Artistic Style 3.1

A Free, Fast, and Small Automatic Formatter
for C, C++, C++/CLI, Objective-C, C#, and Java Source Code

Contents

General Information

Line Endings

Line endings in the formatted file will be the same as the input file. If there are mixed line endings the most frequent occurrence will be used. There is also an option to specify or change the line endings.

File Type

Top

Artistic Style will determine the file type from the file extension. The extension ".java" indicates a Java file, and ".cs" indicates a C# file. Everything else is a C type file (C, C++, C++/CLI, or Objective-C). If

you are using a non-standard file extension for Java or C#, use one of the --mode= options.

Wildcards and Recursion

Artistic Style can process directories recursively. Wildcards (such as "*.cpp" or "*.c??") are processed internally. If a shell is used, it should pass the wildcards to Artistic Style instead of resolving them first. For Linux use double quotes around paths whose file name contains wildcards. For Windows use double quotes around paths whose file name contains spaces. The recursive option in the Other Options section contains information on recursive processing.

File Names

When a file is formatted, the newly indented file retains the original file name. A copy of the original file is created with an **.orig** appended to the original file name. (This can be set to a different string by the option –suffix=, or suppressed altogether by the options -n or --suffix=none). Thus, after indenting *SourceFile.cpp* the indented file will be named *SourceFile.cpp*, while the original pre-indented file will be renamed to *SourceFile.cpp.orig*.

Internationalization

Artistic Style has been internationalized to process files and directories in any language.

It has also been translated into several languages. The translation to use is determined by the User Locale for Windows and the LANG environment variable for other systems. The translation will be done automatically from these settings. If no translation is available it will default to English. There is an "ascii" option to use English instead of the system language.

The source code for the translations is at the end of ASLocalizer.cpp in the form of an English-Translation pair. If you make corrections to a translation, send the source as a bug report and it will be included in the next release.

To add a new language, add a new translation class to ASLocalizer.h. Add the English-Translation pair to the constructor in ASLocalizer.cpp. Update the WinLangCode array and add the language code to the function setTranslationClass(). The ASLocalizer.cpp program contains comments that give web pages for obtaining the LCIDs and language codes. Send the source code as a bug report and it will be included in the next release.

Other Considerations

The names of special characters used in programming vary by region. The terminology used by Artistic Style, followed by other common names, is**:**

braces or curly braces { } - also called brackets, or curly brackets.
parens or round brackets ( ) - also called parentheses, brackets, circle brackets, or soft brackets.
square brackets [ ] - also called block parens, brackets, closed brackets, or hard brackets.
angle brackets < > - also called brackets, pointy brackets, triangular brackets, diamond brackets, tuples, or chevrons.

Visual Studio, and possibly other development environments, has extensions that will align assignment operators across multiple lines. There is an extension named "Code alignment" that will align the code on other items as well. Formatting with these options and extensions can be used with Artistic Style. The space padding will be maintained and the alignment will be preserved.

Artistic Style can format standard class library statements such as Open GL, wxWidgets, Qt, and MFC.

Embedded assembler language is formatted correctly. This includes extended assembly and Microsoft specific assembler lines and blocks.

Artistic Style can format embedded SQL statements. The SQL formatting will be maintained as long as the standard hanging indent format is used. If the "exec sql" statement is indented more than the following statements, the SQL will be aligned in a single column.

Unicode files encoded as UTF-16, both big and little endian, will be formatted. The files must begin with a byte order mark (BOM) to be recognized. Files encoded as UTF-32 will be rejected. Some compilers do not support these encodings. These files can be converted to UTF-8 encoding with the program "iconv". There are Linux and Windows versions available (the Windows version does not seem to work for all encodings). Visual Studio can convert the files from the "File > Advanced Save Options" menu. There are other development environments and text editors, such as SciTE, that can convert files to UTF-8.

Top

Embedded statements that are multiple-line and are NOT in a C-type format, such as Python, are usually mal-formatted (a C-type format has blocks enclosed by braces and statements terminated by a semi-colon). Macros that define functions may cause the following code to be mal-formatted because the macro is missing the braces and semi-colons from the definition. If you have source code with these types of statements, exclude them with the exclude=#### option described in the Other Options section.

Quick Start

If you have never used Artistic Style, there are several of ways to get started.

One is to run it with no options at all. This will use the default brace style, 4 spaces per indent, and no formatting changes. This will break the braces for one line blocks and will break one line statements. To change this, use the option keep-one-line-blocks and/or keep-one-line-statements described in the Formatting Options section.

Another way is to use one of the brace styles described in the Brace Style Options section. Select one with a brace formatting style you like. If no indentation option is set, the default option of 4 spaces will be used. These options also break one line blocks and one line statements as described above.

A third option is to use an options file from the "file" folder. If there is a coding style you want to duplicate, input the appropriate option file. Use the option options=#### to specify the file to use. It must contain a path for the file, including the file name.

Once you are familiar with the options you can customize the format to your personal preference.

Usage

Command Line

Artistic style is a console program that receives information from the command line.

Command line format:

```
astyle [OPTIONS] SourceFilePath1 SourceFilePath2 SourceFilePath3 [ . . . ]
```

The square brackets [ ] indicate that more than one option or more than one file name can be entered. They are NOT actually included in the command. For the options format refer to the following Options section.

Example to format a single file:

```
astyle --style=allman /home/project/foo.cpp
```

Example to format C# files recursively:

```
astyle --style=allman --recursive /home/project/*.cs
```

File Extensions

Multiple file extensions may be used if separated by commas or semicolons. An optional space may follow if the entire file path is enclosed in double quotes. There is no limit to the number of extensions used.

Example to format C++ files recursively using multiple file extensions:

```
astyle --style=allman --recursive /home/project/*.cpp,*.h
```

Redirection

The < and > characters may be used to redirect the files into standard input (stdin) and out of standard output (stdout) - don't forget them! With this option, only one file at a time can be formatted. Wildcards are not recognized, there are no console messages, and a backup is not created. On Windows, the output will always have Windows line ends. The options "stdin=" and "stdout=" can be used instead of redirection.

