapping cases.

If `BreakBeforeBraces` is set to `BS_Custom`, use this to specify how each individual brace case should be handled. Otherwise, this is ignored.

```
# Example of usage:
BreakBeforeBraces: Custom
BraceWrapping:
  AfterEnum: true
  AfterStruct: false
  SplitEmptyFunction: false
```

Nested configuration flags:

- `bool AfterCaseLabel` Wrap case labels.

```
false:                                  true:
switch (foo) {                  vs.     switch (foo) {
  case 1: {                               case 1:
    bar();                                {
    break;                                  bar();
  }                                         break;
  default: {                            }
    plop();                             default:
  }                                     {
}                                         plop();
                                        }
                                      }
```

- `bool AfterClass` Wrap class definitions.

```
true:
class foo {};

false:
class foo
{};
```

- `BraceWrappingAfterControlStatementStyle AfterControlStatement` Wrap control statements (`if`/`for`/`while`/`switch`/..).

  Possible values:

  - `BWACS_Never` (in configuration: `Never`) Never wrap braces after a control statement.

```
if (foo()) {
} else {
}
for (int i = 0; i < 10; ++i) {
}
```

  - `BWACS_MultiLine` (in configuration: `MultiLine`) Only wrap braces after a multi-line control statement.

```
if (foo && bar &&
    baz)
{
  quux();
}
while (foo || bar) {
}
```

  - `BWACS_Always` (in configuration: `Always`) Always wrap braces after a control statement.

```
if (foo())
{
} else
{}
for (int i = 0; i < 10; ++i)
{}
```

- `bool AfterEnum` Wrap enum definitions.

```
true:
enum X : int
{
  B
};

false:
enum X : int { B };
```

- `bool AfterFunction` Wrap function definitions.

```
true:
void foo()
{
  bar();
  bar2();
}

false:
void foo() {
  bar();
  bar2();
}
```

- `bool AfterNamespace` Wrap namespace definitions.

```
true:
namespace
{
int foo();
int bar();
}

false:
namespace {
int foo();
int bar();
}
```

- `bool AfterObjCDeclaration` Wrap ObjC definitions (interfaces, implementations...). @autoreleasepool and @synchronized blocks are wrapped according to *AfterControlStatement* flag.

- `bool AfterStruct` Wrap struct definitions.

```
true:
struct foo
{
  int x;
};

false:
struct foo {
  int x;
};
```

- `bool AfterUnion` Wrap union definitions.

```
true:
union foo
{
  int x;
}

false:
union foo {
```

```
  int x;
}
```

- `bool AfterExternBlock` Wrap extern blocks.

```
true:
extern "C"
{
  int foo();
}

false:
extern "C" {
int foo();
}
```

- `bool BeforeCatch` Wrap before `catch`.

```
true:
try {
  foo();
}
catch () {
}

false:
try {
  foo();
} catch () {
}
```

- `bool BeforeElse` Wrap before `else`.

```
true:
if (foo()) {
}
else {
}

false:
if (foo()) {
} else {
}
```

- `bool BeforeLambdaBody` Wrap lambda block.

```
true:
connect(
  []()
  {
    foo();
    bar();
  });

false:
connect([]() {
  foo();
  bar();
});
```

- `bool IndentBraces` Indent the wrapped braces themselves.

- `bool SplitEmptyFunction` If `false`, empty function body can be put on a single line. This option is used only if the opening brace of the function has already been wrapped, i.e. the *AfterFunction* brace wrapping mode is set, and the function could/should not be put on a single line (as per *AllowShortFunctionsOnASingleLine* and constructor formatting options).

```
int f()   vs.   int f()
{}              {
                }
```

- `bool SplitEmptyRecord` If `false`, empty record (e.g. class, struct or union) body can be put on a single line. This option is used only if the opening brace of the record has already been wrapped, i.e. the *AfterClass* (for classes) brace wrapping mode is set.

```
class Foo   vs.  class Foo
{}               {
                 }
```

- `bool SplitEmptyNamespace` If `false`, empty namespace body can be put on a single line. This option is used only if the opening brace of the namespace has already been wrapped, i.e. the *AfterNamespace* brace wrapping mode is set.

```
namespace Foo   vs.  namespace Foo
{}                   {
                     }
```

BreakAfterJavaFieldAnnotations (`bool`)

Break after each annotation on a field in Java files.

```
true:                                false:
@Partial                     vs.     @Partial @Mock DataLoad loader;
@Mock
DataLoad loader;
```

BreakBeforeBinaryOperators (`BinaryOperatorStyle`)

The way to wrap binary operators.

