CC Mode 5.21

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

Barry A. Warsaw

1 Introduction

Welcome to CC Mode. This is a GNU Emacs mode for editing files containing C, C++, Objective-C, Java, and CORBA IDL 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 I have 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.

CC Mode supports the editing of K&R and ANSI C, *ARM*[1] C++, Objective-C, Java and CORBA's Interface Definition Language files. In this way, you can easily set up consistent coding styles for use in editing all C, C++, Objective-C, Java and IDL programs. 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`, and `idl-mode` entry points are provided. This file 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 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 Chapter 10 [Getting the latest CC Mode release], page 43).

Note that CC Mode no longer works with Emacs 18! The '`cc-mode-18.el`' file is no longer distributed with CC Mode. 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 7 [Syntactic Symbols], page 30, for a complete list of currently recognized syntactic symbols and their semantics. The 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++-mode` 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] or C++, Objective-C, Java or IDL code. In general, for the rest of this manual I'll use the term "C code" to refer to all the C-like dialects, unless otherwise noted.

[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))
```

[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.

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.

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` 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` 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 keybindings 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] Remember that the '`C`' could be replaced with '`C++`', '`ObjC`', '`Java`' or '`IDL`'.

[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 insertion of newlines is controlled by the `c-hanging-braces-alist` 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-list-entry`, `block-open`, `block-close`, `substatement-open`, `statement-case-open`, `extern-lang-open`, `extern-lang-close`, `namespace-open`, and `namespace-close`. See Chapter 7 [Syntactic Symbols], page 30, for a more detailed description of these syntactic symbols.

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 6.4.2 [Custom Brace and Colon Hanging], page 26, 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);
    }
}
```

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.

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

```
(defvar c-hanging-braces-alist '((brace-list-open)
                                 (substatement-open after)
                                 (block-close . c-snug-do-while)
                                 (extern-lang-open after)))
```

which says that `brace-list-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` and `extern-lang-open` braces should hang on the right side, but subsequent text should follow on the next line. Here, in the `block-close` entry, you also see an example of using a function as an *ACTION*.

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 such braces can introduce performance problems. See Chapter 8 [Performance Issues], page 40, 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 variable `c-hanging-colons-alist`. The syntactic symbols appropriate for this assocation 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 6.4.2 [Custom Brace and Colon Hanging], page 26 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 6.4.3 [Customizing Semi-colons and Commas], page 27, 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.

4.1.5 Clean-ups

Clean-ups are a mechanism 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, and similarly, clean-ups are only enabled when auto-newline minor mode is enabled. Clean-ups are used however to adjust code "after-the-fact", i.e. to eliminate some whitespace that is inserted by electric commands, or whitespace that contains intervening constructs.

You can configure CC Mode's clean-ups by setting the 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.

There are currently only five specific constructs that CC Mode can clean up, as indicated by these symbols:

- `brace-else-brace` — cleans 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 brace is typed:

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

- `empty-defun-braces` — cleans 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` — cleans 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` — cleans up commas following braces in array and aggregate initializers. Clean up occurs when the comma is typed.
- `scope-operator` — cleans up double colons which may designate a C++ scope operator split across multiple lines[3]. 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.

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[4] 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.

Similarly, hitting the ⟨Delete⟩ key runs the command `c-electric-delete`. When deleting a single character, or when ⟨Delete⟩ is hit in a literal, or when hungry-delete mode is disabled,

---

[3] 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.

[4] 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.

the function contained in the `c-delete-function` variable is called with one argument (the number of characters to delete). This variable is set to `delete-char` by default.

However, if `delete-key-deletes-forward` is `nil`, or your Emacs does not support separation of ⟨Backspace⟩ and ⟨DEL⟩, then `c-electric-delete` simply calls `c-electric-backspace`.

4.3 Auto-fill mode interaction

One other note about minor modes is worth mentioning here. CC Mode now works much better with auto-fill mode (a standard Emacs minor mode) by correctly auto-filling both line (e.g. C++ style) and block (e.g. C style) oriented comments. When `auto-fill-mode` is enabled, line oriented comments will also be auto-filled by inserting a newline at the line break, and inserting '`//`' at the start of the next line.

When auto-filling block oriented comments, the behavior is dependent on the value of the variable `c-comment-continuation-stars`. When this variable is `nil`, the old behavior for auto-filling C comments is in effect. In this case, the line is broken by closing the comment and starting a new comment on the next line.

If you set `c-comment-continuation-stars` to a string, then a long C block comment line is broken by inserting a newline at the line break position, and inserting this string at the beginning of the next comment line. The default value for `c-comment-continuation-stars` is '`* `' (a star followed by a single space)[5].