Top

Example of redirection option to format a single file and change the name:

```
astyle --style=allman < OriginalSourceFile > BeautifiedSourceFile
```

Example of redirection using "stdin=" and "stdout=" to format a single file and change the name:

```
astyle --style=allman --stdin=OriginalSourceFile --stdout=BeautifiedSourceFile
```

The redirection option may be used to display the formatted file without updating:

```
astyle --style=allman < OriginalSourceFile | less
```

Options

Not specifying any options will result in the default brace style, 4 spaces per indent, and no formatting changes.

This program follows the usual GNU command line syntax. Options may be written two different ways.

Long options

These options start with '**--**', and must be written one at a time.
(Example: '--style=allman --indent=spaces=4')

Short Options

These options start with a single '**-**', and may be concatenated together.
(Example: '-bps4' is the same as writing '-b -p -s4'.)

Option Files

An OPTIONAL default option file and/or project option file may be used to supplement or replace the command line options. They may use the computer's standard encoding, UTF-8 or UTF-16 unicode encoding.

Options may be set apart by new-lines, tabs, commas, or spaces. Long options in the option file may be written without the preceding '--'. Lines within the option file that begin with '#' are considered line-comments. The option files used in formatting and their location can be displayed by using the --verbose option.

1. The **command line options** have precedence. If there is a conflict between a command line option and an option in a default or project file, the command line option will be used.
2. The **project option file** has precedence over the default option file but not the command line options. The project option file should be in the top directory of the project being formatted. The file is identified by a file name only. One of the command line project options must be used to indicate a file is available, or it must be referred to by the environment variable. Artistic Style looks for the file in the current directory or one of its parent directories in the following order.
   - the file name indicated by the **--**project= command line option.
   - the file named .astylerc or _ astylerc.
   - the file name identified by the environment variable ARTISTIC_STYLE_PROJECT_OPTIONS if it exists.
   - the file or environment variable can be disabled by specifying --project=none on the command line.

   The file is expected to be in the top directory of the project being formatted. Only one file will be used per execution and all files to be formatted are assumed to be in the same project. Artistic Style will search backward in the directory path to find the project option file. The initial directory path for the search is obtained from one of the following locations in the following order.
   - The first *SourceFilePath* entered on the command line.
   - The value of "--stdin=" if it is used for redirection.
   - The current directory if "<" is used for rediredction. If the file to be formatted is not in the current directory, use the "–stdin=" option instead.
3. The **default option file** can be used for all projects. The file is identified by a file path and a file name. One of the command line options must be used to indicate a file is available, or it must be referred to by the environment variable. Artistic Style looks for a file path and file name in the following order.

Top

- the file path indicated by the --options= command line option.
- the file path indicated by the environment variable ARTISTIC_STYLE_OPTIONS if it exists.
- the file named .astylerc in the directory pointed to by the HOME environment variable (e.g. "$HOME/.astylerc" on Linux);
- the file named astylerc in the directory pointed to by the APPDATA environment variable (e.g. "%APPDATA%\astylerc" on Windows).
- the file or environment variable can be disabled by specifying --options=none on the command line.

Example of a default or project option file:

```
# this line is a comment
--style=allman      # this is a line-end comment
# long options can be written without the preceding '--'
indent-switches     # cannot do this on the command line
# short options must have the preceding '-'
-t -p
# short options can be concatenated together
-M60Ucv
```

Disable Formatting

Formatting and indenting can be disabled with comment tags inserted in the source code.

Disable Block

Blocks of code can be disabled using "off" and "on" tags. The tags are included in the source file as comments. The comment may be a C comment (/* ... */) or a C++ line comment (//). The tag must be included in a single line comment. If the comment exceeds one line the indent tag will be ignored. Additional information can be included with the tag.

The beginning tag is "*INDENT-OFF*" and the ending tag is "*INDENT-ON*". They may be used anywhere in the program with the condition that parsing is partially disabled between the tags. Disabling partial statements may result in incorrect formatting after the ending tag. If this happens, expand the tags to include additional code.

The following retains the format of a preprocessor define:

```
// *INDENT-OFF*
#define FOO_DECLARE_int32_(name) \
        FOO_API_ extern ::Int32 FOO_FLAG(name)
// *INDENT-ON*
```

Disable Line

Artistic Style cannot always determine the usage of symbols with more than one meaning. For example an asterisk (*) can be multiplication, a pointer, or a pointer dereference. The "&" and "&&" symbols are a similar problem.

If a symbol is being padded incorrectly, padding it manually may fix the problem. If it is still being padded incorrectly, then disabling the formatting may be necessary. To avoid having to use the "disable block" tags above, a single line disable is available.

A line-end comment tag "*NOPAD*" will disable the "pad-oper", "align-pointer", and "align-reference" options. Parsing does NOT stop and all other formatting will be applied to the line. The tag applies to the one line only.

The following prevents the operator padding from changing:

```
size_t foo = (unsigned int) -1;  // *NOPAD*
```

Basic Brace Styles

Top

There are three basic brace styles.
Attached – The braces are attached to the end of the last line of the previous block. (Java).

Broken – The braces are broken from the previous block. (Allman).
Linux – The braces are attached except for the opening brace of a function, class, or namespace (K&R, Linux).

Other brace styles are variations of these. Some will use variations on the placement of class, namespace, or other braces. (Stroustrup, Google, One True Brace, Lisp). Others will indent the braces (Whitesmith, VTK, Banner, and GNU). Others will use run-in braces where the following statement is on the same line as the brace (Horstmann and Pico).

There are technical arguments for selecting one style over another. But the usual reason comes down to personal preference. Some like broken braces with vertical whitespace that makes the code easy to read. Others like attached braces with code that is more compact. Sometimes programmers just want a change. It is easier to select a preference if you can see an entire file formatted in a certain brace style. With Artistic Style you can easily modify source code to suit your preference.

Brace Style Options

Brace Style options define the brace style to use. All options default to 4 spaces per indent, indented with spaces. By default, none of the styles indent namespaces. Other indentations are indicated in the individual style description. All options will break the braces for one line blocks and will break one line statements. To change this, use the option keep-one-line-blocks and/or keep-one-line-statements described in the Formatting Options section.

default brace style

If no brace style is requested, the default brace style will be used. The opening braces are not changed and the closing braces will be broken from the preceding line. There are a few exceptions to this.