Possible values:

- `BOS_None` (in configuration: `None`) Break after operators.

```
LooooooooooongType loooooooooooooooooooooongVariable =
    someLooooooooooooooooongFunction();

bool value = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa +
                     aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa ==
                 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa &&
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa >
                 ccccccccccccccccccccccccccccccccccccccc;
```

- `BOS_NonAssignment` (in configuration: `NonAssignment`) Break before operators that aren't assignments.

```
LooooooooooongType loooooooooooooooooooooongVariable =
    someLooooooooooooooooongFunction();

bool value = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                     + aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                 == aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             && aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                    > ccccccccccccccccccccccccccccccccccccccc;
```

- `BOS_All` (in configuration: `All`) Break before operators.

```
LooooooooooongType loooooooooooooooooooooongVariable
    = someLooooooooooooooooongFunction();

bool value = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                     + aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                 == aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             && aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                    > ccccccccccccccccccccccccccccccccccccccc;
```

BreakBeforeBraces (`BraceBreakingStyle`)

The brace breaking style to use.

Possible values:

- `BS_Attach` (in configuration: `Attach`) Always attach braces to surrounding context.

```
try {
  foo();
} catch () {
}
void foo() { bar(); }
class foo {};
```

```
if (foo()) {
} else {
}
enum X : int { A, B };
```

- `BS_Linux` (in configuration: `Linux`) Like `Attach`, but break before braces on function, namespace and class definitions.

```
try {
  foo();
} catch () {
}
void foo() { bar(); }
class foo
{
};
if (foo()) {
} else {
}
enum X : int { A, B };
```

- `BS_Mozilla` (in configuration: `Mozilla`) Like `Attach`, but break before braces on enum, function, and record definitions.

```
try {
  foo();
} catch () {
}
void foo() { bar(); }
class foo
{
};
if (foo()) {
} else {
}
enum X : int { A, B };
```

- `BS_Stroustrup` (in configuration: `Stroustrup`) Like `Attach`, but break before function definitions, `catch`, and `else`.

```
try {
  foo();
}
catch () {
}
void foo() { bar(); }
class foo {
};
if (foo()) {
}
else {
}
enum X : int { A, B };
```

- `BS_Allman` (in configuration: `Allman`) Always break before braces.

```
try
{
  foo();
}
catch ()
{
}
void foo() { bar(); }
class foo
{
};
if (foo())
{
}
else
{
}
enum X : int
{
  A,
  B
};
```

- `BS_Whitesmiths` (in configuration: `Whitesmiths`) Like `Allman` but always indent braces and line up code with braces.

```
try
  {
  foo();
  }
catch ()
  {
  }
void foo() { bar(); }
class foo
  {
  };
if (foo())
  {
  }
else
  {
  }
enum X : int
  {
  A,
  B
  };
```

- `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.

```
try
  {
    foo();
  }
catch ()
  {
  }
void foo() { bar(); }
class foo
{
};
if (foo())
  {
  }
else
  {
  }
enum X : int
{
  A,
  B
};
```

- `BS_WebKit` (in configuration: `WebKit`) Like `Attach`, but break before functions.

```
try {
  foo();
} catch () {
}
void foo() { bar(); }
class foo {
};
if (foo()) {
} else {
}
enum X : int { A, B };
```

- `BS_Custom` (in configuration: `Custom`) Configure each individual brace in *BraceWrapping*.

BreakBeforeTernaryOperators (`bool`)

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

```
true:
veryVeryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongDescription
    ? firstValue
    : SecondValueVeryVeryVeryVeryLong;

false:
veryVeryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongDescription ?
```

```
    firstValue :
    SecondValueVeryVeryVeryVeryLong;
```

BreakConstructorInitializers (`BreakConstructorInitializersStyle`)

The constructor initializers style to use.

