CC Mode 5.28

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

Barry A. Warsaw, Martin Stjernholm

1 Introduction

Welcome to CC Mode, a GNU Emacs mode for editing files containing C, C++, Objective-C, Java, CORBA IDL, and Pike code. This incarnation of the mode is descendant from '`c-mode.el`' (also called `"`Boring Old C Mode`"` or BOCM `:-)`, and '`c++-mode.el`' version 2, which Barry has been maintaining since 1992. CC Mode represents a significant milestone in the mode's life. It has been fully merged back with Emacs 19's '`c-mode.el`'. Also a new, more intuitive and flexible mechanism for controlling indentation has been developed. Late in 1997, Martin joined the CC Mode Maintainers Team, and implemented the Pike support. As of 2000 Martin has taken over as the sole maintainer.

This manual describes CC Mode version 5.28.

CC Mode supports the editing of K&R and ANSI C, *ARM*[1] C++, Objective-C, Java, CORBA's Interface Definition Language, and Pike[2] files. In this way, you can easily set up consistent coding styles for use in editing all of these languages. CC Mode does *not* handle font-locking (a.k.a. syntax coloring, keyword highlighting) or anything of that nature, for any of these modes. Font-locking is handled by other Emacs packages.

This manual will describe the following:

- How to get started using CC Mode.
- How the new indentation engine works.
- How to customize the new indentation engine.

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`, and `pike-mode` entry points are provided. This package is intended to be a replacement for '`c-mode.el`' and '`c++-mode.el`'.

This distribution also contains a file called '`cc-compat.el`' which should ease your transition from BOCM to CC Mode. If you have a BOCM configuration you are really happy with, and want to postpone learning how to configure CC Mode, take a look at that file. It maps BOCM configuration variables to CC Mode's new indentation model. It is not actively supported so for the long run, you should learn how to customize CC Mode to support your coding style.

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] *The Annotated C++ Reference Manual*, by Ellis and Stroustrup.

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

2 Getting Connected

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 may not have the latest CC Mode release and may want to upgrade your copy.

If you are upgrading an existing CC Mode installation, please see the '`README`' file for installation details. CC Mode may not work with older versions of Emacs or XEmacs. See the CC Mode release notes Web pages for the latest information on Emacs version and package compatibility (see Appendix B [Getting the Latest CC Mode Release], page 59).

Note that CC Mode no longer works with Emacs 18!, so if you haven't upgraded from Emacs 18 by now, you are out of luck.

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.

3 New Indentation Engine

CC Mode has a new indentation engine, providing a simplified, yet flexible and general mechanism for customizing indentation. It separates indentation calculation into two steps: first, CC Mode analyzes the line of code being indented to determine the kind of language construct it's looking at, then it applies user defined offsets to the current line based on this analysis.

This section will briefly cover how indentation is calculated in CC Mode. It is important to understand the indentation model being used so that you will know how to customize CC Mode for your personal coding style.

3.1 Syntactic Analysis

The first thing CC Mode does when indenting a line of code, is to analyze the line, determining the *syntactic component list* of the construct on that line. A syntactic component consists of a pair of information (in lisp parlance, a *cons cell*), where the first part is a *syntactic symbol*, and the second part is a *relative buffer position*. Syntactic symbols describe elements of C code[1], e.g. `statement`, `substatement`, `class-open`, `class-close`, etc. See Chapter 8 [Syntactic Symbols], page 35, for a complete list of currently recognized syntactic symbols and their semantics. The style variable `c-offsets-alist` also contains the list of currently supported syntactic symbols.

Conceptually, a line of C code is always indented relative to the indentation of some line higher up in the buffer. This is represented by the relative buffer position in the syntactic component.

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 the command *C-c C-s* (`c-show-syntactic-information`) to simply report what the syntactic analysis is for the current line. Running this command on line 4 of this example, we'd see in the echo area[3]:

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

[1] Unless otherwise noted, the term "C code" to refers to all the C-like languages.

[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.

This tells us that the line is a statement and it is indented relative to buffer position 35, which happens to be the ‘`i`’ in `int` on line 3. 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: }
```

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 component lists can contain more than one component, and individual syntactic components need not have relative buffer positions. The most common example of this is a line that contains 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 component list contains two syntactic components. Also notice that the first component, ‘`(comment-intro)`’ has no relative buffer position.

[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.

3.2 Indentation Calculation

Indentation for a line is calculated using the syntactic component list derived in step 1 above (see Section 3.1 [Syntactic Analysis], page 3). Each component contributes to the final total indentation of the line in two ways.

First, the syntactic symbols are looked up in the `c-offsets-alist` style variable, which is an association list of syntactic symbols and the offsets to apply for those symbols. These offsets are added to a running total.

Second, if the component has a relative buffer position, CC Mode adds the column number of that position to the running total. By adding up the offsets and columns for every syntactic component on the list, the final total indentation for the current line is computed.

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 re-indent the line. Remember that the syntactic component list for that line is:

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

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 running total (initialized to zero), yielding a running total indentation of 4 spaces.

Next CC Mode goes to buffer position 29 and asks for the current column. This brace is in column zero, so CC Mode adds '`0`' to the running total. Since there is only one syntactic component on the list for this line, indentation calculation is complete, and the total indentation for the line is 4 spaces.

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: }
```

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 component list. Remember that the list for this line is:

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

Here, CC Mode first looks up the `substatement-open` symbol in `c-offsets-alist`. Let's say it finds the value '`4`'. This yields a running total of 4. CC Mode then 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 adding this to the running total yields an indentation for the line of 8 spaces.

Simple, huh?

Actually, 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 component list and calculated offset will always be echoed in the minibuffer when you hit *TAB*.

4 Minor Modes

CC Mode contains two minor-mode-like features that you should find useful while you enter new C code. The first is called *auto-newline* mode, and the second is called *hungry-delete* mode. These minor modes can be toggled on and off independently, and CC Mode can be configured so that it starts up with any combination of these minor modes. By default, both of these minor modes are turned off.

The state of the minor modes is always reflected in the minor mode list on the modeline of the CC Mode buffer. When auto-newline mode is enabled, you will see '`C/a`' on the mode line[1]. When hungry delete mode is enabled you would see '`C/h`' and when both modes are enabled, you'd see '`C/ah`'.

CC Mode provides key bindings which allow you to toggle the minor modes on the fly while editing code. To toggle just the auto-newline state, hit *`C-c C-a`* (`c-toggle-auto-state`). When you do this, you should see the '`a`' indicator either appear or disappear on the modeline. Similarly, to toggle just the hungry-delete state, use *`C-c C-d`* (`c-toggle-hungry-state`), and to toggle both states, use *`C-c C-t`* (`c-toggle-auto-hungry-state`).

To set up the auto-newline and hungry-delete states to your preferred values, you would need to add some lisp to your '`.emacs`' file that called one of the `c-toggle-*-state` functions directly. When called programmatically, each function takes a numeric value, where a positive number enables the minor mode, a negative number disables the mode, and zero toggles the current state of the mode.

So for example, if you wanted to enable both auto-newline and hungry-delete for all your C file editing, you could add the following to your '`.emacs`' file:

```
(add-hook 'c-mode-common-hook
  (lambda () (c-toggle-auto-hungry-state 1)))
```

4.1 Auto-newline Insertion

Auto-newline minor mode works by enabling certain *electric commands*. Electric commands are typically bound to special characters such as the left and right braces, colons, semi-colons, etc., which when typed, perform some magic formatting in addition to inserting the typed character. As a general rule, electric commands are only electric when the following conditions apply:

- Auto-newline minor mode is enabled, as evidenced by a '`C/a`' or '`C/ah`' indicator on the modeline.
- The character was not typed inside of a literal[2].
- No numeric argument was supplied to the command (i.e. it was typed as normal, with no *`C-u`* prefix).