--style=allman / --style=bsd / --style=break / -A1

Allman style uses broken braces.

```
int Foo(bool isBar)
{
    if (isBar)
    {
        bar();
        return 1;
    }
    else
        return 0;
}
```

--style=java / --style=attach / -A2

Java style uses attached braces.

```
int Foo(bool isBar) {
    if (isBar) {
        bar();
        return 1;
    } else
        return 0;
}
```

--style=kr / --style=k&r / --style=k/r / -A3

Kernighan & Ritchie style uses linux braces. Opening braces are broken from namespaces, classes, and function definitions. The braces are attached to everything else, including arrays, structs, enums, and statements within a function.

Using the k&r option may cause problems because of the &. This can be resolved by enclosing the k&r in quotes (e.g. --style="k&r") or by using one of the alternates --style=kr or –style=k/r.

```
int Foo(bool isBar)
{
    if (isBar) {
        bar();
```

Top

```
        return 1;
    } else
        return 0;
}
```

--style=stroustrup / -A4

Stroustrup style uses linux braces with closing headers broken from closing braces (e.g. –break-closing-headers). Opening braces are broken from function definitions only. The opening braces are attached to everything else, including namespaces, classes, arrays, structs, enums, and statements within a function. This style frequently is used with "attach-closing-while", tabbed indents, and an indent of 5 spaces.

```
int Foo(bool isBar)
{
    if (isBar) {
        bar();
        return 1;
    }
    else
        return 0;
}
```

--style=whitesmith / -A5

Whitesmith style uses broken, indented braces. Switch blocks and class blocks are indented to prevent a 'hanging indent' with the following case statements and C++ class modifiers (public, private, protected).

```
int Foo(bool isBar)
    {
    if (isBar)
        {
        bar();
        return 1;
        }
    else
        return 0;
    }
```

--style=vtk / -A15

VTK (Visualization Toolkit) style uses broken, indented braces, except for the opening brace of classes, arrays, structs, enums, and function definitions.. Switch blocks are indented to prevent a 'hanging indent' with following case statements.

```
int Foo(bool isBar)
{
    if (isBar)
        {
        bar();
        return 1;
        }
    else
        return 0;
}
```

--style=ratliff / --style=banner / -A6

Ratliff style uses attached, indented braces. Switch blocks and class blocks are indented to prevent a 'hanging indent' with following case statements and C++ class modifiers (public, private, protected).

```
int Foo(bool isBar) {
    if (isBar) {
        bar();
        return 1;
        }
    else
        return 0;
    }
```

--style=gnu / -A7

Top

GNU style uses broken braces. Extra indentation is added to blocks **within a function** only. The entire block is indented, not just the brace. This style frequently is used with an indent of 2 spaces.

```
int Foo(bool isBar)
{
    if (isBar)
        {
            bar();
            return 1;
        }
    else
        return 0;
}
```

--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;
        } else
                return 0;
}
```

--style=horstmann / --style=run-in / -A9

Horstmann style uses broken braces and run-in statements. Switches are indented to allow a run-in to the opening switch block. This style frequently is used with an indent of 3 spaces.

```
int Foo(bool isBar)
{   if (isBar)
    {   bar();
        return 1;
    }
    else
        return 0;
}
```

--style=1tbs / --style=otbs / -A10

"One True Brace Style" uses linux braces and adds braces to unbraced one line conditional statements. Opening braces are broken from namespaces, classes, and function definitions. The braces are attached to everything else, including arrays, structs, enums, and statements within a function.

In the following example, braces have been added to the "return 0;" statement. The option –add-one-line-braces can also be used with this style.

```
int Foo(bool isBar)
{
    if (isFoo) {
        bar();
        return 1;
    } else {
        return 0;
    }
}
```

--style=google / -A14

Google style uses attached braces and indented class access modifiers. See the indent-modifiers option for an example of the indented modifiers format. This is not actually a unique brace style, but is Java style with a non-brace variation. This style frequently is used with an indent of 2 spaces.

```
int Foo(bool isBar) {
    if (isBar) {
        bar();
        return 1;
    } else
        return 0;
}
```

Top

--style=mozilla / -A16

Mozilla style uses linux braces. Opening braces are broken from classes, structs, enums, and function definitions. The braces are attached to everything else, including namespaces, arrays, and statements within a function. This style frequently is used with an indent of 2 spaces and --break-return-type.

```
int Foo(bool isBar)
{
    if (isBar) {
        bar();
        return 1;
    } else
        return 0;
}
```

--style=pico / -A11

Pico style uses broken braces and run-in statements with attached closing braces. The closing brace is attached to the last line in the block. Switches are indented to allow a run-in to the opening switch block. The style implies keep-one-line-blocks and keep-one-line-statements. If add-braces is used they will be added as one-line braces. This style frequently is used with an indent of 2 spaces.

```
int Foo(bool isBar)
{   if (isBar)
    {   bar();
        return 1; }
    else
        return 0; }
```

--style=lisp / --style=python / -A12

Lisp style uses attached opening and closing braces. The closing brace is attached to the last line in the block. The style implies keep-one-line-statements, but NOT keep-one-line-blocks. This style does not support one-line braces. If add-one-line-braces is used they will be added as multiple-line braces.

```
int Foo(bool isBar) {
    if (isBar) {
        bar()
        return 1; }
    else
        return 0; }
```

Tab Options

The following examples show whitespace characters. A space is indicated with a **.** (dot), a tab is indicated by a > (greater than).

default indent

If no indentation option is set, the default option of 4 spaces will be used (e.g. -s4 --indent=spaces=4).

with default values:

```
void Foo() {
....if (isBar1
............&& isBar2)    // indent of this line can be changed with min-conditional-indent
........bar();
}
```

--indent=spaces / --indent=spaces=# / -s#

Indent using # **spaces** per indent (e.g. -s3 --indent=spaces=3). # must be between 2 and 20. Not specifying # will result in a default of 4 spaces per indent.

with indent=spaces=3

```
void Foo() {
...if (isBar1
.........&& isBar2)    // indent of this line can be changed with min-conditional-indent
......bar();
}
```

Top

`--indent=tab / --indent=tab=# / -t / -t#`

Indent using **tabs for indentation, and spaces for continuation line alignment**. This ensures that the code is displayed correctly regardless of the viewer's tab size. Treat each indent as # spaces (e.g. -t6 / --indent=tab=6). # must be between 2 and 20. If no # is set, treats indents as 4 spaces.

with indent=tab:

```
void Foo() {
>   if (isBar1
>   ........&& isBar2)    // indent of this line can be changed with min-conditional-indent
>   >   bar();
}
```

with style=linux, indent=tab=8:

```
void Foo()
{
>       if (isBar1
>       ....&& isBar2)    // indent of this line can NOT be changed with style=Linux
>       >       bar();
}
```

`--indent=force-tab / --indent=force-tab=# / -T / -T#`

Indent using **all tab** characters, if possible. If a continuation line is not an even number of tabs, spaces will be added at the end. Treat each tab as # spaces (e.g. -T6 / --indent=force-tab=6). # must be between 2 and 20. If no # is set, treats tabs as 4 spaces.

with indent=force-tab:

```
void Foo() {
>   if (isBar1
>   >   >   && isBar2)    // indent of this line can be changed with min-conditional-indent
>   >   bar();
}
```

`--indent=force-tab-x / --indent=force-tab-x=# / -xT / -xT#`

This force-tab option allows the tab length to be set to a length that is different than the indent length. This may cause the indentation to be **a mix of both tabs and spaces.** Tabs will be used to indent, if possible. If a tab indent cannot be used, spaces will be used instead.

This option sets the **tab length.** Treat each tab as # spaces (e.g. -xT6 / --indent=force-tab-x=6. # must be between 2 and 20. If no # is set, treats tabs as 8 spaces. To change the **indent length** from the default of 4 spaces the option "indent=force-tab" must also be used.

with indent=force-tab-x (default tab length of 8 and default indent length of 4):

```
void Foo() {
....if (isBar1
>       ....&& isBar2)    // indent of this line can be changed with min-conditional-indent
>       bar();
}
```

Brace Modify Options

`--attach-namespaces / -xn`

Attach braces to a namespace statement. This is done regardless of the brace style being used. It will also attach braces to CORBA IDL module statements.

the brace is always attached to a namespace statement:

```
namespace FooName {
...
}
```

Top

--attach-classes / -xc

Attach braces to a class statement. This is done regardless of the brace style being used.

the brace is always attached to a class statement:

```
class FooClass {
...
};
```

--attach-inlines / -xl

Attach braces to class and struct inline function definitions. This option has precedence for all styles except Horstmann and Pico (run-in styles). It is effective for C++ files only.

all braces are attached to class and struct inline method definitions:

```
class FooClass
{
    void Foo() {
    ...
    }
};
```

--attach-extern-c / -xk

Attach braces to a braced extern "C" statement. This is done regardless of the brace style being used. This option is effective for C++ files only.

An extern "C" statement that is part of a function definition is formatted according to the requested brace style. Braced extern "C" statements are unaffected by the brace style and this option is the only way to change them.

this option attaches braces to a braced extern "C" statement:

```
#ifdef __cplusplus
extern "C" {
#endif
```

but function definitions are formatted according to the requested brace style:

```
extern "C" EXPORT void STDCALL Foo()
{}
```

--attach-closing-while / -xV

Attach the closing 'while' of a 'do-while' statement to the closing brace. This has precedence over both the brace style and the break closing braces option.

```
do
{
    bar();
    ++x;
}
while x == 1;
```

becomes:

```
do
{
    bar();
    ++x;
} while x == 1;
```

Indentation Options

--indent-classes / -C

Indent 'class' and 'struct' blocks so that the entire block is indented. The struct blocks are indented only if an access modifier, 'public:', 'protected:' or 'private:', is declared somewhere in the struct. This option is effective for C++ files only.

Top

```
class Foo
{
public:
    Foo();
    virtual ~Foo();
};
```

becomes:

```
class Foo
{
    public:
        Foo();
        virtual ~Foo();
};
```

--indent-modifiers / -xG

Indent 'class ' and 'struct' access modifiers, 'public:', 'protected:' and 'private:', one half indent. The rest of the class is not indented. This option is effective for C++ files only. If used with indent-classes this option will be ignored.

```
class Foo
{
public:
    Foo();
    virtual ~Foo();
};
```

becomes:

```
class Foo
{
  public:
    Foo();
    virtual ~Foo();
};
```

--indent-switches / -S

Indent 'switch' blocks so that the 'case X:' statements are indented in the switch block. The entire case block is indented.

```
switch (foo)
{
case 1:
    a += 1;
    break;

case 2:
{
    a += 2;
    break;
}
}
```

becomes:

```
switch (foo)
{
    case 1:
        a += 1;
        break;

    case 2:
    {
        a += 2;
        break;
    }
}
```

--indent-cases / -K

Indent '`case X:`' blocks from the '`case X:`' headers. Case statements not enclosed in blocks are NOT indented.

Top

```
switch (foo)
{
    case 1:
```

```
        a += 1;
        break;

    case 2:
    {
        a += 2;
        break;
    }
}
```

becomes:

```
switch (foo)
{
    case 1:
        a += 1;
        break;

    case 2:
        {
            a += 2;
            break;
        }
}
```

--indent-namespaces / -N

Add extra indentation to namespace blocks. This option has no effect on Java files. It will also indent CORBA IDL module statements.

```
namespace foospace
{
class Foo
{
    public:
        Foo();
        virtual ~Foo();
};
}
```

becomes:

```
namespace foospace
{
    class Foo
    {
        public:
            Foo();
            virtual ~Foo();
    };
}
```

--indent-after-parens / -xU

Indent, instead of align, continuation lines following lines that contain an opening paren '(' or an assignment '='. This includes function definitions and declarations and return statements. The indentation can be modified by using the following indent-continuation option. This option may be preferred for editors displaying proportional fonts.

```
void Foo(bool bar1,
         bool bar2)
{
    isLongFunction(bar1,
                   bar2);

    isLongVariable = foo1
                     || foo2;
}
```

becomes:

```
void Foo(bool bar1,
    bool bar2)
{
    isLongFunction(bar1,
        bar2);

    isLongVariable = foo1
        || foo2;
}
```

Top

--indent-continuation=# / -xt#

Set the continuation indent for a line that ends with an opening paren '(' or an assignment '='. This includes function definitions and declarations. It will also modify the previous indent-after-paren option. The value for # indicates a **number of indents**. The valid values are the integer values from **0 thru 4**. If this option is not used, the default value of **1** is used.