[5] To get block comment continuation lines indented under the block comment starter (e.g. the '`/*`'), it is not enough to set `c-comment-continuation-stars` to the empty string. You need to do this, but you also need to set the offset for the `c` syntactic symbol to be zero.

5 Commands

5.1 Indentation Commands

Various commands are provided which allow you to conveniently re-indent C constructs. There are several things to note about these indentation commands. First, when you change your programming style, either interactively or through some other means, your file does *not* automatically get re-indented. When you change style parameters, you will typically need to reformat the line, expression, or buffer to see the effects of your changes.

Second, changing some variables have no effect on existing code, even when you do re-indent. For example, the `c-hanging-*` variables and `c-cleanup-list` only affect new code as it is typed in on-the-fly, so changing `c-hanging-braces-alist` and re-indenting the buffer will not adjust placement of braces already in the file.

Third, re-indenting large portions of code is currently rather inefficient. Improvements have been made since previous releases of CC Mode, and much more radical improvements are planned, but for now you need to be aware of this[1]. Some provision has been made to at least inform you as to the progress of the re-indentation. The variable `c-progress-interval` controls how often a progress message is displayed. Set this variable to `nil` to inhibit progress messages, including messages normally printed when indentation is started and completed.

Also, except as noted below, re-indentation is always driven by the same mechanisms that control on-the-fly indentation of code. See Chapter 3 [New Indentation Engine], page 3, for details.

To indent a single line of code, use *TAB* (`c-indent-command`). The behavior of this command is controlled by the variable `c-tab-always-indent`. When this variable is `t`, *TAB* always just indents the current line. When `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 *something else happens*[2]. If the value of `c-tab-always-indent` is something other than `t` or `nil` (e.g. `'other`), then a real tab character[3] is inserted only when point is inside a literal (see Section 4.1 [Auto-newline insertion], page 7), otherwise the line is indented.

To indent an entire balanced brace or parenthesis expression, use *M-C-q* (`c-indent-exp`). Note that point should be on the opening brace or parenthesis of the expression you want to indent.

Another very convenient keystroke is *C-c C-q* (`c-indent-defun`) when re-indents the entire top-level function or class definition that encompasses point. It leaves point at the same position within the buffer.

---

[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. *Don't* try to indent this stuff!

[2] Actually what 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 the setting of the variable `indent-tabs-mode`. If you preferred, you could set `c-insert-tab-function` to `tab-to-tab-stop` for example.

[3] The caveat about `indent-tabs-mode` in the previous footnote also applies here.

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

While not strictly an indentation function, *M-C-h* (`c-mark-function`) is useful for marking the current top-level function or class definition as the current region.

5.2 Other Commands

CC Mode contains other 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. When going backwards, `#elif` is treated like `#else` followed by `#if`. When going forwards, `#elif` is 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, it moves 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 a comment, or next to a comment, this command moves by sentences instead of statements.

When called from a program, this function takes three optional arguments: the numeric prefix argument, a buffer position limit (used as a starting point for syntactic parsing and as a limit for backward movement), and a flag to indicate whether movement should be by statements (if `nil`) or sentence (if non-`nil`).

`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 a comment, or next to a comment, this command moves by sentences instead of statements.

When called from a program, this function takes three optional arguments: the numeric prefix argument, a buffer position limit (used as a starting point for syntactic parsing and as a limit for backward movement), and a flag to indicate whether movement should be by statements (if `nil`) or sentence (if non-`nil`).

`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.

`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.

`M-q (fill-paragraph)`
: The command is used to fill a block style (C) or line style (C++) comment, in much the same way that text in the various text modes can be filled[4]. You should never attempt to fill non-comment code sections; you'll end up with garbage! Two variables control how C style block comments are filled, specifically how the comment start and end delimiters are handled.

The variable `c-hanging-comment-starter-p` controls whether comment start delimiters which appear on a line by themselves, end up on a line by themselves

[4] You should not use specialized filling packages such as `filladapt` with CC Mode. They don't work as well for filling as `c-fill-paragraph`

after the fill. When the value is `nil`, the comment starter will remain on its own line[5]. Otherwise, text on the next line will be put on the same line as the comment starter. This is called *hanging* because the following text hangs on the line with the comment starter[6]

The variable `c-hanging-comment-ender-p` controls the analogous behavior for the block comment end delimiter. When the value is `nil`, the comment ender will remain on its own line after the file[7]. Otherwise, the comment end delimiter will be placed at the end of the previous line.

