
















Description	Keystroke	Function	Note
<b>Toggle Electric state</b> 	<ul style="list-style-type: none"><li><b>C-c C-1</b></li><li><b>&lt;f12&gt; &lt;f4&gt; e</b></li></ul>	<b>(c-toggle-electric-state</b> &optional ARG)	Toggle the electric indentation feature done with the electric character keys. <ul style="list-style-type: none"><li>Optional numeric ARG, if supplied, turns on electric indentation when positive, turns it off when negative, and just toggles it when zero or left out.</li></ul>
<b>Set indentation style</b> 	<ul style="list-style-type: none"><li><b>C-c .</b></li><li><b>&lt;f12&gt; &lt;f4&gt; s</b></li></ul>	<b>(c-set-style</b> STYLENAME &optional DONT-OVERRIDE)	Set the <u>bracket/indentation style</u> for the current buffer. <ul style="list-style-type: none"><li>Prompts for the name.</li><li>Supports tab completion (so use tab to see the list). Can be one of the <u>values supported by Emacs</u> but you can also add your customized mode with some Emacs Lisp code.</li></ul>
<b>Change indentation width for current buffer</b> 	<b>&lt;f12&gt; &lt;f4&gt; TAB</b>	<b>(pel-cc-set-indent-width</b> &optional NEW-WIDTH)	Interactively change the Indentation with for current buffer to NEW-WIDTH. <ul style="list-style-type: none"><li>Prompt for new value.<ul style="list-style-type: none"><li>Use 0 to restore value specified by configuration (<b>pel-c-indent-width</b>).</li></ul></li><li>👉 This can be used to change indentation several times in a file.</li></ul>
<b>Toggle syntactic indentation</b>	<b>&lt;f12&gt; &lt;f4&gt; i</b>	<b>(c-toggle-syntactic-indentation</b> &optional ARG)	Toggle syntactic indentation. Toggle if no ARG or if ARG is 0. <ul style="list-style-type: none"><li>With positive ARG turn on syntactic indentation, turns it off when negative.</li></ul> <div><ul style="list-style-type: none"><li>When syntactic indentation turned on (the default), the indentation functions and electric keys indent according to syntactic context keys, when applicable.</li><li>When it's turned off, the electric keys don't reindent, the indentation functions indents every new line to the same level as the previous nonempty line, and M-x c-indent-command adjusts the indentation in steps specified by 'c-basic-offset'. The indentation style has no effect in this mode, nor any of the indentation associated variables, e.g. 'c-special-indent-hook'.</li></ul></div>
<b>Toggle Comment Style</b> 	<ul style="list-style-type: none"><li><b>C-c C-k</b></li><li><b>&lt;f12&gt; &lt;f4&gt; M-;</b></li></ul>	<b>(c-toggle-comment-style</b> &optional ARG)	Toggle the comment style between block and line comments. <ul style="list-style-type: none"><li>Optional numeric ARG, if supplied, switches to block comment style when positive, to line comment style when negative, and just toggles it when zero or left out.</li><li>The C++ style <code>//</code> comments (also called line comments) are compatible with C since C-99.</li></ul>
<b>Toggle Hungry Delete mode</b> 	<b>&lt;f12&gt; &lt;f4&gt; DEL</b>	<b>(c-toggle-hungry-state</b> &optional ARG)	Toggle hungry-delete-key feature. Affects <b>&lt;DEL&gt;</b> and <b>C-d</b> keys. <ul style="list-style-type: none"><li>Optional numeric ARG, if supplied, turns on hungry-delete when positive, turns it off when negative, and just toggles it when zero or left out.</li><li>When the hungry-delete-key feature is enabled (indicated by "/h" on the mode line after the mode name) the delete key gobbles all preceding whitespace in one fell swoop.</li></ul>
<b>Toggle text alignment on pel-newline-and-indent-below</b> See also: <ul style="list-style-type: none"><li><a href="#">Align</a></li><li><a href="#">Indentation</a></li></ul> 	<b>&lt;f11&gt; M-RET</b>	<b>(pel-toggle-newline-indent-align)</b>	Toggle variable <i>pel-newline-does-align</i> for the local buffer. This toggles the way function 'pel-newline-and-indent-below' operates. <ul style="list-style-type: none"><li>If <i>pel-newline-does-align</i> is t, it aligns several syntactic element in the current block: the comments, the assignments.</li><li> Identify modes where <i>pel-newline-does-align</i> is automatically activated (set to t) by adding the major mode to the list in the <b>pel-modes-activating-align-on-return</b> user option.</li><li>This affects the behaviour of the following commands:<ul style="list-style-type: none"><li>pel-cc-newline (assigned to <b>RET</b> in CC modes like c-mode, c++-mode and d-mode).</li><li>pel-newline-and-indent-below (assigned the <b>M-RET</b>)</li></ul></li></ul>
<b>Toggle auto-newline insertion mode</b> 	<ul style="list-style-type: none"><li><b>C-c C-a</b></li><li><b>&lt;f12&gt; &lt;f4&gt; M-RET</b></li></ul>	<b>(c-toggle-auto-newline</b> &optional ARG)	Toggle <b>auto-newline</b> feature. <ul style="list-style-type: none"><li>Optional numeric ARG, if supplied, turns on auto-newline when positive, turns it off when negative, and just toggles it when zero or left out.</li><li>Turning on auto-newline automatically enables <i>electric indentation</i>.</li><li>When the auto-newline feature is enabled (indicated by "/la" on the mode line after the mode name) newlines are automatically inserted after special characters such as brace, comma, semi-colon, and colon.</li></ul>
<b>Change RET key behaviour: select return mode.</b> 	<b>&lt;f12&gt; &lt;f4&gt; RET</b>	<b>(pel-cc-change-newline-mode)</b>	Change the way the RET key behaves in the CC modes and display the new mode in the echo area. Changes from one mode to the next and then rotate to the first one. The modes are: <ol style="list-style-type: none"><li>context-newline : the default : uses <b>(c-context-line-break)</b> with the extra ability to repeat its execution with an argument.</li><li>newline-and-indent: uses <b>(newline ARG t)</b> to insert newline and indent.</li><li>just-newline-no-indent: uses <b>(electric-indent-just-newline ARG)</b></li></ol>  Emacs default is to use newline. PEL sets the default to c-context-line-break which provides more functionality for CC modes. A mode change is local to the current buffer and does not affect RET key behaviour in the other buffers using the same mode.  PEL user option <b>pel-initial-c-newline-mode</b> can be set to change the default for c-mode.
