Indent style

From Wikipedia, the free encyclopedia

In computer programming, an **indent style** is a convention governing the indentation of blocks of code to convey the program's structure. This article largely addresses the free-form languages, such as C programming language and its descendants, but can be (and frequently is) applied to most other programming languages (especially those in the curly bracket family), where whitespace is otherwise insignificant. Indent style is just one aspect of programming style.

Indentation is not a requirement of most programming languages, where it is used as secondary notation. Rather, programmers indent to better convey the structure of their programs to human readers. In particular, indentation is used to show the relationship between control flow constructs such as conditions or loops and code contained within and outside them. However, some programming languages (such as Python and occam) use the indentation to determine the structure instead of using braces or keywords; this is known as the off-side rule, and in these languages indentation is meaningful to the compiler/interpreter, not just a matter of style.

This article uses "brackets" to refer to what are known as "parentheses" in American English, and "braces" to refer to what are known as "curly brackets" in American English.

Contents

Placement of braces

The main difference between indent styles lies in the placement of the braces of the compound statement (`{...}`) that often follows a control statement (`if`, `while`, `for`...). The table below shows this placement for all the styles discussed in this article. Note that, for consistency, the indent depth has been kept constant at 4 spaces, irrespective of the preferred indent depth of each style.

brace placement	styles

`while (x == y) {`
`    something();`
`    somethingelse();`
`}`
	K&R and variants:
	1TBS, Stroustrup, Linux kernel, BSD KNF

`while (x == y)`
`{`
`    something();`
`    somethingelse();`
`}`
	Allman

`while (x == y)`
`  {`
`    something();`
`    somethingelse();`
`  }`
	GNU

`while (x == y)`
`    {`
`    something();`
`    somethingelse();`
`    }`
	Whitesmiths

`while (x == y)`
`{   something();`
`    somethingelse();`
`}`
	Horstmann

`while (x == y)`
`{   something();`
`    somethingelse(); }`
	Pico

`while (x == y) {`
`    something();`
`    somethingelse();`
`    }`
	Ratliff

`while (x == y) {`
`    something();`
`    somethingelse(); }`
	Lisp

`while (x == y)`
`    { something()`
`    ; somethingelse()`
`    ;}`
	Haskell[1]

Tabs, spaces, and size of indent

The size of the indent is usually independent of the style. Many early programs used tab characters for indentation, for simplicity and to save on source file size. Unix editors generally view tabs as equivalent to eight characters, while Macintosh and Microsoft Windows environments would set them to four, creating confusion when code was transferred back and forth. Modern programming editors can now often set arbitrary indentation sizes, and will insert the appropriate combination of tabs and spaces. For Ruby, many shell programming languages, and some forms of HTML formatting, two spaces per indent level is generally used.

The issue of using hard tabs or spaces is an ongoing debate in the programming community. Some programmers such as Jamie Zawinski[2] feel that spaces instead of tabs increase cross-platform functionality. Others, such as the writers of the WordPress coding standards,[3] believe the opposite, that hard tabs increase cross-platform functionality.

Tools

There are a number of computer programs that automatically correct indent styles (according to the preferences of the program author) as well as the length of indents associated with tabs. A famous one among them is `indent`, a program included with many Unix-like operating systems.

In Emacs, various commands are available to automatically fix indentation problems, including just hitting Tab on a given line (in the default configuration). "M-x indent-region" can be used to properly indent large sections of code. Depending on the mode, Emacs can also replace leading indentation spaces with the appropriate number of tabs followed by spaces, which results in the minimal number of characters for indenting each source line.

Elastic tabstops is a tabulation style which requires support from the text editor, where entire blocks of text are kept automatically aligned when the length of one line in the block changes.

Styles

K&R style

The K&R style is commonly used in C. It is also used for C++ and other curly brace programming languages. Used in Kernighan and Ritchie's book *The C Programming Language*, it had its origins in Kernighan and Plauger's The Elements of Programming Style and *Software Tools*.