[5] It will not be placed on a separate line if it is not already on a separate line.

[6] This variable is `t` by default, except in `java-mode`. Hanging comment starters mess up Javadoc style comments.

[7] The same caveat as above holds true.

6 Customizing Indentation

The variable `c-offsets-alist` contains the mappings between syntactic symbols and the offsets to apply for those symbols. You should never modify this variable directly though. Use the function `c-set-offset` instead (see below for details).

The `c-offsets-alist` variable is where you customize all your indentations. You simply need to decide what additional offset you want to add for every syntactic symbol. 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. Also, you can set up *styles* of indentatio. Most likely, you'll find one of the pre-defined styles will suit your needs, but if not, this section will describe how to set up basic editing configurations. See Section 6.3 [Styles], page 21, for an explanation of how to set up named styles.

As mentioned previously, the variable `c-offsets-alist` is an association list of syntactic symbols and the offsets to be applied for those symbols. In fact, these offset values can be any of an integer, a function or lambda expression, a variable name, or one of the following symbols: `+`, `-`, `++`, `--`, `*`, or `/`. These symbols describe offset in multiples of the value of the 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 leaving the same relationship between levels. 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

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 (in your '`.emacs`' file):

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

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 value associated with the syntactic symbols in the `c-offsets-alist` variable. 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.

6.1 Interactive Customization

As an example of how to customize indentation, let's change the style of this example[1]:

```
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:

---

[1] In this an subsequent examples, the original code is formatted using the '`gnu`' style unless otherwise indicated. See Section 6.3 [Styles], page 21.

```
((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* (`c-set-offset`). 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` in the `c-offsets-alist` variable.

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.

6.2 Permanent Customization

To make your changes permanent, you need to add some lisp code to your '`.emacs`' file, but first you need to decide whether your styles should be global in every buffer, or local to each specific buffer.

If you edit primarily one style of code, you may want to make the CC Mode style variables have global values so that every buffer will share the style settings. This will allow you to set the CC Mode variables at the top level of your '`.emacs`' file, and is the way CC Mode works by default.

If you edit many different styles of code at the same time, you might want to make the CC Mode style variables have buffer local values. If you do this, then you will need to set any CC Mode style variables in a hook function (e.g. off of `c-mode-common-hook` instead of at the top level of your '`.emacs`' file). The recommended way to do this is to set the variable `c-style-variables-are-local-p` to `t` **before** CC Mode is loaded into your Emacs session.

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 IDL 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[2].

Here's a simplified example of what you can add to your '`.emacs`' file to make the changes described in the previous section (Section 6.1 [Interactive Customization], page 19) more permanent. See the Emacs manuals for more information on customizing Emacs via hooks. See Chapter 11 [Sample .emacs File], page 44, for a more complete sample '`.emacs`' file.

```
(defun my-c-mode-common-hook ()
  ;; my customizations for all of c-mode and related modes
  (c-set-offset 'substatement-open 0)
  ;; other customizations can go here
  )
(add-hook 'c-mode-common-hook 'my-c-mode-common-hook)
```

For complex customizations, you will probably want to set up a *style* that groups all your customizations under a single name.

6.3 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 on C code. 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

[2] The interaction between `java-mode` and the hook variables is slightly different than for the other modes. `java-mode` sets the style (see Section 6.3 [Styles], page 21) of the buffer to '`java`' *before* running the `c-mode-common-hook` or `java-mode-hook`. You need to be aware of this so that style settings in `c-mode-common-hook` don't clobber your Java style.

called *styles*, associate a single name for any particular style, and pretty easily start editing new or existing code using these styles.

6.3.1 Built-in Styles

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

- `gnu` — coding style blessed by the Free Software Foundation for C code in GNU programs. This is the default style for all newly created buffers, but you can change this by setting the variable `c-default-style`.
- `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[3].
- `linux` — C coding standard for Linux development.
- `python` — C coding standard for Python extension modules[4].
- `java` — The style for editing Java code. Note that this style is automatically installed when you enter `java-mode`.
- `user` — This is a special style for several reasons. First, if you customize CC Mode by using either the new Custom interface or by doing `setq`'s at the top level of your '`.emacs`' file, these settings 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 6.3.2 [Adding Styles], page 23) you need only define the differences between your new style and `user` style.

  Note however that `user` style is *not* the default style. `gnu` is the default style for all newly created buffers, but you can change this by setting variable `c-default-style`. Be careful if you customize CC Mode as described above; since your changes will be captured in the `user` style, you will also have to change `c-default-style` to "user" to see the effect of your customizations.

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.