<b>Display current Mode settings</b>	<b>&lt;f12&gt; &lt;f4&gt; ?</b>	<b>(pel-cc-mode-info)</b>	Display information about current <b>CC mode</b> derivative for the current c-mode buffer.
	<div>The information includes the following:<ul style="list-style-type: none"><li>CC mode style currently active, along with a list of styles associated with current mode. Change it for the current buffer with <b>C-c .</b> or <b>&lt;f12&gt; &lt;f4&gt; s</b>. The Emacs the <b>c-default-style</b> user option defines associations between major modes and the style to use. PEL provides the <b>pel-c-bracket-style</b> that is used to set the style for c-mode. Use <b>&lt;f12&gt; &lt;f2&gt;</b> from a c-mode buffer to access the customization buffer to change it.</li><li>Return key behaviour:<ul style="list-style-type: none"><li>RET (return key) mode. Change with pel-cc-change-newline-mode (<b>&lt;f12&gt; &lt;f4&gt; RET</b>).</li><li>Whether return performs alignment. Change that with pel-toggle-indent-align (<b>&lt;f11&gt; M-RET</b>).</li></ul></li><li>State of <b>electric C characters</b> (toggle it on/off with c-toggle-electric-state (<b>C-c C-1</b> or <b>&lt;f12&gt; &lt;f4&gt; e</b>):<ul style="list-style-type: none"><li>whether it is active or not, and when active what character(s) exhibit electric behaviour.</li><li>if auto-newline on some characters (',' and some other based on style) is active. Toggle it with <b>C-c C-a</b> or <b>&lt;f12&gt; &lt;f4&gt; M-RET</b>.</li></ul></li><li>The fill column: the column where force line wrap is done when the auto-fill-mode is active. Toggle auto fill mode with <b>&lt;f11&gt; RET</b>.</li><li>Tab width and whether hard tabs are used. These are set by the user options <b>pel-c-tab-width</b> and <b>pel-c-use-tabs</b>.<ul style="list-style-type: none"><li>In c-mode buffer use <b>&lt;f12&gt; &lt;f2&gt;</b> to open the appropriate customization buffer to change them.</li></ul></li><li>👉 Remember that tab width does <b>not</b> identify the indentation. It controls the spacing used in some commands moving point to the next tab stop column. Indentation is controlled separately. See next line.</li><li>Indentation width controlled by <b>c-basic-offset</b> normally set by <b>pel-c-indent-width</b> in PEL and whether syntactic indentation mode is active. Shows how it is set and whether it was override by executing the <b>pel-cc-set-indent-width</b> command for this buffer (use <b>&lt;f12&gt; &lt;f4&gt; TAB</b>) for that command.</li><li>The style currently used for indentation and bracket positioning (they should have the same value). Emacs identifies several built-in styles but you can create your own. The example below shows “bsd” with is another name for the <b>Allman style</b>. You can dynamically change for the current buffer with c-set-style command (<b>C-c .</b> or <b>&lt;f12&gt; &lt;f4&gt; s</b>).<ul style="list-style-type: none"><li>👉 CC Mode styles identify everything, including the number of indentation columns. PEL configures the style from the requested pel-c-bracket-style and then updates the indentation and other settings from the PEL user option requested. This allows you to slightly modify an existing style without having to create a new style name for it.</li></ul></li><li>The comment style. Supports C-style (<code>/* */</code>) and C++-style (<code>//</code>) comments since <b>both are now accepted in C since C99</b>.<ul style="list-style-type: none"><li>This can be changed dynamically for the current buffer with the c-toggle-comment-style command (<b>C-c C-k</b> or <b>&lt;f12&gt; &lt;f4&gt; M-;</b> ). C comment continuation lines can use 1 or 2 star characters: if a second one is used on a comment continuation line the remainder of the comment continuation lines used two stars, otherwise only one is used.</li></ul></li><li>Whether hungry delete is used by <b>DEL</b> and <b>C-d</b>. Toggle this for the current buffer with <b>c-toggle-hungry-state</b> (<b>&lt;f12&gt; &lt;f4&gt; DEL</b>).</li><li>The file search methods and parameters used by <b>pel-open-at-point</b> (see sections below).</li></ul><div><div>--UU-:----F1 c file.c All (1,0) (C/*1a- WK Anzu Fly ^ E1Doc Abv) 10:35am 1.97 -----</div><div>c-mode state: - active style : bsd. c-default-style: (bsd) - RET mode : context-newline - Electric characters : active on: #*/({}{};, - Auto newline : on - fill column : 80, auto-filling: off. - Tab width : 8 Set via: pel-c-tab-width(8) ==&gt; tab-width(8) when c-mode buffer is opened. - Indentation chars : spaces only Set via: pel-c-use-tabs(nil) ==&gt; indent-tabs-mode(nil) when c-mode buffer is opened. - Indent width : 4 Set via: pel-c-indent-width(4) ==&gt; c-basic-offset(4) when c-mode buffer is opened. - Syntactic indent : on - c-indentation-style : bsd - PEL Bracket style : bsd - Comment style : Block comments: /* */ , continued line start with * - Hungry delete : off, but the F11-⌘ and F11-⌘ keys are available.</div></div></div>		