```
isLongVariable =
    foo1 ||
    foo2;

isLongFunction(
    bar1,
    bar2);
```

becomes (with indent-continuation=3):

```
isLongVariable =
            foo1 ||
            foo2;

isLongFunction(
            bar1,
            bar2);
```

--indent-labels / -L

Add extra indentation to labels so they appear 1 indent less than the current indentation, rather than being flushed to the left (the default).

```
void Foo() {
    while (isFoo) {
        if (isFoo)
            goto error;
        ...
error:
        ...
        }
}
```

becomes (with indented 'error:'):

```
void Foo() {
    while (isFoo) {
        if (isFoo)
            goto error;
        ...
    error:
        ...
        }
}
```

--indent-preproc-block / -xW

Indent preprocessor blocks at brace level zero and immediately within a namespace. There are restrictions on what will be indented. Blocks within methods, classes, arrays, etc., will not be indented. Blocks containing braces or multi-line define statements will not be indented. Without this option the preprocessor block is not indented.

```
#ifdef _WIN32
#include <windows.h>
#ifndef NO_EXPORT
#define EXPORT
#endif
#endif
```

becomes:

```
#ifdef _WIN32
    #include <windows.h>
    #ifndef NO_EXPORT
        #define EXPORT
    #endif
#endif
```

Top

--indent-preproc-define / -w

Indent multi-line preprocessor definitions ending with a backslash. Should be used with –convert-tabs for proper results. Does a pretty good job, but cannot perform miracles in obfuscated preprocessor definitions. Without this option the preprocessor statements remain unchanged.

```
#define Is_Bar(arg,a,b) \
(Is_Foo((arg), (a)) \
|| Is_Foo((arg), (b)))
```

becomes:

```
#define Is_Bar(arg,a,b) \
    (Is_Foo((arg), (a)) \
     || Is_Foo((arg), (b)))
```

--indent-preproc-cond / -xw

Indent preprocessor conditional statements to the same level as the source code.

```
        isFoo = true;
#ifdef UNICODE
        text = wideBuff;
#else
        text = buff;
#endif
```

becomes:

```
        isFoo = true;
        #ifdef UNICODE
        text = wideBuff;
        #else
        text = buff;
        #endif
```

--indent-col1-comments / -Y

Indent C++ comments beginning in column one. By default C++ comments beginning in column one are assumed to be commented-out code and not indented. This option will allow the comments to be indented with the code.

```
void Foo()\n"
{
// comment
    if (isFoo)
        bar();
}
```

becomes:

```
void Foo()\n"
{
    // comment
    if (isFoo)
        bar();
}
```

--min-conditional-indent=# / -m#

Set the minimal indent that is added when a header is built of multiple lines. This indent helps to easily separate the header from the command statements that follow. The value for # indicates a **number of indents** and is a minimum value. The indent may be greater to align with the data on the previous line. The valid values are:
0 - no minimal indent. The lines will be aligned with the paren on the preceding line.
1 - indent at least one additional indent.
2 - indent at least two additional indents.
3 - indent at least one-half an additional indent. This is intended for large indents (e.g. 8).
The default value is **2**, two additional indents.

```
// default setting makes this non-braced code clear
if (a < b
        || c > d)
    foo++;

// but creates an exaggerated indent in this braced code
if (a < b
        || c > d)
```

Top

```
{
    foo++;
}
```

becomes (when setting `--min-conditional-indent=0`):

```
// setting makes this non-braced code less clear
if (a < b
    || c > d)
    foo++;

// but makes this braced code clearer
if (a < b
    || c > d)
{
    foo++;
}
```

`--max-continuation-indent=# / -M#`

Set the maximum of # spaces to indent a continuation line. The # indicates a number of columns and must not be less than **40** or greater than **120**. If no value is set, the default value of **40** will be used. This option will prevent continuation lines from extending too far to the right. Setting a larger value will allow the code to be extended further to the right.

```
fooArray[] = { red,
               green,
               blue };

fooFunction(barArg1,
            barArg2,
            barArg3);
```

becomes (with larger value):

```
fooArray[] = { red,
                  green,
                  blue };

fooFunction(barArg1,
               barArg2,
               barArg3);
```

Padding Options

`--break-blocks / -f`

Pad empty lines around header blocks (e.g. 'if', 'for', 'while'...).

```
isFoo = true;
if (isFoo) {
    bar();
} else {
    anotherBar();
}
isBar = false;
```

becomes:

```
isFoo = true;

if (isFoo) {
    bar();
} else {
    anotherBar();
}

isBar = false;
```

`--break-blocks=all / -F`

Pad empty lines around header blocks (e.g. 'if', 'for', 'while'...). Treat closing header blocks (e.g. 'else', 'catch') as stand-alone blocks.

Top

```
isFoo = true;
if (isFoo) {
    bar();
} else {
    anotherBar();
}
isBar = false;
```

becomes:

```
isFoo = true;

if (isFoo) {
    bar();

} else {
    anotherBar();
}

isBar = false;
```

--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--);
```

--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);
```

--pad-paren / -P

Insert space padding around parens on both the **outside** and the **inside**. Any end of line comments will remain in the original column, if possible.

```
if (isFoo((a+2), b))
    bar(a, b);
```

becomes:

```
if ( isFoo ( ( a+2 ), b ) )
    bar ( a, b );
```

--pad-paren-out / -d

Insert space padding around parens on the **outside** only. Parens that are empty will not be padded. Any end of line comments will remain in the original column, if possible. This can be used with unpad-paren below to remove unwanted spaces.

```
if (isFoo((a+2), b))
    bar(a, b);
```

becomes:

```
if (isFoo ( (a+2), b) )
    bar (a, b);
```

Top

--pad-first-paren-out / -xd

Insert space padding around the **first** paren in a series on the **outside** only. Parens that are empty will not be padded. Any end of line comments will remain in the original column, if possible. This can be used with unpad-paren below to remove unwanted spaces. If used with pad-paren or pad-paren-out, this option will be ignored. If used with pad-paren-in, the result will be the pad-paren.