---

[3] This document is ftp'able from `euagate.eua.ericsson.se`

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

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 5 [Commands], page 14.

Once you find a built-in style you like, you can make the change permanent by adding some lisp to your '`.emacs`' file. Let's say for example that you want to use the '`ellemtel`' style in all your files. You would add this:

```
(defun my-c-mode-common-hook ()
  ;; use Ellemtel style for all C like languages
  (c-set-style "ellemtel")
  ;; other customizations can go here
  )
(add-hook 'c-mode-common-hook 'my-c-mode-common-hook)
```

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 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.

6.3.2 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 Chapter 11 [Sample .emacs File], page 44.

6.3.3 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. It works via the standard Emacs hook variable `hack-local-variables-hook`.

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 offets 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.

6.4 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.

Note that the style controlling variables can either have global values, or can be buffer local (e.g. different in every buffer). If all the C files you edit tend to have the same style, you might want to keep the variables global. If you tend to edit files with many different styles, you will have to make the variables buffer local. The variable `c-style-variables-are-local-p` controls this.

When `c-style-variables-are-local-p` is non-nil, then the style variables will have a different settable value for each buffer, otherwise all buffers will share the same values. By default, its value is `nil` (i.e. global values). You **must** set this variable before CC Mode is loaded into your Emacs session, and once the variables are made buffer local, they cannot be made global again (unless you restart Emacs of course!)

6.4.1 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 7 [Syntactic Symbols], page 30). CC Mode itself uses custom 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))))
```

Custom indent 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.

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`[5]:

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

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

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

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 several standard custom indentation functions, not all of which are used by the default styles.

- `c-lineup-arglist` — lines up function argument lines under the argument on the previous line.
- `c-lineup-arglist-intro-after-paren` — similar to `c-lineup-arglist`, but works for argument lists that begin with an open parenthesis followed by a newline.
- `c-lineup-arglist-close-under-paren` — set your `arglist-close` syntactic symbol to this line-up function so that parentheses that close argument lists will line up under the parenthesis that opened the argument list.

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

- `c-lineup-close-paren` — lines up the closing parenthesis under its corresponding open parenthesis if that one is followed by code. Otherwise, if the open parenthesis ends its line, no indentation is added. Works with any `...-close` symbol.
- `c-lineup-streamop` — lines up C++ stream operators (e.g. '`<<`' and '`>>`').
- `c-lineup-multi-inher` — lines up multiple inheritance lines.
- `c-indent-one-line-block` — adds `c-basic-offset` to the indentation if the line is a one line block, otherwise 0. Intended to be used with any opening brace symbol, e.g. `substatement-open`.
- `c-lineup-C-comments` — lines up C block comment continuation lines.
- `c-lineup-comment` — lines up comment only lines according to the variable `c-comment-only-line-offset`.
- `c-lineup-runin-statements` — lines up `statements` for coding standards which place the first statement in a block on the same line as the block opening brace[6].
- `c-lineup-math` — lines up math `statement-cont` lines under the previous line after the equals sign.
- `c-lineup-ObjC-method-call` — for Objective-C code, lines up selector arguments just after the message receiver.
- `c-lineup-ObjC-method-args` — for Objective-C code, lines up the colons that separate arguments by aligning colons vertically.
- `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.
- `c-lineup-dont-change` — this lineup function returns the indentation of the current line. Think of it as an identity function for lineups; it is used for `cpp-macro-cont` lines.

6.4.2 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` 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`. The function can also return `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:

---

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

```
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` variable. Since no use has actually been found for this feature, it isn't currently implemented!

6.4.3 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 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 default value of `c-hanging-semi&comma-criteria` is a list containing just the function `c-semi&comma-inside-parenlist`, which suppresses newlines after semicolons inside parenthesis lists (e.g. `for`-loops). 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).

6.4.4 Other Special Indentations

In '`gnu`' style (see Section 6.3.1 [Built-in Styles], page 22), a minimum indentation is imposed on lines inside top-level constructs. This minimum indentation is controlled by the variable `c-label-minimum-indentation`. The default value for this variable is 1.

One other customization variable is available in CC Mode: `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.

7 Syntactic Symbols

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

- `string` — inside multi-line string
- `c` — inside a multi-line C style block comment
- `defun-open` — brace that opens a function definition
- `defun-close` — brace that closes a 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 between them; in C++ and Java, `throws` declarations and other things can appear in this context.
- `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
- `statement` — a C statement
- `statement-cont` — a continuation of a C 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 starting with brace
- `substatement` — the first line after a conditional
- `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' that ends a `do-while` construct
- `else-clause` — the 'else' of an `if-else` 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 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
- `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

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 a `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 . 1) (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:

```
((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 . 1) (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:

[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.