When adhering to K&R, each function has its opening brace at the next line on the same indentation level as its header, the statements within the braces are indented, and the closing brace at the end is on the same indentation level as the header of the function at a line of its own. The blocks inside a function, however, have their opening braces at the same line as their respective control statements; closing braces remain in a line of their own, unless followed by an **else** or **while** keyword.

```
int main(int argc, char *argv[])
{
    ...
    while (x == y) {
        something();
        somethingelse();

        if (some_error)
            do_correct();
        else
            continue_as_usual();
    }

    finalthing();
    ...
}
```

In old versions of the C programming language, the functions, however, were braced distinctly. The opening function brace of a function was placed on the line following after the declaration section and at the same indentation level as the declaration (header of the function). This is because in the original C language, argument types needed to be declared on the subsequent line (i. e., just after the header of the function), whereas when no arguments were necessary, the opening brace would not appear in the same line with the function declaration. The opening brace for function declarations was an exception to the currently basic rule stating that the statements and blocks of a function are all enclosed in the function braces.

```
/* Original pre-ISO C style without function prototypes */
int main(argc, argv)
    int   argc;
    char  *argv[];
{
    ...
}
```

Variant: 1TBS

Advocates of this style sometimes refer to it as "the one true brace style"[4] (abbreviated as 1TBS or OTBS) because of the precedent set by C (although advocates of other styles have been known to use similarly strong language). The main difference from the K&R style is that the braces are not omitted for a control statement with only a single statement in its scope.

In this style, the constructs that allow insertions of new code lines are on separate lines, and constructs that prohibit insertions are on a single line. This principle is amplified by bracing every if, else, while, etc.—including single-line conditionals—so that insertion of a new line of code anywhere is always "safe" (i.e., such an insertion will not make the flow of execution disagree with the source code indentation).

Suggested advantages of this style are that the beginning brace does not require an extra line by itself; and the ending brace lines up with the statement it conceptually belongs to. One cost of this style is that the ending brace of a block takes up an entire line by itself, which can be partially resolved in if/else blocks and do/while blocks:

```
//...
    if (x < 0) {
        puts("Negative");
        negative(x);
    } else {
        puts("Non-negative");
        nonnegative(x);
    }
```

It is not usually the opening brace itself that is interesting, but rather the controlling statement that introduced the block, and as such a suggested advantage with this style is that it makes it easier to find them.

While Java is sometimes written in other styles, a significant body of Java code uses a minor variant of the K&R style in which the opening brace is on the same line as the class or method declaration, largely because Sun's original style guides[5][6][7] used this K&R variant, and as a result most of the standard source code for the Java API is written in this style. It is also a popular indent style for ActionScript and JavaScript, along with the Allman style.

It should be noted that *The C Programming Language* does not explicitly specify this style, though it is followed consistently throughout the book. Of note from the book:

> The position of braces is less important, although people hold passionate beliefs. We have chosen one of several popular styles. Pick a style that suits you, then use it consistently.

Variant: Stroustrup

Stroustrup style is Bjarne Stroustrup's adaptation of K&R style for C++, as used in his books, such as *Programming: Principles and Practice using C++* and *The C++ Programming Language*.[8]

Unlike the variants above, Stroustrup does not use a “cuddled else”. Thus, Stroustrup would write[8]

```
if (x < 0) {
    puts("Negative");
    negative(x);
}
else {
    puts("Non-negative");
    nonnegative(x);
}
```

Stroustrup extends K&R style for classes, writing them as follows:

```
class Vector {
public:
    Vector(int s) :elem(new double[s]), sz(s) { }   // construct a Vector
    double& operator[](int i) { return elem[i]; }   // element access: subscripting
    int size() { return sz; }
private:
    double elem[lowast];    // pointer to the elements
    int sz;                 // the number of elements
};
```

Note that Stroustrup does not indent the labels `public:` and `private:`. Also note that in Stroustrup style, even though the opening brace of a *function* starts on a new line, the opening brace of a *class* is on the same line as the class name.

Also note that Stroustrup is okay with writing short functions all on one line. Stroustrup style is a named indentation style available in the editor Emacs. Currently, Bjarne Stroustrup encourages a more Allman-style approach with C++ as noted in his modern C++ Core Guidelines.[9]

Variant: Linux kernel

The kernel style is known for its extensive usage in the source tree of the Linux kernel. Linus Torvalds strongly advises all contributors to follow it. A detailed description of the style (which not only considers indentation, but naming conventions, comments and various other aspects as well) can be found on kernel.org (https://www.kernel.org/doc/Documentation/CodingStyle). The style borrows some elements from K&R, below.