[1] The '`C`' would be replaced with '`C++`', '`ObjC`', '`Java`', '`IDL`', or '`Pike`' for the respective languages.

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

4.1.1 Hanging Braces

When you type either an open or close brace (i.e. `{` or `}`), the electric command `c-electric-brace` gets run. This command has two electric formatting behaviors. First, it will perform some re-indentation of the line the brace was typed on, and second, it will add various newlines before and/or after the typed brace. Re-indentation occurs automatically whenever the electric behavior is enabled. If the brace ends up on a line other than the one it was typed on, then that line is also re-indented.

The default in auto-newline mode is to insert newlines both before and after a brace, but that can be controlled by the `c-hanging-braces-alist` style variable. This variable contains a mapping between syntactic symbols related to braces, and a list of places to insert a newline. The syntactic symbols that are useful for this list are: `class-open`, `class-close`, `defun-open`, `defun-close`, `inline-open`, `inline-close`, `brace-list-open`, `brace-list-close`, `brace-list-intro`, `brace-entry-open`, `block-open`, `block-close`, `substatement-open`, `statement-case-open`, `extern-lang-open`, `extern-lang-close`, `namespace-open`, `namespace-close`, `inexpr-class-open`, and `inexpr-class-close`[3]. See Chapter 8 [Syntactic Symbols], page 35, for a more detailed description of these syntactic symbols, except for `inexpr-class-open` and `inexpr-class-close`, which aren't actual syntactic symbols.

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[4].

The value associated with each syntactic symbol in this association list is called an *ACTION* which can be either a function or a list. See Section 7.5.2 [Custom Brace and Colon Hanging], page 31, for a more detailed discussion of using a function as a brace hanging *ACTION*.

When the *ACTION* is a list, it can contain any 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 is said to *hang* on the right side of the line, as in:

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

[3] 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.

[4] The braces of anonymous classes produces 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 no newlines are added either before or 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)
 (substatement-open after)
 (block-close . c-snug-do-while)
 (extern-lang-open after)
 (inexpr-class-open after)
 (inexpr-class-close before))
```

which says that `brace-list-open` and `brace-entry-open` 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.

A word of caution: it is not a good idea to hang top-level construct introducing braces, such as `class-open` or `defun-open`. Emacs makes an assumption that such braces will always appear in column zero, hanging them can introduce performance problems. See Chapter 10 [Performance Issues], page 54, for more information.

4.1.2 Hanging Colons

Using a mechanism similar to brace hanging (see Section 4.1.1 [Hanging Braces], page 8), colons can also be made to hang using the style variable `c-hanging-colons-alist`. The syntactic symbols appropriate for this association list are: `case-label`, `label`, `access-label`, `member-init-intro`, and `inher-intro`. Note however that for `c-hanging-colons-alist`, *ACTION*s as functions are not supported. See also Section 7.5.2 [Custom Brace and Colon Hanging], page 31 for details.

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 Section 4.1.5 [Clean-ups], page 10, for details.

4.1.3 Hanging Semi-colons and Commas

Semicolons and commas are also electric in CC Mode, but since these characters do not correspond directly to syntactic symbols, a different mechanism is used to determine whether newlines should be automatically inserted after these characters. See Section 7.5.3 [Customizing Semi-colons and Commas], page 32, for details.

4.1.4 Other Electric Commands

A few other keys also provide electric behavior. For example `#` (`c-electric-pound`) is electric when typed as the first non-whitespace character on a line. 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 C preprocessor macro definitions.

Stars and slashes (i.e. `*` and `/`, `c-electric-star` and `c-electric-slash` respectively) are also electric under certain circumstances. If a star is inserted as the second character of a C style block comment on a comment-only line, then the comment delimiter is indented as defined by `c-offsets-alist`. A comment-only line is defined as a line which contains only a comment, as in:

```
void spam( int i )
{
        // this is a comment-only line...
    if( i == 7 )                             // but this is not
    {
        dosomething(i);
    }
}
```

Likewise, if a slash is inserted as the second slash in a C++ style line comment (also only on a comment-only line), then the line is indented as defined by `c-offsets-alist`.

Less-than and greater-than signs (`c-electric-lt-gt`) are also electric, but only in C++ mode. Hitting the second of two `<` or `>` keys re-indents the line if it is a C++ style stream operator.

The normal parenthesis characters '`(`' and '`)`' also reindent the current line if they are used in normal code. This is useful for getting the closing parenthesis of an argument list aligned automatically.

4.1.5 Clean-ups

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

Most of the clean-ups are only applicable to counteract automatically inserted newlines, and will therefore only have any effect if the auto-newline minor mode is turned on. Others will work all the time.

You can 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. Note that clean-ups are only performed when the construct does not occur within a literal (see Section 4.1 [Auto-newline Insertion], page 7), and when there is nothing but whitespace appearing between the individual components of the construct.

These are the clean-ups that only are active in the auto-newline minor mode:

- `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 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 open parenthesis is typed:

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

  and like this after the 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 semi-colon on top-level function or class definitions when they follow a close brace. Clean up occurs when the semi-colon is typed. So for example, the following:

  ```
  class Spam
  {
  }
  ;
  ```

  is transformed into this when the semi-colon 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.
- `scope-operator` — Clean up double colons which may designate a C++ scope operator split across multiple lines[5]. 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.

The following clean-ups are always active when they occur on `c-cleanup-list`, and are thus not affected by the auto-newline minor mode:

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

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

4.2 Hungry-deletion of Whitespace

Hungry deletion of whitespace, or as it more commonly called, *hungry-delete mode*, is a simple feature that some people find extremely useful. In fact, you might find yourself wanting hungry-delete in **all** your editing modes!

In a nutshell, when hungry-delete mode is enabled, hitting the ⟨Backspace⟩ key[6] will consume all preceding whitespace, including newlines and tabs. This can really cut down on the number of ⟨Backspace⟩'s you have to type if, for example you made a mistake on the preceding line.

By default, when you hit the ⟨Backspace⟩ key CC Mode runs the command `c-electric-backspace`, which deletes text in the backwards direction. When deleting a single character, or when ⟨Backspace⟩ is hit in a literal (see Section 4.1 [Auto-newline Insertion], page 7), or when hungry-delete mode is disabled, the function contained in the `c-backspace-function` variable is called with one argument (the number of characters to delete). This variable is set to `backward-delete-char-untabify` by default.

The default behavior of the ⟨Delete⟩ key depends on the flavor of Emacs you are using. By default in XEmacs 20.3 and beyond, the ⟨Delete⟩ key is bound to `c-electric-delete`. You control the direction that the ⟨Delete⟩ key deletes by setting the variable `delete-key-deletes-forward`, a standard XEmacs variable. When this variable is non-`nil` and hungry-delete mode is enabled, `c-electric-delete` will consume all whitespace *following* point. When `delete-key-deletes-forward` is `nil`, it deletes all whitespace *preceding* point[7] When deleting a single character, or if ⟨Delete⟩ is hit in a literal, or hungry-delete mode is disabled, the function contained in `c-delete-function` is called with one argument: the number of characters to delete. This variable is set to `delete-char` by default.

In Emacs 19 or Emacs 20, both the ⟨Delete⟩ and ⟨Backspace⟩ keys are bound to `c-electric-backspace`, however you can change this by explicitly binding `[delete]`[8].

---

[6] I say "hit the ⟨Backspace⟩ key" but what I really mean is "when Emacs receives the `BackSpace` key event." The difference usually isn't significant to most users, but advanced users will realize that under window systems such as X, any physical key (keycap) on the keyboard can be configured to generate any keysym, and thus any Emacs key event. Also, the use of Emacs on TTYs will affect which keycap generates which key event. From a pedantic point of view, here we are only concerned with the key event that Emacs receives.

[7] i.e. it literally calls `c-electric-backspace`.

[8] E.g. to `c-electric-delete` in your '`.emacs`' file. Note however, that Emacs 20 does not have a standard variable such as `delete-key-deletes-forward`.

XEmacsen older than 20.3 behave similar to Emacs 19 and Emacs 20.

5 Text 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. The goal is to do it as seamlessly as possible, 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 should keep the indentation, fix the comment line decorations, and so on, for you. It does that 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 style of line decoration that starts every line in a comment. The style variable `c-comment-prefix-regexp` contains the regexp used to recognize this *comment line prefix*. The default is '`//+\\|\\**`', which matches C++ style line comments like

