CC-MODE Version 4

A GNU Emacs mode for editing C, C++, and Objective-C code.
(manual revision: 2.35)

Barry A. Warsaw

1 Introduction

Welcome to `cc-mode`, version 4. This is a GNU Emacs mode for editing files containing C, C++, Objective-C, and Java 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` version 4 supports the editing of K&R and ANSI C, *ARM*[1] C++, Objective-C, and Java files. In this way, you can easily set up consistent coding styles for use in editing all C, C++, Objective-C, and Java programs.

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 file is '`cc-mode.el`', and I'll often refer to the package as `cc-mode`, but there really is no top level `cc-mode` entry point. I call it `cc-mode` simply to differentiate it from '`c-mode.el`'. All of the variables, commands, and functions in `cc-mode` are prefixed with `c-<thing>`, and `c-mode`, `c++-mode`, `objc-mode`, and `java-mode` entry points are provided. This file is intended to be a replacement for '`c-mode.el`' and '`c++-mode.el`'.

The major version number was incremented to 4 with the addition of `objc-mode`. To find the minor revision number of this release, use *`M-x c-version RET`*. Work has already begun on `cc-mode` version 5, in which Emacs 18 will not be supported.

As of this writing (19-Jan-1996), both Emacs 19.30 and XEmacs 19.13 are distributed with `cc-mode`. Emacs 19.31 and XEmacs 19.14 will both contain the latest version of cc-mode when it is released. If you are running older versions of these Emacsen, you may want to upgrade your copy of `cc-mode`. See Chapter 10 [Getting the latest cc-mode release], page 35.

This distribution also contains a file called '`cc-compat.el`' which should ease your transition from BOCM to `cc-mode`. It currently comes unguaranteed and unsupported, but this may change for future versions.

A special word of thanks goes to Krishna Padmasola for his work in converting the original '`README`' file to texinfo format. `cc-mode` users have been clamoring for a manual for a long time, and thanks to Krishna, it is now available <clap> <clap> <clap>! `:-)`

[1] i.e. "The Annotated C++ Reference Manual", by Ellis and Stroustrup.

2 Getting Connected

'`cc-mode.el`' works well with the 2 main branches of Emacs 19: XEmacs and the Emacs 19 maintained by the FSF. Emacs 19 users will want to use Emacs version 19.21 or better, XEmacs users will want 19.6 or better. Earlier versions of these Emacsen have deficiencies and/or bugs which will adversely affect the performance and usability of `cc-mode`.

Similarly if you use the '`cc-mode-18.el`' compatibility file, '`cc-mode.el`' will work with Emacs 18, but only moderately well. A word of warning though, *Emacs 18 lacks some fundamental functionality and that ultimately means using Emacs 18 is a losing battle*. Hence `cc-mode` under Emacs 18 is no longer supported and it is highly recommended that you upgrade to Emacs 19. If you use `cc-mode` under Emacs 18, you're on your own. With `cc-mode` version 5, Emacs 18 support will be dropped altogether.

Note that as of XEmacs 19.13 and Emacs 19.30, your Emacs already comes with `cc-mode` version 4 preconfigured for your use. You should be able to safely skip the rest of the setup information in this chapter.

The first thing you will want to do is put '`cc-mode.el`' somewhere on your `load-path` so Emacs can find it. Do a *`C-h v load-path RET`* to see all the directories Emacs looks at when loading a file. If none of these directories are appropriate, create a new directory and add it to your `load-path`:

[in the shell]

```
% cd
% mkdir mylisp
% mv cc-mode.el mylisp
% cd mylisp
```

[in your .emacs file add]

```
(setq load-path (cons "~/mylisp" load-path))
```

Next you want to *byte compile* '`cc-mode.el`'. The mode uses a lot of macros so if you don't byte compile it, things will be unbearably slow. *You can ignore all byte-compiler warnings!* They are the result of the supporting different versions of Emacs, and none of the warnings have any effect on operation. Let me say this again: **You really can ignore all byte-compiler warnings!**

Here's what to do to byte-compile the file [in emacs]:

```
M-x byte-compile-file RET ~/mylisp/cc-mode.el RET
```

If you are running a version of Emacs or XEmacs that comes with `cc-mode` by default, you can simply add the following to your '`.emacs`' file in order to upgrade to the latest version of `cc-mode`:

```
(load "cc-mode")
```

Users of even older versions of Emacs 19, Emacs 18, or of the older Lucid Emacs will probably be running an Emacs that has BOCM '`c-mode.el`' and possible '`c++-mode.el`' pre-dumped. If your Emacs is dumped with either of these files you first need to make Emacs "forget" about those older modes.