Possible values:

- `BCIS_BeforeColon` (in configuration: `BeforeColon`) Break constructor initializers before the colon and after the commas.

```
Constructor()
    : initializer1(),
      initializer2()
```

- `BCIS_BeforeComma` (in configuration: `BeforeComma`) Break constructor initializers before the colon and commas, and align the commas with the colon.

```
Constructor()
    : initializer1()
    , initializer2()
```

- `BCIS_AfterColon` (in configuration: `AfterColon`) Break constructor initializers after the colon and commas.

```
Constructor() :
    initializer1(),
    initializer2()
```

BreakInheritanceList (`BreakInheritanceListStyle`)

The inheritance list style to use.

Possible values:

- `BILS_BeforeColon` (in configuration: `BeforeColon`) Break inheritance list before the colon and after the commas.

```
class Foo
    : Base1,
      Base2
{};
```

- `BILS_BeforeComma` (in configuration: `BeforeComma`) Break inheritance list before the colon and commas, and align the commas with the colon.

```
class Foo
    : Base1
    , Base2
{};
```

- `BILS_AfterColon` (in configuration: `AfterColon`) Break inheritance list after the colon and commas.

```
class Foo :
    Base1,
    Base2
{};
```

BreakStringLiterals (`bool`)

Allow breaking string literals when formatting.

```
true:
const char* x = "veryVeryVeryVeryVeryVe"
                "ryVeryVeryVeryVeryVery"
                "VeryLongString";

false:
const char* x =
  "veryVeryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongString";
```

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.

```
// CommentPragmas: '^ FOOBAR pragma:'
// Will leave the following line unaffected
#include <vector> // FOOBAR pragma: keep
```

CompactNamespaces (`bool`)

If `true`, consecutive namespace declarations will be on the same line. If `false`, each namespace is declared on a new line.

```
true:
namespace Foo { namespace Bar {
}}

false:
namespace Foo {
namespace Bar {
}
}
```

If it does not fit on a single line, the overflowing namespaces get wrapped:

```
namespace Foo { namespace Bar {
namespace Extra {
}}}
```

ConstructorInitializerAllOnOneLineOrOnePerLine (`bool`)

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

```
true:
SomeClass::Constructor()
    : aaaaaaaa(aaaaaaaa), aaaaaaaa(aaaaaaaa), aaaaaaaa(aaaaaaaaaaaaaaaaaaaaaaaa) {
  return 0;
}

false:
SomeClass::Constructor()
    : aaaaaaaa(aaaaaaaa), aaaaaaaa(aaaaaaaa),
      aaaaaaaa(aaaaaaaaaaaaaaaaaaaaaaaa) {
  return 0;
}
```

ConstructorInitializerIndentWidth (`unsigned`)

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

ContinuationIndentWidth (`unsigned`)

Indent width for line continuations.

```
ContinuationIndentWidth: 2

int i =         //  VeryVeryVeryVeryVeryLongComment
  longFunction( // Again a long comment
    arg);
```

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.

```
true:                                  false:
vector<int> x{1, 2, 3, 4};     vs.     vector<int> x{ 1, 2, 3, 4 };
vector<T> x{{}, {}, {}, {}};           vector<T> x{ {}, {}, {}, {} };
f(MyMap[{composite, key}]);            f(MyMap[{ composite, key }]);
new int[3]{1, 2, 3};                   new int[3]{ 1, 2, 3 };
```

DeriveLineEnding (`bool`)

Analyze the formatted file for the most used line ending (`\r\n` or `\n`). `UseCRLF` is only used as a fallback if none can be derived.

DerivePointerAlignment (`bool`)

If `true`, analyze the formatted file for the most common alignment of `&` and `*`. Pointer and reference alignment styles are going to be updated according to the preferences found in the file. `PointerAlignment` is then used only as fallback.

DisableFormat (`bool`)

Disables formatting completely.

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.

FixNamespaceComments (`bool`)

If `true`, clang-format adds missing namespace end comments and fixes invalid existing ones.

```
true:                                  false:
namespace a {                  vs.     namespace a {
foo();                                 foo();
} // namespace a                       }
```

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:

```
FOREACH(<variable-declaration>, ...)
  <loop-body>
```

In the .clang-format configuration file, this can be configured like:

```
ForEachMacros: ['RANGES_FOR', 'FOREACH']
```

For example: BOOST_FOREACH.

IncludeBlocks (`IncludeBlocksStyle`)

Dependent on the value, multiple `#include` blocks can be sorted as one and divided based on category.