👉 Notice the name of the PEL user-options that set the significant feature controlling Emacs variables in the message 			

Description	Keystroke	Function	Note
C Code Help	There are several Emacs extension packages that can help writing C code.		
Get man help about C code See: <a href="#">⌘ Help/Info</a>	<div>&lt;f12&gt; ? ?</div> <div> <ul style="list-style-type: none"> <li>&lt;f11&gt; ? m</li> <li>⌘-M</li> </ul> </div>	(man MAN-ARGS)	Open a Man page inside an Emacs window. See <a href="#">⌘ Help/Info</a> for more info about man. <ul style="list-style-type: none"> <li>Inside a C buffer, you can use it to request man help about a C function or structure.</li> <li>A large amount of information about C library code is available in man form under the various Unix-like platforms.</li> </ul>
Toggle c-eldoc mode	<f12> ? e	(pel-toggle-c-eldoc-mode)	Toggle c-eldoc mode on/off. <ul style="list-style-type: none"> <li>The c-eldoc mode provides the C prototype information in the echo area for the function at point. It currently appears when typing a new function with its arguments inside the code.</li> </ul>
	<f11> SPC c ? e		
	<div>📦 Requires <a href="#">c-eldoc</a> external package. <a href="#">🔧</a> Activated when pel-use-c-eldoc is set to t. ⚠️ The extra processing required may slow Emacs.</div> <ul style="list-style-type: none"> <li>🔧 This package could be improved into providing the information only on demand but a LSP-based system might be more performant anyway. I am currently looking at this to see if I can improve the performances and the feature set. c-eldoc uses the cpp command to preprocess the buffer content.</li> </ul>		
Electric Keys	The following <b>electric C characters</b> have special meaning when the electrical state is active in a buffer using c-mode. <ul style="list-style-type: none"> <li>Toggle electric behaviour in the current buffer with: with c-toggle-electric-state (<b>C-c C-1</b> or <b>&lt;f12&gt; &lt;f4&gt; e</b>).</li> </ul>		
#	#	(c-electric-pound ARG)	Insert a "#". <ul style="list-style-type: none"> <li>If 'c-electric-flag' is set, handle it specially according to the variable 'c-electric-pound-behavior', which can only be nil or 'alignleft'. If a numeric ARG is supplied, or if point is inside a literal or a macro, nothing special happens.</li> </ul>
()	( )	(c-electric-paren ARG)	Insert a parenthesis. <ul style="list-style-type: none"> <li>If 'c-syntactic-indentation' and 'c-electric-flag' are both non-nil, the line is reindented unless a numeric ARG is supplied, or the parenthesis is inserted inside a literal.</li> <li>Whitespace between a function name and the parenthesis may get added or removed; see the variable 'c-cleanup-list'.</li> <li>Also, if 'c-electric-flag' and 'c-auto-newline' are both non-nil, some newline cleanups are done if appropriate; see the variable 'c-cleanup-list'.</li> </ul>
{ }	{ }	(c-electric-brace ARG)	Insert a brace. <ul style="list-style-type: none"> <li>If 'c-electric-flag' is non-nil, the brace is not inside a literal and a numeric ARG hasn't been supplied, the command performs several electric actions: <ol style="list-style-type: none"> <li>If the auto-newline feature is turned on (indicated by "/la" on the mode line) newlines are inserted before and after the brace as directed by the settings in 'c-hanging-braces-alist'.</li> <li>Any auto-newlines are indented. The original line is also reindented unless 'c-syntactic-indentation' is nil.</li> <li>If auto-newline is turned on, various newline cleanups based on the settings of 'c-cleanup-list' are done.</li> </ol> </li> </ul>
:	:	(c-electric-colon ARG)	Insert a colon. <ul style="list-style-type: none"> <li>If 'c-electric-flag' is non-nil, the colon is not inside a literal and a numeric ARG hasn't been supplied, the command performs several electric actions: <ol style="list-style-type: none"> <li>If the auto-newline feature is turned on (indicated by "/la" on the mode line) newlines are inserted before and after the colon based on the settings in 'c-hanging-colons-alist'.</li> <li>Any auto-newlines are indented. The original line is also reindented unless 'c-syntactic-indentation' is nil.</li> <li>If auto-newline is turned on, whitespace between two colons will be "cleaned up" leaving a scope operator, if this action is set in 'c-cleanup-list'.</li> </ol> </li> </ul>
;,	;, ,	(c-electric-semi&comma ARG)	Insert a comma or semicolon. <ul style="list-style-type: none"> <li>If 'c-electric-flag' is non-nil, point isn't inside a literal and a numeric ARG hasn't been supplied, the command performs several electric actions: <ol style="list-style-type: none"> <li>When the auto-newline feature is turned on (indicated by "/la" on the mode line) a newline might be inserted. See the variable 'c-hanging-semi&amp;comma-criteria' for how newline insertion is determined.</li> <li>Any auto-newlines are indented. The original line is also reindented unless 'c-syntactic-indentation' is nil.</li> <li>If auto-newline is turned on, a comma following a brace list or a semicolon following a defun might be cleaned up, depending on the settings of 'c-cleanup-list'.</li> </ol> </li> </ul>