```
// blah blah
```

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

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

with zero or more stars at the beginning of every line. If you change that 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, you need to reinitialize the program mode if you change it inside a CC Mode buffer.

Line breaks are by default handled (almost) the same regardless whether they are made by auto fill mode (see section "Auto Fill" in *The Emacs Editor*), 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 (may be changed with the `string` syntactic symbol). 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 *The Emacs Editor*) to make Emacs correctly keep the line prefix when filling paragraphs. That also makes 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 kept intact:

[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`.

```
/* 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)
```

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 clue about how the prefix should look, namely when a block comment is broken for the first time. The string in the style variable `c-block-comment-prefix`[3] is used in that case. It defaults to '`* `', 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 justify the indentation by putting leading spaces in the `c-block-comment-prefix` string, since CC Mode still uses the normal indentation engine to indent the line. Thus, the right way to fix the indentation is by setting the `c` syntactic symbol. It defaults to `c-lineup-C-comments`, which handles the indentation of most common comment styles, see Chapter 9 [Indentation Functions], page 48.

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 a string literal. This behavior can be controlled with the `c-ignore-auto-fill` variable. It takes a list of symbols for the different contexts where auto-filling never should occur:

[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 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.

- `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 auto-filling only occurs in comments when auto-fill mode is activated. 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.

The commands that does the actual work follows.

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

This is the replacement for `fill-paragraph` in CC Mode buffers. It's used to fill 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 function keeps the comment starters and enders of block comments as they were before the filling. This means that a comment ender on the same line as the paragraph being filled will be filled with the paragraph, and one on a line by itself will stay as it is. The comment starter is handled similarly[4].

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

This is the replacement for `indent-new-comment-line`. It breaks the line at point and indents the new line like the current one.

If inside a comment and `comment-multi-line` 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.

M-x c-context-line-break

This is a function that works like `indent-new-comment-line` in comments and `newline-and-indent` elsewhere, thus combining those two in a way that uses each one in the context it's best suited for. I.e. in comments the comment line prefix and indentation is kept for the new line, and in normal code it's indented according to context by the indentation engine.

It's 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 might consider switching to this function.

[4] This means that the variables `c-hanging-comment-starter-p` and `c-hanging-comment-ender-p`, which controlled this behavior in earlier versions of CC Mode, are now obsolete.

6 Commands

See also Chapter 5 [Text Filling and Line Breaking], page 15, for commands concerning that bit.

6.1 Indentation Commands

The following list of commands re-indent C constructs. Note that when you change your coding style, either interactively or through some other means, your file does *not* automatically get re-indented. 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` only affect how on-the-fly code is formatted. Changing the "hanginess" of a brace and then re-indenting, will not move the brace to a different line. For this, you're better off getting an external program like GNU `indent`, which will re-arrange brace location, among other things.

Re-indenting 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. Especially vulnerable is code generator output[1].

These commands are useful when indenting code:

TAB (`c-indent-command`)
: Indents the current line. The actual behavior is controlled by several variables, described below. See `c-tab-always-indent`, `c-insert-tab-function`, and `indent-tabs-mode`. With a numeric argument, this command rigidly indents the region, preserving the relative indentation among the lines.

M-C-q (`c-indent-exp`)
: Indent 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 or class definition encompassing point. It leaves point unchanged. This function can't be used to re-indent a nested brace construct, such as a nested class or function, or a Java method. The top-level construct being re-indented must be complete, i.e. it must have both a beginning brace and an ending brace.

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

M-C-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-`

[1] In particular, I have had people complain about the speed with which `lex(1)` output is re-indented. Lex, yacc, and other code generators usually output some pretty perversely formatted code. Re-indenting such code will be slow.

`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:

`c-tab-always-indent`
: This variable controls how *TAB* `c-indent-command` operates. When this variable is `t`, *TAB* always just indents the current line. When it is `nil`, the line is indented only if point is at the left margin, or on or before the first non-whitespace character on the line, otherwise some whitespace is inserted. If this variable is the symbol `other`, then some whitespace is inserted only within strings and comments (literals), an inside preprocessor directives, but the line is always reindented.

`c-insert-tab-function`
: 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 just 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.

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

`c-progress-interval`
: 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 the interval in seconds that progress messages are displayed.

6.2 Movement Commands

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

`M-x c-beginning-of-defun`
: Moves point back to the least-enclosing brace. This function is analogous to the Emacs built-in command `beginning-of-defun`, except it eliminates the constraint that the top-level opening brace must be in column zero. See `beginning-of-defun` for more information.

  Depending on the coding style being used, you might prefer `c-beginning-of-defun` to `beginning-of-defun`. If so, consider binding *`C-M-a`* to the former instead. For backwards compatibility reasons, the default binding remains in effect.

`M-x c-end-of-defun`
: Moves point to the end of the current top-level definition. This function is analogous to the Emacs built-in command `end-of-defun`, except it eliminates

the constraint that the top-level opening brace of the defun must be in column zero. See `beginning-of-defun` for more information.

Depending on the coding style being used, you might prefer `c-end-of-defun` to `end-of-defun`. If so, consider binding *C-M-e* to the former instead. For backwards compatibility reasons, the default binding remains in effect.

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

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 point forward into the next nested preprocessor conditional, leaving the mark behind. A prefix argument acts as a repeat count. With a negative argument, move point 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`)
: Move point back over a preprocessor conditional, leaving the mark behind. A prefix argument acts as a repeat count. With a negative argument, move forward.

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

M-a (`c-beginning-of-statement`)
: Move point to the beginning of the innermost C statement. If point is already at the beginning of a statement, move to the beginning of the closest preceding statement, even if that means moving into a block (you can use *M-C-b* to move over a balanced block). With prefix argument n, move back $n - 1$ statements.

  If point is within or next to a comment or a string which spans more than one line, this command moves by sentences instead of statements.

  When called from a program, this function takes 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.

M-e (`c-end-of-statement`)
: Move point to the end of the innermost C statement. If point is at the end of a statement, move to the end of the next statement, even if it's inside a nested block (use *M-C-f* to move to the other side of the block). With prefix argument *n*, move forward $n - 1$ statements.

  If point is within or next to a comment or a string which spans more than one line, this command moves by sentences instead of statements.

  When called from a program, this function takes 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.

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`'.

  This command moves point forward to next capitalized word. With prefix argument *n*, move *n* times.

M-x c-backward-into-nomenclature
: Move point backward to beginning of the next capitalized word. With prefix argument *n*, move *n* times. If *n* is negative, move forward.

6.3 Other Commands

CC Mode contains a few other useful commands:

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-\ (`c-backslash-region`)
: This function is handy when editing macros split over several lines by ending each line with a backslash. It inserts and aligns, or deletes these end-of-line backslashes in the current region.

  With no prefix argument, it inserts any missing backslashes and aligns them to the column specified by the `c-backslash-column` style variable. 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.

7 Customizing Indentation

The style variable `c-offsets-alist` contains the mappings between syntactic symbols and the offsets to apply for those symbols. It's set at mode initialization from a *style* you may specify. Styles are groupings of syntactic symbol offsets and other style variable values. Most likely, you'll find that one of the pre-defined styles will suit your needs. See Section 7.4 [Styles], page 26, for an explanation of how to set up named styles.

Only syntactic symbols not already bound on `c-offsets-alist` will be set from styles. This means that any association you set on it, be it before or after mode initialization, will not be changed. The `c-offsets-alist` variable may therefore be used from e.g. the Customization interface[1] to easily change indentation offsets without having to bother about styles. Initially `c-offsets-alist` is empty, so that all syntactic symbols are set by the style system.

You can use the command *`C-c C-o`* (`c-set-offset`) as the way to set offsets, both interactively and from your mode hook[2].

The offset associated with any particular syntactic symbol can be any of an integer, a function or lambda expression, a variable name, a vector, a list, or one of the following symbols: `+`, `-`, `++`, `--`, `*`, or `/`.

Those last special symbols describe an offset in multiples of the value of the style variable `c-basic-offset`. By defining a style's indentation in terms of this fundamental variable, 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

When a function is used as offset, it's called an *indentation function*. Such functions are useful when more context than just the syntactic symbol is needed to get the desired indentation. See Chapter 9 [Indentation Functions], page 48, and Section 7.5.1 [Custom Indentation Functions], page 30, for details about them.

If the offset is a vector, its first element sets the absolute indentation column, which will override any relative indentation.

The offset can also be a list, in which case it is evaluated recursively using the semantics described above. The first element of the list that returns a non-`nil` value succeeds and

[1] Available in Emacs 20 and later, and XEmacs 19.15 and later.

[2] Obviously, you use the key binding interactively, and the function call programmatically!

the evaluation stops. If none of the list elements return a non-`nil` value, then an offset of 0 (zero) is used[3].

So, for example, because most of the default offsets are defined in terms of `+`, `-`, and `0`, if you like the general indentation style, but you use 4 spaces instead of 2 spaces per level, you can probably achieve your style just by changing `c-basic-offset` like so[4]:

```
M-x set-variable RET
Set variable: c-basic-offset RET
Set c-basic-offset to value: 4 RET
```

This would change

```
int add( int val, int incr, int doit )
{
  if( doit )
    {
      return( val + incr );
    }
  return( val );
}
```

to

```
int add( int val, int incr, int doit )
{
    if( doit )
        {
            return( val + incr );
        }
    return( val );
}
```

To change indentation styles more radically, you will want to change the offsets associated with other syntactic symbols. First, I'll show you how to do that interactively, then I'll describe how to make changes to your '`.emacs`' file so that your changes are more permanent.

7.1 Interactive Customization

As an example of how to customize indentation, let's change the style of this example[5]:

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

[4] You can try this interactively in a C buffer by typing the text that appears in italics.

[5] In this an subsequent examples, the original code is formatted using the '`gnu`' style unless otherwise indicated. See Section 7.4 [Styles], page 26.

```
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 components 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:

```
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 may not always work. The general approach to take is to always start adjusting offsets for lines higher up in the file, then re-indent and see if any following lines need further adjustments.

7.2 Permanent Customization

To make your changes permanent, you need to add some lisp code to your '`.emacs`' file. CC Mode supports many different ways to be configured, from the straightforward way by setting variables globally in '`.emacs`' or in the Customization interface, to the complex and precisely controlled way by using styles and hook functions.

The simplest way of customizing CC Mode permanently is to set the variables in your '`.emacs`' with `setq` and similar commands. So to make the setting of `substatement-open` permanent, add this to the '`.emacs`' file:

```
(require 'cc-mode)
(c-set-offset 'substatement-open 0)
```

The `require` line is only needed once in the beginning to make sure CC Mode is loaded so that the `c-set-offset` function is defined.

You can also use the more user friendly Customization interface, but this manual does not cover how that works.

Variables set like this at the top level in '`.emacs`' take effect in all CC Mode buffers, regardless of language. The indentation style related variables, e.g. `c-basic-offset`, that you don't set this way get their value from the style system (see Section 7.4 [Styles], page 26), and they therefore depend on the setting of `c-default-style`. Note that if you use Customize, this means that the greyed-out default values presented there might not be the ones you actually get, since the actual values depend on the style, which may very well be different for different languages.

If you want to make more advanced configurations, e.g. language-specific customization, global variable settings isn't enough. For that you can use the language hooks, see Section 7.3 [Hooks], page 26, and/or the style system, see Section 7.4 [Styles], page 26.

By default, all style variables are global, so that every buffer will share the same style settings. This is fine if you primarily edit one style of code, but if you edit several languages and want to use different styles for them, you need finer control by making the style variables buffer local. The recommended way to do this is to set the variable `c-style-variables-are-local-p` to `t`. The variables will be made buffer local when CC Mode is activated in a buffer for the first time in the Emacs session. Note that once the style variables are made buffer local, they cannot be made global again, without restarting Emacs.

7.3 Hooks

CC Mode provides several hooks that you can use to customize the mode according to your coding style. Each language mode has its own hook, adhering to standard Emacs major mode conventions. There is also one general hook and one package initialization hook:

- `c-mode-hook` — For C buffers only.
- `c++-mode-hook` — For C++ buffers only.
- `objc-mode-hook` — For Objective-C buffers only.
- `java-mode-hook` — For Java buffers only.
- `idl-mode-hook` — For CORBA IDL buffers only.
- `pike-mode-hook` — For Pike buffers only.
- `c-mode-common-hook` — Common across all languages.
- `c-initialization-hook` — Hook run only once per Emacs session, when CC Mode is initialized.

The language hooks get run as the last thing when you enter that language mode. The `c-mode-common-hook` is run by all supported modes *before* the language specific hook, and thus can contain customizations that are common across all languages. Most of the examples in this section will assume you are using the common hook.

Note that all the language-specific mode setup that CC Mode does is done prior to both `c-mode-common-hook` and the language specific hook. That includes installing the indentation style, which can be mode specific (and also is by default for Java mode). Thus, any style settings done in `c-mode-common-hook` will override whatever language-specific style is chosen by `c-default-style`.

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 D [Sample .emacs File], page 61, 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)
```

7.4 Styles

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.

The variables that the style system affect are called *style variables*. They are handled specially in several ways:

- Style variables are by default global variables, i.e. they have the same value in all Emacs buffers. However, they can instead be made always buffer local by setting `c-style-variables-are-local-p` to non-`nil` before CC Mode is initialized.
- The default value of any style variable (with two exceptions — see below) is the special symbol `set-from-style`. Variables that are still set to that symbol when a CC Mode buffer is initialized will be set according to the current style, otherwise they will keep their current value[6].

  Note that when we talk about the "default value" for a style variable, we don't mean the `set-from-style` symbol that all style variables are set to initially, but instead the value it will get at mode initialization when neither a style nor a global setting has set its value.

  The style variable `c-offsets-alist` is handled a little differently from the other style variables. It's an association list, and is thus by default set to the empty list, `nil`. When the style system is initialized, any syntactic symbols already on it are kept — only the missing ones are filled in from the chosen style.

  The style variable `c-special-indent-hook` is also handled in a special way. Styles may only add more functions on this hook, so the global settings on it are always preserved[7].
- The global settings of style variables get captured in the special `user` style, which is used as the base for all the other styles. See Section 7.4.1 [Built-in Styles], page 27, for details.