If you can do a *`C-h v c-mode-map RET`* without getting an error, you need to add these lines at the top of your '`.emacs`' file:

```
(fmakunbound 'c-mode)
(makunbound  'c-mode-map)
(fmakunbound 'c++-mode)
(makunbound  'c++-mode-map)
(makunbound  'c-style-alist)
```

After those lines you will want to add the following autoloads to your '`.emacs`' file so that `cc-mode` gets loaded at the right time:

```
(autoload 'c++-mode  "cc-mode" "C++ Editing Mode" t)
(autoload 'c-mode    "cc-mode" "C Editing Mode" t)
(autoload 'objc-mode "cc-mode" "Objective-C Editing Mode" t)
(autoload 'java-mode "cc-mode" "Java Editing Mode" t)
```

Alternatively, if you want to make sure `cc-mode` is loaded when Emacs starts up, you could use this line instead of the three autoloads above:

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

Next, you will want to set up Emacs so that it edits C files in `c-mode`, C++ files in `c++-mode`, and Objective-C files in `objc-mode`. All users should add the following to their '`.emacs`' file. Note that this assumes you'll be editing `.h` and `.c` files as C, `.hh`, `.cc`, `.H`, and `.C` files as C++, `.m` files as Objective-C, and `.java` files as Java code. YMMV:

```
(setq auto-mode-alist
  (append
    '(("\\.C$"    . c++-mode)
      ("\\.H$"    . c++-mode)
      ("\\.cc$"   . c++-mode)
      ("\\.hh$"   . c++-mode)
      ("\\.c$"    . c-mode)
      ("\\.h$"    . c-mode)
      ("\\.m$"    . objc-mode)
      ("\\.java$" . java-mode)
     ) auto-mode-alist))
```

You may already have some or all of these settings on your `auto-mode-alist`, but it won't hurt to put them on there again.

That's all you need – I know, I know, it sounds like a lot `:-)`, but after you've done all this, you should only need to quit and restart Emacs. The next time you visit a C, C++, Objective-C, or Java file you should be using `cc-mode`. You can check this easily by hitting *`M-x c-version RET`* in the `c-mode`, `c++-mode`, or `objc-mode` buffer. You should see this message in the echo area:

> `Using cc-mode version 4.`*`xxx`*
>
> `Where `*`xxx`*` is the latest release minor number.`

3 New Indentation Engine

`cc-mode` has a new indentation engine, providing a simplified, yet flexible and general mechanism for customizing indentation. It breaks indentation calculation into two steps. First for the line of code being indented, `cc-mode` analyzes what 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 26, 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.

It might help to see 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 this example, we'd see in the echo area:

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

[1] or C++, Objective-C, or Java 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!

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.[3]

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 example 3 gives us:

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

[3] A *substatement* indicates the line after an `if`, `else`, `while`, `do`, `switch`, or `for` statement, and a *substatement block* is a brace block following one of those constructs.

so 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 the current line is calculated using the syntactic component list derived in step 1 above (see Section 3.1 [Syntactic Analysis], page 5). 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 the 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. Since the brace at buffer position 29 is in column zero, it 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.

To help you configure `cc-mode`, you can 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 comes 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 while editing code on the fly. 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 together, 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].

[1] Remember that the '`C`' would be replaced with '`C++`' or '`ObjC`' if you were editing C++ or Objective-C code.

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

- No numeric argument was supplied to the command (i.e. it was typed as normal, with no `C-u` prefix).

Certain other conditions may apply on a language specific basis. For example, the second slash (`/`) of a C++ style line comment is electric in `c++-mode`, `objc-mode`, and `java-mode`, but not in `c-mode`.

