CC Mode 5.35

A GNU Emacs mode for editing C and C-like languages

Barry A. Warsaw, Martin Stjernholm, Alan Mackenzie

This manual was generated from $Revision$ of cc-mode.texi. It can be downloaded from the URL formed by substituting that 12 digit hexadecimal revision for "<rev>" in http://cc-mode.hg.sourceforge.net/hgweb/cc-mode/cc-mode/raw-file/<rev>/cc-mode.texi.

1 Introduction

Welcome to CC Mode, a GNU Emacs mode for editing files containing C, C++, Objective-C, Java, CORBA IDL (and the variants CORBA PSDL and CIDL), Pike and AWK code. This incarnation of the mode is descended from `c-mode.el` (also called "Boring Old C Mode" or BOCM `:-)`, `c++-mode.el` version 2, which Barry Warsaw had been maintaining since 1992, and `awk-mode.el`, a long neglected mode in the (X)Emacs base.

Late in 1997, Martin Stjernholm joined Barry on the CC Mode Maintainers Team, and implemented the Pike support. In 2000 Martin took over as the sole maintainer. In 2001 Alan Mackenzie joined the team, implementing AWK support in version 5.30. CC Mode did not originally contain the font lock support for its languages — that was added in version 5.30.

This manual describes CC Mode version 5.35.

CC Mode supports the editing of K&R and ANSI C, C++, Objective-C, Java, CORBA's Interface Definition Language, Pike[1] and AWK files. In this way, you can easily set up consistent font locking and coding styles for use in editing all of these languages, although AWK is not yet as uniformly integrated as the other languages.

Note that the name of this package is "CC Mode", but there is no top level `cc-mode` entry point. All of the variables, commands, and functions in CC Mode are prefixed with `c-`*`thing`*, and `c-mode`, `c++-mode`, `objc-mode`, `java-mode`, `idl-mode`, `pike-mode`, and `awk-mode` entry points are provided. This package is intended to be a replacement for `c-mode.el`, `c++-mode.el` and `awk-mode.el`.

A special word of thanks goes to Krishna Padmasola for his work in converting the original `README` file to Texinfo format. I'd also like to thank all the CC Mode victims who help enormously during the early beta stages of CC Mode's development.

[1] A C-like scripting language with its roots in the LPC language used in some MUD engines. See `http://pike.ida.liu.se/`.

2 Overview of the Manual

The manual starts with several introductory chapters (including this one).

The next chunk of the manual describes the day to day *use* of CC Mode (as contrasted with how to customize it).

- The chapter "Commands" describes in detail how to use (nearly) all of CC Mode's features. There are extensive cross-references from here to the corresponding sections later in the manual which tell you how to customize these features.
- "Font Locking" describes how "syntax highlighting" is applied to your buffers. It is mainly background information and can be skipped over at a first reading.

The next chunk of the manual describes how to *customize* CC Mode. Typically, an overview of a topic is given at the chapter level, then the sections and subsections describe the material in increasing detail.

- The chapter "Configuration Basics" tells you *how* to write customizations - whether in hooks, in styles, in both, or in neither, depending on your needs. It describes the CC Mode style system and lists the standard styles that CC Mode supplies.
- The next few chapters describe in detail how to customize the various features of CC Mode.
- Finally, there is a sample `.emacs` fragment, which might help you in creating your own customization.

The manual ends with "this and that", things that don't fit cleanly into any of the previous chunks.

- Two chapters discuss the performance of CC Mode and known bugs/limitations.
- The FAQ contains a list of common problems and questions.
- The next two chapters tell you how to get in touch with the CC Mode project - whether for updating CC Mode or submitting bug reports.

Finally, there are the customary indices.

3 Getting Started

If you got this version of CC Mode with Emacs or XEmacs, it should work just fine right out of the box. Note however that you might not have the latest CC Mode release and might want to upgrade your copy (see below).

You should probably start by skimming through the entire chapter Chapter 4 [Commands], page 5, to get an overview of CC Mode's capabilities.

After trying out some commands, you may dislike some aspects of CC Mode's default configuration. Here is an outline of how to change some of the settings that newcomers to CC Mode most often want to change:

c-basic-offset

This Lisp variable holds an integer, the number of columns CC Mode indents nested code. To set this value to 6, customize `c-basic-offset` or put this into your `.emacs`:

```
(setq c-basic-offset 6)
```

The (indentation) style

The basic "shape" of indentation created by CC Mode—by default, this is `gnu` style (except for Java and AWK buffers). A list of the available styles and their descriptions can be found in Section 6.3.1 [Built-in Styles], page 31. A complete specification of the CC Mode style system, including how to create your own style, can be found in the chapter Section 6.3 [Styles], page 30. To set your style to `linux`, either customize `c-default-style` or put this into your `.emacs`:

```
(setq c-default-style '((java-mode . "java")
                        (awk-mode . "awk")
                        (other . "linux")))
```

Electric Indentation

Normally, when you type "punctuation" characters such as '`;`' or '`{`', CC Mode instantly reindents the current line. This can be disconcerting until you get used to it. To disable *electric indentation* in the current buffer, type *`C-c C-l`*. Type the same thing to enable it again. To have electric indentation disabled by default, put the following into your `.emacs` file[1]:

```
(setq-default c-electric-flag nil)
```

Details of this and other similar "Minor Modes" appear in the section Section 4.5 [Minor Modes], page 11.

Making the `RET` key indent the new line

The standard Emacs binding for `RET` just adds a new line. If you want it to reindent the new line as well, rebind the key. Note that the action of rebinding would fail if the pertinent keymap didn't yet exist—we thus need to delay the action until after CC Mode has been loaded. Put the following code into your `.emacs`:

```
(defun my-make-CR-do-indent ()
  (define-key c-mode-base-map "\C-m" 'c-context-line-break))
```

[1] There is no "easy customization" facility for making this change.

```
(add-hook 'c-initialization-hook 'my-make-CR-do-indent)
```

This example demonstrates the use of a very powerful CC Mode (and Emacs) facility, the hook. The use of CC Mode's hooks is described in Section 6.1 [CC Hooks], page 28.

All these settings should occur in your `.emacs` *before* any CC Mode buffers get loaded—in particular, before any call of `desktop-read`.

As you get to know the mode better, you may want to make more ambitious changes to your configuration. For this, you should start reading the chapter Chapter 6 [Config Basics], page 26.

If you are upgrading an existing CC Mode installation, please see the `README` file for installation details. In particular, if you are going to be editing AWK files, `README` describes how to configure your (X)Emacs so that CC Mode will supersede the obsolete `awk-mode.el` which might have been supplied with your (X)Emacs. CC Mode might not work with older versions of Emacs or XEmacs. See the CC Mode release notes at `http://cc-mode.sourceforge.net` for the latest information on Emacs version and package compatibility (see Appendix E [Updating CC Mode], page 99).

`c-version` [Command]

You can find out what version of CC Mode you are using by visiting a C file and entering *M-x c-version RET*. You should see this message in the echo area:

```
Using CC Mode version 5.XX
```

where '`XX`' is the minor release number.

4 Commands

This chapter specifies all of CC Mode's commands, and thus contains nearly everything you need to know to *use* CC Mode (as contrasted with configuring it). *Commands* here means both control key sequences and *electric keys*, these being characters such as '`;`' which, as well as inserting themselves into the buffer, also do other things.

You might well want to review Section "Moving by Parens" in *GNU Emacs Manual*, which describes commands for moving around brace and parenthesis structures.

4.1 Indentation Commands

The following commands reindent C constructs. Note that when you change your coding style, either interactively or through some other means, your file does *not* automatically get reindented. You will need to execute one of the following commands to see the effects of your changes.

Also, variables like `c-hanging-*` and `c-cleanup-list` (see Chapter 8 [Custom Auto-newlines], page 39) only affect how on-the-fly code is formatted. Changing the "hanginess" of a brace and then reindenting, will not move the brace to a different line. For this, you're better off getting an external program like GNU `indent`, which will rearrange brace location, amongst other things.

Preprocessor directives are handled as syntactic whitespace from other code, i.e. they can be interspersed anywhere without affecting the indentation of the surrounding code, just like comments.

The code inside macro definitions is, by default, still analyzed syntactically so that you get relative indentation there just as you'd get if the same code was outside a macro. However, since there is no hint about the syntactic context, i.e. whether the macro expands to an expression, to some statements, or perhaps to whole functions, the syntactic recognition can be wrong. CC Mode manages to figure it out correctly most of the time, though.

Some macros, when invoked, "have their own semicolon". To get the next line indented correctly, rather than as a continuation line, See Section 12.2 [Macros with ;], page 88.

Reindenting large sections of code can take a long time. When CC Mode reindents a region of code, it is essentially equivalent to hitting `TAB` on every line of the region.

These commands indent code:

`TAB` (`c-indent-command`)

: This command indents the current line. That is all you need to know about it for normal use.

  `c-indent-command` does different things, depending on the setting of `c-syntactic-indentation` (see Chapter 10 [Indentation Engine Basics], page 49):

  - When it's non-`nil` (which it normally is), the command indents the line according to its syntactic context. With a prefix argument (*`C-u TAB`*), it will re-indent the entire expression[1] that begins at the line's left margin.

---

[1] this is only useful for a line starting with a comment opener or an opening brace, parenthesis, or string quote.

- When it's `nil`, the command indents the line by an extra `c-basic-offset` columns. A prefix argument acts as a multiplier. A bare prefix (*C-u TAB*) is equivalent to -1, removing `c-basic-offset` columns from the indentation.

The precise behavior is modified by several variables: With `c-tab-always-indent`, you can make `TAB` insert whitespace in some circumstances—`c-insert-tab-function` then defines precisely what sort of "whitespace" this will be. Set the standard Emacs variable `indent-tabs-mode` to `t` if you want real '`tab`' characters to be used in the indentation, to `nil` if you want only spaces. See Section "Just Spaces" in *GNU Emacs Manual*.

`c-tab-always-indent` [User Option]

This variable modifies how `TAB` operates.

- When it is `t` (the default), `TAB` simply indents the current line.
- When it is `nil`, `TAB` (re)indents the line only if point is to the left of the first non-whitespace character on the line. Otherwise it inserts some whitespace (a tab or an equivalent number of spaces - see below) at point.
- With some other value, the line is reindented. Additionally, if point is within a string or comment, some whitespace is inserted.

`c-insert-tab-function` [User Option]

When "some whitespace" is inserted as described above, what actually happens is that the function stored in `c-insert-tab-function` is called. Normally, this is `insert-tab`, which inserts a real tab character or the equivalent number of spaces (depending on `indent-tabs-mode`). Some people, however, set `c-insert-tab-function` to `tab-to-tab-stop` so as to get hard tab stops when indenting.

The kind of indentation the next five commands do depends on the setting of `c-syntactic-indentation` (see Chapter 10 [Indentation Engine Basics], page 49):

- when it is non-`nil` (the default), the commands indent lines according to their syntactic context;
- when it is `nil`, they just indent each line the same amount as the previous non-blank line. The commands that indent a region aren't very useful in this case.

C-j (`newline-and-indent`)

Inserts a newline and indents the new blank line, ready to start typing. This is a standard (X)Emacs command.

C-M-q (`c-indent-exp`)

Indents an entire balanced brace or parenthesis expression. Note that point must be on the opening brace or parenthesis of the expression you want to indent.

C-c C-q (`c-indent-defun`)

Indents the entire top-level function, class or macro definition encompassing point. It leaves point unchanged. This function can't be used to reindent a nested brace construct, such as a nested class or function, or a Java method.

The top-level construct being reindented must be complete, i.e. it must have both a beginning brace and an ending brace.

`C-M-\` (`indent-region`)
: Indents an arbitrary region of code. This is a standard Emacs command, tailored for C code in a CC Mode buffer. Note, of course, that point and mark must delineate the region you want to indent.

`C-M-h` (`c-mark-function`)
: While not strictly an indentation command, this is useful for marking the current top-level function or class definition as the current region. As with `c-indent-defun`, this command operates on top-level constructs, and can't be used to mark say, a Java method.

These variables are also useful when indenting code:

`indent-tabs-mode` [User Option]
: This is a standard Emacs variable that controls how line indentation is composed. When it's non-`nil`, tabs can be used in a line's indentation, otherwise only spaces are used.

`c-progress-interval` [User Option]
: When indenting large regions of code, this variable controls how often a progress message is displayed. Set this variable to `nil` to inhibit the progress messages, or set it to an integer which is how often (in seconds) progress messages are to be displayed.

4.2 Comment Commands

When the commands in this section add comment delimiters, they use either line comments or block comments depending on the setting of the comment style (see Section 4.5 [Minor Modes], page 11).

`C-c C-c` (`comment-region`)
: This command comments out the lines that start in the region. With a negative argument, it does the opposite - it deletes the comment delimiters from these lines. See Section "Multi-Line Comments" in *GNU Emacs Manual*, for fuller details. `comment-region` isn't actually part of CC Mode - it is given a CC Mode binding for convenience.

`M-;` (`comment-dwim` or `indent-for-comment`[2])
: Insert a comment at the end of the current line, if none is there already. Then reindent the comment according to `comment-column` (see Section "Options for Comments" in *GNU Emacs Manual*) and the variables below. Finally, position the point after the comment starter. `C-u M-;` kills any comment on the current line, together with any whitespace before it. This is a standard Emacs command, but CC Mode enhances it a bit with two variables:

    `c-indent-comment-alist` [User Option]
    : This style variable allows you to vary the column that `M-;` puts the comment at, depending on what sort of code is on the line, and possibly the

[2] The name of this command varies between (X)Emacs versions.

indentation of any similar comment on the preceding line. It is an association list that maps different types of lines to actions describing how they should be handled. If a certain line type isn't present on the list then the line is indented to the column specified by `comment-column`.

See the documentation string for a full description of this variable (use *C-h v c-indent-comment-alist*).

`c-indent-comments-syntactically-p` [User Option]

Normally, when this style variable is `nil`, *M-;* will indent comment-only lines according to `c-indent-comment-alist`, just as it does with lines where other code precede the comments. However, if you want it to act just like `TAB` for comment-only lines you can get that by setting `c-indent-comments-syntactically-p` to non-`nil`.

If `c-indent-comments-syntactically-p` is non-`nil` then `c-indent-comment-alist` won't be consulted at all for comment-only lines.

4.3 Movement Commands

CC Mode contains some useful commands for moving around in C code.

C-M-a (`c-beginning-of-defun`)
C-M-e (`c-end-of-defun`)

Move to the beginning or end of the current or next function. Other constructs (such as a structs or classes) which have a brace block also count as "functions" here. To move over several functions, you can give these commands a repeat count.

The start of a function is at its header. The end of the function is after its closing brace, or after the semicolon of a construct (such as a `struct`) which doesn't end at the brace. These two commands try to leave point at the beginning of a line near the actual start or end of the function. This occasionally causes point not to move at all.

By default, these commands will recognize functions contained within a *declaration scope* such as a C++ `class` or `namespace` construct, should the point start inside it. If CC Modefails to find function beginnings or ends inside the current declaration scope, it will search the enclosing scopes. If you want CC Modeto recognize functions only at the top level[3], set `c-defun-tactic` to `t`.

These functions are analogous to the Emacs built-in commands `beginning-of-defun` and `end-of-defun`, except they eliminate the constraint that the top-level opening brace of the defun must be in column zero. See Section "Defuns" in *GNU Emacs Manual*, for more information.

C-M-a (AWK Mode) (`c-awk-beginning-of-defun`)
C-M-e (AWK Mode) (`c-awk-end-of-defun`)

Move to the beginning or end of the current or next AWK defun. These commands can take prefix-arguments, their functionality being entirely equivalent to `beginning-of-defun` and `end-of-defun`.

[3] this was CC Mode's behavior prior to version 5.32.

AWK Mode *defuns* are either pattern/action pairs (either of which might be implicit) or user defined functions. Having the '`{`' and '`}`' (if there are any) in column zero, as is suggested for some modes, is neither necessary nor helpful in AWK mode.

M-a (`c-beginning-of-statement`)
M-e (`c-end-of-statement`)
: Move to the beginning or end of the innermost C statement. If point is already there, move to the next beginning or end of a statement, even if that means moving into a block. (Use *C-M-b* or *C-M-f* to move over a balanced block.) A prefix argument *n* means move over *n* statements.

  If point is within or next to a comment or a string which spans more than one line, these commands move by sentences instead of statements.

  When called from a program, these functions take three optional arguments: the repetition count, a buffer position limit which is the farthest back to search for the syntactic context, and a flag saying whether to do sentence motion in or near comments and multiline strings.

C-c C-u (`c-up-conditional`)
: Move back to the containing preprocessor conditional, leaving the mark behind. A prefix argument acts as a repeat count. With a negative argument, move forward to the end of the containing preprocessor conditional.

  '`#elif`' is treated like '`#else`' followed by '`#if`', so the function stops at them when going backward, but not when going forward.

  This key sequence is not bound in AWK Mode, which doesn't have preprocessor statements.

M-x c-up-conditional-with-else
: A variety of `c-up-conditional` that also stops at '`#else`' lines. Normally those lines are ignored.

M-x c-down-conditional
: Move forward into the next nested preprocessor conditional, leaving the mark behind. A prefix argument acts as a repeat count. With a negative argument, move backward into the previous nested preprocessor conditional.

  '`#elif`' is treated like '`#else`' followed by '`#if`', so the function stops at them when going forward, but not when going backward.

M-x c-down-conditional-with-else
: A variety of `c-down-conditional` that also stops at '`#else`' lines. Normally those lines are ignored.

C-c C-p (`c-backward-conditional`)
C-c C-n (`c-forward-conditional`)
: Move backward or forward across a preprocessor conditional, leaving the mark behind. A prefix argument acts as a repeat count. With a negative argument, move in the opposite direction.

  These key sequences are not bound in AWK Mode, which doesn't have preprocessor statements.

M-x c-backward-into-nomenclature
M-x c-forward-into-nomenclature

A popular programming style, especially for object-oriented languages such as C++ is to write symbols in a mixed case format, where the first letter of each word is capitalized, and not separated by underscores. E.g. '`SymbolsWithMixedCaseAndNoUnderlines`'.

These commands move backward or forward to the beginning of the next capitalized word. With prefix argument *n*, move *n* times. If *n* is negative, move in the opposite direction.

Note that these two commands have been superseded by `c-subword-mode`, which you should use instead. See Section 4.9 [Subword Movement], page 17. They might be removed from a future release of CC Mode.

4.4 Filling and Line Breaking Commands

Since there's a lot of normal text in comments and string literals, CC Mode provides features to edit these like in text mode. The goal is to do it seamlessly, i.e. you can use auto fill mode, sentence and paragraph movement, paragraph filling, adaptive filling etc. wherever there's a piece of normal text without having to think much about it. CC Mode keeps the indentation, fixes suitable comment line prefixes, and so on.

You can configure the exact way comments get filled and broken, and where Emacs does auto-filling (see see Chapter 7 [Custom Filling and Breaking], page 36). Typically, the style system (see Section 6.3 [Styles], page 30) will have set this up for you, so you probably won't have to bother.

Line breaks are by default handled (almost) the same regardless of whether they are made by auto fill mode (see Section "Auto Fill" in *GNU Emacs Manual*), by paragraph filling (e.g. with *M-q*), or explicitly with *M-j* or similar methods. In string literals, the new line gets the same indentation as the previous nonempty line.[4].

M-q (`c-fill-paragraph`)

This command fills multiline string literals and both block and line style comments. In Java buffers, the Javadoc markup words are recognized as paragraph starters. The line oriented Pike autodoc markup words are recognized in the same way in Pike mode.

The formatting of the starters (`/*`) and enders (`*/`) of block comments are kept as they were before the filling. I.e., if either the starter or ender were on a line of its own, then it stays on its own line; conversely, if the delimiter has comment text on its line, it keeps at least one word of that text with it on the line.

This command is the replacement for `fill-paragraph` in CC Mode buffers.

M-j (`c-indent-new-comment-line`)

This breaks the current line at point and indents the new line. If point was in a comment, the new line gets the proper comment line prefix. If point was inside a macro, a backslash is inserted before the line break. It is the replacement for `indent-new-comment-line`.

[4] You can change this default by setting the `string` syntactic symbol (see Section 10.2 [Syntactic Symbols], page 51, and see Chapter 11 [Customizing Indentation], page 68)

M-x c-context-line-break
Insert a line break suitable to the context: If the point is inside a comment, the new line gets the suitable indentation and comment line prefix like `c-indent-new-comment-line`. In normal code it's indented like `newline-and-indent` would do. In macros it acts like `newline-and-indent` but additionally inserts and optionally aligns the line ending backslash so that the macro remains unbroken. See Chapter 12 [Custom Macros], page 87, for details about the backslash alignment. In a string, a backslash is inserted only if the string is within a macro[5].

This function is not bound to a key by default, but it's intended to be used on the *RET* key. If you like the behavior of `newline-and-indent` on *RET*, you should consider switching to this function. See Appendix A [Sample .emacs File], page 92.

M-x c-context-open-line
This is to *C-o* (*M-x open-line*) as `c-context-line-break` is to *RET*. I.e. it works just like `c-context-line-break` but leaves the point before the inserted line break.

4.5 Minor Modes

CC Mode contains several minor-mode-like features that you might find useful while writing new code or editing old code:

comment style
This specifies whether comment commands (such as *M-;*) insert block comments or line comments[6].

electric mode
When this is enabled, certain visible characters cause reformatting as they are typed. This is normally helpful, but can be a nuisance when editing chaotically formatted code. It can also be disconcerting, especially for users who are new to CC Mode.

auto-newline mode
This automatically inserts newlines where you'd probably want to type them yourself, e.g. after typing '`}`'s. Its action is suppressed when electric mode is disabled.

hungry-delete mode
This lets you delete a contiguous block of whitespace with a single key - for example, the newline and indentation just inserted by auto-newline when you want to back up and write a comment after the last statement.

subword mode
This mode makes basic word movement commands like *M-f* (`forward-word`) and *M-b* (`backward-word`) treat the parts of sillycapsed symbols as different

[5] In GCC, unescaped line breaks within strings are valid.

[6] You can emphasize non-default style comments in your code by giving their delimiters `font-lock-warning-face`. See Section 5.4 [Wrong Comment Style], page 23.

words. E.g. '`NSGraphicsContext`' is treated as three words '`NS`', '`Graphics`', and '`Context`'.

syntactic-indentation mode
: When this is enabled (which it normally is), indentation commands such as *`C-j`* indent lines of code according to their syntactic structure. Otherwise, a line is simply indented to the same level as the previous one and `TAB` adjusts the indentation in steps of 'c-basic-offset'.

Full details on how these minor modes work are at Section 4.6 [Electric Keys], page 13, Section 4.7 [Auto-newlines], page 15, Section 4.8 [Hungry WS Deletion], page 15, Section 4.9 [Subword Movement], page 17, and Chapter 10 [Indentation Engine Basics], page 49.

You can toggle each of these minor modes on and off, and you can configure CC Mode so that it starts up with your favorite combination of them (see Appendix A [Sample .emacs File], page 92). By default, when you initialize a buffer, the comment style is set to the default for the major mode, electric mode and syntactic-indentation mode are enabled, but the other two modes are disabled.

CC Mode displays the current state of the first five of these minor modes on the mode line by appending characters to the major mode's name: '`/`' or '`*`' to indicate the comment style (respectively line or block), and one letter for each of the other minor modes which is enabled - '`l`' for electric mode, '`a`' for auto-newline mode, '`h`' for hungry delete mode, and '`w`' for subword mode. If the comment style was block and all the other modes were enabled, you'd see '`C/*lahw`'[7].

Here are the commands to toggle these modes:

`C-c C-k` (`c-toggle-comment-style`)
: Toggle the comment style between line style and block style. In modes (such as AWK Mode) which only have one of these styles, this function does nothing.

`C-c C-l` (`c-toggle-electric-state`)
: Toggle electric minor mode. When the command turns the mode off, it also suppresses auto-newline mode.