The style variables are: `c-basic-offset`, `c-comment-only-line-offset`, `c-block-comment-prefix`, `c-comment-prefix-regexp`, `c-cleanup-list`, `c-hanging-braces-alist`, `c-hanging-colons-alist`, `c-hanging-semi&comma-criteria`, `c-backslash-column`, `c-special-indent-hook`, `c-label-minimum-indentation`, and `c-offsets-alist`.

7.4.1 Built-in Styles

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

[6] 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`.

[7] This did not change in version 5.26.

- `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[8].
- `linux` — C coding standard for Linux (the kernel).
- `python` — C coding standard for Python extension modules[9].
- `java` — The style for editing Java code. Note that the default value for `c-default-style` installs this style when you enter `java-mode`.
- `user` — This is a special style for several reasons. First, the CC Mode customizations you do by using either the Customization interface, or by writing `setq`’s at the top level of your ‘`.emacs`’ file, will be captured in the `user` style. Also, all other styles implicitly inherit their settings from `user` style. This means that for any styles you add via `c-add-style` (see Section 7.4.2 [Adding Styles], page 29) you need only define the differences between your new style and `user` style.

The default style in all newly created buffers is `gnu`, but you can change this by setting variable `c-default-style`. Although the `user` style is not the default style, any style variable settings you do with the Customization interface or on the top level in your ‘`.emacs`’ file will by default override the style system, so you don’t need to set `c-default-style` to `user` to see the effect of these settings.

`c-default-style` takes either a style name string, or an association list of major mode symbols to style names. Thus you can control exactly which default style is used for which CC Mode language mode. Here are the rules:

1. When `c-default-style` is a string, it must be an existing style name as found in `c-style-alist`. This style is then used for all modes.
2. When `c-default-style` is an association list, the current major mode is looked up to find a style name string. In this case, this style is always used exactly as specified and an error will occur if the named style does not exist.
3. If `c-default-style` is an association list, but the current major mode isn’t found, then the special symbol ‘`other`’ is looked up. If this value is found, the associated style is used.
4. If ‘`other`’ is not found, then the ‘`gnu`’ style is used.
5. 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`.

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

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

If you'd like to experiment with these built-in styles you can simply type the following in a CC Mode buffer:

```
C-c . STYLE-NAME RET
```

`C-c .` runs the command `c-set-style`. Note that all style names are case insensitive, even the ones you define.

Setting a style in this way does *not* automatically re-indent your file. For commands that you can use to view the effect of your changes, see Chapter 6 [Commands], page 18.

Note that for BOCM compatibility, '`gnu`' is the default style, and any non-style based customizations you make (i.e. in `c-mode-common-hook` in your '`.emacs`' file) will be based on '`gnu`' style unless you set `c-default-style` or do a `c-set-style` as the first thing in your hook. The variable `c-indentation-style` always contains the buffer's current style name, as a string.

7.4.2 Adding Styles

If none of the built-in styles is appropriate, you'll probably want to add a new *style definition*. 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` that you can use to easily add new styles or change existing styles. This function takes two arguments, a *stylename* string, and an association list *description* of style customizations. If *stylename* is not already in `c-style-alist`, the new style is added, otherwise the style is changed to the new *description*. This function also takes an optional third argument, which if non-`nil`, automatically applies the new style to the current buffer.

The sample '`.emacs`' file provides a concrete example of how a new style can be added and automatically set. See Appendix D [Sample .emacs File], page 61.

7.4.3 File Styles

The Emacs manual describes how you can customize certain variables on a per-file basis by including a *Local Variable* block at the end of the file. So far, you've only seen a functional interface to CC Mode customization, which is highly inconvenient for use in a Local Variable block. CC Mode provides two variables that make it easier for you to customize your style on a per-file basis.

The variable `c-file-style` can be set to a style name string. When the file is visited, CC Mode will automatically set the file's style to this style using `c-set-style`.

Another variable, `c-file-offsets`, takes an association list similar to what is allowed in `c-offsets-alist`. When the file is visited, 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`). 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.

7.5 Advanced Customizations

For most users, CC Mode will support their coding styles with very little need for more advanced customizations. Usually, one of the standard styles defined in `c-style-alist` will do the trick. At most, perhaps one of the syntactic symbol offsets will need to be tweaked slightly, or maybe `c-basic-offset` will need to be changed. However, some styles require a more flexible framework for customization, and one of the real strengths of CC Mode is that the syntactic analysis model provides just such a framework. This allows you to implement custom indentation calculations for situations not handled by the mode directly.

7.5.1 Custom Indentation Functions

The most flexible way to customize CC Mode is by writing custom indentation functions, and associating them with specific syntactic symbols (see Chapter 8 [Syntactic Symbols], page 35). CC Mode itself uses indentation functions to provide more sophisticated indentation, for example when lining up C++ stream operator blocks:

```
1: void main(int argc, char**)
2: {
3:   cout << "There were "
4:     << argc
5:     << "arguments passed to the program"
6:     << endl;
7: }
```

In this example, lines 4 through 6 are assigned the `stream-op` syntactic symbol. Here, `stream-op` has an offset of `+`, and with a `c-basic-offset` of 2, you can see that lines 4 through 6 are simply indented two spaces to the right of line 3. But perhaps we'd like CC Mode to be a little more intelligent so that it aligns all the '`<<`' symbols in lines 3 through 6. To do this, we have to write a custom indentation function which finds the column of first stream operator on the first line of the statement. Here is sample lisp code implementing this:

```
(defun c-lineup-streamop (langelem)
  ;; lineup stream operators
  (save-excursion
    (let* ((relpos (cdr langelem))
           (curcol (progn (goto-char relpos)
                          (current-column))))
      (re-search-forward "<<\\|>>" (c-point 'eol) 'move)
      (goto-char (match-beginning 0))
      (- (current-column) curcol))))
```

Indentation functions take a single argument, which is a syntactic component cons cell (see Section 3.1 [Syntactic Analysis], page 3). The function returns an integer offset value that will be added to the running total indentation for the line. Note that what actually gets returned is the difference between the column that the first stream operator is on, and the column of the buffer relative position passed in the function's argument. Remember that

CC Mode automatically adds in the column of the component's relative buffer position and we don't the column offset added in twice.

The function should return `nil` if it's used in a situation where it doesn't want to do any decision. If the function is used in a list expression (see Chapter 7 [Customizing Indentation], page 22), that will cause CC Mode to go on and check the next entry in the list.

Now, to associate the function `c-lineup-streamop` with the `stream-op` syntactic symbol, we can add something like the following to our `c++-mode-hook`[10]:

```
(c-set-offset 'stream-op 'c-lineup-streamop)
```

Now the function looks like this after re-indenting (using *C-c C-q*):

```
void main(int argc, char**)
{
  cout << "There were "
       << argc
       << " arguments passed to the program"
       << endl;
}
```

Custom indentation 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 indentation function associated with it.

CC Mode comes with an extensive set of predefined indentation functions, not all of which are used by the default styles. So there's a good chance the function you want already exists. See Chapter 9 [Indentation Functions], page 48, for a list of them. If you have written an indentation function that you think is generally useful, you're very welcome to contribute it; please contact `bug-cc-mode@gnu.org`.

7.5.2 Custom Brace and Colon Hanging

Syntactic symbols aren't the only place where you can customize CC Mode with the lisp equivalent of callback functions. Brace "hanginess" can also be determined by custom functions associated with syntactic symbols on the `c-hanging-braces-alist` style variable. Remember that *ACTION*'s are typically a list containing some combination of the symbols `before` and `after` (see Section 4.1.1 [Hanging Braces], page 8). However, an *ACTION* can also be a function which gets called when a brace matching that syntactic symbol is entered.