Possible values:

- `IBS_Preserve` (in configuration: `Preserve`) Sort each `#include` block separately.

  ```
  #include "b.h"               into      #include "b.h"

  #include <lib/main.h>                  #include "a.h"
  #include "a.h"                         #include <lib/main.h>
  ```

- `IBS_Merge` (in configuration: `Merge`) Merge multiple `#include` blocks together and sort as one.

```
#include "b.h"               into      #include "a.h"
                                       #include "b.h"
#include <lib/main.h>                  #include <lib/main.h>
#include "a.h"
```

- `IBS_Regroup` (in configuration: `Regroup`) Merge multiple `#include` blocks together and sort as one. Then split into groups based on category priority. See `IncludeCategories`.

```
#include "b.h"               into      #include "a.h"
                                       #include "b.h"
#include <lib/main.h>
#include "a.h"                         #include <lib/main.h>
```

IncludeCategories (`std::vector<IncludeCategory>`)

Regular expressions denoting the different `#include` categories used for ordering `#includes`.

POSIX extended regular expressions are supported.

These regular expressions are matched against the filename of an include (including the <> or “”) in order. The value belonging to the first matching regular expression is assigned and `#includes` are sorted first according to increasing category number and then alphabetically within each category.

If none of the regular expressions match, INT_MAX is assigned as category. The main header for a source file automatically gets category 0. so that it is generally kept at the beginning of the `#includes` (**https://llvm.org/docs/CodingStandards.html#include-style**). However, you can also assign negative priorities if you have certain headers that always need to be first.

There is a third and optional field `SortPriority` which can used while `IncludeBloks = IBS_Regroup` to define the priority in which `#includes` should be ordered, and value of `Priority` defines the order of `#include blocks` and also enables to group `#includes` of different priority for order.``SortPriority`` is set to the value of `Priority` as default if it is not assigned.

To configure this in the .clang-format file, use:

```
IncludeCategories:
  - Regex:           '^"(llvm|llvm-c|clang|clang-c)/'
    Priority:        2
    SortPriority:    2
  - Regex:           '^(<|"(gtest|gmock|isl|json)/)'
    Priority:        3
  - Regex:           '<[[:alnum:].]+>'
    Priority:        4
  - Regex:           '.*'
    Priority:        1
    SortPriority:    0
```

IncludeIsMainRegex (`std::string`)

Specify a regular expression of suffixes that are allowed in the file-to-main-include mapping.

When guessing whether a #include is the “main” include (to assign category 0, see above), use this regex of allowed suffixes to the header stem. A partial match is done, so that: - “” means “arbitrary suffix” - “$” means “no suffix”

For example, if configured to “(_test)?$”, then a header a.h would be seen as the “main” include in both a.cc and a_test.cc.

IncludeIsMainSourceRegex (`std::string`)

Specify a regular expression for files being formatted that are allowed to be considered “main” in the file-to-main-include mapping.

By default, clang-format considers files as “main” only when they end with: `.c`, `.cc`, `.cpp`, `.c++`, `.cxx`, `.m` or `.mm` extensions. For these files a guessing of “main” include takes place (to assign category 0, see above). This config option allows for additional suffixes and extensions for files to be considered as “main”.

For example, if this option is configured to `(Impl\.hpp)$`, then a file `ClassImpl.hpp` is considered “main” (in addition to `Class.c`, `Class.cc`, `Class.cpp` and so on) and “main include file” logic will be executed (with *IncludeIsMainRegex* setting also being respected in later phase). Without this option set, `ClassImpl.hpp` would not have the main include file put on top before any other include.

IndentCaseBlocks (`bool`)

Indent case label blocks one level from the case label.

When `false`, the block following the case label uses the same indentation level as for the case label, treating the case label the same as an if-statement. When `true`, the block gets indented as a scope block.

```
false:                                  true:
switch (fool) {                vs.      switch (fool) {
case 1: {                               case 1:
  bar();                                  {
} break;                                    bar();
default: {                                  }
  plop();                                   break;
}                                       default:
}                                         {
                                            plop();
                                          }
                                        }
```

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 (except the first block following the case label, which itself indents the code - unless IndentCaseBlocks is enabled).

```
false:                                  true:
switch (fool) {                vs.      switch (fool) {
case 1:                                   case 1:
  bar();                                    bar();
  break;                                    break;
default:                                  default:
  plop();                                   plop();
}                                       }
```

IndentGotoLabels (`bool`)

Indent goto labels.