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 on is also indented according to `c-offsets-alist`.

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`, and `statement-case-open`. See Chapter 7 [Syntactic Symbols], page 26 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 23 for a more detailed discussion of using a function as a brace hanging *ACTION*.

When *ACTION* is a list, it can contain any combination of the symbols `before` or `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)))
```

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` 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*.

4.1.2 Hanging Colons

Using a mechanism similar to brace hanging (see Section 4.1.1 [Hanging Braces], page 10), colons can also be made to hang using the variable `c-hanging-colons-alist`. The syntactic symbols appropriate for this association list are: `case-label`, `label`, `access-label`, `member-init-intro`, and `inher-intro`. See Section 4.1.1 [Hanging Braces], page 10 and Section 6.4.2 [Custom Brace and Colon Hanging], page 23 for details. Note however, that `c-hanging-colons-alist` does not implement functions as *ACTION*s.

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 12 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 25 for details.

4.1.4 Other electric commands

A few other keys also provide electric behavior. For example the `#` key (`c-electric-pound`) is electric when it is 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 cpp macro definitions.

Stars and slashes (i.e. `*` and `/`) 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`.

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 isn't 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 9), 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 {
```

- `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.

[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.

4.2 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 *DEL* character will consume all preceding whitespace, including newlines and tabs. This can really cut down on the number of *DEL*'s you have to type if, for example you made a mistake on the preceding line.

By default, `cc-mode` actually runs the command `c-electric-delete` when you hit *DEL*. When this command is used to delete a single character (i.e. when it is called interactively with no numeric argument), it really runs the function contained in the variable `c-delete-function`. This function is called with a single argument, which is the number of characters to delete. `c-delete-function` is also called when the *DEL* key is typed inside a literal (see Section 4.1 [Auto-newline insertion], page 9. Inside a literal, `c-electric-delete` is not electric, which is typical of all the so-called electric commands.

5 Indentation Commands

Various commands are provided which allow you to conveniently re-indent C constructs, and these are outlined below. There are several things to note about these indentation commands. First, when you change your programming style, either though `c-set-style` or 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 newly entered code. So for example, 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 will be made for the next release, 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 your large re-indentation command. The variable `c-progress-interval` controls how often a progress message is displayed. Set this variable to `nil` to inhibit progress messages. Note that this feature only works with Emacs 19.

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 5 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 a real tab character is inserted. If this variable's value is something other that `t` or `nil` (e.g. `'other`), then a real tab character is inserted only when point is inside a literal (see Section 4.1 [Auto-newline insertion], page 9), 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.

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.

---

[1] In particular, I have had people complain about the speed that `cc-mode` re-indents `lex(1)` output. Lex, yacc, and other code generators usually output some pretty perverse code. *Don't* try to indent this stuff with `cc-mode`!

6 Customizing Indentation

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 indentation just like in BOCM. 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 19 for an explanation of how to set up named styles.

As mentioned previously, the variable `c-offsets-alist` is an association list between syntactic symbols and the offsets to be applied for those symbols. In fact, these offset values can be an integer, a function or 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 multiples of `c-basic-offset` 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)[1]:

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

This would change

[1] The reason you need to use `setq-default` instead of `setq` is that `c-basic-offset` is a buffer local variable, as are most of the `cc-mode` configuration variables. If you were to put this code in, e.g. your `c-mode-common-hook` function, you could use `setq`.

```
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 example 2 above from:

```
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 component affect the offset calculations for that line. Hitting *C-c C-s* on line 4 yields:

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

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 '`+`', so hit backspace to delete the '`+`', then hit '`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 Indentation

To make this change permanent, you need to add some lisp code to your '`.emacs`' file. `cc-mode` provides four hooks that you can use to customize your language editing styles. Four language specific hooks are provided, according to Emacs major mode conventions: `c-mode-hook`, `c++-mode-hook`, `objc-mode-hook`, and `java-mode-hook`. These get run as the last thing when you enter `c-mode`, `c++-mode`, `objc-mode`, or `java-mode-hook` respectively. `cc-mode` also provides a hook called `c-mode-common-hook` which is run by all three modes *before* the language specific hook. Thus, to make changes consistently across all supported `cc-mode` modes, use `c-mode-common-hook`. Most of the examples in this section will assume you are using the common hook.