`C-c C-a` (`c-toggle-auto-newline`)
: Toggle auto-newline minor mode. When the command turns the mode on, it also enables electric minor mode.

`M-x c-toggle-hungry-state`[8]
: Toggle hungry-delete minor mode.

`M-x c-toggle-auto-hungry-state`[9]
: Toggle both auto-newline and hungry delete minor modes.

`C-c C-w` (`M-x c-subword-mode`)
: Toggle subword mode.

---

[7] The '`C`' would be replaced with the name of the language in question for the other languages CC Mode supports.

[8] Prior to CC Mode 5.31, this command was bound to *`C-c C-d`*.

[9] Prior to CC Mode 5.31, this command was bound to *`C-c C-t`*.

M-x c-toggle-syntactic-indentation
: Toggle syntactic-indentation mode.

Common to all the toggle functions above is that if they are called programmatically, they take an optional numerical argument. For `c-toggle-comment style`, a positive value will select block comments, a negative value will select line comments. For the other functions, a positive value will turn on the minor mode (or both of them in the case of `c-toggle-auto-hungry-state`) and a negative value will turn it (or them) off.

4.6 Electric Keys and Keywords

Most punctuation keys provide *electric* behavior - as well as inserting themselves they perform some other action, such as reindenting the line. This reindentation saves you from having to reindent a line manually after typing, say, a '`}`'. A few keywords, such as `else`, also trigger electric action.

You can inhibit the electric behavior described here by disabling electric minor mode (see Section 4.5 [Minor Modes], page 11).

Common to all these keys is that they only behave electrically when used in normal code (as contrasted with getting typed in a string literal or comment). Those which cause re-indentation do so only when `c-syntactic-indentation` has a non-`nil` value (which it does by default).

These keys and keywords are:

`#`
: Pound (bound to `c-electric-pound`) is electric when typed as the first non-whitespace character on a line and not within a macro definition. In this case, the variable `c-electric-pound-behavior` is consulted for the electric behavior. This variable takes a list value, although the only element currently defined is `alignleft`, which tells this command to force the '`#`' character into column zero. This is useful for entering preprocessor macro definitions.

  Pound is not electric in AWK buffers, where '`#`' starts a comment, and is bound to `self-insert-command` like any typical printable character.

`*`
`/`
: A star (bound to `c-electric-star`) or a slash (`c-electric-slash`) causes reindentation when you type it as the second component of a C style block comment opener ('`/*`') or a C++ line comment opener ('`//`') respectively, but only if the comment opener is the first thing on the line (i.e. there's only whitespace before it).

  Additionally, you can configure CC Mode so that typing a slash at the start of a line within a block comment will terminate the comment. You don't need to have electric minor mode enabled to get this behavior. See Chapter 9 [Clean-ups], page 45.

  In AWK mode, '`*`' and '`/`' do not delimit comments and are not electric.

`<`
`>`
: A less-than or greater-than sign (bound to `c-electric-lt-gt`) is electric in two circumstances: when it is an angle bracket in a C++ '`template`' declaration (and similar constructs in other languages) and when it is the second of two

`<` or `>` characters in a C++ style stream operator. In either case, the line is reindented. Angle brackets in C '`#include`' directives are not electric.

`(`
`)`
: The normal parenthesis characters '`(`' and '`)`' (bound to `c-electric-paren`) reindent the current line. This is useful for getting the closing parenthesis of an argument list aligned automatically.

  You can also configure CC Mode to insert a space automatically between a function name and the '`(`' you've just typed, and to remove it automatically after typing '`)`', should the argument list be empty. You don't need to have electric minor mode enabled to get these actions. See Chapter 9 [Clean-ups], page 45.

`{`
`}`
: Typing a brace (bound to `c-electric-brace`) reindents the current line. Also, one or more newlines might be inserted if auto-newline minor mode is enabled. See Section 4.7 [Auto-newlines], page 15. Additionally, you can configure CC Mode to compact excess whitespace inserted by auto-newline mode in certain circumstances. See Chapter 9 [Clean-ups], page 45.

`:`
: Typing a colon (bound to `c-electric-colon`) reindents the current line. Additionally, one or more newlines might be inserted if auto-newline minor mode is enabled. See Section 4.7 [Auto-newlines], page 15. If you type a second colon immediately after such an auto-newline, by default the whitespace between the two colons is removed, leaving a C++ scope operator. See Chapter 9 [Clean-ups], page 45.

  If you prefer, you can insert '`::`' in a single operation, avoiding all these spurious reindentations, newlines, and clean-ups. See Section 4.10 [Other Commands], page 18.

`;`
`,`
: Typing a semicolon or comma (bound to `c-electric-semi&comma`) reindents the current line. Also, a newline might be inserted if auto-newline minor mode is enabled. See Section 4.7 [Auto-newlines], page 15. Additionally, you can configure CC Mode so that when auto-newline has inserted whitespace after a '`}`', it will be removed again when you type a semicolon or comma just after it. See Chapter 9 [Clean-ups], page 45.

`c-electric-continued-statement` [Command]

Certain keywords are electric, causing reindentation when they are preceded only by whitespace on the line. The keywords are those that continue an earlier statement instead of starting a new one: `else`, `while`, `catch` (only in C++ and Java) and `finally` (only in Java).

An example:

```
for (i = 0; i < 17; i++)
  if (a[i])
    res += a[i]->offset;
else
```

Here, the `else` should be indented like the preceding `if`, since it continues that statement. CC Mode will automatically reindent it after the `else` has been typed in full, since only then is it possible to decide whether it's a new statement or a continuation of the preceding `if`.

CC Mode uses Abbrev mode (see Section "Abbrevs" in *GNU Emacs Manual*) to accomplish this. It's therefore turned on by default in all language modes except IDL mode, since CORBA IDL doesn't have any statements.

4.7 Auto-newline Insertion

When you have *Auto-newline minor mode* enabled (see Section 4.5 [Minor Modes], page 11), CC Mode inserts newlines for you automatically (in certain syntactic contexts) when you type a left or right brace, a colon, a semicolon, or a comma. Sometimes a newline appears before the character you type, sometimes after it, sometimes both.

Auto-newline only triggers when the following conditions hold:

- Auto-newline minor mode is enabled, as evidenced by the indicator '`a`' after the mode name on the mode line (e.g. '`C/a`' or '`C/la`').
- The character was typed at the end of a line, or with only whitespace after it, and possibly a '\' escaping the newline.
- The character is not on its own line already. (This applies only to insertion of a newline *before* the character.)
- The character was not typed inside of a literal[10].
- No numeric argument was supplied to the command (i.e. it was typed as normal, with no *C-u* prefix).

You can configure the precise circumstances in which newlines get inserted (see see Chapter 8 [Custom Auto-newlines], page 39). Typically, the style system (see Section 6.3 [Styles], page 30) will have set this up for you, so you probably won't have to bother.

Sometimes CC Mode inserts an auto-newline where you don't want one, such as after a '`}`' when you're about to type a '`;`'. Hungry deletion can help here (see Section 4.8 [Hungry WS Deletion], page 15), or you can activate an appropriate *clean-up*, which will remove the excess whitespace after you've typed the '`;`'. See Chapter 9 [Clean-ups], page 45, for a full description. See also Section 4.6 [Electric Keys], page 13, for a summary of clean-ups listed by key.

4.8 Hungry Deletion of Whitespace

If you want to delete an entire block of whitespace at point, you can use *hungry deletion*. This deletes all the contiguous whitespace either before point or after point in a single operation. "Whitespace" here includes tabs and newlines, but not comments or preprocessor commands. Hungry deletion can markedly cut down on the number of times you have to hit deletion keys when, for example, you've made a mistake on the preceding line and have already pressed *C-j*.

[10] A *literal* is defined as any comment, string, or preprocessor macro definition. These constructs are also known as *syntactic whitespace* since they are usually ignored when scanning C code.

Hungry deletion is a simple feature that some people find extremely useful. In fact, you might find yourself wanting it in **all** your editing modes!

Loosely speaking, in what follows, `DEL` means "the backspace key" and `DELETE` means "the forward delete key". This is discussed in more detail below.

There are two different ways you can use hungry deletion:

Using *Hungry Delete Mode* with `DEL` and *C-d*

Here you toggle Hungry Delete minor mode with *M-x c-toggle-hungry-state*[11] (see Section 4.5 [Minor Modes], page 11.) This makes `DEL` and *C-d* do backwards and forward hungry deletion.

`DEL` (`c-electric-backspace`)

This command is run by default when you hit the *DEL* key. When hungry delete mode is enabled, it deletes any amount of whitespace in the backwards direction. Otherwise, or when used with a prefix argument or in a literal (see Section 4.7 [Auto-newlines], page 15), the command just deletes backwards in the usual way. (More precisely, it calls the function contained in the variable `c-backspace-function`, passing it the prefix argument, if any.)

`c-backspace-function`

Hook that gets called by `c-electric-backspace` when it doesn't do an "electric" deletion of the preceding whitespace. The default value is `backward-delete-char-untabify` (see Section "Deletion" in *GNU Emacs Lisp Reference Manual*, the function which deletes a single character.

C-d (`c-electric-delete-forward`)

This function, which is bound to *C-d* by default, works just like `c-electric-backspace` but in the forward direction. When it doesn't do an "electric" deletion of the following whitespace, it just does `delete-char`, more or less. (Strictly speaking, it calls the function in `c-delete-function` with the prefix argument.)

`c-delete-function`

Hook that gets called by `c-electric-delete-forward` when it doesn't do an "electric" deletion of the following whitespace. The default value is `delete-char`.

Using Distinct Bindings

The other (newer and recommended) way to use hungry deletion is to perform `c-hungry-delete-backwards` and `c-hungry-delete-forward` directly through their key sequences rather than using the minor mode toggling.

C-c C-DEL, or *C-c DEL* (`c-hungry-delete-backwards`)[12]

Delete any amount of whitespace in the backwards direction (regardless whether hungry-delete mode is enabled or not). This command is bound to both *C-c C-DEL* and *C-c DEL*, since the more

---

[11] Prior to CC Mode 5.31, this command was bound to *C-c C-d*. *C-c C-d* is now the default binding for `c-hungry-delete-forward`.

[12] This command was formerly known as `c-hungry-backspace`.

natural one, *C-c C-DEL*, is sometimes difficult to type at a character terminal.

C-c C-d, *C-c C-DELETE*, or *C-c DELETE* (`c-hungry-delete-forward`)
: Delete any amount of whitespace in the forward direction (regardless whether hungry-delete mode is enabled or not). This command is bound to both *C-c C-DELETE* and *C-c DELETE* for the same reason as for `DEL` above.

When we talk about `DEL`, and `DELETE` above, we actually do so without connecting them to the physical keys commonly known as `Backspace` and `Delete`. The default bindings to those two keys depends on the flavor of (X)Emacs you are using.

In XEmacs 20.3 and beyond, the `Backspace` key is bound to `c-electric-backspace` and the `Delete` key is bound to `c-electric-delete`. You control the direction it deletes in by setting the variable `delete-key-deletes-forward`, a standard XEmacs variable. When this variable is non-`nil`, `c-electric-delete` will do forward deletion with `c-electric-delete-forward`, otherwise it does backward deletion with `c-electric-backspace`. Similarly, *C-c Delete* and *C-c C-Delete* are bound to `c-hungry-delete` which is controlled in the same way by `delete-key-deletes-forward`.

Emacs 21 and later automatically binds `Backspace` and `Delete` to *DEL* and *C-d* according to your environment, and CC Mode extends those bindings to *C-c C-Backspace* etc. If you need to change the bindings through `normal-erase-is-backspace-mode` then CC Mode will also adapt its extended bindings accordingly.

In earlier (X)Emacs versions, CC Mode doesn't bind either `Backspace` or `Delete` directly. Only the key codes *DEL* and *C-d* are bound, and it's up to the default bindings to map the physical keys to them. You might need to modify this yourself if the defaults are unsuitable.

Getting your `Backspace` and `Delete` keys properly set up can sometimes be tricky. The information in Section "DEL Does Not Delete" in *GNU Emacs Manual*, might be helpful if you're having trouble with this in GNU Emacs.

4.9 Subword Movement and Editing

In spite of the GNU Coding Standards, it is popular to name a symbol by mixing uppercase and lowercase letters, e.g. '`GtkWidget`', '`EmacsFrameClass`', or '`NSGraphicsContext`'. Here we call these mixed case symbols *nomenclatures*. Also, each capitalized (or completely uppercase) part of a nomenclature is called a *subword*. Here are some examples:

Nomenclature	**Subwords**
'`GtkWindow`'	'`Gtk`' and '`Window`'
'`EmacsFrameClass`'	'`Emacs`', '`Frame`', and '`Class`'
'`NSGraphicsContext`'	'`NS`', '`Graphics`', and '`Context`'

The subword minor mode replaces the basic word oriented movement and editing commands with variants that recognize subwords in a nomenclature and treat them as separate words:

Key	**Word oriented command**	**Subword oriented command**
M-f	`forward-word`	`c-forward-subword`

M-b	`backward-word`	`c-backward-subword`
M-@	`mark-word`	`c-mark-subword`
M-d	`kill-word`	`c-kill-subword`
M-DEL	`backward-kill-word`	`c-backward-kill-subword`
M-t	`transpose-words`	`c-transpose-subwords`
M-c	`capitalize-word`	`c-capitalize-subword`
M-u	`upcase-word`	`c-upcase-subword`
M-l	`downcase-word`	`c-downcase-subword`

Note that if you have changed the key bindings for the word oriented commands in your `.emacs` or a similar place, the keys you have configured are also used for the corresponding subword oriented commands.

Type *C-c C-w* to toggle subword mode on and off. To make the mode turn on automatically, put the following code in your `.emacs`:

```
(add-hook 'c-mode-common-hook
          (lambda () (c-subword-mode 1)))
```

As a bonus, you can also use `c-subword-mode` in non-CC Mode buffers by typing *M-x c-subword-mode*.

4.10 Other Commands

Here are the various other commands that didn't fit anywhere else:

C-c . (`c-set-style`)
: Switch to the specified style in the current buffer. Use like this:

  C-c . style-name RET

  You can use the `TAB` in the normal way to do completion on the style name. Note that all style names are case insensitive, even the ones you define yourself.

  Setting a style in this way does *not* automatically reindent your file. For commands that you can use to view the effect of your changes, see Section 4.1 [Indentation Commands], page 5, and Section 4.4 [Filling and Breaking], page 10.

  For details of the CC Mode style system, see Section 6.3 [Styles], page 30.

C-c : (`c-scope-operator`)
: In C++, it is also sometimes desirable to insert the double-colon scope operator without performing the electric behavior of colon insertion. *C-c :* does just this.

C-c C-z (`c-display-defun-name`)
: Display the current function name, if any, in the minibuffer. Additionally, if a prefix argument is given, push the function name to the kill ring. If there is no current function, `c-display-defun-name` does nothing. In Emacs, you can use this command in the middle of an interactive search if you set the customizable option `isearch-allow-scoll` to non-`nil`. See Section "Not Exiting Isearch" in *GNU Emacs Manual*.

C-c C- (`c-backslash-region`)
: This function inserts and aligns or deletes end-of-line backslashes in the current region. These are typically used in multi-line macros.

With no prefix argument, it inserts any missing backslashes and aligns them according to the `c-backslash-column` and `c-backslash-max-column` variables. With a prefix argument, it deletes any backslashes.

The function does not modify blank lines at the start of the region. If the region ends at the start of a line, it always deletes the backslash (if any) at the end of the previous line.

To customize the precise workings of this command, Chapter 12 [Custom Macros], page 87.

The recommended line breaking function, `c-context-line-break` (see Section 4.4 [Filling and Breaking], page 10), is especially nice if you edit multiline macros frequently. When used inside a macro, it automatically inserts and adjusts the mandatory backslash at the end of the line to keep the macro together, and it leaves the point at the right indentation column for the code. Thus you can write code inside macros almost exactly as you can elsewhere, without having to bother with the trailing backslashes.

C-c C-e (`c-macro-expand`)

This command expands C, C++, Objective C or Pike macros in the region, using an appropriate external preprocessor program. Normally it displays its output in a temporary buffer, but if you give it a prefix arg (with *C-u C-c C-e*) it will overwrite the original region with the expansion.

The command does not work in any of the other modes, and the key sequence is not bound in these other modes.

`c-macro-expand` isn't actually part of CC Mode, even though it is bound to a CC Mode key sequence. If you need help setting it up or have other problems with it, you can either read its source code or ask for help in the standard (X)Emacs forums.

5 Font Locking

CC Mode provides font locking for its supported languages by supplying patterns for use with Font Lock mode. This means that you get distinct faces on the various syntactic parts such as comments, strings, keywords and types, which is very helpful in telling them apart at a glance and discovering syntactic errors. See Section “Font Lock” in *GNU Emacs Manual*, for ways to enable font locking in CC Mode buffers.

Please note: The font locking in AWK mode is currently not integrated with the rest of CC Mode. Only the last section of this chapter, Section 5.6 [AWK Mode Font Locking], page 24, applies to AWK. The other sections apply to the other languages.

5.1 Font Locking Preliminaries

The font locking for most of the CC Mode languages were provided directly by the Font Lock package prior to version 5.30 of CC Mode. In the transition to CC Mode the patterns have been reworked completely and are applied uniformly across all the languages except AWK mode, just like the indentation rules (although each language still has some peculiarities of its own, of course). Since the languages previously had completely separate font locking patterns, this means that it’s a bit different in most languages now.

The main goal for the font locking in CC Mode is accuracy, to provide a dependable aid in recognizing the various constructs. Some, like strings and comments, are easy to recognize while others, like declarations and types, can be very tricky. CC Mode can go to great lengths to recognize declarations and casts correctly, especially when the types aren’t recognized by standard patterns. This is a fairly demanding analysis which can be slow on older hardware, and it can therefore be disabled by choosing a lower decoration level with the variable `font-lock-maximum-decoration` (see Section “Font Lock” in *GNU Emacs Manual*).

The decoration levels are used as follows:

1. Minimal font locking: Fontify only comments, strings and preprocessor directives (in the languages that use cpp).
2. Fast font locking: In addition to level 1, fontify keywords, simple types and declarations that are easy to recognize. The variables `*-font-lock-extra-types` (where ‘`*`’ is the name of the language) are used to recognize types (see below). Documentation comments like Javadoc are fontified according to `c-doc-comment-style` (see Section 5.3 [Doc Comments], page 22).

   Use this if you think the font locking is too slow. It’s the closest corresponding level to level 3 in the old font lock patterns.
3. Accurate font locking: Like level 2 but uses a different approach that can recognize types and declarations much more accurately. The `*-font-lock-extra-types` variables are still used, but user defined types are recognized correctly anyway in most cases. Therefore those variables should be fairly restrictive and not contain patterns that are uncertain.

   This level is designed for fairly modern hardware and a font lock support mode like Lazy Lock or Just-in-time Lock mode that only fontifies the parts that are actually shown. Fontifying the whole buffer at once can easily get bothersomely slow even on contemporary hardware.

Since user defined types are hard to recognize you can provide additional regexps to match those you use:

`c-font-lock-extra-types` [User Option]
`c++-font-lock-extra-types` [User Option]
`objc-font-lock-extra-types` [User Option]
`java-font-lock-extra-types` [User Option]
`idl-font-lock-extra-types` [User Option]
`pike-font-lock-extra-types` [User Option]

> For each language there's a variable `*-font-lock-extra-types`, where '`*`' stands for the language in question. It contains a list of regexps that matches identifiers that should be recognized as types, e.g. '`\\sw+_t`' to recognize all identifiers ending with '`_t`' as is customary in C code. Each regexp should not match more than a single identifier.
>
> The default values contain regexps for many types in standard runtime libraries that are otherwise difficult to recognize, and patterns for standard type naming conventions like the '`_t`' suffix in C and C++. Java, Objective-C and Pike have as a convention to start class names with capitals, so there are patterns for that in those languages.
>
> Despite the names of these variables, they are not only used for fontification but in other places as well where CC Mode needs to recognize types.

5.2 Faces

CC Mode attempts to use the standard faces for programming languages in accordance with their intended purposes as far as possible. No extra faces are currently provided, with the exception of a replacement face `c-invalid-face` for emacsen that don't provide `font-lock-warning-face`.

- Normal comments are fontified in `font-lock-comment-face`.
- Comments that are recognized as documentation (see Section 5.3 [Doc Comments], page 22) get `font-lock-doc-face` (Emacs) or `font-lock-doc-string-face` (XEmacs) if those faces exist. If they don't then `font-lock-comment-face` is used.
- String and character literals are fontified in `font-lock-string-face`.
- Keywords are fontified with `font-lock-keyword-face`.
- `font-lock-function-name-face` is used for function names in declarations and definitions, and classes in those contexts. It's also used for preprocessor defines with arguments.
- Variables in declarations and definitions, and other identifiers in such variable contexts, get `font-lock-variable-name-face`. It's also used for preprocessor defines without arguments.
- Builtin constants are fontified in `font-lock-constant-face` if it exists, `font-lock-reference-face` otherwise. As opposed to the preceding two faces, this is used on the names in expressions, and it's not used in declarations, even if there happen to be a '`const`' in them somewhere.
- `font-lock-type-face` is put on types (both predefined and user defined) and classes in type contexts.

- Label identifiers get `font-lock-constant-face` if it exists, `font-lock-reference-face` otherwise.
- Name qualifiers and identifiers for scope constructs are fontified like labels.
- Special markup inside documentation comments are also fontified like labels.
- Preprocessor directives get `font-lock-preprocessor-face` if it exists (i.e. XEmacs). In Emacs they get `font-lock-builtin-face` or `font-lock-reference-face`, for lack of a closer equivalent.
- Some kinds of syntactic errors are fontified with `font-lock-warning-face` in Emacs. In older XEmacs versions there's no corresponding standard face, so there a special `c-invalid-face` is used, which is defined to stand out sharply by default.

  Note that it's not used for '`#error`' or '`#warning`' directives, since those aren't syntactic errors in themselves.

5.3 Documentation Comments

There are various tools to supply documentation in the source as specially structured comments, e.g. the standard Javadoc tool in Java. CC Mode provides an extensible mechanism to fontify such comments and the special markup inside them.

`c-doc-comment-style` [User Option]

> This is a style variable that specifies which documentation comment style to recognize, e.g. `javadoc` for Javadoc comments.
>
> The value may also be a list of styles, in which case all of them are recognized simultaneously (presumably with markup cues that don't conflict).
>
> The value may also be an association list to specify different comment styles for different languages. The symbol for the major mode is then looked up in the alist, and the value of that element is interpreted as above if found. If it isn't found then the symbol 'other' is looked up and its value is used instead.
>
> The default value for `c-doc-comment-style` is `((java-mode . javadoc) (pike-mode . autodoc) (c-mo`
>
> Note that CC Mode uses this variable to set other variables that handle fontification etc. That's done at mode initialization or when you switch to a style which sets this variable. Thus, if you change it in some other way, e.g. interactively in a CC Mode buffer, you will need to do *`M-x java-mode`* (or whatever mode you're currently using) to reinitialize.
>
> Note also that when CC Mode starts up, the other variables are modified before the mode hooks are run. If you change this variable in a mode hook, you'll have to call `c-setup-doc-comment-style` afterwards to redo that work.

CC Mode currently provides handing of the following doc comment styles:

`javadoc` Javadoc comments, the standard tool in Java.

`autodoc` For Pike autodoc markup, the standard in Pike.

`gtkdoc` For GtkDoc markup, widely used in the Gnome community.

`doxygen` For Doxygen markup, which can be used with C, C++, Java and variety of other languages.

The above is by no means complete. If you'd like to see support for other doc comment styles, please let us know (see Appendix F [Mailing Lists and Bug Reports], page 100).