Electric pairs	It is also possible to control the insertion of character pairs by activating the <b>electric-pair-mode</b> in the buffer. <ul style="list-style-type: none"> <li>Type the first of a pair to insert this one and its matching character for (), [], {}, "" and "".</li> <li>When the electric-pair-mode is active in a buffer the mode-line lighter set by the pel-electric-pair-lighter is shown. This defaults to <math>\mathcal{E}(1)</math></li> </ul>		
Toggle electric-pair-mode in current buffer   Lighter:= $\mathcal{E}(1)$	<f11> M-e	(electric-pair-local-mode &optional ARG)	Toggle automatic parens pairing (Electric Pair mode) and org-mode special pair electric keys only in this buffer. With this typing ( inserts the matching ). Same for other pairs. <ul style="list-style-type: none"> <li>With a prefix argument ARG, enable Electric Pair mode if ARG is positive, and disable it otherwise.</li> <li>Electric Pair mode is a global minor mode. When enabled, typing an open parenthesis automatically inserts the corresponding closing parenthesis, and vice versa. (Likewise for brackets, etc.). If the region is active, the parentheses (brackets, etc.) are inserted around the region instead.</li> </ul>
Insert New Line(s)	The behaviour of the RET key depends on whether the CC Mode electric mode is active or not. When it is not active it simply inserts a new line. When it is active the point also moves to the proper indentation according to the syntactic context. The following commands can also be used. <ul style="list-style-type: none"> <li>With PEL the default behaviour can be selected by customization and modified dynamically for the current buffer with the <b>pel-cc-change-newline-mode</b> command (bound to <b>&lt;F12&gt; M-RET</b>) see the CC-Mode behaviour control section above.</li> <li>The pel-cc-newline command also aligns comments and assignment in the code block if the <a href="#">pel-modes-activating-align-on-return</a> user option list includes the current major mode. The state for the current buffer can also be modified by the <b>pel-cc-change-newline-mode</b> command (<b>&lt;f11&gt; M-RET</b>).</li> </ul>		
Insert a new line and operate according to the currently active selected return mode.  With PEL, modify behaviour with <b>&lt;F12&gt; M-RET</b> .  See also: <ul style="list-style-type: none"> <li><a href="#">⌘ Filling/Justification</a></li> </ul>	RET	(pel-cc-newline &optional N)	Insert a newline and perhaps align. With argument N repeat N times. <ul style="list-style-type: none"> <li>For newline insertion, operate according to the value of the variable 'pel-cc-newline-mode' which selects one of 3 commands (see the full description in the 3 row below): <ul style="list-style-type: none"> <li>c-context-line-break (PEL default for RET)</li> <li>newline (Emacs default for RET)</li> <li>electric-indent-just-newline</li> </ul> </li> <li>If 'pel-newline-does-align' is t, then perform the text alignment done by the function 'align'.</li> </ul>
	Use : (c-context-line-break) : Do a line break suitable to the context. <ul style="list-style-type: none"> <li>When point is outside a comment or macro, insert a newline and indent according to the syntactic context, unless 'c-syntactic-indentation' is nil, in which case the new line is indented as the previous non-empty line instead.</li> <li>When point is inside the content of a preprocessor directive, a line continuation backslash is inserted before the line break and aligned appropriately. The end of the cpp directive doesn't count as inside it.</li> <li>When point is inside a comment, continue it with the appropriate comment prefix (see the 'c-comment-prefix-regexp' and 'c-block-comment-prefix' variables for details). The end of a C++-style line comment doesn't count as inside it.</li> <li>When point is inside a string, only insert a backslash when it is also inside a preprocessor directive.</li> </ul>		
	Use: (newline &optional ARG INTERACTIVE): Insert a newline, and move to left margin of the new line if it's blank. <ul style="list-style-type: none"> <li>With ARG, insert that many newlines.</li> <li>If option 'use-hard-newlines' is non-nil, the newline is marked with the text-property 'hard'.</li> <li>If 'electric-indent-mode' is enabled, this indents the final new line that it adds, and reindents the preceding line. <ul style="list-style-type: none"> <li>To just insert a newline, use M-x electric-indent-just-newline.</li> </ul> </li> </ul> Calls 'auto-fill-function' if the current column number is greater than the value of 'fill-column' and ARG is nil.		