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 17) more permanent. See the Emacs manuals for more information on customizing Emacs via hooks. See Chapter 11 [Sample .emacs File], page 36 for a more complete sample '`.emacs`' file.[2]

```
(defun my-c-mode-common-hook ()
  ;; my customizations for all of c-mode, c++-mode, objc-mode, java-mode
  (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.

The offset value can also be a function, and this is how power users gain enormous flexibility in customizing indentation. See Section 6.4 [Advanced Customizations], page 21 for details.

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 some variety of coding styles, and as programmers come and go, there could be a number of styles in use. For this reason, `cc-mode` makes it convenient for you to set up logical groupings of customizations called *styles*, associate a single name for any particular style, and pretty easily start editing new or existing code using these styles. This chapter describes how to set up styles and how to edit your C code using styles.

[2] The use of `add-hook` in this example only works for Emacs 19. Workarounds are available if you are using Emacs 18.

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. Some of the most common C and C++ styles are already built-in. These include:

- `gnu` – coding style blessed by the Free Software Foundation for C code in GNU programs.
- `k&r` – The classic Kernighan and Ritchie style for C code.
- `bsd` – **<TBD> Anybody know anything about the history of this style?**
- `stroustrup` – The classic Stroustrup style for C++ code.
- `whitesmith` – **<TBD> Anybody know anything about the history of this style?**
- `ellemtel` – Popular C++ coding standards as defined by "Programming in C++, Rules and Recommendations", Erik Nyquist and Mats Henricson, Ellemtel[3].
- `java` – The style for editing Java code. Note that this style is automatically installed when you enter `java-mode`.
- `CC-MODE` – Style that encapsulates the default values of the `cc-mode` variables. See below for details.

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

M-x c-set-style RET STYLE-NAME RET

Note that all style names are case insensitive, even the ones you define.

Setting a style in this way does *not* automatically re-indent your file. For commands that you can use to view the effect of your changes, see Chapter 5 [Indentation Commands], page 15.

Once you find a built-in style you like, you can make the change permanent by adding a call 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, C++, and Objective-C code
  (c-set-style "ellemtel")
  ;; other customizations can go here
  )
(add-hook 'c-mode-common-hook 'my-c-mode-common-hook)
```

There is one other special style you can use, called `CC-MODE`. This is a style that is calculated by `cc-mode` when it starts up. The `CC-MODE` style is also special because all other styles implicitly inherit from it; in other words, whenever you set a style, `cc-mode` first re-instates the `CC-MODE` style, then applies your new style configurations.

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

The `CC-MODE` style exists because once `cc-mode` initializes, it institutes the `gnu` style for compatibility with BOCM's defaults. Any customizations you make in mode hooks will be based on the `gnu` style, unless you first do a `c-set-style` to `CC-MODE` or some other built-in style.

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 probably won't want to modify this variable directly. `cc-mode` provides a function, called `c-add-style`, that you can use to easily add new styles or update 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 already associated with *stylename* is changed to the new *description*. This function also takes an optional third argument, which if non-`nil`, automatically institutes the new style in 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 36.

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`, which is highly inconvenient for use in a Local Variable block. `cc-mode` provides two variables that make it easier for you to customize your style on a per-file basis.

The variable `c-file-style` can be set to a style name string as described in Section 6.3.1 [Built-in Styles], page 20. When the file is visited, `cc-mode` will automatically set the file's style to this style using `c-set-style`.

Another variable, `c-file-offsets`, takes an association list similar to what is allowed in `c-offsets-alist`. When the file is visited, `cc-mode` will automatically institute these offsets using `c-set-offset`.

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

6.4 Advanced Customizations

For most users, `cc-mode` will support their coding styles with very little need for 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 advanced ability for customization, and one of the real strengths of `cc-mode` is that the syntactic analysis model provides a very flexible framework for customizing indentation.

This allows you to perform special indentation calculations for situations not handled by the mode directly.