You can also write your own doc comment fontification support to use with `c-doc-comment-style`: Supply a variable or function `*-font-lock-keywords` where '`*`' is the name you want to use in `c-doc-comment-style`. If it's a variable, it's prepended to `font-lock-keywords`. If it's a function, it's called at mode initialization and the result is prepended. For an example, see `javadoc-font-lock-keywords` in `cc-fonts.el`. It is even possible, to a limited extent, to fontify constructs inside a doc comment with other faces. For an example, see pike autodoc comment style towards the end of `cc-fonts-el`.

If you add support for another doc comment style, please consider contributing it - send a note to `bug-cc-mode@gnu.org`.

5.4 Marking "Wrong" style comments

Most languages supported by CC Mode have two styles of comments, namely block comments and line comments. Your project may have such a strong preference for one of them, that you wish "wrong" style comments to be clearly marked.

You can get CC Mode to do this by setting the default comment style, if necessary, (see Section 4.5 [Minor Modes], page 11) and setting the customizable option `c-mark-wrong-style-of-comment` to non-`nil`.

`c-mark-wrong-style-of-comment` [Variable]

> When this customizable option is non-`nil`, comment delimiters which aren't of the default style will be fontified with `font-lock-warning-face`.

5.5 Miscellaneous Font Locking

In some languages, particularly in C++, there are constructs which are syntactically ambiguous—they could be either declarations or expressions, and CC Mode cannot tell for sure which. Often such a construct is one of the operators '`*`' or '`&`' surrounded by two identifiers.

Experience shows that very often when such a construct is a declaration it will be written with the operator touching exactly one of the identifiers, like:

```
foo *bar
```

or

```
foo& bar
```

. Whether such code is fontified depends on the setting of `c-asymmetry-fontification-flag`.

`c-asymmetry-fontification-flag` [Variable]

> When `c-asymmetry-fontification-flag` is non-nil (which it is by default), code like the above, with white space either before or after the operator, but not both, is fontified as a declaration. When the variable is nil, such a construct gets the default face.

When the construct is an expression there will often be white space both before and after the operator or there will be no white space around it at all, like:

```
foo * bar
```

or

```
foo&bar
```

.

Such code is not fontified as a declaration. (Typically, the identifiers don't get a non-default face.)

For clarity's sake, we emphasize that the "asymmetry" rule in this section only applies when CC Mode cannot disambiguate a construct in any other way.

5.6 AWK Mode Font Locking

The general appearance of font-locking in AWK mode is much like in any other programming mode. See Section "Faces For Font Lock" in *GNU Emacs Lisp Reference Manual*.

The following faces are, however, used in a non-standard fashion in AWK mode:

`font-lock-variable-name-face`
: This face was intended for variable declarations. Since variables are not declared in AWK, this face is used instead for AWK system variables (such as `NF`) and "Special File Names" (such as `"/dev/stderr"`).

`font-lock-builtin-face` (Emacs)/`font-lock-preprocessor-face` (XEmacs)
: This face is normally used for preprocessor directives in CC Mode. There are no such things in AWK, so this face is used instead for standard functions (such as `match`).

`font-lock-string-face`
: As well as being used for strings, including localizable strings, (delimited by '`"`' and '`_"`'), this face is also used for AWK regular expressions (delimited by '`/`').

`font-lock-warning-face` (Emacs)/`c-invalid-face` (XEmacs)
: This face highlights the following syntactically invalid AWK constructs:

- An unterminated string or regular expression. Here the opening delimiter ('`"`' or '`/`' or '`_"`') is displayed in `font-lock-warning-face`. This is most noticeable when typing in a new string/regular expression into a buffer, when the warning-face serves as a continual reminder to terminate the construct.

  AWK mode fontifies unterminated strings/regular expressions differently from other modes: Only the text up to the end of the line is fontified as a string (escaped newlines being handled correctly), rather than the text up to the next string quote.

- A space between the function name and opening parenthesis when calling a user function. The last character of the function name and the opening parenthesis are highlighted. This font-locking rule will spuriously highlight a valid concatenation expression where an identifier precedes a parenthesized expression. Unfortunately.

- Whitespace following the '\' in what otherwise looks like an escaped newline. The '\' is highlighted.

6 Configuration Basics

You configure CC Mode by setting Lisp variables and calling (and perhaps writing) Lisp functions[1], which is usually done by adding code to an Emacs initialization file. This file might be `site-start.el` or `.emacs` or `init.el` or `default.el` or perhaps some other file. See Section "Init File" in *GNU Emacs Manual*. For the sake of conciseness, we just call this file "your `.emacs`" throughout the rest of the manual.

Several of these variables (currently 16), are known collectively as *style variables*. CC Mode provides a special mechanism, known as *styles* to make it easier to set these variables as a group, to "inherit" settings from one style into another, and so on. Style variables remain ordinary Lisp variables, whose values can be read and changed independently of the style system. See Section 6.2 [Style Variables], page 29.

There are several ways you can write the code, depending on the precise effect you want—they are described further down on this page. If you are new to CC Mode, we suggest you begin with the simplest method, "Top-level commands or the customization interface".

If you make conflicting settings in several of these ways, the way that takes precedence is the one that appears latest in this list:

Style

File Style[2]

Top-level command or "customization interface"

Hook

File Local Variable setting

Here is a summary of the different ways of writing your configuration settings:

Top-level commands or the "customization interface"

Most simply, you can write `setq` and similar commands at the top level of your `.emacs` file. When you load a CC Mode buffer, it initializes its configuration from these global values (at least, for those settings you have given values to), so it makes sense to have these `setq` commands run *before* CC Mode is first initialized—in particular, before any call to `desktop-read` (see Section "Saving Emacs Sessions" in *GNU Emacs Manual*). For example, you might set c-basic-offset thus:

```
(setq c-basic-offset 4)
```

You can use the more user friendly Customization interface instead, but this manual does not cover in detail how that works. To do this, start by typing *M-x customize-group RET c RET*. See Section "Easy Customization" in *GNU Emacs Manual*. Emacs normally writes the customizations at the end of your `.emacs` file. If you use `desktop-read`, you should edit your `.emacs` to place the call to `desktop-read` *after* the customizations.

[1] DON'T PANIC!!! This isn't difficult.

[2] In earlier versions of CC Mode, a File Style setting took precedence over any other setting apart from a File Local Variable setting.

The first initialization of CC Mode puts a snapshot of the configuration settings into the special style `user`. See Section 6.3.1 [Built-in Styles], page 31.

For basic use of Emacs, either of these ways of configuring is adequate. However, the settings are then the same in all CC Mode buffers and it can be clumsy to communicate them between programmers. For more flexibility, you'll want to use one (or both) of CC Mode's more sophisticated facilities, hooks and styles.

Hooks
: An Emacs *hook* is a place to put Lisp functions that you want Emacs to execute later in specific circumstances. See Section "Hooks" in *GNU Emacs Lisp Reference Manual*. CC Mode supplies a main hook and a language-specific hook for each language it supports - any functions you put onto these hooks get executed as the last part of a buffer's initialization. Typically you put most of your customization within the main hook, and use the language-specific hooks to vary the customization settings between language modes. For example, if you wanted different (non-standard) values of `c-basic-offset` in C Mode and Java Mode buffers, you could do it like this:

```
(defun my-c-mode-hook ()
  (setq c-basic-offset 3))
(add-hook 'c-mode-hook 'my-c-mode-hook)

(defun my-java-mode-hook ()
  (setq c-basic-offset 6))
(add-hook 'java-mode-hook 'my-java-mode-hook)
```

See Section 6.1 [CC Hooks], page 28, for more details on the use of CC Mode hooks.

Styles
: A CC Mode *style* is a coherent collection of customizations with a name. At any time, exactly one style is active in each CC Mode buffer, either the one you have selected or a default. CC Mode is delivered with several existing styles. Additionally, you can create your own styles, possibly based on these existing styles. If you worked in a programming team called the "Free Group", which had its own coding standards, you might well have this in your `.emacs` file:

```
(setq c-default-style '((java-mode . "java")
                        (awk-mode . "awk")
                        (other . "free-group-style")))
```

See Section 6.3 [Styles], page 30, for fuller details on using CC Mode styles and how to create them.

File Local Variable setting
: A *file local variable setting* is a setting which applies to an individual source file. You put this in a *local variables list*, a special block at the end of the source file (see Section "Specifying File Variables" in `emacs`).

File Styles
: A *file style* is a rarely used variant of the "style" mechanism described above, which applies to an individual source file. See Section 6.3.5 [File Styles], page 35. You use this by setting certain special variables in a local variables list (see Section "Specifying File Variables" in `emacs`).

Hooks with Styles

For ultimate flexibility, you can use hooks and styles together. For example, if your team were developing a product which required a Linux driver, you'd probably want to use the "linux" style for the driver, and your own team's style for the rest of the code. You could achieve this with code like this in your `.emacs`:

```
(defun my-c-mode-hook ()
  (c-set-style
   (if (and (buffer-file-name)
            (string-match "/usr/src/linux" (buffer-file-name)))
       "linux"
     "free-group-style")))
(add-hook 'c-mode-hook 'my-c-mode-hook)
```

In a programming team, a hook is a also a good place for each member to put his own personal preferences. For example, you might be the only person in your team who likes Auto-newline minor mode. You could have it enabled by default by placing the following in your `.emacs`:

```
(defun my-turn-on-auto-newline ()
  (c-toggle-auto-newline 1))
(add-hook 'c-mode-common-hook 'my-turn-on-auto-newline)
```

6.1 Hooks

CC Mode provides several hooks that you can use to customize the mode for your coding style. The main hook is `c-mode-common-hook`; typically, you'll put the bulk of your customizations here. In addition, each language mode has its own hook, allowing you to fine tune your settings individually for the different CC Mode languages, and there is a package initialization hook. Finally, there is `c-special-indent-hook`, which enables you to solve anomalous indentation problems. It is described in Section 11.5 [Other Indentation], page 85, not here. All these hooks adhere to the standard Emacs conventions.

When you open a buffer, CC Mode first initializes it with the currently active style (see Section 6.3 [Styles], page 30). Then it calls `c-mode-common-hook`, and finally it calls the language-specific hook. Thus, any style settings done in these hooks will override those set by `c-default-style`.

`c-initialization-hook` [Variable]

Hook run only once per Emacs session, when CC Mode is initialized. This is a good place to change key bindings (or add new ones) in any of the CC Mode key maps. See Appendix A [Sample .emacs File], page 92.

`c-mode-common-hook` [Variable]

Common hook across all languages. It's run immediately before the language specific hook.

`c-mode-hook` [Variable]
`c++-mode-hook` [Variable]
`objc-mode-hook` [Variable]

`java-mode-hook` [Variable]
`idl-mode-hook` [Variable]
`pike-mode-hook` [Variable]
`awk-mode-hook` [Variable]

The language specific mode hooks. The appropriate one is run as the last thing when you enter that language mode.

Although these hooks are variables defined in CC Mode, you can give them values before CC Mode's code is loaded—indeed, this is the only way to use `c-initialization-hook`. Their values aren't overwritten when CC Mode gets loaded.

Here's a simplified example of what you can add to your `.emacs` file to do things whenever any CC Mode language is edited. See the Emacs manuals for more information on customizing Emacs via hooks. See Appendix A [Sample .emacs File], page 92, for a more complete sample `.emacs` file.

```
(defun my-c-mode-common-hook ()
  ;; my customizations for all of c-mode and related modes
  (no-case-fold-search)
  )
(add-hook 'c-mode-common-hook 'my-c-mode-common-hook)
```

6.2 Style Variables

The variables that CC Mode's style system control are called *style variables*. Note that style variables are ordinary Lisp variables, which the style system initializes; you can change their values at any time (e.g. in a hook function). The style system can also set other variables, to some extent. See Section 6.3 [Styles], page 30.

Style variables are handled specially in several ways:

- Style variables are by default buffer-local variables. However, they can instead be made global by setting `c-style-variables-are-local-p` to `nil` before CC Mode is initialized.
- The default global binding of any style variable (with two exceptions - see below) is the special symbol `set-from-style`. When the style system initializes a buffer-local copy of a style variable for a CC Mode buffer, if its global binding is still that symbol then it will be set from the current style. Otherwise it will retain its global default[3]. This "otherwise" happens, for example, when you've set the variable with `setq` at the top level of your `.emacs` (see Chapter 6 [Config Basics], page 26).
- The style variable `c-offsets-alist` (see Section 11.1 [c-offsets-alist], page 68) is an association list with an element for each syntactic symbol. It's handled a little differently from the other style variables. It's default global binding is the empty list `nil`, rather than `set-from-style`. Before the style system is initialized, you can add individual elements to `c-offsets-alist` by calling `c-set-offset`(see Section 11.1 [c-offsets-alist], page 68) just like you would set other style variables with `setq`. Those

[3] This is a big change from versions of CC Mode earlier than 5.26, where such settings would get overridden by the style system unless special precautions were taken. That was changed since it was counterintuitive and confusing, especially to novice users. If your configuration depends on the old overriding behavior, you can set the variable `c-old-style-variable-behavior` to non-`nil`.

elements will then prevail when the style system later initializes a buffer-local copy of `c-offsets-alist`.

- The style variable `c-special-indent-hook` is also handled in a special way. Styles can only add functions to this hook, not remove them, so any global settings you put on it are always preserved[4]. The value you give this variable in a style definition can be either a function or a list of functions.
- The global bindings of the style variables get captured in the special `user` style when the style system is first initialized. See Section 6.3.1 [Built-in Styles], page 31, for details.

The style variables are:
`c-indent-comment-alist`, `c-indent-comments-syntactically-p` (see Section 4.1 [Indentation Commands], page 5);
`c-doc-comment-style` (see Section 5.3 [Doc Comments], page 22);
`c-block-comment-prefix`, `c-comment-prefix-regexp` (see Chapter 7 [Custom Filling and Breaking], page 36);
`c-hanging-braces-alist` (see Section 8.1 [Hanging Braces], page 39);
`c-hanging-colons-alist` (see Section 8.2 [Hanging Colons], page 42);
`c-hanging-semi&comma-criteria` (see Section 8.3 [Hanging Semicolons and Commas], page 43);
`c-cleanup-list` (see Chapter 9 [Clean-ups], page 45);
`c-basic-offset` (see Chapter 11 [Customizing Indentation], page 68);
`c-offsets-alist` (see Section 11.1 [c-offsets-alist], page 68);
`c-comment-only-line-offset` (see Section 11.3.4 [Comment Line-Up], page 80);
`c-special-indent-hook`, `c-label-minimum-indentation` (see Section 11.5 [Other Indentation], page 85);
`c-backslash-column`, `c-backslash-max-column` (see Chapter 12 [Custom Macros], page 87).

6.3 Styles

By *style* we mean the layout of the code—things like how many columns to indent a block of code, whether an opening brace gets indented to the level of the code it encloses, or of the construct that introduces it, or "hangs" at the end of a line.

Most people only need to edit code formatted in just a few well-defined and consistent styles. For example, their organization might impose a "blessed" style that all its programmers must conform to. Similarly, people who work on GNU software will have to use the GNU coding style. Some shops are more lenient, allowing a variety of coding styles, and as programmers come and go, there could be a number of styles in use. For this reason, CC Mode makes it convenient for you to set up logical groupings of customizations called *styles*, associate a single name for any particular style, and pretty easily start editing new or existing code using these styles.

As an alternative to writing a style definition yourself, you can have CC Mode *guess* (at least part of) your style by looking at an already formatted piece of your code, Section 6.3.4 [Guessing the Style], page 33.

---

[4] This did not change in version 5.26.

6.3.1 Built-in Styles

If you're lucky, one of CC Mode's built-in styles might be just what you're looking for. These are:

`gnu` Coding style blessed by the Free Software Foundation for C code in GNU programs.

`k&r` The classic Kernighan and Ritchie style for C code.

`bsd` Also known as "Allman style" after Eric Allman.

`whitesmith`
Popularized by the examples that came with Whitesmiths C, an early commercial C compiler.

`stroustrup`
The classic Stroustrup style for C++ code.

`ellemtel` Popular C++ coding standards as defined by "Programming in C++, Rules and Recommendations," Erik Nyquist and Mats Henricson, Ellemtel[5].

`linux` C coding standard for Linux (the kernel).

`python` C coding standard for Python extension modules[6].

`java` The style for editing Java code. Note that the default value for `c-default-style` installs this style when you enter `java-mode`.

`awk` The style for editing AWK code. Note that the default value for `c-default-style` installs this style when you enter `awk-mode`.

`user` This is a special style created by you. It consists of the factory defaults for all the style variables as modified by the customizations you do either with the Customization interface or by writing `setq`s and `c-set-offset`s at the top level of your `.emacs` file (see Chapter 6 [Config Basics], page 26). The style system creates this style as part of its initialization and doesn't modify it afterwards.

6.3.2 Choosing a Style

When you create a new buffer, its style will be set from `c-default-style`. The factory default is the style `gnu`, except in Java and AWK modes where it's `java` and `awk`.

Remember that if you set a style variable with the Customization interface or at the top level of your `.emacs` file before the style system is initialized (see Chapter 6 [Config Basics], page 26), this setting will override the one that the style system would have given the variable.

To set a buffer's style interactively, use the command *`C-c .`* (see Section 4.10 [Other Commands], page 18). To set it from a file's local variable list, Section 6.3.5 [File Styles], page 35.

---

[5] This document is available at `http://www.doc.ic.ac.uk/lab/cplus/c++.rules/` among other places.

[6] Python is a high level scripting language with a C/C++ foreign function interface. For more information, see `http://www.python.org/`.

`c-default-style` [User Option]

This variable specifies which style to install by default in new buffers. It takes either a style name string, or an association list of major mode symbols to style names:

1. When `c-default-style` is a string, it must be an existing style name. This style is then used for all modes.
2. When `c-default-style` is an association list, the mode language is looked up to find a style name string.
3. If `c-default-style` is an association list where the mode language mode isn't found then the special symbol '`other`' is looked up. If it's found then the associated style is used.
4. If '`other`' is not found then the '`gnu`' style is used.

In all cases, the style described in `c-default-style` is installed *before* the language hooks are run, so you can always override this setting by including an explicit call to `c-set-style` in your language mode hook, or in `c-mode-common-hook`.

The standard value of `c-default-style` is `((java-mode . "java") (awk-mode . "awk") (other . "gnu`

`c-indentation-style` [Variable]

This variable always contains the buffer's current style name, as a string.

6.3.3 Adding and Amending Styles

If none of the built-in styles is appropriate, you'll probably want to create a new *style definition*, possibly based on an existing style. To do this, put the new style's settings into a list with the following format - the list can then be passed as an argument to the function `c-add-style`. You can see an example of a style definition in Appendix A [Sample .emacs File], page 92.

Structure of a Style Definition List

([*base-style*] [(*variable* . *value*) . . .])

Optional *base-style*, if present, must be a string which is the name of the *base style* from which this style inherits. At most one *base-style* is allowed in a style definition. If *base-style* is not specified, the style inherits from the table of factory default values[7] instead. All styles eventually inherit from this internal table. Style loops generate errors. The list of pre-existing styles can be seen in Section 6.3.1 [Built-in Styles], page 31.

The dotted pairs (*variable* . *value*) each consist of a variable and the value it is to be set to when the style is later activated.[8] The variable can be either a CC Mode style variable or an arbitrary Emacs variable. In the latter case, it is *not* made buffer-local by the CC Mode style system.

Two variables are treated specially in the dotted pair list:

`c-offsets-alist`

The value is in turn a list of dotted pairs of the form

(*syntactic-symbol* . *offset*)

---

[7] This table is stored internally in the variable c-fallback-style.

[8] Note that if the variable has been given a value by the Customization interface or a `setq` at the top level of your `.emacs`, this value will override the one the style system tries to give it. See Chapter 6 [Config Basics], page 26.

as described in Section 11.1 [c-offsets-alist], page 68. These are passed to `c-set-offset` so there is no need to set every syntactic symbol in your style, only those that are different from the inherited style.

`c-special-indent-hook`
: The value is added to `c-special-indent-hook` using `add-hook`, so any functions already on it are kept. If the value is a list, each element of the list is added with `add-hook`.

Styles are kept in the `c-style-alist` variable, but you should never modify this variable directly. Instead, CC Mode provides the function `c-add-style` for this purpose.

`c-add-style` *stylename description* **&optional** *set-p* [Function]

Add or update a style called *stylename*, a string. *description* is the new style definition in the form described above. If *stylename* already exists in `c-style-alist` then it is replaced by *description*. (Note, this replacement is total. The old style is *not* merged into the new one.) Otherwise, a new style is added.

If the optional *set-p* is non-`nil` then the new style is applied to the current buffer as well. The use of this facility is deprecated and it might be removed from CC Mode in a future release. You should use `c-set-style` instead.

The sample `.emacs` file provides a concrete example of how a new style can be added and automatically set. See Appendix A [Sample .emacs File], page 92.

`c-style-alist` [Variable]

This is the variable that holds the definitions for the styles. It should not be changed directly; use `c-add-style` instead.

6.3.4 Guessing the Style

Instead of specifying a style, you can get CC Mode to *guess* your style by examining an already formatted code buffer. CC Mode then determines the "most frequent" offset (see Section 11.1 [c-offsets-alist], page 68) for each of the syntactic symbols (see Chapter 10 [Indentation Engine Basics], page 49) encountered in the buffer, and the "most frequent" value of c-basic-offset (see Chapter 11 [Customizing Indentation], page 68), then merges the current style with these "guesses" to form a new style. This combined style is known as the *guessed style*.

To do this, call `c-guess` (or one of the other 5 guessing commands) on your sample buffer. The analysis of your code may take some time.

You can then set the guessed style in any CC Mode buffer with `c-guess-install`. You can display the style with `c-guess-view`, and preserve it by copying it into your `.emacs` for future use, preferably after editing it.

`M-x c-guess-no-install`
`M-x c-guess-buffer-no-install`
`M-x c-guess-region-no-install`
: These commands analyze a part of the current buffer and guess the style from it.

The part of the buffer examined is either the region (`c-guess-region-no-install`), the entire buffer (`c-guess-buffer-no-install`), or the first `c-guess-region-max` bytes (`c-guess-no-install`).

Each of these commands can be given an optional prefix argument. This instructs CC Mode to combine the new guesses with the current guesses before forming the guessed style.

M-x c-guess
M-x c-guess-buffer
M-x c-guess-region

These commands analyze a part of the current buffer, guess the style from it, then install the guessed style on the buffer. The guessed style is given a name based on the buffer's absolute file name, and you can then set this style on any CC Mode buffer with *C-c .*.

The part of the buffer examined is either the region (`c-guess-region`), the entire buffer (`c-guess-buffer`), or the first `c-guess-region-max` bytes (`c-guess`).

Each of these commands can be given an optional prefix argument. This instructs CC Mode to combine the new guesses with the current guesses before forming the guessed style.

`c-guess-region-max` [User Option]

This variable, default 50000, is the size in bytes of the buffer portion examined by c-guess and c-guess-no-install. If set to `nil`, the entire buffer is examined.

`c-guess-offset-threshold` [User Option]

This variable, default 10, is the maximum offset, either outwards or inwards, which will be taken into account by the analysis process. Any offset bigger than this will be ignored. For no limit, set this variable to a large number.

M-x c-guess-install

Set the current buffer's style to the guessed style. This prompts you to enter an optional new style name to give to the guessed style. By default, this name is based on the buffer's absolute file name. You can then use this style like any other.

M-x c-guess-view

Display the most recently guessed style in a temporary buffer. This display is in the form of a `c-add-style` form (see Section 6.3.3 [Adding Styles], page 32) which can be easily copied to your `.emacs`. You will probably want to edit it first.

The display of the guessed style contains these elements:

Placeholder Name

You should replace this with a style name of your own.

Parent Style

The style current when the guessing began, from which the guessed style inherits (see Chapter 6 [Config Basics], page 26) the settings which weren't guessed.

Guessed Offsets
: These are the core result of the guessing process. Each of them is marked by a comment.

Inherited Offsets
: These are syntactic offsets which have been taken over from the parent style. To avoid possible future conflicts, you should remove either these offsets or the parent style name.