```
if (isFoo((a+2), b))
    bar(a, b);
```

becomes:

```
if (isFoo ((a+2), b))
    bar (a, b);
```

--pad-paren-in / -D

Insert space padding around paren on the **inside** only. Any end of line comments will remain in the original column, if possible. This can be used with unpad-paren below to remove unwanted spaces.

```
if (isFoo((a+2), b))
    bar(a, b);
```

becomes:

```
if ( isFoo( ( a+2 ), b ) )
    bar( a, b );
```

--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);
```

--unpad-paren / -U

Remove extra space padding around parens on the inside and outside. Any end of line comments will remain in the original column, if possible. This option can be used in combination with the paren padding options pad-paren, pad-paren-out, pad-paren-in, and pad-header above. Only padding that has not been requested by other options will be removed.

For example, if a source has parens padded on both the inside and outside, and you want inside only. You need to use unpad-paren to remove the outside padding, and pad-paren-in to retain the inside padding. Using only pad-paren-in> would not remove the outside padding.

```
if ( isFoo( ( a+2 ), b ) )
    bar ( a, b );
```

becomes (with no padding option requested):

```
if(isFoo((a+2), b))
    bar(a, b);
```

--delete-empty-lines / -xe

Delete empty lines within a function or method. Empty lines outside of functions or methods are NOT deleted. If used with break-blocks or break-blocks=all it will delete all lines EXCEPT the lines added by the break-blocks options.

```
void Foo()
{

    foo1 = 1;

    foo2 = 2;

}
```

Top

becomes:

```
void Foo()
{
    foo1 = 1;
    foo2 = 2;
}
```

--fill-empty-lines / -E
Fill empty lines with the white space of the previous line.

--align-pointer=type / -k1
--align-pointer=middle / -k2
--align-pointer=name / -k3
Attach a pointer or reference operator (*, &, or ^) to either the variable type (left) or variable name (right), or place it between the type and name (middle). The spacing between the type and name will be preserved, if possible. This option is for C/C++, C++/CLI, and C# files. To format references separately, use the following align-reference option.

```
char* foo1;
char & foo2;
string ^s1;
```

becomes (with align-pointer=type):

```
char* foo1;
char& foo2;
string^ s1;

char* foo1;
char & foo2;
string ^s1;
```

becomes (with align-pointer=middle):

```
char * foo1;
char & foo2;
string ^ s1;

char* foo1;
char & foo2;
string ^s1;
```

becomes (with align-pointer=name):

```
char *foo1;
char &foo2;
string ^s1;
```

--align-reference=none / -W0
--align-reference=type / -W1
--align-reference=middle / -W2
--align-reference=name / -W3
This option will align references separate from pointers. Pointers are not changed by this option. If pointers and references are to be aligned the same, use the previous align-pointer option. The option align-reference=none will not change the reference alignment. The other options are the same as for align-pointer. This option is for C/C++, C++/CLI, and C# files.

```
char &foo1;
```

becomes (with align-reference=type):

```
char& foo1;

char& foo2;
```

becomes (with align-reference=middle):

```
char & foo2;

char& foo3;
```

becomes (with align-reference=name):

Top

```
char &foo3;
```

Formatting Options

--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 (a broken 'else'):

```
void Foo(bool isFoo) {
    if (isFoo) {
        bar();
    }
    else {
        anotherBar();
    }
}
```

--break-elseifs / -e

Break "else if" header combinations into separate lines. This option has no effect if keep-one-line-statements is used, the "else if" statements will remain as they are.

If this option is NOT used, "else if" header combinations will be placed on a single line.

```
if (isFoo) {
    bar();
}
else if (isFoo1()) {
    bar1();
}
else if (isFoo2()) {
    bar2;
}
```

becomes:

```
if (isFoo) {
    bar();
}
else
    if (isFoo1()) {
        bar1();
    }
    else
        if (isFoo2()) {
            bar2();
        }
```

--break-one-line-headers / -xb

Break one line headers (e.g. 'if', 'while', 'else', ...) from a statement residing on the same line. If the statement is enclosed in braces, the braces will be formatted according to the requested brace style.

A multi-statement line will NOT be broken if keep-one-line-statements is requested. One line blocks will NOT be broken if keep-one-line-blocks is requested and the header is enclosed in the block.

```
void Foo(bool isFoo)
{
    if (isFoo1) bar1();

    if (isFoo2) { bar2(); }
}
```

Top

becomes:

```
void Foo(bool isFoo)
{
    if (isFoo1)
        bar1();

    if (isFoo2) {
        bar2();
    }
}
```

--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;
}
```

--add-one-line-braces / -J

Add one line braces to unbraced one line conditional statements (e.g. 'if', 'for', 'while'...). The statement must be on a single line. The option implies --keep-one-line-blocks and will not break the one line blocks.

```
if (isFoo)
    isFoo = false;
```

becomes:

```
if (isFoo)
    { isFoo = false; }
```

--remove-braces / -xj

Remove braces from conditional statements (e.g. 'if', 'for', 'while'...). The statement must be a single statement on a single line. If --add-braces or —add-one-line-braces is also used the result will be to add braces. Braces will not be removed from "One True Brace Style", --style=1tbs.

```
if (isFoo)
{
    isFoo = false;
}
```

becomes:

```
if (isFoo)
    isFoo = false;
```

--break-return-type / -xB
--break-return-type-decl / -xD

Break the return type from the function name. The two options are for the function definitions (-xB), and the function declarations or signatures (-xD). If used with --attach-return-type, the result will be to break the return type. This option has no effect on Objective-C functions.

```
void Foo(bool isFoo);
```

becomes:

```
void
Foo(bool isFoo);
```

Top

--attach-return-type / -xf
--attach-return-type-decl / -xh
Attach the return type to the function name. The two options are for the function definitions (-xf), and the function declarations or signatures (-xh). They are intended to undo the --break-return-type options. If used with --break-return-type, the result will be to break the return type. This option has no effect on Objective-C functions.

```
void
Foo(bool isFoo);
```

becomes:

```
void Foo(bool isFoo);
```

--keep-one-line-blocks / -O
Don't break one-line blocks.

```
if (isFoo)
{ isFoo = false; cout << isFoo << endl; }
```

remains unchanged.

--keep-one-line-statements / -o
Don't break complex statements and multiple statements residing on a single line.