When `false`, goto labels are flushed left.

```
true:                                   false:
int f() {                      vs.      int f() {
  if (foo()) {                            if (foo()) {
  label1:                                 label1:
    bar();                                  bar();
  }                                       }
label2:                                 label2:
  return 1;                               return 1;
}                                       }
```

IndentPPDirectives (`PPDirectiveIndentStyle`)

The preprocessor directive indenting style to use.

Possible values:

- `PPDIS_None` (in configuration: `None`) Does not indent any directives.

```
#if FOO
#if BAR
#include <foo>
#endif
#endif
```

- `PPDIS_AfterHash` (in configuration: `AfterHash`) Indents directives after the hash.

```
#if FOO
#  if BAR
```

```
#    include <foo>
#  endif
#endif
```

- `PPDIS_BeforeHash` (in configuration: `BeforeHash`) Indents directives before the hash.

```
#if FOO
  #if BAR
    #include <foo>
  #endif
#endif
```

IndentWidth (`unsigned`)

The number of columns to use for indentation.

```
IndentWidth: 3

void f() {
   someFunction();
   if (true, false) {
      f();
   }
}
```

IndentWrappedFunctionNames (`bool`)

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

```
true:
LooooooooooooooooooooooooooooooooooooooongReturnType
    LooooooooooooooooooooooooooooooongFunctionDeclaration();

false:
LooooooooooooooooooooooooooooooooooooooongReturnType
LooooooooooooooooooooooooooooooongFunctionDeclaration();
```

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

A vector of prefixes ordered by the desired groups for Java imports.

Each group is separated by a newline. Static imports will also follow the same grouping convention above all non-static imports. One group's prefix can be a subset of another - the longest prefix is always matched. Within a group, the imports are ordered lexicographically.

In the .clang-format configuration file, this can be configured like in the following yaml example. This will result in imports being formatted as in the Java example below.

```
JavaImportGroups: ['com.example', 'com', 'org']
```

```
import static com.example.function1;

import static com.test.function2;

import static org.example.function3;

import com.example.ClassA;
import com.example.Test;
import com.example.a.ClassB;

import com.test.ClassC;

import org.example.ClassD;
```

JavaScriptQuotes (`JavaScriptQuoteStyle`)

The JavaScriptQuoteStyle to use for JavaScript strings.

Possible values:

- `JSQS_Leave` (in configuration: `Leave`) Leave string quotes as they are.

```
string1 = "foo";
string2 = 'bar';
```

- `JSQS_Single` (in configuration: `Single`) Always use single quotes.

```
string1 = 'foo';
string2 = 'bar';
```

- `JSQS_Double` (in configuration: `Double`) Always use double quotes.

```
string1 = "foo";
string2 = "bar";
```

JavaScriptWrapImports (`bool`)

Whether to wrap JavaScript import/export statements.

```
true:
import {
    VeryLongImportsAreAnnoying,
    VeryLongImportsAreAnnoying,
    VeryLongImportsAreAnnoying,
} from 'some/module.js'

false:
import {VeryLongImportsAreAnnoying, VeryLongImportsAreAnnoying, VeryLongImportsAreAnnoying,} from "some/module.js"
```

KeepEmptyLinesAtTheStartOfBlocks (`bool`)

If true, the empty line at the start of blocks is kept.

```
true:                                false:
if (foo) {                    vs.    if (foo) {
                                       bar();
  bar();                               }
}
```

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++.
- `LK_CSharp` (in configuration: `CSharp`) Should be used for C#.
- `LK_Java` (in configuration: `Java`) Should be used for Java.
- `LK_JavaScript` (in configuration: `JavaScript`) Should be used for JavaScript.
- `LK_ObjC` (in configuration: `ObjC`) Should be used for Objective-C, Objective-C++.
- `LK_Proto` (in configuration: `Proto`) Should be used for Protocol Buffers (**https://developers.google.com/protocol-buffers/**).
- `LK_TableGen` (in configuration: `TableGen`) Should be used for TableGen code.
- `LK_TextProto` (in configuration: `TextProto`) Should be used for Protocol Buffer messages in text format (**https://developers.google.com/protocol-buffers/**).