These *ACTION* functions are called with two arguments: the syntactic symbol for the brace, and the buffer position at which the brace was inserted. The *ACTION* function is expected to return a list containing some combination of `before` and `after`, including neither of them (i.e. `nil`). This return value has the normal brace hanging semantics.

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

[10] It probably makes more sense to add this to `c++-mode-hook` than `c-mode-common-hook` since stream operators are only relevant for C++.

```
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.
Using this function, ‘while’ clauses that end a ‘do-while’ block will
remain on the same line as the brace that closes that block.

See ‘c-hanging-braces-alist’ for how to utilize this function as an
ACTION associated with ‘block-close’ syntax."
  (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.

During the call to the brace hanging *ACTION* function, the variable `c-syntactic-context` is bound to the full syntactic analysis list.

Note that for symmetry, colon hanginess should be customizable by allowing function symbols as *ACTION*s on the `c-hanging-colon-alist` style variable. Since no use has actually been found for this feature, it isn't currently implemented!

7.5.3 Customizing Semi-colons and Commas

You can also customize the insertion of newlines after semi-colons and commas, when the auto-newline minor mode is enabled (see Chapter 4 [Minor Modes], page 7). This is controlled by the style variable `c-hanging-semi&comma-criteria`, which contains a list of functions that are called in the order they appear. Each function is called with zero arguments, and is expected to return one of the following values:

- non-`nil` — A newline is inserted, and no more functions from the list are called.
- `stop` — No more functions from the list are called, but no newline is inserted.
- `nil` — No determination is made, and the next function in the list is called.

If every function in the list is called without a determination being made, then no newline is added. The default value for this variable is a list containing a single function which inserts newlines only after semi-colons which do not appear inside parenthesis lists (i.e. those that separate `for`-clause statements).

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

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

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 (i.e. in C++ or Java).

7.5.4 Other Special Indentations

In '`gnu`' style (see Section 7.4.1 [Built-in Styles], page 27), a minimum indentation is imposed on lines inside top-level constructs. This minimum indentation is controlled by the style variable `c-label-minimum-indentation`. The default value for this variable is 1.

One other customization variable is available in CC Mode: The style variable `c-special-indent-hook`. This is a standard hook variable that is called after every line is indented by CC Mode. You can use it to do any special indentation or line adjustments your style dictates, such as adding extra indentation to constructors or destructor declarations in a class definition, etc. Note however, that you should not change point or mark inside your `c-special-indent-hook` functions (i.e. you'll probably want to wrap your function in a `save-excursion`).

Setting `c-special-indent-hook` in your style definition is handled slightly differently than other variables. In your style definition, you should set the value for `c-special-indent-hook` to a function or list of functions, which will be appended to `c-special-indent-hook` using `add-hook`. That way, the current setting for the buffer local value of `c-special-indent-hook` won't be overridden.

Normally, the standard Emacs command *M-;* (`indent-for-comment`) will indent comment only lines to `comment-column`. Some users however, prefer that *M-;* act just like *TAB* for purposes of indenting comment-only lines; i.e. they want the comments to always indent as they would for normal code, regardless of whether *TAB* or *M-;* were used. This behavior is controlled by the variable `c-indent-comments-syntactically-p`. When `nil` (the default), *M-;* indents comment-only lines to `comment-column`, otherwise, they are indented just as they would be if *TAB* were typed.

Note that this has no effect for comment lines that are inserted with *M-;* at the end of regular code lines. These comments will always start at `comment-column`.

8 Syntactic Symbols

Here is a complete list of the recognized syntactic symbols as described in the `c-offsets-alist` style variable, along with a brief description. More detailed descriptions follow.

`string` Inside a multi-line string.

`c` Inside a multi-line C style block comment.

`defun-open`
: Brace that opens a top-level function definition.

`defun-close`
: Brace that closes a top-level function definition.

`defun-block-intro`
: The first line in a top-level defun.

`class-open`
: Brace that opens a class definition.

`class-close`
: Brace that closes a class definition.

`inline-open`
: Brace that opens an in-class inline method.

`inline-close`
: Brace that closes an in-class inline method.

`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.

`knr-argdecl-intro`
: First line of a K&R C argument declaration.

`knr-argdecl`
: Subsequent lines in a K&R C argument declaration.

`topmost-intro`
: The first line in a "topmost" definition.

`topmost-intro-cont`
: Topmost definition continuation lines.

`member-init-intro`
: First line in a member initialization list.

`member-init-cont`
: Subsequent member initialization list lines.

`inher-intro`
: First line of a multiple inheritance list.

`inher-cont`
: Subsequent multiple inheritance lines.

`block-open`
: Statement block open brace.

`block-close`
: Statement block close brace.

`brace-list-open`
: Open brace of an enum or static array list.

`brace-list-close`
: Close brace of an enum or static array list.

`brace-list-intro`
: First line in an enum or static array list.

`brace-list-entry`
: Subsequent lines in an enum or static array list.

`brace-entry-open`
: Subsequent lines in an enum or static array list where the line begins with an open brace.

`statement`
: A statement.

`statement-cont`
: A continuation of a statement.

`statement-block-intro`
: The first line in a new statement block.

`statement-case-intro`
: The first line in a case block.

`statement-case-open`
: The first line in a case block that starts with a brace.

`substatement`
: The first line after a conditional or loop construct.

`substatement-open`
: The brace that opens a substatement block.

`case-label`
: A `case` or `default` label.

`access-label`
: C++ access control label.

`label`
: Any non-special C label.

`do-while-closure`
: The `while` line that ends a `do-while` construct.

`else-clause`
: The `else` line of an `if-else` construct.

`catch-clause`
: The `catch` or `finally` (in Java) line of a `try-catch` construct.

`comment-intro`
: A line containing only a comment introduction.

`arglist-intro`
: The first line in an argument list.

`arglist-cont`
: Subsequent argument list lines when no arguments follow on the same line as the arglist opening paren.

`arglist-cont-nonempty`
: Subsequent argument list lines when at least one argument follows on the same line as the arglist opening paren.

`arglist-close`
: The solo close paren of an argument list.

`stream-op`
: Lines continuing a stream operator (C++ only).

`inclass` The line is nested inside a class definition.

`cpp-macro`
: The start of a C preprocessor macro definition.

`cpp-macro-cont`
: Subsequent lines of a multi-line C preprocessor macro definition.

`friend` A C++ friend declaration.

`objc-method-intro`
: The first line of an Objective-C method. definition.

`objc-method-args-cont`
: Lines continuing an Objective-C method. definition

`objc-method-call-cont`
: Lines continuing an Objective-C method call.

`extern-lang-open`
: Brace that opens an external language block.

`extern-lang-close`
: Brace that closes an external language block.

`inextern-lang`
: Analogous to `inclass` syntactic symbol, but used inside external language blocks (e.g. `extern "C" {`).

`namespace-open`
: Brace that opens a C++ namespace block.

`namespace-close`
: Brace that closes a C++ namespace block.

`innamespace`
: Analogous to `inextern-lang` syntactic symbol, but used inside C++ namespace blocks.

`template-args-cont`
: C++ template argument list continuations.

`inlambda`
: Analogous to `inclass` syntactic symbol, but used inside lambda (i.e. anonymous) functions. Only used in Pike mode.

`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.

`inexpr-statement`
: A statement block inside an expression. The gcc C extension of this is recognized. It's also used for the special functions that takes a statement block as an argument in Pike.

`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.

Most syntactic symbol names follow a general naming convention. When a line begins with an open or close brace, the syntactic symbol will contain the suffix `-open` or `-close` respectively.

Usually, a distinction is made between the first line that introduces a construct and lines that continue a construct, and the syntactic symbols that represent these lines will contain the suffix `-intro` or `-cont` respectively. As a sub-classification of this scheme, a line which is the first of a particular brace block construct will contain the suffix `-block-intro`.

Let's look at some examples to understand how this works. Remember that you can check the syntax of any line by using *`C-c C-s`*.