The kernel style utilizes tabs (with tab stops set at 8 characters) for indentation. Opening curly braces of a function go to the beginning of the line following the function header. Any other opening curly braces are to be put on the same line as the corresponding statement, separated by a space. Labels in a "switch" statement are aligned with the enclosing block (there's only one level of indentation). A single-statement body of a compound statement (such as if, while and do-while) need not be surrounded by curly braces. If, however, one or more of the sub-statements in an "if-else" statement require braces, then both sub-statements should be wrapped inside curly braces. Line length is limited to 80 characters.

```
int power(int x, int y)
{
        int result;

        if (y < 0) {
                result = 0;
        } else {
                result = 1;
                while (y-- > 0)
                        result *= x;

        }
        return result;
}
```

Variant: BSD KNF

Also known as Kernel Normal Form, this is currently the form of most of the code used in the Berkeley Software Distribution operating systems. Although mostly intended for kernel code, it is widely used as well in userland code. It is essentially a thoroughly-documented variant of K&R style as used in the Bell Labs Version 6 & 7 UNIX source code.[10]

The SunOS kernel and userland uses a similar indentation style.[10] Like KNF, this also was based on AT&T style documents and that is sometimes known as Bill Joy Normal Form.[11] The SunOS guideline was published in 1996; ANSI C is discussed briefly. The correctness of the indentation of a list of source files can be verified by the *cstyle* program written by Bill Shannon.[12]

The hard tabulator (ts in vi) is kept at eight columns, while a soft tabulator is often defined as a helper as well (sw in vi), and set at four. The hard tabulators are used to indent code blocks, while a soft tabulator (four spaces) of additional indent is used for all continuing lines that must be split over multiple lines.

Moreover, function calls do not use a space before the parenthesis, although C language native statements such as `if`, `while`, `do`, `switch` and `return` do (in the case where `return` is used with parens). Functions that declare no local variables in their top-level block should also leave an empty line after their opening block brace.

Here follow a few samples:

```
while (x == y) {
        something();
        somethingelse();
}
finalthing();
```

```
if (data != NULL && res > 0) {
        if (JS_DefineProperty(cx, o, "data",
            STRING_TO_JSVAL(JS_NewStringCopyN(cx, data, res)),
            NULL, NULL, JSPROP_ENUMERATE) != 0) {
                QUEUE_EXCEPTION("Internal error!");
                goto err;
        }
        PQfreemem(data);
} else {
        if (JS_DefineProperty(cx, o, "data", OBJECT_TO_JSVAL(NULL),
            NULL, NULL, JSPROP_ENUMERATE) != 0) {
                QUEUE_EXCEPTION("Internal error!");
                goto err;
        }
}
```

```
static JSBool
pgresult_constructor(JSContext *cx, JSObject *obj, uintN argc,
    jsval *argv, jsval *rval)
{

        QUEUE_EXCEPTION("PGresult class not user-instantiable");

        return (JS_FALSE);
}
```

Allman style

The Allman style is named after Eric Allman. It is also sometimes known as "BSD style" since Allman wrote many of the utilities for BSD Unix (although this should not be confused with the different "BSD KNF style"; see above).

This style puts the brace associated with a control statement on the next line, indented to the same level as the control statement. Statements within the braces are indented to the next level.

```
while (x == y)
{
    something();
    somethingelse();
}

finalthing();
```

This style is similar to the standard indentation used by the Pascal programming language and Transact-SQL, where the braces are equivalent to the `begin` and `end` keywords.

```
(* Example Allman code indentation style in Pascal *)
procedure dosomething(x: integer, y: integer)
begin
    while x = y do
    begin
        something;
        somethingelse
    end
end
```

Purported advantages of this style are that the indented code is clearly set apart from the containing statement by lines that are almost completely whitespace and the closing brace lines up in the same column as the opening brace. Some people feel this makes it easy to find matching braces. The blocking style also delineates the actual block of code from the associated control statement itself. Commenting out the control statement, removing the control statement entirely, refactoring, or removing of the block of code is less likely to introduce syntax errors because of dangling or missing braces. Furthermore, it's consistent with brace placement for the outer/function block.

For example, the following is still syntactically correct:

```
// while (x == y)
{
    something();
    somethingelse();
}
```

As is this:

```
// for (int i=0; i < x; i++)
// while (x == y)
if (x == y)
{
    something();
    somethingelse();
}
```

Even like this, with conditional compilation:

```
    int c;
#ifdef HAS_GETCH
    while ((c = getch()) != EOF)
#else
    while ((c = getchar()) != EOF)
#endif
    {
        do_something(c);
    }
```

Whitesmiths style

The Whitesmiths style, also called Wishart style, to a lesser extent was originally used in the documentation for the first commercial C compiler, the Whitesmiths Compiler. It was also popular in the early days of Windows, since it was used in three influential Windows programming books, *Programmer's Guide to Windows* by Durant, Carlson & Yao, *Programming Windows* by Petzold, and *Windows 3.0 Power Programming Techniques* by Norton & Yao.