MacroBlockBegin (`std::string`)

A regular expression matching macros that start a block.

```
# With:
MacroBlockBegin: "^NS_MAP_BEGIN|\
NS_TABLE_HEAD$"
MacroBlockEnd: "^\
NS_MAP_END|\
NS_TABLE_.*_END$"

NS_MAP_BEGIN
```

```
  foo();
NS_MAP_END

NS_TABLE_HEAD
  bar();
NS_TABLE_FOO_END

# Without:
NS_MAP_BEGIN
foo();
NS_MAP_END

NS_TABLE_HEAD
bar();
NS_TABLE_FOO_END
```

MacroBlockEnd (`std::string`)

A regular expression matching macros that end a block.

MaxEmptyLinesToKeep (`unsigned`)

The maximum number of consecutive empty lines to keep.

```
MaxEmptyLinesToKeep: 1         vs.     MaxEmptyLinesToKeep: 0
int f() {                              int f() {
  int = 1;                                 int i = 1;
                                           i = foo();
  i = foo();                               return i;
                                       }
  return i;
}
```

NamespaceIndentation (`NamespaceIndentationKind`)

The indentation used for namespaces.

Possible values:

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

```
namespace out {
int i;
namespace in {
int i;
}
}
```

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

```
namespace out {
int i;
namespace in {
  int i;
}
}
```

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

```
namespace out {
  int i;
  namespace in {
    int i;
  }
}
```

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

A vector of macros which are used to open namespace blocks.

These are expected to be macros of the form:

```
NAMESPACE(<namespace-name>, ...) {
  <namespace-content>
}
```

For example: TESTSUITE

ObjCBinPackProtocolList (`BinPackStyle`)

Controls bin-packing Objective-C protocol conformance list items into as few lines as possible when they go over `ColumnLimit`.

If `Auto` (the default), delegates to the value in `BinPackParameters`. If that is `true`, bin-packs Objective-C protocol conformance list items into as few lines as possible whenever they go over `ColumnLimit`.

If `Always`, always bin-packs Objective-C protocol conformance list items into as few lines as possible whenever they go over `ColumnLimit`.

If `Never`, lays out Objective-C protocol conformance list items onto individual lines whenever they go over `ColumnLimit`.

```
Always (or Auto, if BinPackParameters=true):
@interface ccccccccccccc () <
    ccccccccccccc, ccccccccccccc,
    ccccccccccccc, ccccccccccccc> {
}

Never (or Auto, if BinPackParameters=false):
@interface ddddddddddddd () <
    ddddddddddddd,
    ddddddddddddd,
    ddddddddddddd,
    ddddddddddddd> {
}
```

Possible values:

- `BPS_Auto` (in configuration: `Auto`) Automatically determine parameter bin-packing behavior.
- `BPS_Always` (in configuration: `Always`) Always bin-pack parameters.
- `BPS_Never` (in configuration: `Never`) Never bin-pack parameters.

ObjCBlockIndentWidth (`unsigned`)

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

```
ObjCBlockIndentWidth: 4

[operation setCompletionBlock:^{
    [self onOperationDone];
}];
```

ObjCBreakBeforeNestedBlockParam (`bool`)

Break parameters list into lines when there is nested block parameters in a fuction call.

```
false:
 - (void)_aMethod
 {
     [self.test1 t:self w:self callback:^(typeof(self) self, NSNumber *u, NSNumber *v) {
         u = c;
     }]
 }
 true:
 - (void)_aMethod
 {
    [self.test1 t:self
                w:self
         callback:^(typeof(self) self, NSNumber *u, NSNumber *v) {
             u = c;
         }]
 }
```

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>`.

PenaltyBreakAssignment (`unsigned`)

The penalty for breaking around an assignment operator.

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.

PenaltyBreakTemplateDeclaration (`unsigned`)

The penalty for breaking after template declaration.

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.

```
int* a;
```

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

```
int *a;
```

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

```
int * a;
```

RawStringFormats (`std::vector<RawStringFormat>`)

Defines hints for detecting supported languages code blocks in raw strings.

A raw string with a matching delimiter or a matching enclosing function name will be reformatted assuming the specified language based on the style for that language defined in the .clang-format file. If no style has been defined in the .clang-format file for the specific language, a predefined style given by 'BasedOnStyle' is used. If 'BasedOnStyle' is not found, the formatting is based on llvm style. A matching delimiter takes precedence over a matching enclosing function name for determining the language of the raw string contents.

If a canonical delimiter is specified, occurrences of other delimiters for the same language will be updated to the canonical if possible.