```
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.

Here's another 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[1]. 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 . 67))
```

The primary syntactic symbol for this line is `access-label` as this 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:

[1] 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.

```
((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.

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. If though, 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:

```
class Bass
    : public Guitar,
      public Amplifiable
{
public:
    Bass();
}

inline
Bass::Bass()
    : eString( new BassString( 0.105 )),
      aString( new BassString( 0.085 )),
      dString( new BassString( 0.065 )),
      gString( new BassString( 0.045 ))
{
    eString.tune( 'E' );
    aString.tune( 'A' );
    dString.tune( 'D' );
    gString.tune( 'G' );
}
```

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:

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

The `friend` syntactic symbol is a modifier that typically does not have a relative buffer position.

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.

Here is another (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:         {
 7:             do_something_special();
 8:         }
 9:         else
10:             do_something( i );
11:     }
12:     do {
13:         another_thing( i-- );
14:     }
15:     while( i > 0 );
16: }
```

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 `substatement-block-intro` syntax. Lines 6 and 7 are assigned similar syntax. Line 8 contains the brace that closes the inner substatement block. It is given the syntax `block-close`, as are lines 11 and 14.

Line 9 is a little different — since it contains the keyword `else` matching the `if` statement introduced on line 5, it is given the `else-clause` syntax. The `try-catch` constructs in C++ and Java are treated this way too, with the only difference that the `catch`, and in Java also `finally`, is marked with `catch-clause`.

Line 10 is also slightly different. Because `else` is considered a conditional introducing keyword[2], and because the following substatement is not a brace block, line 10 is assigned the `substatement` syntax.

[2] The list of conditional keywords are (in C, C++, Objective-C, Java, and Pike): `for`, `if`, `do`, `else`, `while`, and `switch`. C++ and Java have two additional conditional keywords: `try` and `catch`. Java also has the `finally` and `synchronized` keywords.

One other difference is seen on line 15. The `while` construct 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 have been assigned `block-close` syntax instead.

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

```
 1: void spam( enum Ingredient i )
 2: {
 3:     switch( i ) {
 4:     case Ham:
 5:         be_a_pig();
 6:         break;
 7:     case Salt:
 8:         drink_some_water();
 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.

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 (and line 9) 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.

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`.

Similarly, C++ namespace constructs have their own associated syntactic symbols. In this example:

```
1: namespace foo
2: {
3:     void xxx() {}
4: }
```

line 2 is given the `namespace-open` syntax, while line 4 is given the `namespace-close` syntax. The analysis for line 3 yields: `((innamespace) (topmost-intro . 17))`, where `innamespace` is a modifier similar in purpose to `inextern-lang` and `inclass`.

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:     );
 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.

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.

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

```
 1: void Bass::play( int volume )
 2: const
 3: {
 4:     /* this line starts a multi-line
 5:      * comment.  This line should get ‘c’ syntax */
 6:
 7:     char* a_multiline_string = "This line starts a multi-line \
 8: string.  This line should get ‘string’ syntax.";
 9:
10:   note:
11:     {
12: #ifdef LOCK
13:         Lock acquire();
14: #endif // LOCK
15:         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.
- 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` syntax.
- 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.
- Line 17 is assigned `stream-op` syntax.

Multi-line C preprocessor macros are now (somewhat) supported. At least CC Mode now recognizes the fact that it is inside a multi-line macro, and it properly skips such macros as syntactic whitespace. 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`. This first line of a macro is always given this symbol. The second and subsequent lines (e.g. lines 2 through 5) are given the `cpp-macro-cont` syntactic symbol, with a relative buffer position pointing to the `#` which starts the macro definition.

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.

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

```
1: public void watch(Observable o) {
2:     o.addObserver(new Observer() {
3:             public void update(Observable o, Object arg) {
4:                 history.addElement(arg);
5:             }
6:         });
7: }
```

The brace following the `new` operator opens the anonymous class. Lines 3 and 6 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`.

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

```
1: int res = ({
2:         int y = foo (); int z;
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.

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`[3].

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.

Two other syntactic symbols can appear in old style, non-prototyped C code[4]:

```
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.

[3] 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.

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

9 Indentation Functions

Often there are cases when a simple offset setting on a syntactic symbol isn't enough to get the desired indentation. Therefore, it's also possible to use a *indentation function* (a.k.a. line-up function) for a syntactic symbol.

CC Mode comes with many predefined indentation functions for common situations. If none of these does what you want, you can write your own, see Section 7.5.1 [Custom Indentation Functions], page 30. If you do, it's probably a good idea to start working from one of these predefined functions, they can be found in the file '`cc-align.el`'.

For every function below there is a "works with" list that indicates which syntactic symbols the function is intended to be used with.

`c-lineup-arglist`
: Line up the current argument line under the first argument.

  Works with: `arglist-cont-nonempty`.

`c-lineup-arglist-intro-after-paren`
: Line up a line 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-arglist-close-under-paren`
: Set e.g. 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.

  Works with: `defun-close`, `class-close`, `inline-close`, `block-close`, `brace-list-close`, `arglist-close`, `extern-lang-close`, `namespace-close` (for most of these, a zero offset will normally produce the same result, though).

`c-lineup-close-paren`
: 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
```

  Works with: `defun-close`, `class-close`, `inline-close`, `block-close`, `brace-list-close`, `arglist-close`, `extern-lang-close`, `namespace-close`.

`c-lineup-streamop`

Line up C++ stream operators (i.e. '`<<`' and '`>>`').

Works with: `stream-op`.

`c-lineup-multi-inher`

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`

Line up Java implements and extends declarations. If class names follows 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`

Line up Java throws declarations. If exception names follows 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-indent-one-line-block`

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`

Indent a multi line 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 multi line block, which makes the function usable in list expressions.

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

`c-lineup-C-comments`

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`

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` 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-runin-statements`