	Use: (electric-indent-just-newline ARG): Insert just a newline, without any auto-indentation. <ul style="list-style-type: none"> <li>With ARG, insert that many newlines.</li> </ul>		
Insert an indented line below unbroken current line See also: <a href="#">⌘ Indentation</a>	<ul style="list-style-type: none"> <li>M-RET</li> <li>&lt;f11&gt; &lt;tab&gt; RET</li> </ul>	(pel-newline-and-indent-below)	Insert an indented line just below current line regardless of the position of point and move point to the beginning of the next line. Does not break current line. For example if point is at the beginning, middle or end of the line it just insert a new line below the current one at the proper indentation. <ul style="list-style-type: none"> <li>If <i>pel-newline-does-align</i> is t, it aligns several syntactic element in the current block: the comments, the assignments. <ul style="list-style-type: none"> <li>You can toggle this on/off with <b>&lt;f11&gt; M-RET</b>.</li> </ul> </li> <li>🔍 Identify modes where <i>pel-newline-does-align</i> is automatically activated (set to t) by adding the c-mode to the list in the <b>pel-modes-activating-align-on-return</b> user option.</li> </ul>





Description	Keystroke	Function	Note
<a href="#">Hungry Deletion of Whitespace</a>	<p>The CC mode provides two commands that can perform “hungry whitespace deletion” that can also be used in every mode.</p> <ul style="list-style-type: none"> <li>👉 PEL provides the convenient keys with the <b>&lt;f11&gt;</b> prefix keys for those 2 commands, available in <b>all</b> modes.</li> <li>In modes compatible with the CC Mode (e.g. for C, C++, D, Java, Pike, etc..) it is also possible to activate the Hungry Delete Mode to modify the behaviour of the simple <b>&lt;DEL&gt;</b> and <b>C-d</b>, to perform hungry deletions. That’s not currently supported in other modes. <ul style="list-style-type: none"> <li>When the Hungry Delete Mode is on, the mode-line displays a ‘h’ to the right of the ‘/!’ indication of electric mode.</li> </ul> </li> <li>The Hungry Mode also activates the key prefixes below that start with <b>C-c</b>. They are listed but remember they are only available once the Hungry state mode is activated (and that can only be done in modes that are CC Mode compatible).</li> <li>In modes derived from CC Mode you can also activate the hungry state to make standard delete commands delete hungrily, but that does not work for other modes. PEL provides the <b>&lt;f12&gt; M-DEL</b> key for those modes (like C).</li> </ul> <p>• Toggle hurry deletion mode of the <b>DEL</b> and <b>C-d</b> key for the current buffer with <b>c-toggle-hungry-state (&lt;f12&gt; M-DEL)</b>.</p>		
Delete preceding char or all preceding whitespace.  See also: <ul style="list-style-type: none"> <li>🔗 <a href="#">Cut &amp; Paste</a></li> </ul>	<ul style="list-style-type: none"> <li><b>C-c DEL</b></li> <li><b>C-c</b> </li> <li><b>C-c C-</b></li> <li><b>C-c &lt;C-backspace&gt;</b></li> <li><b>C-c C-DEL</b></li> </ul>	(c-hungry-delete-backwards)	Delete the preceding character or all preceding whitespace back to the previous non-whitespace character. <p>🖥️➡️ In terminal mode, even though <b>C-</b>, <b>&lt;C-backspace&gt;</b> and <b>C-DEL</b> are not available, they are mapped to the non-control key so attempting to type them end up invoking the command anyway because the first key bindings are recognized.</p> <p>👉 With PEL, the <b>&lt;f11&gt;  </b> binding is always available, in all modes.</p> <p>The other keys are only available in modes derived from the CC Mode. This prevents conflicts with other modes that may use the popular C-c bindings.</p>
Delete next char or all following whitespace.  See also: <ul style="list-style-type: none"> <li>🔗 <a href="#">Cut &amp; Paste</a></li> </ul>	<ul style="list-style-type: none"> <li><b>C-c C-d</b></li> <li><b>C-c</b> </li> <li><b>C-c C-</b></li> <li><b>C-c &lt;C-delete&gt;</b></li> </ul>	(c-hungry-delete-forward)	Delete the following character or all following whitespace up to the next non-whitespace character. <p>🖥️➡️ In terminal mode, even though <b>C-</b> and <b>&lt;C-delete&gt;</b> are not available, they are mapped to the non-control key so attempting to type them end up invoking the command anyway because the first key bindings are recognized.</p> <p>👉 With PEL, the <b>&lt;f11&gt; </b> binding is always available, in all modes. The other keys are only available in modes derived from the CC Mode. This prevents conflicts with other modes that may use the popular C-c bindings.</p>
<a href="#">Indentation</a>	All syntactic indentation control for C is controlled by the CC-Mode state, the style and whether electric mode for some characters is active. See CC Mode behaviour control section above. You can also explicitly request indentation using the commands below. <ul style="list-style-type: none"> <li>The first set of commands perform syntactic indentations s controlled by the CC Mode.</li> <li>Rigid indentation commands are also available and listed at the end of this list. They are also listed in the 🔗 <a href="#">Indentation</a> table.</li> </ul>		