There should be at most one specification per language and each delimiter and enclosing function should not occur in multiple specifications.

To configure this in the .clang-format file, use:

```
RawStringFormats:
  - Language: TextProto
      Delimiters:
```

```
        - 'pb'
        - 'proto'
      EnclosingFunctions:
        - 'PARSE_TEXT_PROTO'
      BasedOnStyle: google
  - Language: Cpp
      Delimiters:
        - 'cc'
        - 'cpp'
      BasedOnStyle: llvm
      CanonicalDelimiter: 'cc'
```

ReflowComments (`bool`)

If `true`, clang-format will attempt to re-flow comments.

```
false:
// veryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongComment with plenty of information
/* second veryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongComment with plenty of information */

true:
// veryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongComment with plenty of
// information
/* second veryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongComment with plenty of
 * information */
```

SortIncludes (`bool`)

If `true`, clang-format will sort `#includes`.

```
false:                                  true:
#include "b.h"                 vs.     #include "a.h"
#include "a.h"                          #include "b.h"
```

SortUsingDeclarations (`bool`)

If `true`, clang-format will sort using declarations.

The order of using declarations is defined as follows: Split the strings by "::" and discard any initial empty strings. The last element of each list is a non-namespace name; all others are namespace names. Sort the lists of names lexicographically, where the sort order of individual names is that all non-namespace names come before all namespace names, and within those groups, names are in case-insensitive lexicographic order.

```
false:                                  true:
using std::cout;               vs.      using std::cin;
using std::cin;                         using std::cout;
```

SpaceAfterCStyleCast (`bool`)

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

```
true:                                   false:
(int) i;                       vs.      (int)i;
```

SpaceAfterLogicalNot (`bool`)

If `true`, a space is inserted after the logical not operator (`!`).

```
true:                                   false:
! someExpression();           vs.      !someExpression();
```

SpaceAfterTemplateKeyword (`bool`)

If `true`, a space will be inserted after the 'template' keyword.

```
true:                                   false:
template <int> void foo();     vs.      template<int> void foo();
```

SpaceBeforeAssignmentOperators (`bool`)

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

```
true:                                   false:
int a = 5;                     vs.      int a= 5;
a += 42;                                a+= 42;
```

SpaceBeforeCpp11BracedList (`bool`)

If `true`, a space will be inserted before a C++11 braced list used to initialize an object (after the preceding identifier or type).

```
true:                                   false:
Foo foo { bar };               vs.      Foo foo{ bar };
Foo {};                                 Foo{};
vector<int> { 1, 2, 3 };                vector<int>{ 1, 2, 3 };
new int[3] { 1, 2, 3 };                 new int[3]{ 1, 2, 3 };
```

SpaceBeforeCtorInitializerColon (`bool`)

If `false`, spaces will be removed before constructor initializer colon.

```
true:                                   false:
Foo::Foo() : a(a) {}                    Foo::Foo(): a(a) {}
```

SpaceBeforeInheritanceColon (`bool`)

If `false`, spaces will be removed before inheritance colon.

```
true:                                   false:
class Foo : Bar {}             vs.      class Foo: Bar {}
```

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.

```
void f() {
  if(true) {
    f();
  }
}
```

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

```
void f() {
  if (true) {
    f();
  }
}
```

- `SBPO_NonEmptyParentheses` (in configuration: `NonEmptyParentheses`) Put a space before opening parentheses only if the parentheses are not empty i.e. '()'

```
void() {
  if (true) {
    f();
    g (x, y, z);
  }
}
```

- `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.)

```
void f () {
  if (true) {
    f ();
```

```
  }
}
```

SpaceBeforeRangeBasedForLoopColon (`bool`)

If `false`, spaces will be removed before range-based for loop colon.

```
true:                                   false:
for (auto v : values) {}       vs.      for(auto v: values) {}
```

SpaceBeforeSquareBrackets (`bool`)

If `true`, spaces will be before `[`. Lambdas will not be affected. Only the first `[` will get a space added.

```
true:                                   false:
int a [5];                    vs.       int a[5];
int a [5][5];                 vs.       int a[5][5];
```

SpaceInEmptyBlock (`bool`)

If `true`, spaces will be inserted into `{}`.

```
true:                                   false:
void f() { }                   vs.      void f() {}
while (true) { }                        while (true) {}
```