6.4.1 Custom Indentation Functions

One of the most common ways to customize `cc-mode` is by writing *custom indentation functions* and associating them with specific syntactic symbols (see Chapter 7 [Syntactic Symbols], page 26). `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. If `stream-op` had an offset of `+`, and `c-basic-offset` was 2, lines 4 through 6 would simply be 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 offsets the stream operators under the operator in line 3. 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 the lisp code (from the '`cc-mode.el`' source file) that implements 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 5). 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 want that value added into the final total 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`[4]:

```
(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.

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 10). However, an *ACTION* can also be a function symbol which gets called when a brace matching that syntactic symbol is typed.

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 described in Section 4.1.1 [Hanging Braces], page 10.

As an example, `cc-mode` itself uses this feature to dynamically determine the hanginess of braces which close '`do-while`' constructs:

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

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

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

```
(defun c-snug-do-while (syntax pos)
  "Dynamically calculate brace hanginess for do-while statements.
Using this function, `while' clauses that end a `do-while' block will
remain on the same line as the brace that closes that block.

See `c-hanging-braces-alist' for how to utilize this function as an
ACTION associated with `block-close' syntax."
  (save-excursion
    (let (langelem)
      (if (and (eq syntax 'block-close)
               (setq langelem (assq 'block-close c-syntactic-context))
               (progn (goto-char (cdr langelem))
                      (if (= (following-char) ?{)
                          (forward-sexp -1))
                      (looking-at "\\<do\\>[^_]")))
          '(before)
        '(before after)))))
```

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

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

Note that for symmetry, colon hanginess should be customizable by allowing function symbols as *ACTION*s on the `c-hanging-colon-alist` 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 9). 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).

6.4.4 Other Special Indentations

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`).

7 Syntactic Symbols

The complete list of recognized syntactic symbols is described in the `c-offsets-alist` variable. This chapter will provide some examples to help clarify these symbols.

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, which happens to be 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: }
```

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++, 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 17 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.

Things get interesting at line 5. The primary syntactic symbol for this line is `access-label` since this a label keyword that specifies access protection in C++. However, this line actually shows two syntactic symbols when you hit `C-c C-s`. This is because it is also a top-level construct inside a class definition. Thus the other syntactic symbol assigned to this line is `inclass`. Similarly, line 6 is given both `inclass` and `topmost-intro` syntax.

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 is assigned `inline-open` 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. For example, if the definition of the `Bass` constructor appeared outside the class definition, line 11 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 three 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' );
}
```

Similarly, line 16 is given `inline-close` syntax.

As in the first example above, line 12 is given `defun-block-open` syntax and lines 13 through 15 are all given `statement` syntax.

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 generic 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:

```
 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:

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

```
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.

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.

The other lines with relevant syntactic symbols include lines 2 and 17 which 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 *non-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[3].

[3] The need for this somewhat confusing arrangement is that the typical indentation desired for these lines is calculated very differently. This should be simplified in version 5 of `cc-`

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 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_long_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 which is assigned the `ansi-funcdecl-cont` syntax;
- line 4 which is assigned both `defun-block-intro` *and* `comment-intro` syntax[4];
- line 5 which is assigned `c` syntax;
- line 6 which, even though it contains nothing but whitespace, is assigned `defun-block-intro`. Note that the appearance of the comment on lines 4 and 5 do not cause line 6 to be assigned `statement` syntax because comments are considered to be *syntactic whitespace*, which are essentially ignored when analyzing code;
- line 8 which is assigned `string` syntax;
- line 10 which is assigned `label` syntax;
- line 11 which is assigned `block-open` syntax;
- lines 12 and 14 which are assigned `cpp-macro` syntax;
- line 17 which is assigned `stream-op` syntax[5].

`mode`, along with the added distinction between argument lists in function declarations, and argument lists in function calls.

[4] The `comment-intro` syntactic symbol is known generically as a *modifier* since it always appears on a syntactic analysis list with other symbols, and never has a relative buffer position.

[5] In C++ only.

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.

Other syntactic symbols may be recognized by `cc-mode`, but these are more obscure and so I haven't included examples of them. These include: `knr-argdecl-intro`, `knr-argdecl`, and the `friend` modifier.

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. Some pathological code can cause `cc-mode` to slow down considerably. This section identifies some of the coding styles to watch out for, and suggests some workarounds that you can use to improve performance.

Note that this is an area that will get a lot of attention in `cc-mode` version 5. The mode should end up being much faster, at the expense of dropping Emacs 18 support, owing to the implementation of syntactic analysis caching. This is the last release of `cc-mode` that will be compatible with Emacs 18.

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!

One of the simplest things you can do to reduce scan time, is make sure any brace that opens a top-level block construct 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 should set the variable `defun-prompt-regexp` to something reasonable[1], however that "something reasonable" is difficult to define, so `cc-mode` doesn't do it for you.

You will probably notice pathological behavior from `cc-mode` when working in files containing large amounts of cpp macros. This is because `cc-mode` cannot quickly skip backwards over these lines, which do not contribute to the syntactic calculations. You'll probably also have problems if you are editing "K&R" C code, i.e. C code that does not use function prototypes. This is because there are ambiguities in the C syntax when K&R style argument lists are used, and `cc-mode` has to use a slower scan to determine what it's looking at.

For the latter problem, I would suggest converting to ANSI style protocols, and turning the variable `c-recognize-knr-p` to `nil` (this is its default value for all modes).

For the former problem, you might want to investigate some of the speed-ups provided for you in the file '`cc-lobotomy.el`', which is part of the canonical `cc-mode` distribution. As mentioned previously, `cc-mode` always trades accuracy for speed; 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 speed for accuracy. 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.

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

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 ⟨LFD⟩ 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 36):