Indent current line or region  See also: <ul style="list-style-type: none"> <li>🔗 <a href="#">Indentation</a></li> </ul>	<b>&lt;tab&gt;</b>	(c-indent-line-or-region &optional ARG REGION)	Indent active region, current line, or block starting on this line. <p>• Behaviour depends on syntactic-indentation mode (enabled by default but can be toggled on/off with the <b>&lt;f12&gt; M-i</b> key):</p> <ul style="list-style-type: none"> <li>With syntactic-indentation on (the default): <ul style="list-style-type: none"> <li>In Transient Mark mode, when the region is active, reindent the region.</li> <li>Otherwise, with a prefix argument, rigidly reindent the expression starting on the current line.</li> <li>Otherwise reindent just the current line.</li> </ul> </li> <li>👉 This might seem strange for new Emacs users, but it ends up being very useful. You can type <b>&lt;tab&gt;</b> anywhere in the line to adjust the indentation of the current line or everything in the marked area if a block is marked.</li> <li>With syntactic-indentation off: <ul style="list-style-type: none"> <li><b>&lt;tab&gt;</b> always indent current line by one level</li> <li><b>C-u - &lt;tab&gt;</b> or <b>M-- &lt;tab&gt;</b> always un-indent current line by one level.</li> <li>Indenting marked region is done without syntax knowledge and at the same level as previous line.</li> </ul> </li> </ul> <p>👉 If you want to indent rigidly you can use:</p> <ul style="list-style-type: none"> <li><b>pel-indent-rigidly</b>, bound to <b>C-x &lt;tab&gt;</b> and to <b>&lt;f11&gt; &lt;tab&gt;&lt;tab&gt;</b> to indent the line or region rigidly.</li> <li><b>tab-to-tab-stop</b>, bound to <b>M-i</b> to insert spaces to the next tab stop column.</li> </ul>
Indent lines of list after point See also: <ul style="list-style-type: none"> <li>🔗 <a href="#">Indentation</a></li> </ul>	<b>C-M-q</b>	(indent-pp-sexp &optional ARG)	Indent each line of the list starting just after point, or pretty-print it. <ul style="list-style-type: none"> <li>A prefix argument (<b>C-u</b>) specifies pretty-printing. Pretty-printing essentially uses more lines as it places the beginning of each list on a new line.</li> </ul>
Indent current function or class	<b>C-c C-q</b>	(c-indent-defun)	Indent the content of the current top-level function or class. Leaves point unchanged.
Indent a region	<b>C-M-\</b>	(indent-region START END &optional COLUMN)	Indent each nonblank line in the region. <ul style="list-style-type: none"> <li>A numeric prefix argument specifies a column: indent each line to that column.</li> <li>With no prefix argument, the command chooses one of these methods and indents all the lines with it: <ol style="list-style-type: none"> <li>If ‘fill-prefix’ is non-nil, insert ‘fill-prefix’ at the beginning of each line in the region that does not already begin with it.</li> <li>If ‘indent-region-function’ is non-nil, call that function to indent the region.</li> <li>Indent each line via ‘indent-according-to-mode’.</li> </ol> </li> </ul> <p>👉 When a region is marked you can also use the simple <b>&lt;tab&gt;</b> to do the same when syntactic-indentation is active.</p>
<a href="#">Non Syntactic Indentation</a>	Emacs provides the following command to indent without regards to semantics. More information on indentation is available in the 🔗 <a href="#">Indentation</a> table. <p>🌱 For most editing scenarios, it’s best to set <b>pel-c-tab-width</b> and <b>pel-c-indent-width</b> to the same value: the first 2 commands use the value of pel-c-tab-width while the other 2 use pel-c-indent-width.</p>		
Insert spaces or tabs to next defined tab-stop column See also: <ul style="list-style-type: none"> <li>🔗 <a href="#">Indentation</a></li> </ul>	<b>M-i</b>	(tab-to-tab-stop)	Insert spaces or tabs to next defined tab-stop column. <ul style="list-style-type: none"> <li>The exact location of the next tab stop is identified by the value of the <b>tab-stop-list</b> and <b>tab-width</b> for the current buffer.</li> <li>With PEL, the tab-stop interval is controlled by the value of <b>pel-c-tab-width</b>. <ul style="list-style-type: none"> <li>PEL sets <b>tab-width</b> to the value of pel-c-tab-width for each c-mode buffer.</li> </ul> </li> </ul>
Indent/Unindent rigidly  See also: <ul style="list-style-type: none"> <li>🔗 <a href="#">Indentation</a></li> <li>🔗 <a href="#">Key-Chords</a></li> </ul>	<ul style="list-style-type: none"> <li><b>C-x &lt;tab&gt;</b></li> <li><b>&lt;f11&gt; &lt;tab&gt; &lt;tab&gt;</b></li> <li><b>&lt;tab&gt;q</b></li> </ul>	(pel-indent-rigidly &optional N)	Indent rigidly the marked region or current line N times tab-width columns. <ul style="list-style-type: none"> <li>If a region is marked, it uses “indent-rigidly” and provides the same prompts to control indentation changes.</li> <li>If no region is marked, it operates on current line(s) identified by the numeric argument N (or if not specified N=1): <ul style="list-style-type: none"> <li>N = [-1, 0, 1] : operate on current line</li> <li>N &gt; 1 : operate on the current line and N-1 lines below.</li> <li>N &lt; -1 : operate on the current line and (abs N) -1 lines above.</li> </ul> </li> </ul>