Whitesmiths along with Allman have been the most common bracing styles with equal mind shares according to the Jargon File.[13]

This style puts the brace associated with a control statement on the next line, indented. Statements within the braces are indented to the same level as the braces.

```
while (x == y)
    {
    something();
    somethingelse();
    }

finalthing();
```

The advantages of this style are similar to those of the Allman style in that blocks are clearly set apart from control statements. Another advantage is the alignment of the braces with the block that emphasizes the fact that the entire block is conceptually (as well as programmatically) a single compound statement. Furthermore, indenting the braces emphasizes that they are subordinate to the control statement. Another advantage of this style is that the ending brace no longer lines up with the statement, but instead with the opening brace.

An example:

```
if (data != NULL && res > 0)
    {
    if (!JS_DefineProperty(cx, o, "data", STRING_TO_JSVAL(JS_NewStringCopyN(cx, data, res)), NULL, NULL,
        {
        QUEUE_EXCEPTION("Internal error!");
        goto err;
        }
    PQfreemem(data);
    }
else if (!JS_DefineProperty(cx, o, "data", OBJECT_TO_JSVAL(NULL), NULL, NULL, JSPROP_ENUMERATE))
    {
    QUEUE_EXCEPTION("Internal error!");
    goto err;
    }
```

note: "else if" are treated as statement, much like the #elif preprocessor statement.

GNU style

Like the Allman and Whitesmiths styles, GNU style puts braces on a line by themselves, indented by two spaces, except when opening a function definition, where they are not indented.[14] In either case, the contained code is indented by two spaces from the braces.

Popularised by Richard Stallman, the layout may be influenced by his background of writing Lisp code.[15] In Lisp the equivalent to a block (a progn) is a first-class data entity, and giving it its own indent level helps to emphasize that, whereas in C a block is just syntax. Although not directly related to indentation, GNU coding style also includes a space before the bracketed list of arguments to a function.

```
static char *
concat (char *s1, char *s2)
{
  while (x == y)
    {
      something ();
      somethingelse ();
    }
  finalthing ();
}
```

[14]

This style combines the advantages of Allman and Whitesmiths, thereby removing the possible Whitesmiths disadvantage of braces not standing out from the block. One disadvantage is that the ending brace no longer lines up with the statement it conceptually belongs to. Another disadvantage is that the style wastes space resources by using two visual levels of indentation for one conceptual level of indentation.

The GNU Coding Standards recommend this style, and nearly all maintainers of GNU project software use it.

The GNU Emacs text editor and the GNU systems' indent command will reformat code according to this style by default. Those who do not use GNU Emacs, or similarly extensible/customisable editors, may find that the automatic indenting settings of their editor are unhelpful for this style. However, many editors defaulting to KNF style cope well with the GNU style when the tab width is set to two spaces; likewise, GNU Emacs adapts well to KNF style just by setting the tab width to eight spaces. In both cases, automatic reformatting will destroy the original spacing, but automatic line indentation will work correctly.

Steve McConnell, in his book Code Complete, advises against using this style: he marks a code sample which uses it with a "Coding Horror" icon, symbolizing especially dangerous code, and states that it impedes readability.[16]

Horstmann style

The 1997 edition of *Computing Concepts with C++ Essentials* by Cay S. Horstmann adapts Allman by placing the first statement of a block on the same line as the opening brace.

```
while (x == y)
{   something();
    somethingelse();
    //...
    if (x < 0)
    {   printf("Negative");
        negative(x);
    }
    else
    {   printf("Non-negative");
        nonnegative(x);
    }
}
finalthing();
```

This style combines the advantages of Allman by keeping the vertical alignment of the braces for readability and easy identification of blocks, with the saving of a line of the K&R style. However the 2003 edition now uses Allman style throughout. [1] (http://www.horstmann.com/bigcpp/styleguide.html)

Pico style

The style used most commonly in the Pico programming language by its designers is different from the aforementioned styles. The lack of return statements and the fact that semicolons are used in Pico as statement separators, instead of terminators, leads to the following syntax:

```
stuff(n):
{ x: 3 * n;
  y: doStuff(x);
  y + x }
```

The advantages and disadvantages are similar to those of saving screen real estate with K&R style. One additional advantage is that the beginning and closing braces are consistent in application (both share space with a line of code), as opposed to K&R style where one brace shares space with a line of code and one brace has a line to itself.