```
if (isFoo)
{
    isFoo = false; cout << isFoo << endl;
}
```

remains unchanged.

--convert-tabs / -c
Converts tabs into spaces in the non-indentation part of the line. The number of spaces inserted will maintain the spacing of the tab. The current setting for spaces per tab is used. It may not produce the expected results if convert-tabs is used when changing spaces per tab. Tabs are not replaced within quotes.

--close-templates / -xy
Closes whitespace between the ending angle brackets of template definitions. Closing the ending angle brackets is now allowed by the C++11 standard. Be sure your compiler supports this before making the changes.

```
Stack< int, List< int > > stack1;
```

becomes:

```
Stack< int, List< int >> stack1;
```

--remove-comment-prefix / -xp
Remove the preceding '*' in a multi-line comment that begins a line. A trailing '*', if present, is also removed. Text that is less than one indent is indented to one indent. Text greater than one indent is not changed. Multi-line comments that begin a line, but without the preceding '*', are indented to one indent for consistency. This can slightly modify the indentation of commented out blocks of code. Lines containing all '*' are left unchanged. Extra spacing is removed from the comment close '*/'.

```
/*
 * comment line 1
 * comment line 2
 */
```

becomes:

```
/*
    comment line 1
    comment line 2
*/
```

Top

--max-code-length=# / -xC#
--break-after-logical / -xL
The option max-code-length will break a line if the code exceeds # characters. The valid values are **50** thru **200**. Lines without logical conditionals will break on a logical conditional (||, &&, ...), comma, paren, semicolon, or space.

Some code will not be broken, such as comments, quotes, and arrays. If used with keep-one-line-blocks or add-one-line-braces the blocks will NOT be broken. If used with keep-one-line-statements the statements will be broken at a semicolon if the line goes over the maximum length. If there is no available break point within the max code length, the line will be broken at the first available break point after the max code length.

By default logical conditionals will be placed first in the new line. The option **break-after-logical** will cause the logical conditionals to be placed last on the previous line. This option has no effect without max-code-length.

```
if (thisVariable1 == thatVariable1 || thisVariable2 == thatVariable2 || thisVariable3 == thatVariable3)
    bar();
```

becomes:

```
if (thisVariable1 == thatVariable1
        || thisVariable2 == thatVariable2
        || thisVariable3 == thatVariable3)
    bar();
```

becomes (with break-after-logical):

```
if (thisVariable1 == thatVariable1 ||
        thisVariable2 == thatVariable2 ||
        thisVariable3 == thatVariable3)
    bar();
```

--mode=c
--mode=cs
--mode=java
Indent a C type, C#, or Java file. C type files are C, C++, C++/CLI, and Objective-C. The option is usually set from the file extension for each file. You can override the setting with this entry. It will be used for all files, regardless of the file extension. It allows the formatter to identify language specific syntax such as C++ classes, templates, and keywords.

Objective-C Options

These options are effective for Objective-C files only. The standard paren padding options will still apply to the Objective-C method prefix and return type unless overridden by the following options.

Because of the longer indents sometimes needed for Objective-C, the option "max-continuation-indent" may need to be increased. If you are not getting the paren and square bracket alignment you want, try increasing this value. The option is described in the "Indentation Options" section.

--pad-method-prefix / -xQ
Insert space padding **after** the '-' or '+' Objective-C method prefix. This will add exactly one space. Any additional spaces will be deleted.

```
-(void)foo1;
-    (void)foo2;
```

becomes:

```
- (void)foo1;
- (void)foo2;
```

Top

--unpad-method-prefix / -xR
Remove all space padding **after** the '-' or '+' Objective-C method prefix. This option will be ignored if used with pad-method-prefix. This option takes precedence over the pad paren outside option.

```
- (void) foo1;
-     (void) foo2;
```

becomes:

```
-(void) foo1;
-(void) foo2;
```

--pad-return-type / -xq

Insert space padding **after** the Objective-C return type. This will add exactly one space. Any additional spaces will be deleted.

```
-(void)foo1;
-(void)   foo2;
```

becomes:

```
-(void) foo1;
-(void) foo2;
```

--unpad-return-type / -xr

Remove all space padding **after** the Objective-C return type. This option will be ignored if used with pad-return-type. This option takes precedence over the pad paren outside option.

```
-(void) foo1;
-(void)    foo2;
```

becomes:

```
-(void)foo1;
-(void)foo2;
```

--pad-param-type / -xS

Insert space padding around an Objective-C parameter type. This will add exactly one space. Any additional spaces will be deleted. This has precedence over the pad method colon option and will always cause space padding after a method colon.

```
-(void)foo1:(bool)barArg1;
-(void)foo2:    (bool)   barArg2;
```

becomes:

```
-(void)foo1: (bool) barArg1;
-(void)foo2: (bool) barArg2;
```

--unpad-param-type / -xs

Remove all space padding around an Objective-C parameter type. This option takes precedence over the pad paren outside option. The pad method colon option has precedence over the **opening** paren. The closing paren will always be unpadded.

```
-(void)foo1: (bool)    barArg1;
-(void)foo2:     (bool)    barArg2;
```

becomes (with an unpadded method colon):

```
-(void)foo1:(bool)barArg1;
-(void)foo2:(bool)barArg2;
```

becomes (with a padded method colon after):

```
-(void)foo1: (bool)barArg1;
-(void)foo2: (bool)barArg2;
```

--align-method-colon / -xM

Align the colons in Objective-C method declarations and method calls. If this option is not declared, method definitions will be indented uniformly, and method calls will align with the first keyword.

Top

```
-(void)longKeyword: (ID)theArg1
        keyword: (int)theArg2
```

```
        error: (NSError*)theError
{
    [myObj longKeyword: arg1
     keyword: arg2
     error: arg3];
}
```

becomes (with no option declared):

```
-(void)longKeyword: (ID)theArg1
    keyword: (int)theArg2
    error: (NSError*)theError
{
    [myObj longKeyword: arg1
           keyword: arg2
           error: arg3];
}
```

becomes (with align-method-colon):