6.3.5 File Styles

The Emacs manual describes how you can customize certain variables on a per-file basis by including a *file local variable* block at the end of the file (see Section "Local Variables in Files" in *GNU Emacs Manual*).

So far, you've only seen a functional interface for setting styles in CC Mode, and this can't be used here. CC Mode fills the gap by providing two variables for use in a file's local variable list. Don't use them anywhere else! These allow you to customize the style on a per-file basis:

`c-file-style` [Variable]
: Set this variable to a style name string in the Local Variables list. From now on, when you visit the file, CC Mode will automatically set the file's style to this one using `c-set-style`.

`c-file-offsets` [Variable]
: Set this variable (in the Local Variables list) to an association list of the same format as `c-offsets-alist`. From now on, when you visit the file, CC Mode will automatically institute these offsets using `c-set-offset`.

Note that file style settings (i.e. `c-file-style`) are applied before file offset settings (i.e. `c-file-offsets`)[9].

If you set any variable by the file local variables mechanism, that setting takes priority over all other settings, even those in your mode hooks (see Section 6.1 [CC Hooks], page 28). Any individual setting of a variable will override one made through `c-file-style` or `c-file-offsets`.

[9] Also, if either of these are set in a file's local variable section, all the style variable values are made local to that buffer, even if `c-style-variables-are-local-p` is `nil`. Since this variable is virtually always non-`nil` anyhow, you're unlikely to notice this effect.

7 Customizing Filling and Line Breaking

Since there's a lot of normal text in comments and string literals, CC Mode provides features to edit these like in text mode. It does this by hooking in on the different line breaking functions and tuning relevant variables as necessary.

To make Emacs recognize comments and treat text in them as normal paragraphs, CC Mode makes several standard variables[1] buffer-local and modifies them according to the language syntax and the comment line prefix.

`c-comment-prefix-regexp` [User Option]

This style variable contains the regexp used to recognize the *comment line prefix*, which is the line decoration that starts every line in a comment. The variable is either the comment line prefix itself, or (more usually) an association list with different values for different languages. The symbol for the major mode is looked up in the alist to get the regexp for the language, and if it isn't found then the special symbol '`other`' is looked up instead.

When a comment line gets divided by *M-j* or the like, CC Mode inserts the comment line prefix from a neighboring line at the start of the new line. The default value of c-comment-prefix-regexp is '`//+\\|\\**`', which matches C++ style line comments like

```
// blah blah
```

with two or more slashes in front of them, and the second and subsequent lines of C style block comments like

```
/*
 * blah blah
 */
```

with zero or more stars at the beginning of every line. If you change this variable, please make sure it still matches the comment starter (i.e. `//`) of line comments *and* the line prefix inside block comments.

Also note that since CC Mode uses the value of `c-comment-prefix-regexp` to set up several other variables at mode initialization, there won't be any effect if you just change it inside a CC Mode buffer. You need to call the command `c-setup-paragraph-variables` too, to update those other variables. That's also the case if you modify `c-comment-prefix-regexp` in a mode hook, since CC Mode will already have set up these variables before calling the hook.

In comments, CC Mode uses `c-comment-prefix-regexp` to adapt the line prefix from the other lines in the comment.

CC Mode uses adaptive fill mode (see Section "Adaptive Fill" in *GNU Emacs Manual*) to make Emacs correctly keep the line prefix when filling paragraphs. That also makes

---

[1] `comment-start`, `comment-end`, `comment-start-skip`, `paragraph-start`, `paragraph-separate`, `paragraph-ignore-fill-prefix`, `adaptive-fill-mode`, `adaptive-fill-regexp`, and `adaptive-fill-first-line-regexp`.

Emacs preserve the text indentation *inside* the comment line prefix. E.g. in the following comment, both paragraphs will be filled with the left margins of the texts kept intact:

```
/* Make a balanced b-tree of the nodes in the incoming
 * stream.  But, to quote the famous words of Donald E.
 * Knuth,
 *
 *     Beware of bugs in the above code; I have only
 *     proved it correct, not tried it.
 */
```

It's also possible to use other adaptive filling packages, notably Kyle E. Jones' Filladapt package[2], which handles things like bulleted lists nicely. There's a convenience function `c-setup-filladapt` that tunes the relevant variables in Filladapt for use in CC Mode. Call it from a mode hook, e.g. with something like this in your `.emacs`:

```
(defun my-c-mode-common-hook ()
  (c-setup-filladapt)
  (filladapt-mode 1))
(add-hook 'c-mode-common-hook 'my-c-mode-common-hook)
```

`c-block-comment-prefix` [User Option]

Normally the comment line prefix inserted for a new line inside a comment is deduced from other lines in it. However there's one situation when there's no hint about what the prefix should look like, namely when a block comment is broken for the first time. This style variable[3] is used then as the comment prefix. It defaults to '`* `'[4], which makes a comment

```
/* Got O(n^2) here, which is a Bad Thing. */
```

break into

```
/* Got O(n^2) here, which
 * is a Bad Thing. */
```

Note that it won't work to adjust the indentation by putting leading spaces in `c-block-comment-prefix`, since CC Mode still uses the normal indentation engine to indent the line. Thus, the right way to fix the indentation is by customizing the `c` syntactic symbol. It defaults to `c-lineup-C-comments`, which handles the indentation of most common comment styles, see Section 11.3 [Line-Up Functions], page 72.

`c-ignore-auto-fill` [User Option]

When auto fill mode is enabled, CC Mode can selectively ignore it depending on the context the line break would occur in, e.g. to never break a line automatically inside

[2] It's available from `http://www.wonderworks.com/`. As of version 2.12, it does however lack a feature that makes it work suboptimally when `c-comment-prefix-regexp` matches the empty string (which it does by default). A patch for that is available from the CC Mode web site.

[3] In versions before 5.26, this variable was called `c-comment-continuation-stars`. As a compatibility measure, CC Mode still uses the value on that variable if it's set.

[4] Actually, this default setting of `c-block-comment-prefix` typically gets overridden by the default style `gnu`, which sets it to blank. You can see the line splitting effect described here by setting a different style, e.g. `k&r` See Section 6.3.2 [Choosing a Style], page 31.

a string literal. This variable takes a list of symbols for the different contexts where auto-filling never should occur:

`string` Inside a string or character literal.

`c` Inside a C style block comment.

`c++` Inside a C++ style line comment.

`cpp` Inside a preprocessor directive.

`code` Anywhere else, i.e. in normal code.

By default, `c-ignore-auto-fill` is set to `(string cpp code)`, which means that when auto-fill mode is activated, auto-filling only occurs in comments. In literals, it's often desirable to have explicit control over newlines. In preprocessor directives, the necessary '\' escape character before the newline is not automatically inserted, so an automatic line break would produce invalid code. In normal code, line breaks are normally dictated by some logical structure in the code rather than the last whitespace character, so automatic line breaks there will produce poor results in the current implementation.

If inside a comment and `comment-multi-line` (see Section "Auto Fill" in *GNU Emacs Manual* is non-`nil`, the indentation and line prefix are preserved. If inside a comment and `comment-multi-line` is `nil`, a new comment of the same type is started on the next line and indented as appropriate for comments.

Note that CC Mode sets `comment-multi-line` to `t` at startup. The reason is that *M-j* could otherwise produce sequences of single line block comments for texts that should logically be treated as one comment, and the rest of the paragraph handling code (e.g. *M-q* and *M-a*) can't cope with that, which would lead to inconsistent behavior.

8 Customizing Auto-newlines

CC Mode determines whether to insert auto-newlines in two basically different ways, depending on the character just typed:

Braces and Colons
: CC Mode first determines the syntactic context of the brace or colon (see Section 10.2 [Syntactic Symbols], page 51), then looks for a corresponding element in an alist. This element specifies where to put newlines - this is any combination of before and after the brace or colon. If no alist element is found, newlines are inserted both before and after a brace, but none are inserted around a colon. See Section 8.1 [Hanging Braces], page 39, and Section 8.2 [Hanging Colons], page 42.

Semicolons and Commas
: The variable `c-hanging-semi&comma-criteria` contains a list of functions which determine whether to insert a newline after a newly typed semicolon or comma. See Section 8.3 [Hanging Semicolons and Commas], page 43.

The names of these configuration variables contain '`hanging`' because they let you *hang* the pertinent characters. A character which introduces a C construct is said to *hang on the right* when it appears at the end of a line after other code, being separated by a line break from the construct it introduces, like the opening brace in:

```
while (i < MAX) {
    total += entry[i];
    entry [i++] = 0;
}
```

A character *hangs on the left* when it appears at the start of the line after the construct it closes off, like the above closing brace.

The next chapter, "Clean-ups", describes how to configure CC Mode to remove these automatically added newlines in certain specific circumstances. See Chapter 9 [Clean-ups], page 45.

8.1 Hanging Braces

To specify which kinds of braces you want auto-newlines put around, you set the style variable `c-hanging-braces-alist`. Its structure and semantics are described in this section. Details of how to set it up, and its relationship to CC Mode's style system are given in Section 6.2 [Style Variables], page 29.

Say you wanted an auto-newline after (but not before) the following '`{`':

```
if (foo < 17) {
```

First you need to find the *syntactic context* of the brace—type a `RET` before the brace to get it on a line of its own[1], then type *`C-c C-s`*. That will tell you something like:

```
((substatement-open 1061))
```

So here you need to put the entry `(substatement-open . (after))` into `c-hanging-braces-alist`.

[1] Also insert a '\' at the end of the previous line if you're in AWK Mode.

If you don't want any auto-newlines for a particular syntactic symbol, put this into `c-hanging-braces-alist`:

```
(brace-entry-open)
```

If some brace syntactic symbol is not in `c-hanging-brace-alist`, its entry is taken by default as `(before after)`—insert a newline both before and after the brace. In place of a "before/after" list you can specify a function in this alist—this is useful when the auto newlines depend on the code around the brace.

`c-hanging-braces-alist` [User Option]

This variable is an association list which maps syntactic symbols to lists of places to insert a newline. See Section "Association Lists" in *GNU Emacs Lisp Reference Manual*. The key of each element is the syntactic symbol, the associated value is either `nil`, a list, or a function.

The Key - the syntactic symbol

The syntactic symbols that are useful as keys in this list are `brace-list-intro`, `statement-cont`, `inexpr-class-open`, `inexpr-class-close`, and all the `*-open` and `*-close` symbols. See Section 10.2 [Syntactic Symbols], page 51, for a more detailed description of these syntactic symbols, except for `inexpr-class-open` and `inexpr-class-close`, which aren't actual syntactic symbols. Elements with any other value as a key get ignored.

The braces of anonymous inner classes in Java are given the special symbols `inexpr-class-open` and `inexpr-class-close`, so that they can be distinguished from the braces of normal classes[2].

Note that the aggregate constructs in Pike mode, '`({`', '`})`', '`([`', '`])`', and '`(<`', '`>)`', do not count as brace lists in this regard, even though they do for normal indentation purposes. It's currently not possible to set automatic newlines on these constructs.

The associated value - the "ACTION" list or function

The value associated with each syntactic symbol in this association list is called an *action*, which can be either a list or a function which returns a list. See Section 8.1.1 [Custom Braces], page 41, for how to use a function as a brace hanging *action*.

The list *action* (or the list returned by *action* when it's a function) contains some combination of the symbols `before` and `after`, directing CC Mode where to put newlines in relationship to the brace being inserted. Thus, if the list contains only the symbol `after`, then the brace hangs on the right side of the line, as in:

```
// here, open braces always 'hang'
void spam( int i ) {
    if( i == 7 ) {
        dosomething(i);
```

[2] The braces of anonymous classes produce a combination of `inexpr-class`, and `class-open` or `class-close` in normal indentation analysis.

```
        }
    }
```

When the list contains both `after` and `before`, the braces will appear on a line by themselves, as shown by the close braces in the above example. The list can also be empty, in which case newlines are added neither before nor after the brace.

If a syntactic symbol is missing entirely from `c-hanging-braces-alist`, it's treated in the same way as an *action* with a list containing `before` and `after`, so that braces by default end up on their own line.

For example, the default value of `c-hanging-braces-alist` is:

```
((brace-list-open)
 (brace-entry-open)
 (statement-cont)
 (substatement-open after)
 (block-close . c-snug-do-while)
 (extern-lang-open after)
 (namespace-open after)
 (module-open after)
 (composition-open after)
 (inexpr-class-open after)
 (inexpr-class-close before))
```

which says that `brace-list-open`, `brace-entry-open` and `statement-cont`[3] braces should both hang on the right side and allow subsequent text to follow on the same line as the brace. Also, `substatement-open`, `extern-lang-open`, and `inexpr-class-open` braces should hang on the right side, but subsequent text should follow on the next line. The opposite holds for `inexpr-class-close` braces; they won't hang, but the following text continues on the same line. Here, in the `block-close` entry, you also see an example of using a function as an *action*. In all other cases, braces are put on a line by themselves.

8.1.1 Custom Brace Hanging

Syntactic symbols aren't the only place where you can customize CC Mode with the lisp equivalent of callback functions. Remember that *action*s are usually a list containing some combination of the symbols `before` and `after` (see Section 8.1 [Hanging Braces], page 39). For more flexibility, you can instead specify brace "hanginess" by giving a syntactic symbol an *action function* in `c-hanging-braces-alist`; this function determines the "hanginess" of a brace, usually by looking at the code near it.

An action function is called with two arguments: the syntactic symbol for the brace (e.g. `substatement-open`), and the buffer position where the brace has been inserted. Point is undefined on entry to an action function, but the function must preserve it (e.g. by using `save-excursion`). The return value should be a list containing some combination of `before` and `after`, including neither of them (i.e. `nil`).

[3] Brace lists inside statements, such as initializers for static array variables inside functions in C, are recognized as `statement-cont`. All normal substatement blocks are recognized with other symbols.

`c-syntactic-context` [Variable]

During the call to the indentation or brace hanging *action* function, this variable is bound to the full syntactic analysis list. This might be, for example, '`((block-close 73))`'. Don't ever give `c-syntactic-context` a value yourself—this would disrupt the proper functioning of CC Mode.

This variable is also bound in three other circumstances: (i) when calling a c-hanging-semi&comma-criteria function (see Section 8.3 [Hanging Semicolons and Commas], page 43); (ii) when calling a line-up function (see Section 11.4 [Custom Line-Up], page 84); (iii) when calling a c-special-indent-hook function (see Section 11.5 [Other Indentation], page 85).

As an example, CC Mode itself uses this feature to dynamically determine the hanginess of braces which close "do-while" constructs:

```
void do_list( int count, char** atleast_one_string )
{
    int i=0;
    do {
        handle_string( atleast_one_string[i] );
        i++;
    } while( i < count );
}
```

CC Mode assigns the `block-close` syntactic symbol to the brace that closes the `do` construct, and normally we'd like the line that follows a `block-close` brace to begin on a separate line. However, with "do-while" constructs, we want the `while` clause to follow the closing brace. To do this, we associate the `block-close` symbol with the *action* function `c-snug-do-while`:

```
(defun c-snug-do-while (syntax pos)
  "Dynamically calculate brace hanginess for do-while statements."
  (save-excursion
    (let (langelem)
      (if (and (eq syntax 'block-close)
               (setq langelem (assq 'block-close c-syntactic-context))
               (progn (goto-char (cdr langelem))
                      (if (= (following-char) ?{)
                          (forward-sexp -1))
                      (looking-at "\\<do\\>[^_]")))
          '(before)
        '(before after)))))
```

This function simply looks to see if the brace closes a "do-while" clause and if so, returns the list '`(before)`' indicating that a newline should be inserted before the brace, but not after it. In all other cases, it returns the list '`(before after)`' so that the brace appears on a line by itself.

8.2 Hanging Colons

Using a mechanism similar to brace hanging (see Section 8.1 [Hanging Braces], page 39), colons can also be made to hang using the style variable `c-hanging-colons-alist` - When a colon is typed, CC Modedetermines its syntactic context, looks this up in the alist `c-changing-colons-alist` and inserts up to two newlines accordingly. Here, however, If CC Modefails to find an entry for a syntactic symbol in the alist, no newlines are inserted around the newly typed colon.

`c-hanging-colons-alist` [User Option]

The Key - the syntactic symbol
: The syntactic symbols appropriate as keys in this association list are: `case-label`, `label`, `access-label`, `member-init-intro`, and `inher-intro`. See Section 10.2 [Syntactic Symbols], page 51. Elements with any other value as a key get ignored.

The associate value - the "ACTION" list
: The *action* here is simply a list containing a combination of the symbols `before` and `after`. Unlike in `c-hanging-braces-alist`, functions as *actions* are not supported - there doesn't seem to be any need for them.

In C++, double-colons are used as a scope operator but because these colons always appear right next to each other, newlines before and after them are controlled by a different mechanism, called *clean-ups* in CC Mode. See Chapter 9 [Clean-ups], page 45, for details.

8.3 Hanging Semicolons and Commas

`c-hanging-semi&comma-criteria` [User Option]

This style variable takes a list of functions; these get called when you type a semicolon or comma. The functions are called in order without arguments. When these functions are entered, point is just after the newly inserted '`;`' or '`,`' and they must preserve point (e.g., by using `save-excursion`). During the call, the variable `c-syntactic-context` is bound to the syntactic context of the current line[4] see Section 8.1.1 [Custom Braces], page 41. These functions don't insert newlines themselves, rather they direct CC Mode whether or not to do so. They should return one of the following values:

`t` A newline is to be inserted after the '`;`' or '`,`', and no more functions from the list are to be called.

`stop` No more functions from the list are to be called, and no newline is to be inserted.

`nil` No determination has been made, and the next function in the list is to be called.

Note that auto-newlines are never inserted *before* a semicolon or comma. If every function in the list is called without a determination being made, then no newline is added.

In AWK mode, this variable is set by default to `nil`. In the other modes, the default value is a list containing a single function, `c-semi&comma-inside-parenlist`. This inserts newlines after all semicolons, apart from those separating `for`-clause statements.

[4] This was first introduced in CC Mode 5.31.

`c-semi&comma-no-newlines-before-nonblanks` [Function]

This is an example of a criteria function, provided by CC Mode. It prevents newlines from being inserted after semicolons when there is a non-blank following line. Otherwise, it makes no determination. To use, add this function to the front of the `c-hanging-semi&comma-criteria` list.

```
(defun c-semi&comma-no-newlines-before-nonblanks ()
  (save-excursion
    (if (and (= (c-last-command-char) ?\;)
     (zerop (forward-line 1))
     (bolp)      ; forward-line has funny behavior at eob.
     (not (looking-at "^[ \t]*$")))
'stop
      nil)))
```

`c-semi&comma-inside-parenlist` [Function]

`c-semi&comma-no-newlines-for-oneline-inliners` [Function]

The function `c-semi&comma-inside-parenlist` is what prevents newlines from being inserted inside the parenthesis list of `for` statements. In addition to `c-semi&comma-no-newlines-before-nonblanks` described above, CC Mode also comes with the criteria function `c-semi&comma-no-newlines-for-oneline-inliners`, which suppresses newlines after semicolons inside one-line inline method definitions (e.g. in C++ or Java).

9 Clean-ups

Clean-ups are mechanisms which remove (or exceptionally, add) whitespace in specific circumstances and are complementary to colon and brace hanging. You enable a clean-up by adding its symbol into `c-cleanup-list`, e.g. like this:

```
(add-to-list 'c-cleanup-list 'space-before-funcall)
```

On the surface, it would seem that clean-ups overlap the functionality provided by the `c-hanging-*-alist` variables. Clean-ups, however, are used to adjust code "after-the-fact", i.e. to adjust the whitespace in constructs later than when they were typed.

Most of the clean-ups remove automatically inserted newlines, and are only active when auto-newline minor mode is turned on. Others will work all the time. Note that clean-ups are only performed when there is nothing but whitespace appearing between the individual components of the construct, and (apart from `comment-close-slash`) when the construct does not occur within a literal (see Section 4.7 [Auto-newlines], page 15).

`c-cleanup-list` [User Option]

You configure CC Mode's clean-ups by setting the style variable `c-cleanup-list`, which is a list of clean-up symbols. By default, CC Mode cleans up only the `scope-operator` construct, which is necessary for proper C++ support.

These are the clean-ups that are only active when electric and auto-newline minor modes are enabled:

`brace-else-brace`

Clean up '`} else {`' constructs by placing the entire construct on a single line. Clean up occurs when the open brace after the '`else`' is typed. So for example, this:

```
void spam(int i)
{
    if( i==7 ) {
        dosomething();
    }
    else
    {
```

appears like this after the last open brace is typed:

```
void spam(int i)
{
    if( i==7 ) {
        dosomething();
    } else {
```

`brace-elseif-brace`

Similar to the `brace-else-brace` clean-up, but this cleans up '`} else if (...) {`' constructs. For example:

```
void spam(int i)
{
    if( i==7 ) {
        dosomething();
    }
    else if( i==3 )
    {
```

appears like this after the last open parenthesis is typed:

```
void spam(int i)
{
    if( i==7 ) {
        dosomething();
    } else if(
```

and like this after the last open brace is typed:

```
void spam(int i)
{
    if( i==7 ) {
        dosomething();
    } else if( i==3 ) {
```

`brace-catch-brace`
: Analogous to `brace-elseif-brace`, but cleans up '`} catch (...) {`' in C++ and Java mode.

`empty-defun-braces`
: Clean up braces following a top-level function or class definition that contains no body. Clean up occurs when the closing brace is typed. Thus the following:

```
class Spam
{
}
```

is transformed into this when the close brace is typed:

```
class Spam
{}
```

`defun-close-semi`
: Clean up the terminating semicolon on top-level function or class definitions when they follow a close brace. Clean up occurs when the semicolon is typed. So for example, the following:

```
class Spam
{
...
}
;
```

is transformed into this when the semicolon is typed:

```
class Spam
{
...
};
```

`list-close-comma`
: Clean up commas following braces in array and aggregate initializers. Clean up occurs when the comma is typed. The space before the comma is zapped just like the space before the semicolon in `defun-close-semi`.

`scope-operator`
: Clean up double colons which might designate a C++ scope operator split across multiple lines[1]. Clean up occurs when the second colon is typed. You will always want `scope-operator` in the `c-cleanup-list` when you are editing C++ code.

`one-liner-defun`
: Clean up a single line of code enclosed by defun braces by removing the whitespace before and after the code. The clean-up happens when the closing brace is typed. If the variable `c-max-one-liner-length` is set, the cleanup is only done if the resulting line would be no longer than the value of that variable.

  For example, consider this AWK code:

```
BEGIN {
    FS = "\t" # use <TAB> as a field separator
}
```

  It gets compacted to the following when the closing brace is typed:

```
BEGIN {FS = "\t"} # use <TAB> as a field separator
```

  `c-max-one-liner-length` [User Option]
  : The maximum length of the resulting line for which the clean-up `one-liner-defun` will be triggered. This length is that of the entire line, including any leading whitespace and any trailing comment. Its default value is 80. If the value is zero or `nil`, no limit applies.

The following clean-ups are always active when they occur on `c-cleanup-list`, regardless of whether Electric minor mode or Auto-newline minor mode are enabled:

`space-before-funcall`
: Insert a space between the function name and the opening parenthesis of a function call. This produces function calls in the style mandated by the GNU coding standards, e.g. '`signal (SIGINT, SIG_IGN)`' and '`abort ()`'. Clean up occurs when the opening parenthesis is typed. This clean-up should never be active in AWK Mode, since such a space is syntactically invalid for user defined functions.

`compact-empty-funcall`
: Clean up any space between the function name and the opening parenthesis of a function call that has no arguments. This is typically used together with

[1] Certain C++ constructs introduce ambiguous situations, so `scope-operator` clean-ups might not always be correct. This usually only occurs when scoped identifiers appear in switch label tags.