Ratliff style

In the book "Programmers at Work",[17] C. Wayne Ratliff discussed using the style below. The style begins much like 1TBS but then the closing brace lines up with the indentation of the nested block. Ratliff was the original programmer behind the popular dBase-II and dBase-III fourth-generation languages. He indicated that it was originally documented in material from Digital Research Inc. This style has sometimes been referred to as "banner" style,[18] possibly for the resemblance to a banner hanging from a pole. In this style, which is to Whitesmiths as K&R is to Allman, the closing control is indented as the last item in the list (and thus appropriately loses salience). The style can make visual scanning easier for some, since the "headers" of any block are the only thing exdented at that level (the theory being that the closing control of the previous block interferes with the visual flow of the next block header in the K&R and Allman styles).

```
// In C
for (i = 0; i < 10; i++) {
    if (i % 2 == 0) {
        doSomething(i);
        }
    else {
        doSomethingElse(i);
        }
    }
```

or, in a markup language...

```
<table>
  <tr>
    <td> lots of stuff...
      more stuff
      </td>
    <td> alternative for short lines </td>
    <td> etc. </td>
    </tr>
  </table>

<table>
  <tr> ... etc
  </table>
```

Lisp style

A programmer may even go as far as to insert closing braces in the last line of a block. This style makes indentation the only way of distinguishing blocks of code, but has the advantage of containing no uninformative lines. This could easily be called the Lisp style (because this style is very common in Lisp code) or the Python style (Python has no braces, but the layout looks very similar, as evidenced by the following code blocks). In Python, layout is a part of the language, called the off-side rule.

```
// In C
for (i = 0; i < 10; i++) {
    if (i % 2 == 0)
        doSomething(i);
    else {
        doSomethingElse(i);
        doThirdThing(i);}}
```

```
# In Python
for i in range(10):
    if i % 2 == 0:
        do_something(i)
    else:
        do_something_else(i)
        do_third_thing(i)
```

```
;; In Lisp
(dotimes (i 10)
    (if (evenp i)
        (do-something i)
        (progn
            (do-something-else i)
            (do-third-thing i))))
```

Haskell style

Haskell is a braces-optional language,[19] that is, the two following segments are semantically equal:

```
braceless = do
  text <- getContents
  let
    firstWord = head $ words text
    bigWord = map toUpper firstWord
  putStrLn bigWord

braceful = do
  { text <- getContents
  ; let
      { firstWord = head $ words text
      ; bigWord = map toUpper firstWord
      }
  ; putStrLn bigWord
  }
```

Usually the braces and semicolons are omitted for procedural `do` sections and the program text in general, but the style is commonly used for lists, records and other syntactic elements made up of some pair of parens or braces and separated with commas or semicolons.[20]

Other considerations

Losing track of blocks

In certain situations, there is a risk of losing track of block boundaries. This is often seen in large sections of code containing many compound statements nested to many levels of indentation. By the time the programmer scrolls to the bottom of a huge set of nested statements, he may have lost track of which control statements go where. However, overly long code could have other issues such as being too complex, and the programmer should consider whether refactoring the code would help in the longer term.

Programmers who rely on counting the opening braces may have difficulty with indentation styles such as K&R, where the beginning brace is not visually separated from its control statement. Programmers who rely more on indentation will gain more from styles that are vertically compact, such as K&R, because the blocks are shorter.

To avoid losing track of control statements such as for, one can use a large indent, such as an 8-unit wide hard tab, along with breaking up large functions into smaller and more readable functions. Linux is done this way, as well as using the K&R style.

In text editors of the vi family, one method for tracking block boundaries is to position the text cursor over one of the braces, and pressing the "%" key. Vi or vim will then bounce the cursor to the opposing brace. Since the text cursor's "next" key (viz., the "n" key) retained directional positioning information (whether the "up" or "down" key was previously pressed), the dot macro (the "." key) could then be used to place the text cursor on the next brace,[21] given an appropriate coding style. Alternatively, inspection of the block boundaries using the "%" key can be used to enforce a coding standard.