```
-(void)longKeyword: (ID)theArg1
           keyword: (int)theArg2
             error: (NSError*)theError
{
    [myObj longKeyword: arg1
              keyword: arg2
                error: arg3];
}
```

--pad-method-colon=none / -xP0
--pad-method-colon=all / -xP1
--pad-method-colon=after / -xP2
--pad-method-colon=before / -xP3
Add or remove space padding before or after the colons in an Objective-C method call. These options will pad exactly one space. Any additional spaces will be deleted. The space padding after the method colon can be overridden by pad-param-type.

with pad-method-colon=none:

```
-(void)insertKey:(id)key;
```

with pad-method-colon=all:

```
-(void)insertKey : (id)key;
```

with pad-method-colon=after:

```
-(void)insertKey: (id)key;
```

with pad-method-colon=before:

```
-(void)insertKey :(id)key;
```

Other Options

These are non-formatting options available for the command-line. They can also be included in an option file.

--suffix=####
Append the suffix #### instead of '.orig' to original file name (e.g. --suffix=**.bak**. If this is to be a file extension, the dot '.' must be included. Otherwise the suffix will be appended to the current file extension.

--suffix=none / -n
Do not retain a backup of the original file. The original file is purged after it is formatted.

--recursive / -r / -R
For each directory in the command line, process all subdirectories recursively. When using the recursive option the file name statement should contain a wildcard. Linux users should place the file path and name in double quotes so the shell will not resolve the wildcards (e.g. "$HOME/src/*.cpp"). Windows users should place the file path and name in double quotes if the path or name contains spaces.

Top

--dry-run
Perform a trial run with no changes made to the files. The report will be output as usual.

--exclude=####
Specify a file or subdirectory #### to be excluded from processing.

Excludes are matched from the end of the file path. An exclude option of "templates" will exclude ALL directories named "templates". An exclude option of "cpp/templates" will exclude ALL "cpp/templates" directories. You may proceed backward in the directory tree to exclude only the required directories.

Specific files may be excluded in the same manner. An exclude option of "default.cpp" will exclude ALL files named "default.cpp". An exclude option of "python/default.cpp" will exclude ALL files named "default.cpp" contained in a "python" subdirectory. You may proceed backward in the directory tree to exclude only the required files.

Wildcards are NOT allowed. There may be more than one exclude statement. The file path and name may be placed in double quotes (e.g. –exclude="foo bar.cpp").

--ignore-exclude-errors / -i
Allow processing to continue if there are errors in the "exclude=###" options.
This option lets the excludes for several projects be entered in a single option file. This option may be placed in the same option file as the excludes. It will display the unmatched excludes. The following option will not display the unmatched excludes.

--ignore-exclude-errors-x / -xi
Allow processing to continue if there are errors in the "exclude=###" options.
This option lets the excludes for several projects be entered in a single option file. This option may be placed in the same option file as the excludes. It will NOT display the unmatched excludes. The preceding option will display the unmatched excludes.

--errors-to-stdout / -X
Print errors to standard-output rather than to standard-error.
This option should be helpful for systems/shells that do not have a separate output to standard-error, such as in Windows95.

--preserve-date / -Z
Preserve the original file's date and time modified. The time modified will be changed a few microseconds to force the changed files to compile. This option is not effective if redirection is used to rename the input file.

--verbose / -v
Verbose display mode. Display optional information, such as release number, date, option file locations, and statistical data.

--formatted / -Q
Formatted files display mode. Display only the files that have been formatted. Do not display files that are unchanged.

--quiet / -q
Quiet display mode. Suppress all output except error messages.

--lineend=windows / -z1
--lineend=linux / -z2
--lineend=macold / -z3
Force use of the specified line end style. Valid options are windows (CRLF), linux (LF), and macold (CR). MacOld style is the format for Mac OS 9 and earlier. MacOS and OS X uses the Linux style. If one of these options is not used, the line ends will be determined automatically from the input file.

When **redirection** is used on Windows the output will always have Windows line ends. This option will be ignored.

Command Line Only

These options are available for the command-line only. They are NOT available in an options file.

Top

--options=####
--options=none
Specify an options file #### to read and use. It must contain a file path and a file name. This will allow the file name to be changed from astylerc or .astylerc.

The "none" option will disable the default options file if one exists. Only command-line parameters will be used.

Further information is available in the Option Files section.

--project
--project=####
--project=none
Specify a project option file to use. The option file should have the default name of .astylerc or _astylerc and should be in the top directory of the project being formatted.

Specify a project options file #### to use. It must contain a file name only without a directory path. This will allow the project file name to be changed from .astylerc or _astylerc. It should be in the top directory of the project being formatted.

The "none" option will disable a project options file if one exists. In this case, the project option file will not be used.

Further information is available in the Option Files section.

--ascii / -I
The displayed output will be ASCII characters only. The text will be displayed in English and numbers will not be formatted. The short option must be by itself, it cannot be concatenated with other options.

--version / -V
Print version number and quit. The short option must be by itself, it cannot be concatenated with other options.

--help / -h / -?
Print a help message and quit. The short option must be by itself, it cannot be concatenated with other options.

--html / -!
Open the HTML help file "astyle.html" in the default browser and quit. The short option must be by itself, it cannot be concatenated with other options. The documentation must be installed in the standard install path (/usr/share/doc/astyle/html for Linux or %PROGRAMFILES%\AStyle\doc for Windows). If installed to a different path use html=###.

--html=####
Open an HTML help file in the default browser using the file path #### and quit. An HTML file other than "astyle.help" may be specified. The path may include a directory path and a file name, or a file name only (e.g. html=install.html). If only a file name is used, it is assumed to be in the standard install path (/usr/share/doc/astyle/html for Linux or %PROGRAMFILES%\AStyle\doc for Windows). In both cases, the file name must include the html extension. File paths containing spaces must be enclosed in quotes.

On Linux the HTML file is opened using the script "xdg-open" from the install package "xdg-utils". This should be installed by default on most distributions.

Any HTML file can be opened by this option. The files you are likely to need are astyle.html (the default), install.html, and index.html.

--stdin=####
Open a file using the file path #### as input to single file formatting. This is a replacement for redirection. Do not use this with "<" redirection.

--stdout=####
Open a file using the file path #### as output from single file formatting. This is a replacement for redirection. Do not use this with ">" redirection.

Top