`space-before-funcall` if you prefer the GNU function call style for functions with arguments but think it looks ugly when it's only an empty parenthesis pair. I.e. you will get '`signal (SIGINT, SIG_IGN)`', but '`abort()`'. Clean up occurs when the closing parenthesis is typed.

`comment-close-slash`

When inside a block comment, terminate the comment when you type a slash at the beginning of a line (i.e. immediately after the comment prefix). This clean-up removes whitespace preceding the slash and if needed, inserts a star to complete the token '`*/`'. Type *C-q /* in this situation if you just want a literal '`/`' inserted.

10 Indentation Engine Basics

This chapter will briefly cover how CC Mode indents lines of code. It is helpful to understand the indentation model being used so that you will know how to customize CC Mode for your personal coding style. All the details are in Chapter 11 [Customizing Indentation], page 68.

CC Mode has an indentation engine that provides a flexible and general mechanism for customizing indentation. When CC Mode indents a line of code, it separates its calculations into two steps:

1. It analyzes the line to determine its *syntactic symbol(s)* (the kind of language construct it's looking at) and its *anchor position* (the position earlier in the file that CC Mode will indent the line relative to). The anchor position might be the location of an opening brace in the previous line, for example. See Section 10.1 [Syntactic Analysis], page 49.
2. It looks up the syntactic symbol(s) in the configuration to get the corresponding *offset(s)*. The symbol `+`, which means "indent this line one more level" is a typical offset. CC Mode then applies these offset(s) to the anchor position, giving the indentation for the line. The different sorts of offsets are described in Section 11.1 [c-offsets-alist], page 68.

In exceptional circumstances, the syntax directed indentation described here may be a nuisance rather than a help. You can disable it by setting `c-syntactic-indentation` to `nil`. (To set the variable interactively, Section 4.5 [Minor Modes], page 11).

`c-syntactic-indentation` [User Option]

> When this is non-`nil` (which it is by default), the indentation of code is done according to its syntactic structure. When it's `nil`, every line is just indented to the same level as the previous one, and *TAB* (`c-indent-command`) adjusts the indentation in steps of `c-basic-offset`. The current style (see Chapter 6 [Config Basics], page 26) then has no effect on indentation, nor do any of the variables associated with indentation, not even `c-special-indent-hook`.

10.1 Syntactic Analysis

The first thing CC Mode does when indenting a line of code, is to analyze the line by calling `c-guess-basic-syntax`, determining the syntactic context of the (first) construct on that line. Although this function is mainly used internally, it can sometimes be useful in Line-up functions (see Section 11.4 [Custom Line-Up], page 84) or in functions on `c-special-indent-hook` (see Section 11.5 [Other Indentation], page 85).

`c-guess-basic-syntax` [Function]

> Determine the syntactic context of the current line.

The *syntactic context* is a list of *syntactic elements*, where each syntactic element in turn is a list[1] Here is a brief and typical example:

```
((defun-block-intro 1959))
```

---

[1] In CC Mode5.28 and earlier, a syntactic element was a dotted pair; the cons was the syntactic symbol and the cdr was the anchor position. For compatibility's sake, the parameter passed to a line-up function still has this dotted pair form (see Section 11.4 [Custom Line-Up], page 84).

The first thing inside each syntactic element is always a *syntactic symbol*. It describes the kind of construct that was recognized, e.g. `statement`, `substatement`, `class-open`, `class-close`, etc. See Section 10.2 [Syntactic Symbols], page 51, for a complete list of currently recognized syntactic symbols and their semantics. The remaining entries are various data associated with the recognized construct - there might be zero or more.

Conceptually, a line of code is always indented relative to some position higher up in the buffer (typically the indentation of the previous line). That position is the *anchor position* in the syntactic element. If there is an entry after the syntactic symbol in the syntactic element list then it's either nil or that anchor position.

Here is an example. Suppose we had the following code as the only thing in a C++ buffer[2]:

```
1: void swap( int& a, int& b )
2: {
3:     int tmp = a;
4:     a = b;
5:     b = tmp;
6: }
```

We can use *C-c C-s* (`c-show-syntactic-information`) to report what the syntactic analysis is for the current line:

C-c C-s (`c-show-syntactic-information`)
: This command calculates the syntactic analysis of the current line and displays it in the minibuffer. The command also highlights the anchor position(s).

Running this command on line 4 of this example, we'd see in the echo area[3]:

```
((statement 35))
```

and the '`i`' of `int` on line 3 would be highlighted. This tells us that the line is a statement and it is indented relative to buffer position 35, the highlighted position. If you were to move point to line 3 and hit *C-c C-s*, you would see:

```
((defun-block-intro 29))
```

This indicates that the '`int`' line is the first statement in a top level function block, and is indented relative to buffer position 29, which is the brace just after the function header.

Here's another example:

```
1: int add( int val, int incr, int doit )
2: {
3:     if( doit )
4:         {
5:             return( val + incr );
6:         }
7:     return( val );
8: }
```

[2] The line numbers in this and future examples don't actually appear in the buffer, of course!

[3] With a universal argument (i.e. *C-u C-c C-s*) the analysis is inserted into the buffer as a comment on the current line.

Hitting *C-c C-s* on line 4 gives us:

```
((substatement-open 46))
```

which tells us that this is a brace that *opens* a substatement block.[4]

Syntactic contexts can contain more than one element, and syntactic elements need not have anchor positions. The most common example of this is a *comment-only line*:

```
1: void draw_list( List<Drawables>& drawables )
2: {
3:         // call the virtual draw() method on each element in list
4:     for( int i=0; i < drawables.count(), ++i )
5:     {
6:         drawables[i].draw();
7:     }
8: }
```

Hitting *C-c C-s* on line 3 of this example gives:

```
((comment-intro) (defun-block-intro 46))
```

and you can see that the syntactic context contains two syntactic elements. Notice that the first element, '`(comment-intro)`', has no anchor position.

10.2 Syntactic Symbols

This section is a complete list of the syntactic symbols which appear in the `c-offsets-alist` style variable, along with brief descriptions. The previous section (see Section 10.1 [Syntactic Analysis], page 49) states what syntactic symbols are and how the indentation engine uses them.

More detailed descriptions of these symbols, together with snippets of source code to which they apply, appear in the examples in the subsections below. Note that, in the interests of brevity, the anchor position associated with most syntactic symbols is *not* specified. In cases of doubt, type *C-c C-s* on a pertinent line—this highlights the anchor position.

The syntactic symbols which indicate brace constructs follow a general naming convention. When a line begins with an open or close brace, its syntactic symbol will contain the suffix `-open` or `-close` respectively. The first line within the brace block construct will contain the suffix `-block-intro`.

In constructs which can span several lines, a distinction is usually made between the first line that introduces the construct and the lines that continue it. The syntactic symbols that indicate these lines will contain the suffixes `-intro` or `-cont` respectively.

The best way to understand how all this works is by looking at some examples. Remember that you can see the syntax of any source code line by using *C-c C-s*.

`string` Inside a multiline string. Section 10.2.8 [Literal Symbols], page 62.

`c` Inside a multiline C style block comment. Section 10.2.8 [Literal Symbols], page 62.

---

[4] A *substatement* is the line after a conditional statement, such as `if`, `else`, `while`, `do`, `switch`, etc. A *substatement block* is a brace block following one of these conditional statements.

`defun-open`
: Brace that opens a top-level function definition. Section 10.2.1 [Function Symbols], page 56.

`defun-close`
: Brace that closes a top-level function definition. Section 10.2.1 [Function Symbols], page 56.

`defun-block-intro`
: The first line in a top-level defun. Section 10.2.1 [Function Symbols], page 56.

`class-open`
: Brace that opens a class definition. Section 10.2.2 [Class Symbols], page 57.

`class-close`
: Brace that closes a class definition. Section 10.2.2 [Class Symbols], page 57.

`inline-open`
: Brace that opens an in-class inline method. Section 10.2.2 [Class Symbols], page 57.

`inline-close`
: Brace that closes an in-class inline method. Section 10.2.2 [Class Symbols], page 57.

`func-decl-cont`
: The region between a function definition's argument list and the function opening brace (excluding K&R argument declarations). In C, you cannot put anything but whitespace and comments in this region, however in C++ and Java, `throws` declarations and other things can appear here. Section 10.2.8 [Literal Symbols], page 62.

`knr-argdecl-intro`
: First line of a K&R C argument declaration. Section 10.2.13 [K&R Symbols], page 65.

`knr-argdecl`
: Subsequent lines in a K&R C argument declaration. Section 10.2.13 [K&R Symbols], page 65.

`topmost-intro`
: The first line in a "topmost" definition. Section 10.2.1 [Function Symbols], page 56.

`topmost-intro-cont`
: Topmost definition continuation lines. This is only used in the parts that aren't covered by other symbols such as `func-decl-cont` and `knr-argdecl`. Section 10.2.1 [Function Symbols], page 56.

`annotation-top-cont`
: Topmost definition continuation lines where all previous items are annotations. Section 10.2.11 [Java Symbols], page 64.

`member-init-intro`
: First line in a member initialization list. Section 10.2.2 [Class Symbols], page 57.

`member-init-cont`
: Subsequent member initialization list lines. Section 10.2.2 [Class Symbols], page 57.

`inher-intro`
: First line of a multiple inheritance list. Section 10.2.2 [Class Symbols], page 57.

`inher-cont`
: Subsequent multiple inheritance lines. Section 10.2.2 [Class Symbols], page 57.

`block-open`
: Statement block open brace. Section 10.2.8 [Literal Symbols], page 62.

`block-close`
: Statement block close brace. Section 10.2.3 [Conditional Construct Symbols], page 58.

`brace-list-open`
: Open brace of an enum or static array list. Section 10.2.5 [Brace List Symbols], page 60.

`brace-list-close`
: Close brace of an enum or static array list. Section 10.2.5 [Brace List Symbols], page 60.

`brace-list-intro`
: First line after the opening '`{`' in an enum or static array list. Section 10.2.5 [Brace List Symbols], page 60.

`brace-list-entry`
: Subsequent lines in an enum or static array list. Section 10.2.5 [Brace List Symbols], page 60.

`brace-entry-open`
: Subsequent lines in an enum or static array list where the line begins with an open brace. Section 10.2.5 [Brace List Symbols], page 60.

`statement`
: A statement. Section 10.2.1 [Function Symbols], page 56.

`statement-cont`
: A continuation of a statement. Section 10.2.1 [Function Symbols], page 56.

`annotation-var-cont`
: A continuation of a statement where all previous items are annotations. Section 10.2.11 [Java Symbols], page 64.

`statement-block-intro`
: The first line in a new statement block. Section 10.2.3 [Conditional Construct Symbols], page 58.

`statement-case-intro`
: The first line in a case block. Section 10.2.4 [Switch Statement Symbols], page 59.

`statement-case-open`
: The first line in a case block that starts with a brace. Section 10.2.4 [Switch Statement Symbols], page 59.

`substatement`
: The first line after a conditional or loop construct. Section 10.2.3 [Conditional Construct Symbols], page 58.

`substatement-open`
: The brace that opens a substatement block. Section 10.2.3 [Conditional Construct Symbols], page 58.

`substatement-label`
: The first line after a conditional or loop construct if it's a label. Section 10.2.3 [Conditional Construct Symbols], page 58.

`case-label`
: A label in a `switch` block. Section 10.2.4 [Switch Statement Symbols], page 59.

`access-label`
: C++ access control label. Section 10.2.2 [Class Symbols], page 57.

`label`
: Any other label. Section 10.2.8 [Literal Symbols], page 62.

`do-while-closure`
: The `while` line that ends a `do-while` construct. Section 10.2.3 [Conditional Construct Symbols], page 58.

`else-clause`
: The `else` line of an `if-else` construct. Section 10.2.3 [Conditional Construct Symbols], page 58.

`catch-clause`
: The `catch` or `finally` (in Java) line of a `try-catch` construct. Section 10.2.3 [Conditional Construct Symbols], page 58.

`comment-intro`
: A line containing only a comment introduction. Section 10.2.8 [Literal Symbols], page 62.

`arglist-intro`
: The first line in an argument list. Section 10.2.7 [Paren List Symbols], page 61.

`arglist-cont`
: Subsequent argument list lines when no arguments follow on the same line as the arglist opening paren. Section 10.2.7 [Paren List Symbols], page 61.

`arglist-cont-nonempty`
: Subsequent argument list lines when at least one argument follows on the same line as the arglist opening paren. Section 10.2.7 [Paren List Symbols], page 61.

`arglist-close`
: The solo close paren of an argument list. Section 10.2.7 [Paren List Symbols], page 61.

`stream-op`
: Lines continuing a stream operator (C++ only). Section 10.2.8 [Literal Symbols], page 62.

`inclass`
: The line is nested inside a class definition. Section 10.2.2 [Class Symbols], page 57.

`cpp-macro`
: The start of a preprocessor macro definition. Section 10.2.8 [Literal Symbols], page 62.

`cpp-define-intro`
: The first line inside a multiline preprocessor macro if `c-syntactic-indentation-in-macros` is set. Section 10.2.9 [Multiline Macro Symbols], page 63.

`cpp-macro-cont`
: All lines inside multiline preprocessor macros if `c-syntactic-indentation-in-macros` is `nil`. Section 10.2.9 [Multiline Macro Symbols], page 63.

`friend`
: A C++ friend declaration. Section 10.2.2 [Class Symbols], page 57.

`objc-method-intro`
: The first line of an Objective-C method definition. Section 10.2.10 [Objective-C Method Symbols], page 64.

`objc-method-args-cont`
: Lines continuing an Objective-C method definition. Section 10.2.10 [Objective-C Method Symbols], page 64.

`objc-method-call-cont`
: Lines continuing an Objective-C method call. Section 10.2.10 [Objective-C Method Symbols], page 64.

`extern-lang-open`
: Brace that opens an `extern` block (e.g. `extern "C" {...}`). Section 10.2.6 [External Scope Symbols], page 61.

`extern-lang-close`
: Brace that closes an `extern` block. Section 10.2.6 [External Scope Symbols], page 61.

`inextern-lang`
: Analogous to `inclass` syntactic symbol, but used inside `extern` blocks. Section 10.2.6 [External Scope Symbols], page 61.

`namespace-open`
`namespace-close`
`innamespace`
: These are analogous to the three `extern-lang` symbols above, but are returned for C++ namespace blocks. Section 10.2.6 [External Scope Symbols], page 61.

`module-open`
`module-close`
`inmodule`
: Analogous to the above, but for CORBA IDL `module` blocks. Section 10.2.6 [External Scope Symbols], page 61.

`composition-open`
`composition-close`
`incomposition`
: Analogous to the above, but for CORBA CIDL `composition` blocks. Section 10.2.6 [External Scope Symbols], page 61.

`template-args-cont`
: C++ template argument list continuations. Section 10.2.2 [Class Symbols], page 57.

`inlambda`
: Analogous to `inclass` syntactic symbol, but used inside lambda (i.e. anonymous) functions. Used in C++ and Pike modes. Section 10.2.12 [Statement Block Symbols], page 64.

`lambda-intro-cont`
: Lines continuing the header of a lambda function, i.e. between the `lambda` keyword and the function body. Only used in Pike mode. Section 10.2.12 [Statement Block Symbols], page 64.

`inexpr-statement`
: A statement block inside an expression. The gcc C and C++ extension for this is recognized. It's also used for the special functions that take a statement block as an argument in Pike. Section 10.2.12 [Statement Block Symbols], page 64.

`inexpr-class`
: A class definition inside an expression. This is used for anonymous classes in Java. It's also used for anonymous array initializers in Java. Section 10.2.11 [Java Symbols], page 64.

10.2.1 Function Symbols

This example shows a typical function declaration.

```
1: void
2: swap( int& a, int& b )
3: {
4:     int tmp = a;
5:     a = b;
6:     b = tmp;
7:     int ignored =
8:         a + b;
9: }
```

Line 1 shows a `topmost-intro` since it is the first line that introduces a top-level construct. Line 2 is a continuation of the top-level construct introduction so it has the syntax `topmost-intro-cont`. Line 3 shows a `defun-open` since it is the brace that opens a top-level function definition. Line 9 is the corresponding `defun-close` since it contains the brace that closes the top-level function definition. Line 4 is a `defun-block-intro`, i.e. it is the first line of a brace-block, enclosed in a top-level function definition.

Lines 5, 6, and 7 are all given `statement` syntax since there isn't much special about them. Note however that line 8 is given `statement-cont` syntax since it continues the statement begun on the previous line.

10.2.2 Class related Symbols

Here's an example which illustrates some C++ class syntactic symbols:

```
 1: class Bass
 2:     : public Guitar,
 3:       public Amplifiable
 4: {
 5: public:
 6:     Bass()
 7:         : eString( new BassString( 0.105 )),
 8:           aString( new BassString( 0.085 )),
 9:           dString( new BassString( 0.065 )),
10:           gString( new BassString( 0.045 ))
11:     {
12:         eString.tune( 'E' );
13:         aString.tune( 'A' );
14:         dString.tune( 'D' );
15:         gString.tune( 'G' );
16:     }
17:     friend class Luthier;
18: };
```

As in the previous example, line 1 has the `topmost-intro` syntax. Here however, the brace that opens a C++ class definition on line 4 is assigned the `class-open` syntax. Note that in C++, classes, structs, and unions are essentially equivalent syntactically (and are very similar semantically), so replacing the `class` keyword in the example above with `struct` or `union` would still result in a syntax of `class-open` for line 4[5]. Similarly, line 18 is assigned `class-close` syntax.

Line 2 introduces the inheritance list for the class so it is assigned the `inher-intro` syntax, and line 3, which continues the inheritance list is given `inher-cont` syntax.

Hitting *C-c C-s* on line 5 shows the following analysis:

```
((inclass 58) (access-label 58))
```

The primary syntactic symbol for this line is `access-label` as this is a label keyword that specifies access protection in C++. However, because this line is also a top-level construct inside a class definition, the analysis actually shows two syntactic symbols. The other syntactic symbol assigned to this line is `inclass`. Similarly, line 6 is given both `inclass` and `topmost-intro` syntax:

```
((inclass 58) (topmost-intro 60))
```

Line 7 introduces a C++ member initialization list and as such is given `member-init-intro` syntax. Note that in this case it is *not* assigned `inclass` since this is not considered a top-level construct. Lines 8 through 10 are all assigned `member-init-cont` since they continue the member initialization list started on line 7.

---

[5] This is the case even for C and Objective-C. For consistency, structs in all supported languages are syntactically equivalent to classes. Note however that the keyword `class` is meaningless in C and Objective-C.

Line 11's analysis is a bit more complicated:

```
((inclass 58) (inline-open))
```

This line is assigned a syntax of both `inline-open` and `inclass` because it opens an *in-class* C++ inline method definition. This is distinct from, but related to, the C++ notion of an inline function in that its definition occurs inside an enclosing class definition, which in C++ implies that the function should be inlined. However, if the definition of the `Bass` constructor appeared outside the class definition, the construct would be given the `defun-open` syntax, even if the keyword `inline` appeared before the method name, as in:

```
 1: class Bass
 2:     : public Guitar,
 3:       public Amplifiable
 4: {
 5: public:
 6:     Bass();
 7: };
 8:
 9: inline
10: Bass::Bass()
11:     : eString( new BassString( 0.105 )),
12:       aString( new BassString( 0.085 )),
13:       dString( new BassString( 0.065 )),
14:       gString( new BassString( 0.045 ))
15: {
16:     eString.tune( 'E' );
17:     aString.tune( 'A' );
18:     dString.tune( 'D' );
19:     gString.tune( 'G' );
20: }
```

Returning to the previous example, line 16 is given `inline-close` syntax, while line 12 is given `defun-block-open` syntax, and lines 13 through 15 are all given `statement` syntax. Line 17 is interesting in that its syntactic analysis list contains three elements:

```
((inclass 58) (topmost-intro 380) (friend))
```

The `friend` and `inline-open` syntactic symbols are modifiers that do not have anchor positions.

Template definitions introduce yet another syntactic symbol:

```
 1: ThingManager <int,
 2:    Framework::Callback *,
 3:    Mutex> framework_callbacks;
```

Here, line 1 is analyzed as a `topmost-intro`, but lines 2 and 3 are both analyzed as `template-args-cont` lines.

10.2.3 Conditional Construct Symbols

Here is a (totally contrived) example which illustrates how syntax is assigned to various conditional constructs:

```
 1: void spam( int index )
```

```
 2: {
 3:     for( int i=0; i<index; i++ )
 4:     {
 5:         if( i == 10 )
 6:             do_something_special();
 7:         else
 8:           silly_label:
 9:             do_something( i );
10:     }
11:     do {
12:         another_thing( i-- );
13:     }
14:     while( i > 0 );
15: }
```

Only the lines that illustrate new syntactic symbols will be discussed.

Line 4 has a brace which opens a conditional's substatement block. It is thus assigned `substatement-open` syntax, and since line 5 is the first line in the substatement block, it is assigned `statement-block-intro` syntax. Line 10 contains the brace that closes the inner substatement block, and is therefore given the syntax `block-close`[6]. Line 13 is treated the same way.

Lines 6 and 9 are also substatements of conditionals, but since they don't start blocks they are given `substatement` syntax instead of `substatement-open`.

Line 8 contains a label, which is normally given `label` syntax. This one is however a bit special since it's between a conditional and its substatement. It's analyzed as `substatement-label` to let you handle this rather odd case differently from normal labels.

Line 7 start with an `else` that matches the `if` statement on line 5. It is therefore given the `else-clause` syntax and is anchored on the matching `if`. The `try-catch` constructs in C++ and Java are treated this way too, except that `catch` and (in Java) `finally`, are marked with `catch-clause`.

The `while` construct on line 14 that closes a `do` conditional is given the special syntax `do-while-closure` if it appears on a line by itself. Note that if the `while` appeared on the same line as the preceding close brace, that line would still have `block-close` syntax.

10.2.4 Switch Statement Symbols

Switch statements have their own set of syntactic symbols. Here's an example:

```
void spam( enum Ingredient i )
{
    switch( i ) {
    case Ham:
        be_a_pig();
        break;
    case Salt:
        drink_some_water();
```

[6] `block-open` is used only for "free-standing" blocks, and is somewhat rare (see Section 10.2.8 [Literal Symbols], page 62, for an example.)

```
 9:         break;
10:     default:
11:         {
12:             what_is_it();
13:             break;
14:         }
15:     }
14: }
```

Here, lines 4, 7, and 10 are all assigned `case-label` syntax, while lines 5 and 8 are assigned `statement-case-intro`. Line 11 is treated slightly differently since it contains a brace that opens a block — it is given `statement-case-open` syntax.

10.2.5 Brace List Symbols

There are a set of syntactic symbols that are used to recognize constructs inside of brace lists. A brace list is defined as an `enum` or aggregate initializer list, such as might statically initialize an array of structs. The three special aggregate constructs in Pike, `({ })`, `([ ])` and `(< >)`, are treated as brace lists too. An example:

```
1: static char* ingredients[] =
2: {
3:     "Ham",
4:     "Salt",
5:     NULL
6: };
```

Following convention, line 2 in this example is assigned `brace-list-open` syntax, and line 3 is assigned `brace-list-intro` syntax. Likewise, line 6 is assigned `brace-list-close` syntax. Lines 4 and 5 however, are assigned `brace-list-entry` syntax, as would all subsequent lines in this initializer list.

Your static initializer might be initializing nested structures, for example:

```
 1: struct intpairs[] =
 2: {
 3:     { 1, 2 },
 4:     {
 5:         3,
 6:         4
 7:     }
 8:     { 1,
 9:       2 },
10:     { 3, 4 }
11: };
```

Here, you've already seen the analysis of lines 1, 2, 3, and 11. On line 4, things get interesting; this line is assigned `brace-entry-open` syntactic symbol because it's a bracelist entry line that starts with an open brace. Lines 5 and 6 are pretty standard, and line 7 is a `brace-list-close` as you'd expect. Once again, line 8 is assigned as `brace-entry-open` as is line 10. Line 9 is assigned two syntactic elements, `brace-list-intro` with anchor point at the '`{`' of line 8[7], and `brace-list-entry` anchored on the '`1`' of line 8.