SpaceInEmptyParentheses (`bool`)

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

```
true:                                   false:
void f( ) {                    vs.      void f() {
  int x[] = {foo( ), bar( )};             int x[] = {foo(), bar()};
  if (true) {                             if (true) {
    f( );                                   f();
  }                                       }
}                                       }
```

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.

```
SpacesBeforeTrailingComments: 3
void f() {
  if (true) {   // foo1
    f();        // bar
  }             // foo
}
```

SpacesInAngles (`bool`)

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

```
true:                                   false:
static_cast< int >(arg);       vs.      static_cast<int>(arg);
std::function< void(int) > fct;         std::function<void(int)> fct;
```

SpacesInCStyleCastParentheses (`bool`)

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

```
true:                                   false:
x = ( int32 )y                 vs.      x = (int32)y
```

SpacesInConditionalStatement (`bool`)

If `true`, spaces will be inserted around if/for/switch/while conditions.

```
true:                                   false:
if ( a )  { ... }             vs.       if (a) { ... }
while ( i < 5 )  { ... }                while (i < 5) { ... }
```

SpacesInContainerLiterals (`bool`)

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

```
true:                                   false:
var arr = [ 1, 2, 3 ];          vs.     var arr = [1, 2, 3];
f({a : 1, b : 2, c : 3});               f({a: 1, b: 2, c: 3});
```

SpacesInParentheses (`bool`)

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

```
true:                                   false:
t f( Deleted & ) & = delete;    vs.     t f(Deleted &) & = delete;
```

SpacesInSquareBrackets (`bool`)

If `true`, spaces will be inserted after `[` and before `]`. Lambdas without arguments or unspecified size array declarations will not be affected.

```
true:                                   false:
int a[ 5 ];                     vs.     int a[5];
std::unique_ptr<int[]> foo() {} // Won't be affected
```

Standard (`LanguageStandard`)

Parse and format C++ constructs compatible with this standard.

```
c++03:                                  latest:
vector<set<int> > x;            vs.     vector<set<int>> x;
```

Possible values:

- `LS_Cpp03` (in configuration: `c++03`) Parse and format as C++03. `Cpp03` is a deprecated alias for `c++03`
- `LS_Cpp11` (in configuration: `c++11`) Parse and format as C++11.
- `LS_Cpp14` (in configuration: `c++14`) Parse and format as C++14.
- `LS_Cpp17` (in configuration: `c++17`) Parse and format as C++17.
- `LS_Cpp20` (in configuration: `c++20`) Parse and format as C++20.
- `LS_Latest` (in configuration: `Latest`) Parse and format using the latest supported language version. `Cpp11` is a deprecated alias for `Latest`
- `LS_Auto` (in configuration: `Auto`) Automatic detection based on the input.

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

A vector of macros that should be interpreted as complete statements.

Typical macros are expressions, and require a semi-colon to be added; sometimes this is not the case, and this allows to make clang-format aware of such cases.

For example: Q_UNUSED

TabWidth (`unsigned`)

The number of columns used for tab stops.

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

A vector of macros that should be interpreted as type declarations instead of as function calls.

These are expected to be macros of the form:

```
STACK_OF(...)
```

In the .clang-format configuration file, this can be configured like:

```
TypenameMacros: ['STACK_OF', 'LIST']
```

For example: OpenSSL STACK_OF, BSD LIST_ENTRY.

UseCRLF (`bool`)

Use `\r\n` instead of `\n` for line breaks. Also used as fallback if `DeriveLineEnding` is true.

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_ForContinuationAndIndentation` (in configuration: `ForContinuationAndIndentation`) Use tabs only for line continuation and 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.

Adding additional style options

Each additional style option adds costs to the clang-format project. Some of these costs affect the clang-format development itself, as we need to make sure that any given combination of options work and that new features don't break any of the existing options in any way. There are also costs for end users as options become less discoverable and people have to think about and make a decision on options they don't really care about.

The goal of the clang-format project is more on the side of supporting a limited set of styles really well as opposed to supporting every single style used by a codebase somewhere in the wild. Of course, we do want to support all major projects and thus have established the following bar for adding style options. Each new style option must ..

- be used in a project of significant size (have dozens of contributors)
- have a publicly accessible style guide
- have a person willing to contribute and maintain patches

Examples ".clang-format" file 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();
    }
}
```