<p>✂ PEL rebinds this key, but it extends the functionality: pel-indent-rigidly uses the original indent-rigidly.</p> <p><b>indent-rigidly</b> Indent all lines starting in the region.</p> <ul style="list-style-type: none"> <li>If called interactively with no prefix argument, activate a transient mode in which the indentation can be adjusted interactively by typing <b>&lt;left&gt;</b>, <b>&lt;right&gt;</b>, <b>&lt;S-left&gt;</b>, or <b>&lt;S-right&gt;</b>.</li> </ul> <p>Both of these commands activate a transient mode where Emacs prompts for extra keys to control how to indent. Indenting and un-indenting is possible. The capabilities are controlled by the variable <i>indent-rigidly-map</i> with by default provides:</p> <ul style="list-style-type: none"> <li><b>S-&lt;right&gt;</b> indent-rigidly-right-to-tab-stop</li> <li><b>S-&lt;left&gt;</b> indent-rigidly-left-to-tab-stop</li> <li><b>&lt;right&gt;</b> indent-rigidly-right</li> <li><b>&lt;left&gt;</b> indent-rigidly-left</li> </ul> <p>Typing any other key deactivates the transient mode.</p> <ul style="list-style-type: none"> <li>The <b>S-&lt;right&gt;</b> and <b>S-&lt;left&gt;</b> keys indent/de-indent to the next tab-stop position, which is controlled by the <b>tab-width</b> user option.</li> <li>With PEL, the tab-stop interval is controlled by the value of <b>pel-c-tab-width</b>. <ul style="list-style-type: none"> <li>PEL sets <b>tab-width</b> to the value of pel-c-tab-width for each c-mode buffer.</li> </ul> </li> </ul> <p>⚠ If you use the cua-mode: the cua-mode uses <b>C-x</b>, to invoke this command when cua-mode is active, type it really fast or type <b>C-x C-x &lt;tab&gt;</b> (or use the PEL binding <b>&lt;f11&gt; &lt;tab&gt; &lt;tab&gt;</b>).</p>			






Description	Keystroke	Function	Note
<p><b>Tempo skeletons for C</b></p> <p>See also:</p> <ul style="list-style-type: none"> <li>🔗 <b>Inserting Text</b> for more info and information about tempo skeleton and yasnippet template-based text insertion</li> </ul>	<p>PEL provides support for flexible text template insertion through the Emacs built-in <b>tempo skeleton</b> mechanism.</p> <ul style="list-style-type: none"> <li>PEL creates key bindings to invoke the skeletons in the supported major modes, using the same key prefix sequence for each mode: <b>&lt;f12&gt; &lt;f12&gt;</b>, with the same key bindings for equivalent concepts (such as file header block) as much as possible.</li> <li>👁️ Several aspects of the PEL Emacs Lisp Source Code Style is controlled by the user options inside the <b>pel-c-code-style</b> group. This group can be edited with <b>&lt;f12&gt; &lt;f12&gt; &lt;f2&gt;</b> from a C mode buffer and include the following options: <ul style="list-style-type: none"> <li><b>pel-c-skel-module-header-block-style</b> : allows selecting a user-define module-header comment block.</li> <li><b>pel-c-skel-comment-with-2-star</b> : controls the format of C-style continuation comments.</li> <li><b>pel-c-skel-insert-file-timestamp</b> : set whether an automatically updated timestamp is inserted in the file header block.</li> <li><b>pel-c-skel-use-separators</b> : set whether blocks use horizontal separator lines.</li> <li><b>pel-c-skel-doc-markup</b> 📄 : identifies the documentation markup supported by the templates. Currently ‘none’ and ‘Doxygen’ are available.</li> <li><b>pel-c-skel-cfile-section-titles</b> : identifies documentation section titles inserted in code files.</li> <li><b>pel-c-skel-hfile-section-titles</b> : identifies documentation section titles inserted in header files. A section titled “.” split sections placed before and after the include guard. If not present all sections are placed after the include guard.</li> <li><b>pel-c-skel-insert-function-sections</b> : set whether C function templates are inserted in the function description comment.</li> <li><b>pel-c-skel-function-section-titles</b> : identifies the title of the C function templates sections inserted when pel-c-skel-insert-function-sections is <b>t</b>.</li> <li><b>pel-c-skel-function-define-style</b> : select the C function comment block style. Several styles are provided: <ul style="list-style-type: none"> <li>- no special comment</li> <li>- a basic, free-format style to describe the function above its code.</li> <li>- a Man-page style comment block with the sections identified by pel-c-skel-function-section-titles</li> <li>- a user defined tempo skeleton loaded from a user specified file name. See the <b>source code example</b>.</li> </ul> </li> <li><b>pel-c-skel-function-name-on-first-column</b> : identifies whether return type is located on the same line as function name or just above.</li> <li><b>pel-c-skel-with-license</b> : specify whether copy right and code license is specified. An option provide ability to insert open source software license text controlled by 📄 <b>lice</b>.