```
(define-key c-mode-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 36 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 an Emacs 19 feature, controlled by `font-lock-mode`. It is not part of `cc-mode`.

10 Getting the latest cc-mode release

cc-mode is now distributed with both Emacs 19 and XEmacs 19, so you would typically just use the version that comes with your Emacs. Users of older versions of Emacs can get the latest release from this URL:

```
ftp://ftp.python.org/pub/emacs/cc-mode.tar.gz
```

Note that this is a "gzipped" tar file.

If you do not have 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.

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 C, C++, and Objective-C.  We can put these in
  ;; c-mode-map because c++-mode-map and objc-mode-map inherit it
  (define-key c-mode-map "\C-m" 'newline-and-indent)
  )

;; the following only works in Emacs 19
;; Emacs 18ers can use (setq c-mode-common-hook 'my-c-mode-common-hook)
(add-hook 'c-mode-common-hook 'my-c-mode-common-hook)
```

12 Requirements

'`cc-mode.el`' requires '`reporter.el`' for submission of bug reports. '`reporter.el`' is distributed with the latest Emacs 19s. Here is the Emacs Lisp Archive anonymous ftp'ing record for those of you who are using older Emacsen.

```
          GNU Emacs Lisp Code Directory Apropos -- "reporter"
"~/" refers to archive.cis.ohio-state.edu:/pub/gnu/emacs/elisp-archive/

reporter (2.12)       06-Jul-1994
     Barry A. Warsaw, <bwarsaw@cnri.reston.va.us>
     ~/misc/reporter.el.Z
     Customizable bug reporting of lisp programs.
```

13 Limitations and Known Bugs

- Multi-line macros are not handled properly.
- Re-indenting large regions or expressions can be slow.
- Use with Emacs 18 can be slow and annoying. You should seriously consider upgrading to Emacs 19.
- There is still some weird behavior when filling C block comments. My suggestion is to check out add-on fill packages such as `filladapt`, available at the elisp archive.
- Lines following `inline-close` braces which hang "after" do not line up correctly. Hit *`TAB`* to reindent the line.

14 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.

Bug reports are now to be sent to `bug-gnu-emacs@prep.ai.mit.edu` which is mirrored on the Usenet newsgroup `gnu.emacs.bug`. Other questions and suggestions should be mailed to `help-gnu-emacs@prep.ai.mit.edu` which is mirrored on `gnu.emacs.help`.

Note that the `cc-mode` Majordomo mailing lists have been disbanded! With the inclusion of `cc-mode` in both of the latest flavors of Emacs 19, the need for them has ended.

Concept Index

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.

Key Index

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.

Short Contents

Table of Contents