```
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. Note also that line 10 is slightly different too. 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.

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:

---

[2] The list of conditional keywords are (in C, C++, Objective-C, and Java): `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.

```
 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. For 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.

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, assigned the `func-decl-cont` syntax;
- line 4, assigned both `defun-block-intro` *and* `comment-intro` syntax;
- line 5, 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, assigned `string` syntax;
- line 10, assigned `label` syntax;
- line 11, assigned `block-open` syntax;
- lines 12 and 14, assigned `cpp-macro` syntax.
- line 17, 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.

Two other syntactic symbols can appear in old style, non-prototyped C code[3]:

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

8 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[3], 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[4]. 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!

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 comment.

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[5]. 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

---

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

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

[3] Note that this variable is only defined in Emacs 19.

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

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

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.

9 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 `ESC C-\`.

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

A. `ESC C-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 `ESC C-u`, then re-indent that expression with `ESC C-q`.

Q. *Why doesn't the* `RET` *key indent the line to where the new text should go after inserting the newline?*

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` (see the sample '`.emacs`' file Chapter 11 [Sample .emacs File], page 44):

```
(define-key c-mode-base-map "\C-m" 'newline-and-indent)
```

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, mostly 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 add the following to the top of your '`.emacs`' file:

```
(require 'cc-mode)
```

See the sample '`.emacs`' file Chapter 11 [Sample .emacs File], page 44 for details.

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`. It is not part of CC Mode.

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 move into blocks.

10 Getting the latest CC Mode release

CC Mode is now standard with the latest versions of Emacs 19 and XEmacs 19. It is also the standard for Emacs 20 and XEmacs 20. You would typically just use the version that comes with your X/Emacs. These may be slightly out of date due to release schedule skew, so you should always check the canonical site for the latest version.

```
World Wide Web:

    http://www.python.org/ftp/emacs/

Anonymous FTP:

    ftp://ftp.python.org/pub/emacs/
```

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.

If you do not have World Wide Web, or anonymous ftp access, you can get the distribution through an anonymous ftp-to-mail gateway, such as the one run by DEC at:

```
ftpmail@decwrl.dec.com
```

To get CC Mode via email, send the following message in the body of your mail to that address:

```
reply <a valid net address back to you>
connect ftp.python.org
binary
uuencode
chdir pub/emacs
get cc-mode.tar.gz
```

or just send the message "help" for more information on ftpmail. Response times will vary with the number of requests in the queue. I am in no way connected to this service, so I make no claims or guarantees about its availability!

11 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")

;; Customizations for all of c-mode, c++-mode, and objc-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)
  ;; offset customizations not in my-c-style
  (c-set-offset 'member-init-intro '++)
  ;; 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)
  ;; keybindings 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, and idl-mode-map inherit from it.
  (define-key c-mode-base-map "\C-m" 'newline-and-indent)
  )

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

12 Limitations and Known Bugs

- Re-indenting large regions or expressions can be slow.
- Add-on fill packages may not work as well as CC Mode's built-in filling routines. I no longer recommend you use `filladapt` to fill comments.
- `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.

13 Mailing Lists and Submitting Bug Reports

To report bugs, use the *C-c C-b* (`c-submit-bug-report`) command. This provides vital information I 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 my ability to reproduce it.

Bug reports are now sent to the following email addresses: `cc-mode-help@python.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 `cc-mode-help@python.org`, or `help-gnu-emacs@gnu.org` which is mirrored on newsgroup `gnu.emacs.help`.

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@python.org`. Announcements will also be posted to the Usenet newsgroup `gnu.emacs.sources`. Note that the `cc-mode-victims@python.org` mailing list was recently decommissioned.

Concept Index

-

-block-intro syntactic symbols 31
-close syntactic symbols 31
-cont syntactic symbols 31
-intro syntactic symbols 31
-open syntactic symbols 31

.

.emacs file 22

A

access-label syntactic symbol 33
Adding Styles 23
Advanced Customizations 24
announcement mailing list 46
arglist-close syntactic symbol 37
arglist-cont syntactic symbol 37
arglist-cont-nonempty syntactic symbol 37
arglist-intro syntactic symbol 37
Auto-fill mode interaction 13
Auto-newline insertion 7

B

basic-offset (c-) 18
beta testers mailing list 46
block-close syntactic symbol 8, 35
block-open syntactic symbol 8, 38
BOCM 1
brace lists 36
brace-list-close syntactic symbol 8, 36
brace-list-entry syntactic symbol 8, 36
brace-list-intro syntactic symbol 8, 36
brace-list-open syntactic symbol 8, 36
BSD style 22
Built-in Styles 22

C

c syntactic symbol 38
c-basic-offset 18
c-hanging- functions 14
c-indent-exp 45
c-set-offset 18
case-label syntactic symbol 36
'`cc-compat.el`' file 1
'`cc-lobotomy.el`' file 40
'`cc-mode-18.el`' file 2
class-close syntactic symbol 8, 32
class-open syntactic symbol 8, 32
Clean-ups 9, 10
Commands 14
comment-intro syntactic symbol 38
comment-only line 4, 9
cpp-macro syntactic symbol 38
cpp-macro-cont syntactic symbol 38
Custom Brace and Colon Hanging 26
Custom Indentation Functions 8, 24
customizing brace hanging 26
customizing colon hanging 27
Customizing Indentation 18
Customizing Semi-colons and Commas 27, 28

D

defun-block-intro syntactic symbol 32
defun-close syntactic symbol 8, 32
defun-open syntactic symbol 8, 32
do-while-closure syntactic symbol 35

E

electric characters 7
electric commands 7
Ellemtel style 22
else-clause syntactic symbol 35
extern-lang-close syntactic symbol 8, 36
extern-lang-open syntactic symbol 8, 36

F

File Styles 23
Frequently Asked Questions 42
friend syntactic symbol 34
func-decl-cont syntactic symbol 38

G

Getting Connected 2
Getting the latest CC Mode release 43
GNU style 22

H

Hanging Braces 8
Hanging Colons 9
Hanging Semi-colons and commas 9
hooks 20
Hungry-deletion of whitespace 12

I

in-class inline methods 33
inclass syntactic symbol 33, 36
indent-exp (c-) 45
Indentation Calculation 5
Indentation Commands 14
inextern-lang syntactic symbol 36
inher-cont syntactic symbol 33
inher-intro syntactic symbol 33
inline-close syntactic symbol 8, 33
inline-open syntactic symbol 8, 33
innamespace syntactic-symbol 37
Interactive Customization 19
Introduction 1

J

Java style 22
java-mode 22