Another way is to use inline comments added after the closing brace:

```
for (int i = 0; i < total; i++) {
    foo(bar);
} //for (i)
```

```
if (x < 0) {
   bar(foo);
} //if (x < 0)
```

However, maintaining duplicate code in multiple locations is the major disadvantage of this method.

Another solution is implemented in a folding editor, which lets the developer hide or reveal blocks of code by their indentation level or by their compound statement structure. Many editors will also highlight matching brackets or braces when the caret is positioned next to one.

Statement insertion

K&R style prevents another common error suffered when using the standard UNIX line editor, ed. A statement mistakenly inserted between the control statement and the opening brace of the loop block turns the body of the loop into a single trip.

```
for (int i = 0; i < 10; i++)
    whoops(bar);   /* repeated 10 times, with i from 0 to 9 */
{
    only_once();   /* Programmer intended this to be done 10 times */
} //for (i) <-- This comment is no longer valid, and is very misleading!
```

K&R style avoids this problem by keeping the control statement and the opening brace on the same line.

See also

- Secondary notation
- Syntax highlighting

References

1. The last semicolon would not be needed in Haskell.
2. "Tabs versus Spaces: An Eternal Holy War." (http://www.jwz.org/doc/tabs-vs-spaces.html) by Jamie Zawinski 2000
3. "WordPress Coding Standards" (http://codex.wordpress.org/WordPress_Coding_Standards#Indentation)
4. "The Jargon File". Retrieved 18 August 2014.
5. Reddy, Achut (2000-03-30). "Java Coding Style Guide" (PDF). Sun Microsystems. Archived from the original (PDF) on February 28, 2006. Retrieved 2008-05-30.
6. "Java Code Conventions" (PDF). Sun Microsystems. 1997-09-12. Archived from the original (PDF) on May 13, 2008. Retrieved 2008-05-30.
7. "Code Conventions for the Java Programming Language". Sun Microsystems. 1997-03-20. Retrieved 2008-05-30.
8. Bjarne Stroustrup (September 2010). "PPP Style Guide" (PDF).
9. Stroustrup, Bjarne. "C++ Core Guidelines". *GitHub*. Retrieved 17 December 2015.
10. Shannon, Bill (1996). "C Style and Coding Standards for SunOS" (PDF) (Version 1.8 of 96/08/19.). Sun Microsystems, Inc. Retrieved 6 February 2015.
11. Gregg, Brendan. "DTraceToolkit Style Guide". Brendan D. Gregg. Retrieved 6 February 2015.
12. Shannon, Bill. "cstyle 1.58 98/09/09". *http://www.illumos.org/projects/illumos-gate*. Sun Microsystems, Inc. Retrieved 6 February 2015. External link in `|website=` (help)
13. "The Jargon File 4.4.8: indent style". Retrieved 2014-03-31.
14. "Formatting Your Source Code". *GNU Coding Standards*.
15. "My Lisp Experiences and the Development of GNU Emacs (Transcript of Richard Stallman's Speech, 28 Oct 2002, at the International Lisp Conference)".
16. McConnell, Steve (2004). *Code Complete: A practical handbook of software construction*. Redmond, WA: Microsoft Press. pp. 746–747. ISBN 0-7356-1967-0.
17. Lammers, Susan (1986). *Programmers at Work*. Microsoft Press. ISBN 0-914845-71-3.
18. Pattee, Jim. "Artistic Style 2.05 Documentation". *Artistic Style*. Retrieved 24 April 2015.
19. "The Haskell 98 Report". *haskell.org*. Retrieved 2016-02-03.
20. Lipovača, Miran. "Making Our Own Types and Typeclasses". *learnyouahaskell.com*. Retrieved 2016-02-03.
21. Linda Lamb, *Learning the vi editor*. O'Reilly

External links

- *C Style: Standards and Guidelines: Defining Programming Standards for Professional C Programmers* (http://syque.com/cstyle/index.htm), Prentice Hall, ISBN 0-13-116898-3 / ISBN 978-0-13-116898-5 (complete text is also on-line). Straker, David (1992).
- Contextual Indent (http://milan.adamovsky.com/2010/08/contextual-indent.html)
- GNU Coding Standards (http://www.gnu.org/prep/standards/standards.html)
- Jargon File article on indent style (http://www.catb.org/jargon/html/I/indent-style.html)
- Source Code Formatters (https://www.dmoz.org/Computers/Programming/Development_Tools/Source_Code_Formatters/) at DMOZ