Line up statements for coding standards which place the first statement in a block on the same line as the block opening brace[1]. E.g:

```
int main()
{ puts (\"Hello world!\");
  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-math`

Line up the current line after the equal sign on the first line in the statement. If there isn't any, indent with `c-basic-offset`. If the current line contains an equal sign too, try to align it with the first one.

Works with: `statement-cont`.

`c-lineup-template-args`

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`

For Objective-C code, line up selector args as `elisp-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`

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`.

[1] Run-in style doesn't really work too well. You might need to write your own custom indentation functions to better support this style.

`c-lineup-ObjC-method-args-2`
: 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`.

`c-lineup-inexpr-block`
: 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-whitesmith-in-block`
: 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`, `block-close`, `brace-list-close`, `brace-list-intro`, `statement-block-intro`, `inclass`, `inextern-lang`, `innamespace`.

`c-lineup-dont-change`
: This lineup function makes the line stay at whatever indentation it already has; think of it as an identity function for lineups. It is used for `cpp-macro-cont` lines.

  Works with: Any syntactic symbol.

10 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[1] can cause CC Mode to perform fairly badly. This section identifies some of the coding styles to watch out for, and suggests some workarounds that you can use to improve performance.

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. The farther this position is from the current insertion point, the slower the mode gets. Some coding styles can even force CC Mode to scan from the beginning of the buffer for every line of code!

One of the simplest things you can do to reduce scan time, is make sure any brace that opens a top-level construct[2] always appears in the leftmost column. This is actually an Emacs constraint, as embodied in the `beginning-of-defun` function which CC Mode uses heavily. If you insist on hanging top-level open braces on the right side of the line, then you might want to set the variable `defun-prompt-regexp` to something reasonable, however that "something reasonable" is difficult to define, so CC Mode doesn't do it for you.

A special note about `defun-prompt-regexp` in Java mode: while much of the early sample Java code seems to encourage a style where the brace that opens a class is hung on the right side of the line, this is not a good style to pursue in Emacs. CC Mode comes with a variable `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[3]. 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 rely on `defun-prompt-regexp` will definitely slow things down anyway because you'll be doing regular expression searches for every line you indent, so you're probably screwed either way!

Another alternative for XEmacs users, is to 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 circumstances, can locate the top-most opening brace much quicker 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 20.2 or 20.3 as of this writing 27-Apr-1998).

You will probably notice pathological behavior from CC Mode when working in files containing large amounts of C preprocessor macros. This is because Emacs cannot skip backwards over these lines as quickly as it can comments.

[1] such as the output of `lex(1)`!

[2] E.g. a function in C, or outermost class definition in C++ or Java.

[3] This has been observed in Emacs 19.34 and XEmacs 19.15.

Previous versions of CC Mode had potential performance problems when recognizing K&R style function argument declarations. This was because there are ambiguities in the C syntax when K&R style argument lists are used[4]. CC Mode has adopted BOCM's convention for limiting the search: it assumes that argdecls are indented at least one space, and that the function headers are not indented at all. With current versions of CC Mode, user customization of `c-recognize-knr-p` is deprecated. Just don't put argdecls in column zero!

You might want to investigate the speed-ups contained in the file '`cc-lobotomy.el`', which comes as part of the CC Mode distribution, but is completely unsupported. As mentioned previous, CC Mode always trades speed for accuracy, however it is recognized that sometimes you need speed and can sacrifice some accuracy in indentation. The file '`cc-lobotomy.el`' contains hacks that will "dumb down" CC Mode in some specific ways, making that trade-off of accurancy for speed. I won't go into details of its use here; you should read the comments at the top of the file, and look at the variable `cc-lobotomy-pith-list` for details.

[4] It is hard to distinguish them from top-level declarations.

11 Limitations and Known Bugs

- Re-indenting large regions or expressions can be slow.
- `c-indent-exp` has not been fully optimized. It essentially equivalent to hitting *TAB* (`c-indent-command`) on every line. Some information is cached from line to line, but such caching invariable causes inaccuracies in analysis in some bizarre situations.
- XEmacs versions from 19.15 until (as of this writing 12-Mar-1998) 20.4 contain a variable called `signal-error-on-buffer-boundary`. This was intended as a solution to user interface problems associated with buffer movement and the `zmacs-region` deactivation on errors. However, setting this variable to a non-default value had the deleterious side effect of breaking many built-in primitive functions. Most users will not be affected since they never change the value of this variable. **Do not set this variable to `nil`**; you will cause serious problems in CC Mode and probably other XEmacs packages! As of at least XEmacs 20.4, the effects this variable tried to correct have been fixed in other, better ways.

Appendix A Frequently Asked Questions

Q. *How do I re-indent the whole file?*

A. Visit the file and hit `C-x h` to mark the whole buffer. Then hit `C-M-\`.

Q. *How do I re-indent the entire function?* `C-M-x` *doesn't work.*

A. `C-M-x` is reserved for future Emacs use. To re-indent the entire function hit `C-c C-q`.

Q. *How do I re-indent the current block?*

A. First move to the brace which opens the block with `C-M-u`, then re-indent that expression with `C-M-q`.

Q. *Why doesn't the* `RET` *key indent the new line?*

A. 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-mode-common-hook`:

```
(define-key c-mode-base-map "\C-m" 'c-context-line-break)
```

This is a very common question. If you want this to be the default behavior, don't lobby me, lobby RMS! `:-)`

Q. *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.*

A. This means that CC Mode wasn't loaded into your Emacs session by the time the `c-set-offset` call was 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-mode-common-hook`, or simply modify `c-offsets-alist` directly:

```
(setq c-offsets-alist '((substatement-open . 0)))
```

Q. *How do I make strings, comments, keywords, and other constructs appear in different colors, or in bold face, etc.?*

A. "Syntax Colorization" is a standard Emacs feature, controlled by `font-lock-mode`. CC Mode does not contain font-lock definitions for any of its supported languages.

Q. `M-a` *and* `M-e` *used to move over entire balanced brace lists, but now they move into blocks. How do I get the old behavior back?*

A. Use `C-M-f` and `C-M-b` to move over balanced brace blocks. Use `M-a` and `M-e` to move by statements, which will also move into blocks.

Q. *Whenever I try to indent a line or type an "electric" key such as* `;`*,* `{`*, or* `}`*, I get an error that look like this:* `Invalid function: (macro . #[....` *What gives?*

A. This is a common error when CC Mode hasn't been compiled correctly, especially under Emacs 19.34[1]. If you are using the standalone CC Mode distribution, try recompiling it according to the instructions in the '`README`' file.

[1] Technically, it's because some macros wasn't defined during the compilation, so the byte compiler put in function calls instead of the macro expansions. Later, when the interpreter tries to call the macros as functions, it shows this (somewhat cryptic) error message.

Appendix B Getting the Latest CC Mode Release

CC Mode is 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 via the Web at:

```
http://cc-mode.sourceforge.net/
```

Old URLs, including the FTP URLs, should no longer be used.

There are many files under these directories; you can pick up the entire distribution (named `cc-mode.tar.gz`; a gzip'd tar file), or any of the individual files, including PostScript documentation.

Appendix C Mailing Lists and Submitting Bug Reports

To report bugs, use the `C-c C-b` (`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 if you isolate the Emacs Lisp code that trigs the bug and include it in your report.

Bug reports are now sent to the following email addresses: `bug-cc-mode@gnu.org` and `bug-gnu-emacs@gnu.org`; the latter is mirrored on the Usenet newsgroup `gnu.emacs.bug`. You can send other questions and suggestions (kudos? `;-)` to `bug-cc-mode@gnu.org`.

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`. Announcements will also be posted to the Usenet newsgroups `gnu.emacs.sources`, `comp.emacs` and `comp.emacs.xemacs`.

Appendix D Sample .emacs file

```
;; Here's a sample .emacs file that might help you along the way.  Just
;; copy this region and paste it into your .emacs file.  You may want to
;; change some of the actual values.

(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")

;; offset customizations not in my-c-style
(setq c-offsets-alist '((member-init-intro . ++)))

;; Customizations for all modes in CC Mode.
(defun my-c-mode-common-hook ()
  ;; add my personal style and set it for the current buffer
  (c-add-style "PERSONAL" my-c-style t)
  ;; other customizations
  (setq tab-width 8
        ;; this will make sure spaces are used instead of tabs
        indent-tabs-mode nil)
  ;; we like auto-newline and hungry-delete
  (c-toggle-auto-hungry-state 1)
  ;; key bindings for all supported languages.  We can put these in
  ;; c-mode-base-map because c-mode-map, c++-mode-map, objc-mode-map,
  ;; java-mode-map, idl-mode-map, and pike-mode-map inherit from it.
  (define-key c-mode-base-map "\C-m" 'c-context-line-break)
  )

(add-hook 'c-mode-common-hook 'my-c-mode-common-hook)
```

Concept Index

Command Index

Since most CC Mode commands are prepended with the string '`c-`', each appears under its `c-<`*thing*`>` name and its `<`*thing*`> (c-)` name.

Key Index

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.

Short Contents

Table of Contents