10.2.6 External Scope Symbols

External language definition blocks also have their own syntactic symbols. In this example:

```
1: extern "C"
2: {
3:     int thing_one( int );
4:     int thing_two( double );
5: }
```

line 2 is given the `extern-lang-open` syntax, while line 5 is given the `extern-lang-close` syntax. The analysis for line 3 yields:

```
((inextern-lang) (topmost-intro 14))
```

where `inextern-lang` is a modifier similar in purpose to `inclass`.

There are various other top level blocks like `extern`, and they are all treated in the same way except that the symbols are named after the keyword that introduces the block. E.g. C++ namespace blocks get the three symbols `namespace-open`, `namespace-close` and `innamespace`. The currently recognized top level blocks are:

`extern-lang-open`, `extern-lang-close`, `inextern-lang`
: `extern` blocks in C and C++.[8]

`namespace-open`, `namespace-close`, `innamespace`
: `namespace` blocks in C++.

`module-open`, `module-close`, `inmodule`
: `module` blocks in CORBA IDL.

`composition-open`, `composition-close`, `incomposition`
: `composition` blocks in CORBA CIDL.

10.2.7 Parenthesis (Argument) List Symbols

A number of syntactic symbols are associated with parenthesis lists, a.k.a argument lists, as found in function declarations and function calls. This example illustrates these:

```
1: void a_function( int line1,
2:                  int line2 );
3:
4: void a_longer_function(
5:     int line1,
6:     int line2
7:     );
```

[7] This extra syntactic element was introduced in CC Mode 5.33.1 to allow extra flexibility in indenting the second line of such a construct. You can preserve the behaviour resulting from the former syntactic analysis by giving `brace-list-entry` an offset of `c-lineup-under-anchor` (see Section 11.3.5 [Misc Line-Up], page 81).

[8] These should logically be named `extern-open`, `extern-close` and `inextern`, but that isn't the case for historical reasons.

```
 8:
 9: void call_them( int line1, int line2 )
10: {
11:     a_function(
12:         line1,
13:         line2
14:         );
15:
16:     a_longer_function( line1,
17:                        line2 );
18: }
```

Lines 5 and 12 are assigned `arglist-intro` syntax since they are the first line following the open parenthesis, and lines 7 and 14 are assigned `arglist-close` syntax since they contain the parenthesis that closes the argument list.

Lines that continue argument lists can be assigned one of two syntactic symbols. For example, Lines 2 and 17 are assigned `arglist-cont-nonempty` syntax. What this means is that they continue an argument list, but that the line containing the parenthesis that opens the list is *not empty* following the open parenthesis. Contrast this against lines 6 and 13 which are assigned `arglist-cont` syntax. This is because the parenthesis that opens their argument lists is the last character on that line.

Syntactic elements with `arglist-intro`, `arglist-cont-nonempty`, and `arglist-close` contain two buffer positions: the anchor position (the beginning of the declaration or statement) and the position of the open parenthesis. The latter position can be used in a line-up function (see Section 11.3 [Line-Up Functions], page 72).

Note that there is no `arglist-open` syntax. This is because any parenthesis that opens an argument list, appearing on a separate line, is assigned the `statement-cont` syntax instead.

10.2.8 Comment String Label and Macro Symbols

A few miscellaneous syntactic symbols that haven't been previously covered are illustrated by this C++ example:

```
void Bass::play( int volume )
const
{
    /* this line starts a multiline
     * comment.  This line should get ‘c’ syntax */

    char* a_multiline_string = "This line starts a multiline \
string.  This line should get ‘string’ syntax.";

  note:
    {
#ifdef LOCK
        Lock acquire();
#endif // LOCK
        slap_pop();
```

```
16:         cout << "I played "
17:              << "a note\n";
18:     }
19: }
```

The lines to note in this example include:

- Line 2 is assigned the `func-decl-cont` syntax.
- Line 4 is assigned both `defun-block-intro` *and* `comment-intro` syntax. A syntactic element with `comment-intro` has no anchor point — It is always accompanied by another syntactic element which does have one.
- Line 5 is assigned `c` syntax.
- Line 6 which, even though it contains nothing but whitespace, is assigned `defun-block-intro`. Note that the appearance of the comment on lines 4 and 5 do not cause line 6 to be assigned `statement` syntax because comments are considered to be *syntactic whitespace*, which are ignored when analyzing code.
- Line 8 is assigned `string` syntax.
- Line 10 is assigned `label` syntax.
- Line 11 is assigned `block-open` as well as `statement` syntax. A `block-open` syntactic element doesn't have an anchor position, since it always appears with another syntactic element which does have one.
- Lines 12 and 14 are assigned `cpp-macro` syntax in addition to the normal syntactic symbols (`statement-block-intro` and `statement`, respectively). Normally `cpp-macro` is configured to cancel out the normal syntactic context to make all preprocessor directives stick to the first column, but that's easily changed if you want preprocessor directives to be indented like the rest of the code. Like `comment-intro`, a syntactic element with `cpp-macro` doesn't contain an anchor position.
- Line 17 is assigned `stream-op` syntax.

10.2.9 Multiline Macro Symbols

Multiline preprocessor macro definitions are normally handled just like other code, i.e. the lines inside them are indented according to the syntactic analysis of the preceding lines inside the macro. The first line inside a macro definition (i.e. the line after the starting line of the cpp directive itself) gets `cpp-define-intro`. In this example:

```
1: #define LIST_LOOP(cons, listp)                         \
2:   for (cons = listp; !NILP (cons); cons = XCDR (cons)) \
3:     if (!CONSP (cons))                                 \
4:       signal_error ("Invalid list format", listp);     \
5:     else
```

line 1 is given the syntactic symbol `cpp-macro`. The first line of a cpp directive is always given that symbol. Line 2 is given `cpp-define-intro`, so that you can give the macro body as a whole some extra indentation. Lines 3 through 5 are then analyzed as normal code, i.e. `substatement` on lines 3 and 4, and `else-clause` on line 5.

The syntactic analysis inside macros can be turned off with `c-syntactic-indentation-in-macros` (see Chapter 12 [Custom Macros], page 87). In that case, lines 2 through 5 would all be given `cpp-macro-cont` with an anchor position pointing to the `#` which starts the cpp directive[9].

See Chapter 12 [Custom Macros], page 87, for more info about the treatment of macros.

10.2.10 Objective-C Method Symbols

In Objective-C buffers, there are three additional syntactic symbols assigned to various message calling constructs. Here's an example illustrating these:

```
1: - (void)setDelegate:anObject
2:           withStuff:stuff
3: {
4:     [delegate masterWillRebind:self
5:               toDelegate:anObject
6:               withExtraStuff:stuff];
7: }
```

Here, line 1 is assigned `objc-method-intro` syntax, and line 2 is assigned `objc-method-args-cont` syntax. Lines 5 and 6 are both assigned `objc-method-call-cont` syntax.

10.2.11 Java Symbols

Java has a concept of anonymous classes which can look something like this:

```
1:  @Test
2:  public void watch(Observable o) {
3:      @NonNull
4:      Observer obs = new Observer() {
5:          public void update(Observable o, Object arg) {
6:              history.addElement(arg);
7:          }
8:      };
9:      o.addObserver(obs);
10: }
```

The brace following the `new` operator opens the anonymous class. Lines 5 and 8 are assigned the `inexpr-class` syntax, besides the `inclass` symbol used in normal classes. Thus, the class will be indented just like a normal class, with the added indentation given to `inexpr-class`. An `inexpr-class` syntactic element doesn't have an anchor position.

Line 2 is assigned the `annotation-top-cont` syntax, due to it being a continuation of a topmost introduction with an annotation symbol preceding the current line. Similarly, line 4 is assigned the `annotation-var-cont` syntax due to it being a continuation of a variable declaration where preceding the declaration is an annotation.

10.2.12 Statement Block Symbols

There are a few occasions where a statement block might be used inside an expression. One is in C or C++ code using the gcc extension for this, e.g:

```
int res = ({
        int y = foo (); int z;
```

[9] This is how CC Mode 5.28 and earlier analyzed macros.

```
3:         if (y > 0) z = y; else z = - y;
4:         z;
5:     });
```

Lines 2 and 5 get the `inexpr-statement` syntax, besides the symbols they'd get in a normal block. Therefore, the indentation put on `inexpr-statement` is added to the normal statement block indentation. An `inexpr-statement` syntactic element doesn't contain an anchor position.

C++11's lambda expressions involve a block inside a statement. For example:

```
1:  std::for_each(someList.begin(), someList.end(), [&total](int x) {
2:                                                      total += x;
3:                                                  });
```

Here a lambda expressions begins at the open bracket on line 1 and ends at the closing brace on line 3. Line 2, in addition to the familiar `defun-block-intro` syntactic element, is also prefixed by an `inlambda` element, which is typically used to indent the entire lambda expression to under the opening bracket.

In Pike code, there are a few other situations where blocks occur inside statements, as illustrated here:

```
 1: array itgob()
 2: {
 3:     string s = map (backtrace()[-2][3..],
 4:                     lambda
 5:                         (mixed arg)
 6:                     {
 7:                         return sprintf ("%t", arg);
 8:                     }) * ", " + "\n";
 9:     return catch {
10:             write (s + "\n");
11:         };
12: }
```

Lines 4 through 8 contain a lambda function, which CC Mode recognizes by the `lambda` keyword. If the function argument list is put on a line of its own, as in line 5, it gets the `lambda-intro-cont` syntax. The function body is handled as an inline method body, with the addition of the `inlambda` syntactic symbol. This means that line 6 gets `inlambda` and `inline-open`, and line 8 gets `inline-close`[10].

On line 9, `catch` is a special function taking a statement block as its argument. The block is handled as an in-expression statement with the `inexpr-statement` syntax, just like the gcc extended C example above. The other similar special function, `gauge`, is handled like this too.

10.2.13 K&R Symbols

Two other syntactic symbols can appear in old style, non-prototyped C code[11]:

---

[10] You might wonder why it doesn't get `inlambda` too. It's because the closing brace is relative to the opening brace, which stands on its own line in this example. If the opening brace was hanging on the previous line, then the closing brace would get the `inlambda` syntax too to be indented correctly.

[11] a.k.a. K&R C, or Kernighan & Ritchie C

```
1: int add_three_integers(a, b, c)
2:      int a;
3:      int b;
4:      int c;
5: {
6:     return a + b + c;
7: }
```

Here, line 2 is the first line in an argument declaration list and so is given the `knr-argdecl-intro` syntactic symbol. Subsequent lines (i.e. lines 3 and 4 in this example), are given `knr-argdecl` syntax.

10.3 Indentation Calculation

Indentation for a line is calculated from the syntactic context (see Section 10.1 [Syntactic Analysis], page 49).

First, a buffer position is found whose column will be the base for the indentation calculation. It's the anchor position in the first syntactic element that provides one that is used. If no syntactic element has an anchor position then column zero is used.

Second, the syntactic symbols in each syntactic element are looked up in the `c-offsets-alist` style variable (see Section 11.1 [c-offsets-alist], page 68), which is an association list of syntactic symbols and the offsets to apply for those symbols. These offsets are added together with the base column to produce the new indentation column.

Let's use our two code examples above to see how this works. Here is our first example again:

```
1: void swap( int& a, int& b )
2: {
3:     int tmp = a;
4:     a = b;
5:     b = tmp;
6: }
```

Let's say point is on line 3 and we hit the `TAB` key to reindent the line. The syntactic context for that line is:

```
((defun-block-intro 29))
```

Since buffer position 29 is the first and only anchor position in the list, CC Mode goes there and asks for the current column. This brace is in column zero, so CC Mode uses '`0`' as the base column.

Next, CC Mode looks up `defun-block-intro` in the `c-offsets-alist` style variable. Let's say it finds the value '`4`'; it adds this to the base column '`0`', yielding a running total indentation of 4 spaces.

Since there is only one syntactic element on the list for this line, indentation calculation is complete, and the total indentation for the line is 4 spaces.

Here's another example:

```
int add( int val, int incr, int doit )
{
```

```
3:     if( doit )
4:         {
5:             return( val + incr );
6:         }
7:     return( val );
8: }
```

If we were to hit *TAB* on line 4 in the above example, the same basic process is performed, despite the differences in the syntactic context. The context for this line is:

```
((substatement-open 46))
```

Here, CC Mode goes to buffer position 46, which is the '`i`' in `if` on line 3. This character is in the fourth column on that line so the base column is '`4`'. Then CC Mode looks up the `substatement-open` symbol in `c-offsets-alist`. Let's say it finds the value '`4`'. It's added with the base column and yields an indentation for the line of 8 spaces.

Simple, huh?

Actually, it's a bit more complicated than that since the entries on `c-offsets-alist` can be much more than plain offsets. See Section 11.1 [c-offsets-alist], page 68, for the full story.

Anyway, the mode usually just does The Right Thing without you having to think about it in this much detail. But when customizing indentation, it's helpful to understand the general indentation model being used.

As you configure CC Mode, you might want to set the variable `c-echo-syntactic-information-p` to non-`nil` so that the syntactic context and calculated offset always is echoed in the minibuffer when you hit *TAB*.

11 Customizing Indentation

The principal variable for customizing indentation is the style variable `c-offsets-alist`, which gives an *offset* (an indentation rule) for each syntactic symbol. Its structure and semantics are completely described in Section 11.1 [c-offsets-alist], page 68. The various ways you can set the variable, including the use of the CC Mode style system, are described in Chapter 6 [Config Basics], page 26, and its sections, in particular Section 6.2 [Style Variables], page 29.

The simplest and most used kind of "offset" setting in `c-offsets-alist` is in terms of multiples of `c-basic-offset`:

`c-basic-offset` [User Option]

This style variable holds the basic offset between indentation levels. It's factory default is 4, but all the built-in styles set it themselves, to some value between 2 (for `gnu` style) and 8 (for `bsd`, `linux`, and `python` styles).

The most flexible "offset" setting you can make in `c-offsets-alist` is a line-up function (or even a list of them), either one supplied by CC Mode (see Section 11.3 [Line-Up Functions], page 72) or one you write yourself (see Section 11.4 [Custom Line-Up], page 84).

Finally, in Section 11.5 [Other Indentation], page 85, you'll find the tool of last resort: a hook which is called after a line has been indented. You can install functions here to make ad-hoc adjustments to any line's indentation.

11.1 c-offsets-alist

This section explains the structure and semantics of the style variable `c-offsets-alist`, the principal variable for configuring indentation. Details of how to set it up, and its relationship to CC Mode's style system are given in Section 6.2 [Style Variables], page 29.

`c-offsets-alist` [User Option]

This is an alist which associates an offset with each syntactic symbol. This *offset* is a rule specifying how to indent a line whose syntactic context matches the symbol. See Section 10.1 [Syntactic Analysis], page 49.

Note that the buffer-local binding of this alist in a CC Mode buffer contains an entry for *every* syntactic symbol. Its global binding and its settings within style specifications usually contain only a few entries. See Section 6.2 [Style Variables], page 29.

The offset specification associated with any particular syntactic symbol can be an integer, a variable name, a vector, a function or lambda expression, a list, or one of the following special symbols: `+`, `-`, `++`, `--`, `*`, or `/`. The meanings of these values are described in detail below.

Here is an example fragment of a `c-offsets-alist`, showing some of these kinds of offsets:

```
((statement . 0)
 (substatement . +)
 (cpp-macro . [0])
 (topmost-intro-cont . c-lineup-topmost-intro-cont)
```

```
  (statement-block-intro . (add c-lineup-whitesmith-in-block
                                c-indent-multi-line-block))
  ...

)
```

`c-set-offset` (*C-c C-o*) [Command]

This command changes the entry for a syntactic symbol in the current binding of `c-offsets-alist`, or it inserts a new entry if there isn't already one for that syntactic symbol.

You can use `c-set-offset` interactively within a CC Mode buffer to make experimental changes to your indentation settings. *C-c C-o* prompts you for the syntactic symbol to change (defaulting to that of the current line) and the new offset (defaulting to the current offset).

`c-set-offset` takes two arguments when used programmatically: *symbol*, the syntactic element symbol to change and *offset*, the new offset for that syntactic element. You can call the command in your `.emacs` to change the global binding of `c-offsets-alist` (see Section 6.2 [Style Variables], page 29); you can use it in a hook function to make changes from the current style. CC Mode itself uses this function when initializing styles.

The "offset specifications" in `c-offsets-alist` can be any of the following:

An integer
: The integer specifies a relative offset. All relative offsets[1] will be added together and used to calculate the indentation relative to an anchor position earlier in the buffer. See Section 10.3 [Indentation Calculation], page 66, for details. Most of the time, it's probably better to use one of the special symbols like `+` than an integer (apart from zero).

One of the symbols `+`, `-`, `++`, `--`, `*`, or `/`
: These special symbols describe a relative offset in multiples of `c-basic-offset`:

  By defining a style's indentation in terms of `c-basic-offset`, you can change the amount of whitespace given to an indentation level while maintaining the same basic shape of your code. Here are the values that the special symbols correspond to:

  `+` `c-basic-offset` times 1

  `-` `c-basic-offset` times -1

  `++` `c-basic-offset` times 2

  `--` `c-basic-offset` times -2

  `*` `c-basic-offset` times 0.5

  `/` `c-basic-offset` times -0.5

[1] The syntactic context `((defun-block-intro 2724) (comment-intro))` would likely have two relative offsets.

A vector
: The first element of the vector, an integer, sets the absolute indentation column. This will override any previously calculated indentation, but won't override relative indentation calculated from syntactic elements later on in the syntactic context of the line being indented. See Section 10.3 [Indentation Calculation], page 66. Any elements in the vector beyond the first will be ignored.

A function or lambda expression
: The function will be called and its return value will in turn be evaluated as an offset specification. Functions are useful when more context than just the syntactic symbol is needed to get the desired indentation. See Section 11.3 [Line-Up Functions], page 72, and Section 11.4 [Custom Line-Up], page 84, for details about them.

A symbol with a variable binding
: If the symbol also has a function binding, the function takes precedence over the variable. Otherwise the value of the variable is used. It must be an integer (which is used as relative offset) or a vector (an absolute offset).

A list
: The offset can also be a list containing several offset specifications; these are evaluated recursively and combined. A list is typically only useful when some of the offsets are line-up functions. A common strategy is calling a sequence of functions in turn until one of them recognizes that it is appropriate for the source line and returns a non-`nil` value.

  `nil` values are always ignored when the offsets are combined. The first element of the list specifies the method of combining the non-`nil` offsets from the remaining elements:

  `first`
  : Use the first offset that doesn't evaluate to `nil`. Subsequent elements of the list don't get evaluated.

  `min`
  : Use the minimum of all the offsets. All must be either relative or absolute - they can't be mixed.

  `max`
  : Use the maximum of all the offsets. All must be either relative or absolute - they can't be mixed.

  `add`
  : Add all the evaluated offsets together. Exactly one of them may be absolute, in which case the result is absolute. Any relative offsets that preceded the absolute one in the list will be ignored in that case.

  As a compatibility measure, if the first element is none of the above then it too will be taken as an offset specification and the whole list will be combined according to the method `first`.

If an offset specification evaluates to `nil`, then a relative offset of 0 (zero) is used[2].

---

[2] There is however a variable `c-strict-syntax-p` that when set to non-`nil` will cause an error to be signaled in that case. It's now considered obsolete since it doesn't work well with some of the alignment functions that return `nil` instead of zero. You should therefore leave `c-strict-syntax-p` set to `nil`.

11.2 Interactive Customization

As an example of how to customize indentation, let's change the style of this example[3]:

```
1: int add( int val, int incr, int doit )
2: {
3:   if( doit )
4:     {
5:       return( val + incr );
6:     }
7:   return( val );
8: }
```

to:

```
1: int add( int val, int incr, int doit )
2: {
3:   if( doit )
4:   {
5:     return( val + incr );
6:   }
7:   return( val );
8: }
```

In other words, we want to change the indentation of braces that open a block following a condition so that the braces line up under the conditional, instead of being indented. Notice that the construct we want to change starts on line 4. To change the indentation of a line, we need to see which syntactic symbols affect the offset calculations for that line. Hitting *C-c C-s* on line 4 yields:

```
((substatement-open 44))
```

so we know that to change the offset of the open brace, we need to change the indentation for the `substatement-open` syntactic symbol.

To do this interactively, just hit *C-c C-o*. This prompts you for the syntactic symbol to change, providing a reasonable default. In this case, the default is `substatement-open`, which is just the syntactic symbol we want to change!

After you hit return, CC Mode will then prompt you for the new offset value, with the old value as the default. The default in this case is '`+`', but we want no extra indentation so enter '`0`' and *RET*. This will associate the offset 0 with the syntactic symbol `substatement-open`.

To check your changes quickly, just hit *C-c C-q* (`c-indent-defun`) to reindent the entire function. The example should now look like:

---

3 In this and subsequent examples, the original code is formatted using the '`gnu`' style unless otherwise indicated. See Section 6.3 [Styles], page 30.

```
1: int add( int val, int incr, int doit )
2: {
3:   if( doit )
4:   {
5:     return( val + incr );
6:   }
7:   return( val );
8: }
```

Notice how just changing the open brace offset on line 4 is all we needed to do. Since the other affected lines are indented relative to line 4, they are automatically indented the way you'd expect. For more complicated examples, this might not always work. The general approach to take is to always start adjusting offsets for lines higher up in the file, then reindent and see if any following lines need further adjustments.

`c-set-offset` *symbol offset* [Command]

> This is the command bound to *`C-c C-o`*. It provides a convenient way to set offsets on `c-offsets-alist` both interactively (see the example above) and from your mode hook.
>
> It takes two arguments when used programmatically: *symbol* is the syntactic element symbol to change and *offset* is the new offset for that syntactic element.

11.3 Line-Up Functions

Often there are cases when a simple offset setting on a syntactic symbol isn't enough to get the desired indentation—for example, you might want to line up a closing parenthesis with the matching opening one rather than indenting relative to its "anchor point". CC Mode provides this flexibility with *line-up functions*.

The way you associate a line-up function with a syntactic symbol is described in Section 11.1 [c-offsets-alist], page 68. CC Mode comes with many predefined line-up functions for common situations. If none of these does what you want, you can write your own. See Section 11.4 [Custom Line-Up], page 84. Sometimes, it is easier to tweak the standard indentation by adding a function to `c-special-indent-hook` (see Section 11.5 [Other Indentation], page 85).

The line-up functions haven't been adapted for AWK buffers or tested with them. Some of them might work serendipitously. There shouldn't be any problems writing custom line-up functions for AWK mode.

The calling convention for line-up functions is described fully in Section 11.4 [Custom Line-Up], page 84. Roughly speaking, the return value is either an offset itself (such as `+` or `[0]`), another line-up function, or it's `nil`, meaning "this function is inappropriate in this case - try a different one". See Section 11.1 [c-offsets-alist], page 68.

The subsections below describe all the standard line-up functions, categorized by the sort of token the lining-up centers around. For each of these functions there is a "works with" list that indicates which syntactic symbols the function is intended to be used with.

11.3.1 Brace and Parenthesis Line-Up Functions

The line-up functions here calculate the indentation for braces, parentheses and statements within brace blocks.

c-lineup-close-paren [Function]

Line up the closing paren under its corresponding open paren if the open paren is followed by code. If the open paren ends its line, no indentation is added. E.g:

```
main (int,
      char **
     )                <-c-lineup-close-paren
```

and

```
main (
    int, char **
)                     <-c-lineup-close-paren
```

As a special case, if a brace block is opened at the same line as the open parenthesis of the argument list, the indentation is `c-basic-offset` instead of the open paren column. See `c-lineup-arglist` for further discussion of this “DWIM” measure.

Works with: All `*-close` symbols.

c-lineup-arglist-close-under-paren [Function]

Set your `arglist-close` syntactic symbol to this line-up function so that parentheses that close argument lists will line up under the parenthesis that opened the argument list. It can also be used with `arglist-cont` and `arglist-cont-nonempty` to line up all lines inside a parenthesis under the open paren.