K

K&R style 22
knr-argdecl syntactic symbol 39
knr-argdecl-intro syntactic symbol 39

L

label syntactic symbol 38
Limitations and Known Bugs 45
Linux style 22
literal 7, 10, 12, 14
local variables 23

M

Mailing Lists and Submitting Bug Reports 46
member-init-cont syntactic symbol 33
member-init-intro syntactic symbol 33
Minor Modes 7
modifier syntactic symbol 38
multi-line macros 38

N

namespace-close symbol 8
namespace-close syntactic-symbol 37
namespace-open symbol 8
namespace-open syntactic-symbol 37
New Indentation Engine 3

O

objc-method-args-cont syntactic symbol 39
objc-method-call-cont syntactic symbol 39
objc-method-intro syntactic symbol 39
Other Commands 15
Other electric commands 9

P

Performance Issues 40
Permanent Customization 20
Python style 22

R

relative buffer position 3

S

Sample .emacs file 44
set-offset (c-) 18
statement syntactic symbol 32
statement-case-intro syntactic symbol 36
statement-case-open syntactic symbol 8, 36
statement-cont syntactic symbol 32
stream-op syntactic symbol 25, 38
string syntactic symbol 38
Stroustrup style 22
Styles 21
substatement 4

substatement syntactic symbol 35
substatement-block-intro syntactic symbol 35
substatement-open syntactic symbol 8, 35
substatment block 4
Syntactic Analysis 3
syntactic component 3
syntactic component list 3
syntactic symbol 3
Syntactic Symbols 30
syntactic whitespace 7, 38

T

TAB 6
topmost-intro syntactic symbol 32
topmost-intro-cont syntactic symbol 32

U

User style 22

W

Whitesmith style 22

Command Index

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

A

add-style (c-) 23

B

backward-conditional (c-) 15
backward-delete-char-untabify 12
backward-into-nomenclature (c-) 16
beginning-of-defun 15, 40
beginning-of-defun (c-) 15
beginning-of-statement (c-) 15

C

c-add-style 23
c-backward-conditional 15
c-backward-into-nomenclature 16
c-beginning-of-defun 15
c-beginning-of-statement 15
c-electric-backspace 12
c-electric-brace 8
c-electric-delete 12
c-electric-lt-gt 10
c-electric-pound 9
c-electric-slash 9
c-electric-star 9
c-end-of-defun 15
c-end-of-statement 16
c-forward-conditional 15
c-forward-into-nomenclature 16
c-hanging-braces-alist 14
c-indent-command 14
c-indent-defun 14, 20
c-indent-exp 14
c-indent-one-line-block 26
c-lineup-arglist 25
c-lineup-arglist-close-under-paren 25
c-lineup-arglist-intro-after-paren 25
c-lineup-C-comments 26
c-lineup-close-paren 25
c-lineup-comment 26
c-lineup-dont-change 26
c-lineup-math 26
c-lineup-multi-inher 26
c-lineup-ObjC-method-args 26
c-lineup-ObjC-method-args-2 26
c-lineup-ObjC-method-call 26
c-lineup-runin-statements 26
c-lineup-streamop 25, 26
c-mark-function 15
c-mode 1
c-scope-operator 16
c-semi&comma-inside-parenlist 28
c-semi&comma-no-newlines-before-nonblanks 28
c-semi&comma-no-newlines-for-oneline-inliners 28
c-set-offset 20, 23
c-set-style 22
c-show-syntactic-information 3
c-snug-do-while 27
c-submit-bug-report 46
c-toggle-auto-hungry-state 7
c-toggle-auto-state 7
c-toggle-hungry-state 7
c-up-conditional 15
c-version 2
c++-mode 1

D

defun-prompt-regexp 40
delete-char 12

E

electric-backspace (c-) 12
electric-brace (c-) 8
electric-delete (c-) 12
electric-lt-gt (c-) 10
electric-pound (c-) 9
electric-slash (c-) 9
electric-star (c-) 9
end-of-defun 15
end-of-defun (c-) 15
end-of-statement (c-) 16

F

fill-paragraph 16
forward-conditional (c-) 15
forward-into-nomenclature (c-) 16

H

hanging-braces-alist (c-) 14

I

idl-mode 1
indent-command (c-) 14
indent-defun (c-) 14, 20
indent-exp (c-) 14
indent-for-comment 28
indent-one-line-block (c-) 26
indent-region 14

J

java-mode 1

L

lineup-arglist (c-) 25
lineup-arglist-close-under-paren (c-) 25
lineup-arglist-intro-after-paren (c-) 25
lineup-C-comments (c-) 26
lineup-close-paren (c-) 25
lineup-comment (c-) 26
lineup-dont-change (c-) 26
lineup-math (c-) 26
lineup-multi-inher (c-) 26
lineup-ObjC-method-args (c-) 26
lineup-ObjC-method-args-2 (c-) 26
lineup-ObjC-method-call (c-) 26
lineup-runin-statements (c-) 26
lineup-streamop (c-) 25, 26

M

mark-function (c-) 15

N

newline-and-indent 42

O

objc-mode 1

S

scope-operator (c-) 16
semi&comma-inside-parenlist (c-) 28
semi&comma-no-newlines-before-nonblanks (c-) 28
semi&comma-no-newlines-for-oneline-inliners (c-) 28
set-offset (c-) 20, 23
set-style (c-) 22
show-syntactic-information (c-) 3
snug-do-while (c-) 27
submit-bug-report (c-) 46