As a special case, if a brace block is opened at the same line as the open parenthesis of the argument list, the indentation is `c-basic-offset` only. See `c-lineup-arglist` for further discussion of this “DWIM” measure.

Works with: Almost all symbols, but are typically most useful on `arglist-close`, `brace-list-close`, `arglist-cont` and `arglist-cont-nonempty`.

c-indent-one-line-block [Function]

Indent a one line block `c-basic-offset` extra. E.g:

```
if (n > 0)
    {m+=n; n=0;}      <-c-indent-one-line-block
<--> c-basic-offset
```

and

```
if (n > 0)
{                     <-c-indent-one-line-block
    m+=n; n=0;
}
```

The block may be surrounded by any kind of parenthesis characters. `nil` is returned if the line doesn’t start with a one line block, which makes the function usable in list expressions.

Works with: Almost all syntactic symbols, but most useful on the `-open` symbols.

c-indent-multi-line-block [Function]

Indent a multiline block `c-basic-offset` extra. E.g:

```
int *foo[] = {
    NULL,
    {17},             <-c-indent-multi-line-block
```

and

```
int *foo[] = {
    NULL,
        {               <-c-indent-multi-line-block
        17
        },
    <--> c-basic-offset
```

The block may be surrounded by any kind of parenthesis characters. `nil` is returned if the line doesn't start with a multiline block, which makes the function usable in list expressions.

Works with: Almost all syntactic symbols, but most useful on the `-open` symbols.

`c-lineup-runin-statements` [Function]

Line up statements for coding standards which place the first statement in a block on the same line as the block opening brace[4]. E.g:

```
int main()
{ puts ("Hello!");
  return 0;             <-c-lineup-runin-statements
}
```

If there is no statement after the opening brace to align with, `nil` is returned. This makes the function usable in list expressions.

Works with: The `statement` syntactic symbol.

`c-lineup-inexpr-block` [Function]

This can be used with the in-expression block symbols to indent the whole block to the column where the construct is started. E.g. for Java anonymous classes, this lines up the class under the '`new`' keyword, and in Pike it lines up the lambda function body under the '`lambda`' keyword. Returns `nil` if the block isn't part of such a construct.

Works with: `inlambda`, `inexpr-statement`, `inexpr-class`.

`c-lineup-after-whitesmith-blocks` [Function]

Compensate for Whitesmith style indentation of blocks. Due to the way CC Mode calculates anchor positions for normal lines inside blocks, this function is necessary for those lines to get correct Whitesmith style indentation. Consider the following examples:

```
int foo()
    {
    a;
    x;                   <-c-lineup-after-whitesmith-blocks
int foo()
    {
        {
        a;
        }
    x;                   <-c-lineup-after-whitesmith-blocks
```

[4] Run-in style doesn't really work too well. You might need to write your own custom line-up functions to better support this style.

The fact that the line with x is preceded by a Whitesmith style indented block in the latter case and not the first should not affect its indentation. But since CC Mode in cases like this uses the indentation of the preceding statement as anchor position, the x would in the second case be indented too much if the offset for `statement` was set simply to zero.

This lineup function corrects for this situation by detecting if the anchor position is at an open paren character. In that case, it instead indents relative to the surrounding block just like `c-lineup-whitesmith-in-block`.

Works with: `brace-list-entry`, `brace-entry-open`, `statement`, `arglist-cont`.

`c-lineup-whitesmith-in-block` [Function]

Line up lines inside a block in Whitesmith style. It's done in a way that works both when the opening brace hangs and when it doesn't. E.g:

```
something
    {
    foo;                <-c-lineup-whitesmith-in-block
    }
```

and

```
something {
    foo;                <-c-lineup-whitesmith-in-block
    }
<--> c-basic-offset
```

In the first case the indentation is kept unchanged, in the second `c-basic-offset` is added.

Works with: `defun-close`, `defun-block-intro`, `inline-close`, `block-close`, `brace-list-close`, `brace-list-intro`, `statement-block-intro`, `arglist-intro`, `arglist-cont-nonempty`, `arglist-close`, and all `in*` symbols, e.g. `inclass` and `inextern-lang`.

11.3.2 List Line-Up Functions

The line-up functions here calculate the indentation for lines which form lists of items, usually separated by commas.

The function [c-lineup-arglist-close-under-paren], page 73, which is mainly for indenting a close parenthesis, is also useful for the lines contained within parentheses.

`c-lineup-arglist` [Function]

Line up the current argument line under the first argument.

As a special case, if an argument on the same line as the open parenthesis starts with a brace block opener, the indentation is `c-basic-offset` only. This is intended as a "DWIM" measure in cases like macros that contain statement blocks, e.g:

```
A_VERY_LONG_MACRO_NAME ({
        some (code, with + long, lines * in[it]);
    });
<--> c-basic-offset
```

This is motivated partly because it's more in line with how code blocks are handled, and partly since it approximates the behavior of earlier CC Mode versions, which due to inaccurate analysis tended to indent such cases this way.

Works with: `arglist-cont-nonempty`, `arglist-close`.

`c-lineup-arglist-intro-after-paren` [Function]

Line up a line to just after the open paren of the surrounding paren or brace block.

Works with: `defun-block-intro`, `brace-list-intro`, `statement-block-intro`, `statement-case-intro`, `arglist-intro`.

`c-lineup-2nd-brace-entry-in-arglist` [Function]

Line up the second entry of a brace block under the first, when the first line is also contained in an arglist or an enclosing brace *on that line*.

I.e. handle something like the following:

```
set_line (line_t {point_t{0.4, 0.2},
                  point_t{0.2, 0.5},      <-brace-list-intro
                  .....});
         ^ enclosing parenthesis.
```

The middle line of that example will have a syntactic context with three syntactic symbols, `arglist-cont-nonempty`, `brace-list-intro`, and `brace-list-entry` (see Section 10.2.5 [Brace List Symbols], page 60).

This function is intended for use in a list. If the construct being analyzed isn't like the preceding, the function returns nil. Otherwise it returns the function `c-lineup-arglist-intro-after-paren`, which the caller then uses to perform indentation.

Works with: `brace-list-intro`.

`c-lineup-class-decl-init-+` [Function]

Line up the second entry of a class (etc.) initializer `c-basic-offset` characters in from the identifier when:

1. The type is a class, struct, union, etc. (but not an enum);
2. There is a brace block in the type declaration, specifying it; and
3. The first element of the initializer is on the same line as its opening brace.

I.e. we have a construct like this:

```
struct STR {
    int i; float f;
} str_1 = {1, 1.7},
    str_2 = {2,
         3.1          <-brace-list-intro
    };
    <--> c-basic-offset
```

Note that the syntactic context of the `brace-list-intro` line also has a syntactic element with the symbol `brace-list-entry` (see Section 10.2.5 [Brace List Symbols], page 60).

This function is intended for use in a list. If the above structure isn't present, the function returns nil, allowing a different offset specification to indent the line.

Works with: `brace-list-intro`.

c-lineup-class-decl-init-after-brace [Function]

Line up the second entry of a class (etc.) initializer after its opening brace when:

1. The type is a class, struct, union, etc. (but not an enum);
2. There is a brace block in the type declaration, specifying it; and
3. The first element of the initializer is on the same line as its opening brace.

I.e. we have a construct like this:

```
struct STR {
    int i; float f;
} str_1 = {1, 1.7},
    str_2 = {2,
             3.1      <-brace-list-intro
    };
```

Note that the syntactic context of the `brace-list-intro` line also has a syntactic element with the symbol `brace-list-entry` (see Section 10.2.5 [Brace List Symbols], page 60). Also note that this function works by returning the symbol `c-lineup-arglist-intro-after-paren`, which the caller then uses to perform the indentation.

This function is intended for use in a list. If the above structure isn't present, the function returns nil, allowing a different offset specification to indent the line.

Works with: `brace-list-intro`.

c-lineup-multi-inher [Function]

Line up the classes in C++ multiple inheritance clauses and member initializers under each other. E.g:

```
Foo::Foo (int a, int b):
    Cyphr (a),
    Bar (b)           <-c-lineup-multi-inher
```

and

```
class Foo
    : public Cyphr,
      public Bar      <-c-lineup-multi-inher
```

and

```
Foo::Foo (int a, int b)
    : Cyphr (a)
    , Bar (b)         <-c-lineup-multi-inher
```

Works with: `inher-cont`, `member-init-cont`.

c-lineup-java-inher [Function]

Line up Java implements and extends declarations. If class names follow on the same line as the '`implements`'/'`extends`' keyword, they are lined up under each other. Otherwise, they are indented by adding `c-basic-offset` to the column of the keyword. E.g:

```
class Foo
    extends
        Bar           <-c-lineup-java-inher
    <--> c-basic-offset
```

and

```
class Foo
    extends Cyphr,
            Bar      <-c-lineup-java-inher
```

Works with: `inher-cont`.

`c-lineup-java-throws` [Function]

Line up Java throws declarations. If exception names follow on the same line as the throws keyword, they are lined up under each other. Otherwise, they are indented by adding `c-basic-offset` to the column of the '`throws`' keyword. The '`throws`' keyword itself is also indented by `c-basic-offset` from the function declaration start if it doesn't hang. E.g:

```
int foo()
    throws            <-c-lineup-java-throws
        Bar           <-c-lineup-java-throws
<--><--> c-basic-offset
```

and

```
int foo() throws Cyphr,
                 Bar,    <-c-lineup-java-throws
                 Vlod    <-c-lineup-java-throws
```

Works with: `func-decl-cont`.

`c-lineup-template-args` [Function]

Line up the arguments of a template argument list under each other, but only in the case where the first argument is on the same line as the opening '`<`'.

To allow this function to be used in a list expression, `nil` is returned if there's no template argument on the first line.

Works with: `template-args-cont`.

`c-lineup-ObjC-method-call` [Function]

For Objective-C code, line up selector args as Emacs Lisp mode does with function args: go to the position right after the message receiver, and if you are at the end of the line, indent the current line c-basic-offset columns from the opening bracket; otherwise you are looking at the first character of the first method call argument, so lineup the current line with it.

Works with: `objc-method-call-cont`.

`c-lineup-ObjC-method-args` [Function]

For Objective-C code, line up the colons that separate args. The colon on the current line is aligned with the one on the first line.

Works with: `objc-method-args-cont`.

`c-lineup-ObjC-method-args-2` [Function]

Similar to `c-lineup-ObjC-method-args` but lines up the colon on the current line with the colon on the previous line.

Works with: `objc-method-args-cont`.

11.3.3 Operator Line-Up Functions

The line-up functions here calculate the indentation for lines which start with an operator, by lining it up with something on the previous line.

`c-lineup-argcont` [Function]

Line up a continued argument. E.g:

```
foo (xyz, aaa + bbb + ccc
          + ddd + eee + fff);  <-c-lineup-argcont
```

Only continuation lines like this are touched, `nil` is returned on lines which are the start of an argument.

Within a gcc `asm` block, `:` is recognized as an argument separator, but of course only between operand specifications, not in the expressions for the operands.

Works with: `arglist-cont`, `arglist-cont-nonempty`.

`c-lineup-arglist-operators` [Function]

Line up lines starting with an infix operator under the open paren. Return `nil` on lines that don't start with an operator, to leave those cases to other line-up functions. Example:

```
if (  x < 10
   || at_limit (x,     <-c-lineup-arglist-operators
                list)  <-c-lineup-arglist-operators returns nil
   )
```

Since this function doesn't do anything for lines without an infix operator you typically want to use it together with some other lineup settings, e.g. as follows (the `arglist-close` setting is just a suggestion to get a consistent style):

```
(c-set-offset 'arglist-cont
              '(c-lineup-arglist-operators 0))
(c-set-offset 'arglist-cont-nonempty
              '(c-lineup-arglist-operators c-lineup-arglist))
(c-set-offset 'arglist-close
              '(c-lineup-arglist-close-under-paren))
```

Works with: `arglist-cont`, `arglist-cont-nonempty`.

`c-lineup-assignments` [Function]

Line up the current line after the assignment operator on the first line in the statement. If there isn't any, return nil to allow stacking with other line-up functions. If the current line contains an assignment operator too, try to align it with the first one.

Works with: `topmost-intro-cont`, `statement-cont`, `arglist-cont`, `arglist-cont-nonempty`.

`c-lineup-math` [Function]

Like `c-lineup-assignments` but indent with `c-basic-offset` if no assignment operator was found on the first line. I.e. this function is the same as specifying a list `(c-lineup-assignments +)`. It's provided for compatibility with old configurations.

Works with: `topmost-intro-cont`, `statement-cont`, `arglist-cont`, `arglist-cont-nonempty`.

`c-lineup-ternary-bodies` [Function]

Line up true and false branches of a ternary operator (i.e. `?:`). More precisely, if the line starts with a colon which is a part of a said operator, align it with the corresponding question mark. For example:

```
return arg % 2 == 0 ? arg / 2
                    : (3 * arg + 1);    <-c-lineup-ternary-bodies
```

Works with: `arglist-cont`, `arglist-cont-nonempty` and `statement-cont`.

`c-lineup-cascaded-calls` [Function]

Line up "cascaded calls" under each other. If the line begins with `->` or `.` and the preceding line ends with one or more function calls preceded by the same token, then the arrow is lined up with the first of those tokens. E.g:

```
r = proc->add(17)->add(18)
        ->add(19) +         <-c-lineup-cascaded-calls
  offset;                   <-c-lineup-cascaded-calls (inactive)
```

In any other situation `nil` is returned to allow use in list expressions.

Works with: `topmost-intro-cont`, `statement-cont`, `arglist-cont`, `arglist-cont-nonempty`.

`c-lineup-streamop` [Function]

Line up C++ stream operators (i.e. '`<<`' and '`>>`').

Works with: `stream-op`.

`c-lineup-string-cont` [Function]

Line up a continued string under the one it continues. A continued string in this sense is where a string literal follows directly after another one. E.g:

```
result = prefix + "A message "
                  "string.";    <-c-lineup-string-cont
```

`nil` is returned in other situations, to allow stacking with other lineup functions.

Works with: `topmost-intro-cont`, `statement-cont`, `arglist-cont`, `arglist-cont-nonempty`.

11.3.4 Comment Line-Up Functions

The lineup functions here calculate the indentation for several types of comment structure.

`c-lineup-C-comments` [Function]

Line up C block comment continuation lines. Various heuristics are used to handle most of the common comment styles. Some examples:

```
/*                  /**                /*
 * text              * text              text
 */                  */                */

/* text             /*                 /**
   text             ** text             ** text
*/                  */                  */
```

```
/**********************************************************
 * text
 **********************************************************/
/**********************************************************
    Free form text comments:
 In comments with a long delimiter line at the
 start, the indentation is kept unchanged for lines
 that start with an empty comment line prefix.  The
 delimiter line is whatever matches the
 comment-start-skip regexp.
**********************************************************/
```

The style variable `c-comment-prefix-regexp` is used to recognize the comment line prefix, e.g. the '`*`' that usually starts every line inside a comment.

Works with: The `c` syntactic symbol.

`c-lineup-comment` [Function]

Line up a comment-only line according to the style variable `c-comment-only-line-offset`. If the comment is lined up with a comment starter on the previous line, that alignment is preserved.

`c-comment-only-line-offset` [User Option]

This style variable specifies the extra offset for the line. It can contain an integer or a cons cell of the form

(*non-anchored-offset* . *anchored-offset*)

where *non-anchored-offset* is the amount of offset given to non-column-zero anchored lines, and *anchored-offset* is the amount of offset to give column-zero anchored lines. Just an integer as value is equivalent to (*value* . -1000).

Works with: `comment-intro`.

`c-lineup-knr-region-comment` [Function]

Line up a comment in the "K&R region" with the declaration. That is the region between the function or class header and the beginning of the block. E.g:

```
int main()
/* Called at startup. */  <-c-lineup-knr-region-comment
{
  return 0;
}
```

Return `nil` if called in any other situation, to be useful in list expressions.

Works with: `comment-intro`.

11.3.5 Miscellaneous Line-Up Functions

The line-up functions here are the odds and ends which didn't fit into any earlier category.

`c-lineup-dont-change` [Function]

This lineup function makes the line stay at whatever indentation it already has; think of it as an identity function for lineups.

Works with: Any syntactic symbol.

c-lineup-under-anchor [Function]

Line up a line directly underneath its anchor point. This is like '0', except any previously calculated offset contributions are disregarded.

Works with: Any syntactic symbol which has an anchor point.

c-lineup-cpp-define [Function]

Line up macro continuation lines according to the indentation of the construct preceding the macro. E.g:

```
const char msg[] =     <-The beginning of the preceding construct.
  \"Some text.\";

#define X(A, B)  \
do {             \    <-c-lineup-cpp-define
  printf (A, B); \
} while (0)
```

and:

```
int dribble() {
  if (!running)       <-The beginning of the preceding construct.
    error(\"Not running!\");

#define X(A, B)    \
  do {             \  <-c-lineup-cpp-define
    printf (A, B); \
  } while (0)
```

If `c-syntactic-indentation-in-macros` is non-`nil`, the function returns the relative indentation to the macro start line to allow accumulation with other offsets. E.g. in the following cases, `cpp-define-intro` is combined with the `statement-block-intro` that comes from the '`do {`' that hangs on the '`#define`' line:

```
const char msg[] =
  \"Some text.\";

#define X(A, B) do { \
  printf (A, B);     \  <-c-lineup-cpp-define
  this->refs++;      \
} while (0)             <-c-lineup-cpp-define
```

and:

```
int dribble() {
  if (!running)
    error(\"Not running!\");

#define X(A, B) do { \
    printf (A, B);   \  <-c-lineup-cpp-define
    this->refs++;    \
  } while (0)           <-c-lineup-cpp-define
```

The relative indentation returned by `c-lineup-cpp-define` is zero and two, respectively, on the two lines in each of these examples. They are then added to the two column indentation that `statement-block-intro` gives in both cases here.

If the relative indentation is zero, then `nil` is returned instead. That is useful in a list expression to specify the default indentation on the top level.

If `c-syntactic-indentation-in-macros` is `nil` then this function keeps the current indentation, except for empty lines (ignoring the ending backslash) where it takes the indentation from the closest preceding nonempty line in the macro. If there's no such line in the macro then the indentation is taken from the construct preceding it, as described above.

Works with: `cpp-define-intro`.

`c-lineup-gcc-asm-reg` [Function]

Line up a gcc asm register under one on a previous line.

```
asm ("foo %1, %0\n"
     "bar %0, %1"
     : "=r" (w),
       "=r" (x)
     :  "0" (y),
        "1" (z));
```

The '`x`' line is aligned to the text after the '`:`' on the '`w`' line, and similarly '`z`' under '`y`'.

This is done only in an '`asm`' or '`__asm__`' block, and only to those lines mentioned. Anywhere else `nil` is returned. The usual arrangement is to have this routine as an extra feature at the start of arglist lineups, e.g.

```
(c-lineup-gcc-asm-reg c-lineup-arglist)
```

Works with: `arglist-cont`, `arglist-cont-nonempty`.

`c-lineup-topmost-intro-cont` [Function]

Line up declaration continuation lines zero or one indentation step[5]. For lines preceding a definition, zero is used. For other lines, `c-basic-offset` is added to the indentation. E.g:

```
int
neg (int i)           <-c-lineup-topmost-intro-cont
{
    return -i;
}
```

and

[5] This function is mainly provided to mimic the behavior of CC Mode 5.28 and earlier where this case wasn't handled consistently so that those lines could be analyzed as either topmost-intro-cont or statement-cont. It's used for `topmost-intro-cont` by default, but you might consider using `+` instead.

```
struct
larch                <-c-lineup-topmost-intro-cont
{
    double height;
}
    the_larch,       <-c-lineup-topmost-intro-cont
    another_larch;   <-c-lineup-topmost-intro-cont
<--> c-basic-offset
```

and

```
struct larch
the_larch,           <-c-lineup-topmost-intro-cont
    another_larch;   <-c-lineup-topmost-intro-cont
```

Works with: `topmost-intro-cont`.

11.4 Custom Line-Up Functions

The most flexible way to customize indentation is by writing custom line-up functions, and associating them with specific syntactic symbols (see Section 11.1 [c-offsets-alist], page 68). Depending on the effect you want, it might be better to write a `c-special-indent-hook` function rather than a line-up function (see Section 11.5 [Other Indentation], page 85).

CC Mode comes with an extensive set of predefined line-up functions, not all of which are used by the default styles. So there's a good chance the function you want already exists. See Section 11.3 [Line-Up Functions], page 72, for a list of them. If you write your own line-up function, it's probably a good idea to start working from one of these predefined functions, which can be found in the file `cc-align.el`. If you have written a line-up function that you think is generally useful, you're very welcome to contribute it; please contact `bug-cc-mode@gnu.org`.

Line-up functions are passed a single argument, the syntactic element (see below). At the time of the call, point will be somewhere on the line being indented. The return value is a `c-offsets-alist` offset specification: for example, an integer, a symbol such as `+`, a vector, `nil`[6], or even another line-up function. Full details of these are in Section 11.1 [c-offsets-alist], page 68.

Line-up functions must not move point or change the content of the buffer (except temporarily). They are however allowed to do *hidden buffer changes*, i.e. setting text properties for caching purposes etc. Buffer undo recording is disabled while they run.

The syntactic element passed as the parameter to a line-up function is a cons cell of the form

(*syntactic-symbol* . *anchor-position*)

where *syntactic-symbol* is the symbol that the function was called for, and *anchor-position* is the anchor position (if any) for the construct that triggered the syntactic symbol (see Section 10.1 [Syntactic Analysis], page 49). This cons cell is how the syntactic element of a line used to be represented in CC Mode 5.28 and earlier. Line-up functions are still passed this cons cell, so as to preserve compatibility with older configurations. In the future, we

[6] Returning `nil` is useful when the offset specification for a syntactic element is a list containing the line-up function (see Section 11.1 [c-offsets-alist], page 68).

may decide to convert to using the full list format—you can prepare your setup for this by using the access functions (`c-langelem-sym`, etc.) described below.

Some syntactic symbols, e.g. `arglist-cont-nonempty`, have more info in the syntactic element - typically other positions that can be interesting besides the anchor position. That info can't be accessed through the passed argument, which is a cons cell. Instead, you can get this information from the variable `c-syntactic-element`, which is dynamically bound to the complete syntactic element. The variable `c-syntactic-context` might also be useful - it gets dynamically bound to the complete syntactic context. See Section 8.1.1 [Custom Braces], page 41.

CC Mode provides a few functions to access parts of syntactic elements in a more abstract way. Besides making the code easier to read, they also hide the difference between the old cons cell form used in the line-up function argument and the new list form used in `c-syntactic-element` and everywhere else. The functions are:

`c-langelem-sym` *langelem* [Function]

Return the syntactic symbol in *langelem*.

`c-langelem-pos` *langelem* [Function]

Return the anchor position in *langelem*, or nil if there is none.

`c-langelem-col` *langelem* **&optional** *preserve-point* [Function]

Return the column of the anchor position in *langelem*. Also move the point to that position unless *preserve-point* is non-`nil`.

`c-langelem-2nd-pos` *langelem* [Function]

Return the secondary position in *langelem*, or `nil` if there is none.

Note that the return value of this function is always `nil` if *langelem* is in the old cons cell form. Thus this function is only meaningful when used on syntactic elements taken from `c-syntactic-element` or `c-syntactic-context`.

Sometimes you may need to use the syntactic context of a line other than the one being indented. You can determine this by (temporarily) moving point onto this line and calling `c-guess-basic-syntax` (see Section 10.1 [Syntactic Analysis], page 49).

Custom line-up functions can be as simple or as complex as you like, and any syntactic symbol that appears in `c-offsets-alist` can have a custom line-up function associated with it.

11.5 Other Special Indentations

To configure macros which you invoke without a terminating '`;`', see See Section 12.2 [Macros with ;], page 88.