T

tab-to-tab-stop 14
toggle-auto-hungry-state (c-) 7
toggle-auto-state (c-) 7
toggle-hungry-state (c-) 7

U

up-conditional (c-) 15

V

version (c-) 2

Key Index

#

`#` 9

>

`>` 10

<

`<` 10

B

`Backspace` 12

C

`C-c` 22
`C-c :` 16
`C-c C-a` 7
`C-c C-b` 46
`C-c C-d` 7
`C-c C-n` 15
`C-c C-o` 20
`C-c C-p` 15
`C-c C-q` 14, 20, 25, 42
`C-c C-s` 3, 31
`C-c C-t` 7
`C-c C-u` 15
`C-j` 42
`C-u` 7
`C-x h` 42

D

`DEL` 12

E

`ESC a` 15
`ESC C-\` 42
`ESC C-q` 42
`ESC C-u` 42
`ESC C-x` 42
`ESC e` 16
`ESC q` 16

M

`M-;` 28
`M-C-\` 14
`M-C-h` 15
`M-C-q` 14

R

`RET` 42

T

`TAB` 5, 14

Variable Index

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

B

backspace-function (c-) 12
basic-offset (c-) 24

C

c-backspace-function 12
c-basic-offset 24
c-cleanup-list 10
c-comment-continuation-stars 13
c-comment-only-line-offset 26
c-default-style 22
c-delete-function 12
c-echo-syntactic-information-p 6
c-electric-pound-behavior 9
c-file-offsets 23
c-file-style 23
c-hanging-braces-alist 8, 26
c-hanging-colon-alist 27
c-hanging-colons-alist 9
c-hanging-comment-ender-p 16
c-hanging-comment-starter-p 16
c-hanging-semi&comma-criteria 27
c-indent-comments-syntactically-p 28
c-indentation-style 23
c-initialization-hook 20
c-insert-tab-function 14
c-Java-defun-prompt-regexp 40
c-label-minimum-indentation 28
c-mode-common-hook 20
c-mode-hook 20
c-offsets-alist 3, 5, 8, 9, 18, 23, 25, 30
c-progress-interval 14
c-recognize-knr-p 40
c-special-indent-hook 28
c-style-alist 23, 24
c-style-variables-are-local-p 18, 20, 24
c-syntactic-context 27
c-tab-always-indent 14
c++-mode-hook 20
cc-lobotomy-pith-list 40
cleanup-list (c-) 10
comment-column 28
comment-continuation-stars (c-) 13
comment-line-break-function 13
comment-only-line-offset (c-) 26

D

default-style (c-) 22
delete-function (c-) 12
delete-key-deletes-forward 12

E

echo-syntactic-information-p (c-) 6
electric-pound-behavior (c-) 9

F

file-offsets (c-) 23
file-style (c-) 23

H

hanging-braces-alist (c-) 8, 26
hanging-colon-alist (c-) 27
hanging-colons-alist (c-) 9
hanging-comment-ender-p (c-) 16
hanging-comment-starter-p (c-) 16
hanging-semi&comma-criteria (c-) 27

I

idl-mode-hook 20
indent-comments-syntactically-p (c-) 28
indent-tabs-mode 14
indentation-style (c-) 23
initialization-hook (c-) 20
insert-tab-function (c-) 14

J

Java-defun-prompt-regexp (c-) 40
java-mode-hook 20

L

label-minimum-indentation (c-) 28

M

mode-common-hook (c-) 20

O

objc-mode-hook 20
offsets-alist (c-) 3, 5, 8, 9, 18, 23, 25, 30

P

progress-interval (c-) 14

R

recognize-knr-p (c-) 40

S

special-indent-hook (c-) 28
style-alist (c-) 23, 24
style-variables-are-local-p 24
style-variables-are-local-p (c-) 18, 20
syntactic-context (c-) 27

T

tab-always-indent (c-) 14

Short Contents

1 Introduction 1
2 Getting Connected 2
3 New Indentation Engine 3
4 Minor Modes 7
5 Commands 14
6 Customizing Indentation 18
7 Syntactic Symbols 30
8 Performance Issues 40
9 Frequently Asked Questions 42
10 Getting the latest CC Mode release 43
11 Sample .emacs file 44
12 Limitations and Known Bugs 45
13 Mailing Lists and Submitting Bug Reports 46
Concept Index 47
Command Index 50
Key Index 52
Variable Index 53

Table of Contents

1 Introduction 1

2 Getting Connected 2

3 New Indentation Engine 3
- 3.1 Syntactic Analysis 3
- 3.2 Indentation Calculation 5

4 Minor Modes 7
- 4.1 Auto-newline insertion 7
  - 4.1.1 Hanging Braces 8
  - 4.1.2 Hanging Colons 9
  - 4.1.3 Hanging Semi-colons and commas 9
  - 4.1.4 Other electric commands 9
  - 4.1.5 Clean-ups 10
- 4.2 Hungry-deletion of whitespace 12
- 4.3 Auto-fill mode interaction 13

5 Commands 14
- 5.1 Indentation Commands 14
- 5.2 Other Commands 15

6 Customizing Indentation 18
- 6.1 Interactive Customization 19
- 6.2 Permanent Customization 20
- 6.3 Styles 21
  - 6.3.1 Built-in Styles 22
  - 6.3.2 Adding Styles 23
  - 6.3.3 File Styles 23
- 6.4 Advanced Customizations 24
  - 6.4.1 Custom Indentation Functions 24
  - 6.4.2 Custom Brace and Colon Hanging 26
  - 6.4.3 Customizing Semi-colons and Commas 27
  - 6.4.4 Other Special Indentations 28

7 Syntactic Symbols 30

8 Performance Issues 40

9 Frequently Asked Questions 42

10 Getting the latest CC Mode release 43

11 Sample .emacs file 44

12 Limitations and Known Bugs 45

13 Mailing Lists and Submitting Bug Reports 46

Concept Index 47

Command Index 50

Key Index 52

Variable Index 53