Here are the remaining odds and ends regarding indentation:

`c-label-minimum-indentation` [User Option]

In '`gnu`' style (see Section 6.3.1 [Built-in Styles], page 31), a minimum indentation is imposed on lines inside code blocks. This minimum indentation is controlled by this style variable. The default value is 1.

It's the function `c-gnu-impose-minimum` that enforces this minimum indentation. It must be present on `c-special-indent-hook` to work.

`c-special-indent-hook` [User Option]

This style variable is a standard hook variable that is called after every line is indented by CC Mode. It is called only if `c-syntactic-indentation` is non-`nil` (which it is by default (see Chapter 10 [Indentation Engine Basics], page 49)). You can put a function on this hook to do any special indentation or ad hoc line adjustments your style dictates, such as adding extra indentation to constructors or destructor declarations in a class definition, etc. Sometimes it is better to write a custom Line-up Function instead (see Section 11.4 [Custom Line-Up], page 84).

The indentation engine calls each function on this hook with no parameters, with point somewhere on the pertinent line, and with the variable `c-syntactic-context` bound to the current syntactic context (i.e. what you would get by typing *C-c C-s* on the source line. See Section 8.1.1 [Custom Braces], page 41.). Note that you should not change `c-syntactic-context` or point or mark inside a `c-special-indent-hook` function; thus you'll probably want to wrap your function in a `save-excursion`[7].

Setting `c-special-indent-hook` in style definitions is handled slightly differently from other variables—A style can only add functions to this hook, not remove them. See Section 6.2 [Style Variables], page 29.

[7] The numerical value returned by `point` will change if you change the indentation of the line within a `save-excursion` form, but point itself will still be over the same piece of text.

12 Customizing Macros

Preprocessor macros in C, C++, and Objective C (introduced by `#define`) have a syntax different from the main language—for example, a macro declaration is not terminated by a semicolon, and if it is more than a line long, line breaks in it must be escaped with backslashes. CC Mode has some commands to manipulate these, see Section 12.1 [Macro Backslashes], page 87.

Normally, the lines in a multi-line macro are indented relative to each other as though they were code. You can suppress this behavior by setting the following user option:

`c-syntactic-indentation-in-macros` [User Option]

> Enable syntactic analysis inside macros, which is the default. If this is `nil`, all lines inside macro definitions are analyzed as `cpp-macro-cont`.

Sometimes you may want to indent particular directives (e.g. `#pragma`) as though they were statements. To do this, see Section 12.4 [Indenting Directives], page 89.

Because a macro can expand into anything at all, near where one is invoked CC Mode can only indent and fontify code heuristically. Sometimes it gets it wrong. Usually you should try to design your macros so that they "look like ordinary code" when you invoke them. However, two situations are so common that CC Mode handles them specially: that is when certain macros needn't (or mustn't) be followed by a '`;`', and when certain macros (or compiler directives) expand to nothing. You need to configure CC Mode to handle these macros properly, see Section 12.2 [Macros with ;], page 88, and Section 12.3 [Noise Macros], page 88.

12.1 Customizing Macro Backslashes

CC Mode provides some tools to help keep the line continuation backslashes in macros neat and tidy. Their precise action is customized with these variables:

`c-backslash-column` [User Option]
`c-backslash-max-column` [User Option]

> These variables control the alignment columns for line continuation backslashes in multiline macros. They are used by the functions that automatically insert or align such backslashes, e.g. `c-backslash-region` and `c-context-line-break`.
>
> `c-backslash-column` specifies the minimum column for the backslashes. If any line in the macro goes past this column, then the next tab stop (i.e. next multiple of `tab-width`) in that line is used as the alignment column for all the backslashes, so that they remain in a single column. However, if any lines go past `c-backslash-max-column` then the backslashes in the rest of the macro will be kept at that column, so that the lines which are too long "stick out" instead.
>
> Don't ever set these variables to `nil`. If you want to disable the automatic alignment of backslashes, use `c-auto-align-backslashes`.

`c-auto-align-backslashes` [User Option]

> Align automatically inserted line continuation backslashes if non-`nil`. When line continuation backslashes are inserted automatically for line breaks in multiline macros,

e.g. by `c-context-line-break`, they are aligned with the other backslashes in the same macro if this flag is set.

If `c-auto-align-backslashes` is `nil`, automatically inserted backslashes are preceded by a single space, and backslashes get aligned only when you explicitly invoke the command `c-backslash-region` (*C-c C-*\).

12.2 Macros with semicolons

Macros which needn't (or mustn't) be followed by a semicolon when you invoke them, *macros with semicolons*, are very common. These can cause CC Mode to parse the next line wrongly as a `statement-cont` (see Section 10.2.1 [Function Symbols], page 56) and thus mis-indent it. At the top level, a macro invocation before a defun start can cause, for example, `c-beginning-of-defun` (*C-M-a*) not to find the correct start of the current function.

You can prevent these by specifying which macros have semicolons. It doesn't matter whether or not such a macro has a parameter list:

`c-macro-names-with-semicolon` [User Option]

This buffer-local variable specifies which macros have semicolons. After setting its value, you need to call `c-make-macro-with-semi-re` for it to take effect. It should be set to one of these values:

nil There are no macros with semicolons.

a list of strings

Each string is the name of a macro with a semicolon. Only valid `#define` names are allowed here. For example, to set the default value, you could write the following into your `.emacs`:

```
(setq c-macro-names-with-semicolon
      '("Q_OBJECT" "Q_PROPERTY" "Q_DECLARE" "Q_ENUMS"))
```

a regular expression

This matches each symbol which is a macro with a semicolon. It must not match any string which isn't a valid `#define` name. For example:

```
(setq c-macro-names-with-semicolon
      "\\<\\(CLEAN_UP_AND_RETURN\\|Q_[[:upper:]]+\\)\\>")
```

`c-make-macro-with-semi-re` [Function]

Call this (non-interactive) function, which sets internal variables, each time you change the value of `c-macro-names-with-semicolon` after the major mode function has run. It takes no arguments, and its return value has no meaning. This function is called by CC Mode's initialization code, after the mode hooks have run.

12.3 Noise Macros

In CC Mode, *noise macros* are macros which expand to nothing, or compiler directives (such as GCC's `__attribute__`) which play no part in the syntax of the C (etc.) language. Some noise macros are followed by arguments in parentheses (possibly optionally), others are not.

Noise macros can easily confuse CC Mode's analysis of function headers, causing them to be mis-fontified, or even mis-indented. You can prevent this confusion by specifying the identifiers which constitute noise macros.

`c-noise-macro-names` [User Option]

This variable is a list of names of noise macros which never have parenthesized arguments. Each element is a string, and must be a valid identifier. Alternatively, the variable may be a regular expression which matches the names of such macros. Such a noise macro is treated as whitespace by CC Mode. It must not also be in, or be matched by `c-noise-macro-with-parens-names`.

`c-noise-macro-with-parens-names` [User Option]

This variable is a list of names of noise macros which optionally have arguments in parentheses. Each element of the list is a string, and must be a valid identifier. Alternatively, the variable may be a regular expression which matches the names of such macros. Such a noise macro must not also be in, or be matched by `c-noise-macro-names`. For performance reasons, such a noise macro, including any parenthesized arguments, is specially handled, but it is only handled when used in declaration contexts[1].

The two compiler directives `__attribute__` and `__declspec` have traditionally been handled specially in CC Mode; for example they are fontified with font-lock-keyword-face. You don't need to include these directives in `c-noise-macro-with-parens-names`, but doing so is OK.

`c-make-noise-macro-regexps` [Function]

Call this (non-interactive) function, which sets internal variables, on changing the value of `c-noise-macro-names` or `c-noise-macro-with-parens-names` after the major mode's function has run. This function is called by CC Mode's initialization code, after the mode hooks have run.

12.4 Indenting Directives

Sometimes you may want to indent particular preprocessor directives (e.g. `#pragma`) as though they were statements. To do this, first set up `c-cpp-indent-to-body-directives` to include the directive name(s), then enable the "indent to body" feature with `c-toggle-cpp-indent-to-body`.

`c-cpp-indent-to-body-directives` [User Option]

This variable is a list of names of CPP directives (not including the introducing '`#`') which will be indented as though statements. Each element is a string, and must be a valid identifier. The default value is `("pragma")`.

If you add more directives to this variable, or remove directives from it, whilst "indent to body" is active, you need to re-enable the feature by calling `c-toggle-cpp-indent-to-body` for these changes to take effect[2].

[1] If this restriction causes your project difficulties, please get in touch with `bug-cc-mode@gnu.org`.

[2] Note that the removal of directives doesn't work satisfactorally on XEmacs or on very old versions of Emacs

`c-toggle-cpp-indent-to-body` [Function]

With *M-x c-toggle-cpp-indent-to-body*, you enable or disable the "indent to body" feature. When called programmatically, it takes an optional numerical argument. A positive value will enable the feature, a zero or negative value will disable it.

You should set up `c-cpp-indent-to-body-directives` before calling this function, since the function sets internal state which depends on that variable.

13 Odds and Ends

The stuff that didn't fit in anywhere else is documented here.

`c-require-final-newline` [User Option]

Controls whether a final newline is enforced when the file is saved. The value is an association list that for each language mode specifies the value to give to `require-final-newline` (see Section "Saving Buffers" in *GNU Emacs Lisp Reference Manual*) at mode initialization. If a language isn't present on the association list, CC Mode won't touch `require-final-newline` in buffers for that language.

The default is to set `require-final-newline` to `t` in the languages that mandate that source files should end with newlines. These are C, C`++` and Objective-C.

`c-echo-syntactic-information-p` [User Option]

If non-`nil`, the syntactic analysis for the current line is shown in the echo area when it's indented (unless `c-syntactic-indentation` is `nil`). That's useful when finding out which syntactic symbols to modify to get the indentation you want.

`c-report-syntactic-errors` [User Option]

If non-`nil`, certain syntactic errors are reported with a ding and a message, for example when an `else` is indented for which there is no corresponding `if`.

Note however that CC Mode doesn't make any special effort to check for syntactic errors; that's the job of the compiler. The reason it can report cases like the one above is that it can't find the correct anchoring position to indent the line in that case.

Appendix A Sample .emacs File

Here's a sample .emacs file fragment that might help you along the way. Just copy this region and paste it into your .emacs file. You might want to change some of the actual values.

```
;; Make a non-standard key binding.  We can put this in
;; c-mode-base-map because c-mode-map, c++-mode-map, and so on,
;; inherit from it.
(defun my-c-initialization-hook ()
  (define-key c-mode-base-map "\C-m" 'c-context-line-break))
(add-hook 'c-initialization-hook 'my-c-initialization-hook)

;; offset customizations not in my-c-style
;; This will take precedence over any setting of the syntactic symbol
;; made by a style.
(setq c-offsets-alist '((member-init-intro . ++)))

;; Create my personal style.
(defconst my-c-style
  '((c-tab-always-indent        . t)
    (c-comment-only-line-offset . 4)
    (c-hanging-braces-alist     . ((substatement-open after)
                                   (brace-list-open)))
    (c-hanging-colons-alist     . ((member-init-intro before)
                                   (inher-intro)
                                   (case-label after)
                                   (label after)
                                   (access-label after)))
    (c-cleanup-list             . (scope-operator
                                   empty-defun-braces
                                   defun-close-semi))
    (c-offsets-alist            . ((arglist-close . c-lineup-arglist)
                                   (substatement-open . 0)
                                   (case-label        . 4)
                                   (block-open        . 0)
                                   (knr-argdecl-intro . -)))
    (c-echo-syntactic-information-p . t))
  "My C Programming Style")
(c-add-style "PERSONAL" my-c-style)

;; Customizations for all modes in CC Mode.
(defun my-c-mode-common-hook ()
  ;; set my personal style for the current buffer
  (c-set-style "PERSONAL")
  ;; other customizations
  (setq tab-width 8
        ;; this will make sure spaces are used instead of tabs
```

```
        indent-tabs-mode nil)
  ;; we like auto-newline, but not hungry-delete
  (c-toggle-auto-newline 1))
(add-hook 'c-mode-common-hook 'my-c-mode-common-hook)
```

Appendix B Performance Issues

C and its derivative languages are highly complex creatures. Often, ambiguous code situations arise that require CC Mode to scan large portions of the buffer to determine syntactic context. Such pathological code can cause CC Mode to perform fairly badly. This section gives some insight in how CC Mode operates, how that interacts with some coding styles, and what you can use to improve performance.

The overall goal is that CC Mode shouldn't be overly slow (i.e. take more than a fraction of a second) in any interactive operation. I.e. it's tuned to limit the maximum response time in single operations, which is sometimes at the expense of batch-like operations like reindenting whole blocks. If you find that CC Mode gradually gets slower and slower in certain situations, perhaps as the file grows in size or as the macro or comment you're editing gets bigger, then chances are that something isn't working right. You should consider reporting it, unless it's something that's mentioned in this section.

Because CC Mode has to scan the buffer backwards from the current insertion point, and because C's syntax is fairly difficult to parse in the backwards direction, CC Mode often tries to find the nearest position higher up in the buffer from which to begin a forward scan (it's typically an opening or closing parenthesis of some kind). The farther this position is from the current insertion point, the slower it gets.

In earlier versions of CC Mode, we used to recommend putting the opening brace of a top-level construct[1] into the leftmost column. Earlier still, this used to be a rigid Emacs constraint, as embodied in the `beginning-of-defun` function. CC Modenow caches syntactic information much better, so that the delay caused by searching for such a brace when it's not in column 0 is minimal, except perhaps when you've just moved a long way inside the file.

A special note about `defun-prompt-regexp` in Java mode: The common style is to hang the opening braces of functions and classes on the right side of the line, and that doesn't work well with the Emacs approach. CC Mode comes with a constant `c-Java-defun-prompt-regexp` which tries to define a regular expression usable for this style, but there are problems with it. In some cases it can cause `beginning-of-defun` to hang[2]. For this reason, it is not used by default, but if you feel adventurous, you can set `defun-prompt-regexp` to it in your mode hook. In any event, setting and relying on `defun-prompt-regexp` will definitely slow things down because (X)Emacs will be doing regular expression searches a lot, so you'll probably be taking a hit either way!

CC Mode maintains a cache of the opening parentheses of the blocks surrounding the point, and it adapts that cache as the point is moved around. That means that in bad cases it can take noticeable time to indent a line in a new surrounding, but after that it gets fast as long as the point isn't moved far off. The farther the point is moved, the less useful is the cache. Since editing typically is done in "chunks" rather than on single lines far apart from each other, the cache typically gives good performance even when the code doesn't fit the Emacs approach to finding the defun starts.

XEmacs users can set the variable `c-enable-xemacs-performance-kludge-p` to non-`nil`. This tells CC Mode to use XEmacs-specific built-in functions which, in some circum-

---

[1] E.g. a function in C, or outermost class definition in C++ or Java.

[2] This has been observed in Emacs 19.34 and XEmacs 19.15.

stances, can locate the top-most opening brace much more quickly than `beginning-of-defun`. Preliminary testing has shown that for styles where these braces are hung (e.g. most JDK-derived Java styles), this hack can improve performance of the core syntax parsing routines from 3 to 60 times. However, for styles which *do* conform to Emacs' recommended style of putting top-level braces in column zero, this hack can degrade performance by about as much. Thus this variable is set to `nil` by default, since the Emacs-friendly styles should be more common (and encouraged!). Note that this variable has no effect in Emacs since the necessary built-in functions don't exist (in Emacs 22.1 as of this writing in February 2007).

Text properties are used to speed up skipping over syntactic whitespace, i.e. comments and preprocessor directives. Indenting a line after a huge macro definition can be slow the first time, but after that the text properties are in place and it should be fast (even after you've edited other parts of the file and then moved back).

Font locking can be a CPU hog, especially the font locking done on decoration level 3 which tries to be very accurate. Note that that level is designed to be used with a font lock support mode that only fontifies the text that's actually shown, i.e. Lazy Lock or Just-in-time Lock mode, so make sure you use one of them. Fontification of a whole buffer with some thousand lines can often take over a minute. That is a known weakness; the idea is that it never should happen.

The most effective way to speed up font locking is to reduce the decoration level to 2 by setting `font-lock-maximum-decoration` appropriately. That level is designed to be as pretty as possible without sacrificing performance. See Section 5.1 [Font Locking Preliminaries], page 20, for more info.

Appendix C Limitations and Known Bugs

- CC Mode doesn't support trigraphs. (These are character sequences such as '`??(`', which represents '`[`'. They date from a time when some character sets didn't have all the characters that C needs, and are now utterly obsolete.)
- There is no way to apply auto newline settings (see Section 4.7 [Auto-newlines], page 15) on already typed lines. That's only a feature to ease interactive editing.

  To generalize this issue a bit: CC Mode is not intended to be used as a reformatter for old code in some more or less batch-like way. With the exception of some functions like `c-indent-region`, it's only geared to be used interactively to edit new code. There's currently no intention to change this goal.

  If you want to reformat old code, you're probably better off using some other tool instead, e.g. Section "GNU indent" in *The 'indent' Manual*, which has more powerful reformatting capabilities than CC Mode.
- The support for C`++` templates (in angle brackets) is not yet complete. When a non-nested template is used in a declaration, CC Mode indents it and font-locks it OK. Templates used in expressions, and nested templates do not fare so well. Sometimes a workaround is to refontify the expression after typing the closing '`>`'.
- In a *k&r region* (the part of an old-fashioned C function declaration which specifies the types of its parameters, coming between the parameter list and the opening brace), there should be at most 20 top-level parenthesis and bracket pairs. This limit has been imposed for performance reasons. If it is violated, the source file might be incorrectly indented or fontified.
- On loading CC Mode, sometimes this error message appears:

  ```
  File mode specification error: (void-variable c-font-lock-keywords-3)
  ```

  This is due to a bug in the function `eval-after-load` in some versions of (X)Emacs. It can manifest itself when there is a symbolic link in the path of the directory which contains (X)Emacs. As a workaround, put the following into your `.emacs` file, fairly early on:

  ```
  (defun my-load-cc-fonts ()
    (require "cc-fonts"))
  (add-hook 'c-initialization-hook 'my-load-cc-fonts)
  ```

Appendix D Frequently Asked Questions

- *How can I change the indent level from 4 spaces to 2 spaces?*

  Set the variable `c-basic-offset`. See Chapter 3 [Getting Started], page 3.

- *Why doesn't the `RET` key indent the new line?*

  Emacs' convention is that *RET* just adds a newline, and that *C-j* adds a newline and indents it. You can make *RET* do this too by adding this to your `c-initialization-hook`:

  ```
  (define-key c-mode-base-map "\C-m" 'c-context-line-break)
  ```

  See Chapter 3 [Getting Started], page 3. This is a very common question. If you want this to be the default behavior, don't lobby us, lobby RMS! `:-)`

- *How do I get block comments in my C++ files?*

  Interactively, change the comment style with *C-c C-k*. See Section 4.5 [Minor Modes], page 11.

  To configure this setting, say, for files within the gdb project, you could amend your C++ Mode hook like this:

  ```
  (defun my-c++-mode-hook ()
    (if (string-match "/gdb/" (buffer-file-name))
        (c-toggle-comment-style 1)))
  (add-hook 'c++-mode-hook 'my-c++-mode-hook)
  ```

- *How do I stop my C++ lambda expressions being indented way over to the right?*

  This is now the default, so you don't need to do anything. To restore the previous default, indenting lambda expressions to the right of the constructs which introduce them, change the offset associated with `inlambda` from 0 to `c-lineup-inexpr-block`. For example, if you are setting offsets in a hook function you might include the following line:

  ```
  (c-set-offset 'inlambda 'c-lineup-inexpr-block)
  ```

  For details of the different ways you can make this setting, Chapter 6 [Config Basics], page 26.

- *How do I stop my code jumping all over the place when I type?*

  Deactivate "electric minor mode" with *C-c C-l*. See Chapter 3 [Getting Started], page 3.

- *How do I reindent the whole file?*

  Visit the file and hit *C-x h* to mark the whole buffer. Then hit *C-M-*. See Section 4.1 [Indentation Commands], page 5.

- *How do I reindent the current block?*

  First move to the brace which opens the block with *C-M-u*, then reindent that expression with *C-M-q*. See Section 4.1 [Indentation Commands], page 5.

- *I put* `(c-set-offset 'substatement-open 0)` *in my* `.emacs` *file but I get an error saying that* `c-set-offset`*'s function definition is void. What's wrong?*

  This means that CC Mode hasn't yet been loaded into your Emacs session by the time the `c-set-offset` call is reached, most likely because CC Mode is being autoloaded.

Instead of putting the `c-set-offset` line in your top-level `.emacs` file, put it in your `c-initialization-hook` (see Section 6.1 [CC Hooks], page 28), or simply modify `c-offsets-alist` directly:

```
(setq c-offsets-alist '((substatement-open . 0)))
```

- *I have an open paren character at column zero inside a comment or multiline string literal, and it causes the fontification and/or indentation to go haywire. What gives?*

  It's due to the ad-hoc rule in (X)Emacs that such open parens always start defuns (which translates to functions, classes, namespaces or any other top-level block constructs in the CC Mode languages). See Section "Left Margin Paren" in *GNU Emacs Manual*, for details (See Section "Defuns" in *GNU Emacs Manual*, in the Emacs 20 manual).

  This heuristic is built into the core syntax analysis routines in (X)Emacs, so it's not really a CC Mode issue. However, in Emacs 21.1 it became possible to turn it off[1] and CC Mode does so there since it's got its own system to keep track of blocks.

[1] Using the variable `open-paren-in-column-0-is-defun-start`.

Appendix E Getting the Latest CC Mode Release

CC Mode has been standard with all versions of Emacs since 19.34 and of XEmacs since 19.16.

Due to release schedule skew, it is likely that all of these Emacsen have old versions of CC Mode and so should be upgraded. Access to the CC Mode source code, as well as more detailed information on Emacsen compatibility, etc. are all available on the web site:

```
http://cc-mode.sourceforge.net/
```

Appendix F Mailing Lists and Submitting Bug Reports

To report bugs, use the *C-c C-b* (bound to `c-submit-bug-report`) command. This provides vital information we need to reproduce your problem. Make sure you include a concise, but complete code example. Please try to boil your example down to just the essential code needed to reproduce the problem, and include an exact recipe of steps needed to expose the bug. Be especially sure to include any code that appears *before* your bug example, if you think it might affect our ability to reproduce it.

Please try to produce the problem in an Emacs instance without any customizations loaded (i.e. start it with the '`-q --no-site-file`' arguments). If it works correctly there, the problem might be caused by faulty customizations in either your own or your site configuration. In that case, we'd appreciate it if you isolate the Emacs Lisp code that triggers the bug and include it in your report.

Bug reports should be sent to `bug-cc-mode@gnu.org`. You can also send other questions and suggestions (kudos? `;-)`) to that address. It's a mailing list which you can join or browse an archive of; see the web site at `http://cc-mode.sourceforge.net/` for further details.

If you want to get announcements of new CC Mode releases, send the word *subscribe* in the body of a message to `cc-mode-announce-request@lists.sourceforge.net`. It's possible to subscribe from the web site too. Announcements will also be posted to the Usenet newsgroups `gnu.emacs.sources`, `comp.emacs`, `comp.emacs.xemacs`, `comp.lang.c`, `comp.lang.c++`, `comp.lang.objective-c`, `comp.lang.java.softwaretools`, `comp.lang.idl`, and `comp.lang.awk`.

Command and Function Index

Since most CC Mode commands are prepended with the string '`c-`', each appears under its `c-`*`thing`* name and its *`thing`* `(c-)` name.

D

E

F

G

H

I

J

K

L

M

N

O

P

S

T

U

V

Variable Index

Since most CC Mode variables are prepended with the string '`c-`', each appears under its `c-`*`thing`* name and its *`thing`* `(c-)` name.

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

Concept and Key Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

W

Short Contents

Table of Contents