</li> <li><b>pel-c-use-include-guards</b> : specify which type of include guard is inserted in header files. The available choices are: <ul style="list-style-type: none"> <li>- no include guard</li> <li>- use #pragma once statement</li> <li>- use classic #ifdef/#define/#endif block using symbol created from file name</li> <li>- use a #ifdef/#define/#endif block using symbol created from file name and UUID for its uniqueness.</li> </ul> </li> </ul> <p>👉 Emacs user options by default take effect globally. But by using file and directory variables ( see 🔗 <b>File/Directory Variables</b>) they can also be used to take effect on a single file or all files inside a directory tree. So by default, the user options that control the PEL tempo template take effect globally. If you want to change the behaviour for only one file, write the user option control block at the end of that file. If you want to control the behaviour of the PEL tempo templates for all files inside a directory tree create a .dir-locals file and store the values of the relevant options variables inside that file. This allows you to control the user options affecting the format of the tempo templates precisely and does not affect what you actually type.</p> <ul style="list-style-type: none"> <li>Once a skeleton was just entered (or later by activating the pel-tempo-mode) you can move to the next or previous point of interest (so called <i>tempo-marks</i>) with the standard tempo-mode keys <b>C-c M-f</b> and <b>C-c M-b</b> or some other keys like <b>C-c .</b> and <b>C-c ,</b>.</li> </ul> </li></ul>		
🔗 <b>Customize</b> PEL C Skeletons layout	<b>&lt;f12&gt; &lt;f12&gt; &lt;f2&gt;</b>	( <b>pel-customize-pel</b> &optional OTHER-WINDOW)	<p>Customize PEL C skeleton layout.</p> <ul style="list-style-type: none"> <li>If OTHER-WINDOW is non-nil (use <b>C-u</b>), display in another window.</li> </ul>
Insert a file header	<b>&lt;f12&gt; &lt;f12&gt; h</b>	( <b>pel-elisp-file-header</b> )	<p>Insert a file description block. Distinguish between code files and header files.</p> <ul style="list-style-type: none"> <li>Prompts for the file purpose.</li> <li>For header files, include guard is inserted if requested by customization.</li> <li>The layout of the entered text is controlled by user options. It is possible to create a user-specified skeleton this command will used instead of the one provided by PEL.</li> <li>See examples of generated outputs located in <b>example/templates/c/files</b> repo directory.</li> <li>Access the customization buffer by typing: <b>&lt;f12&gt; &lt;f12&gt; &lt;f2&gt;</b></li> </ul>
Insert #define	<b>&lt;f12&gt; &lt;f12&gt; d</b>	( <b>pel-c-define</b> )	<p>Insert a C pre-processor <b>#define</b> statement.</p> <ul style="list-style-type: none"> <li>If there is text between the beginning of the line and point, insert the statement on the next line, otherwise insert it on the current line, even if there is text after point (to allow inserting it before the name of the symbol to define).</li> </ul>
Insert #include <.h>	<b>&lt;f12&gt; &lt;f12&gt; i</b>	( <b>pel-c-include-lib</b> )	<p>Insert a C pre-processor <b>#include &lt;&gt;</b> statement to include a library file.</p> <ul style="list-style-type: none"> <li>If there is text between the beginning of the line and point, insert the statement on the next line, otherwise insert it on the current line.</li> <li>If there is text after point, insert a new line to place that text on the next line.</li> <li>The .h extension is written between the angle brackets and point left right before the period. The next tempo mark is placed at the end of the line (so <b>C-c .</b> move point there).</li> </ul>
Insert #include “.h”	<b>&lt;f12&gt; &lt;f12&gt; I</b>	( <b>pel-c-include-local</b> )	<p>Insert a C pre-processor <b>#include “”</b> statement to include a local file.</p> <ul style="list-style-type: none"> <li>If there is text between the beginning of the line and point, insert the statement on the next line, otherwise insert it on the current line.</li> <li>If there is text after point, insert a new line to place that text on the next line.</li> <li>The .h extension is written between the angle brackets and point left right before the period. The next tempo mark is placed at the end of the line (so <b>C-c .</b> move point there).</li> </ul>
Insert a function definition with comment block	<b>&lt;f12&gt; &lt;f12&gt; f</b>	( <b>pel-c-function</b> )	<p>Insert a C function definition code and comment template.</p> <ul style="list-style-type: none"> <li>The command prompts for the function name and its purpose. <ul style="list-style-type: none"> <li>You can hit return both prompts to specify no text; in that case a tempo skeleton marker is left at the location where the text must be inserted and point is left at the first one.</li> <li>If you enter a function name, it must be a valid C function name (as far as the syntax is concerned). However leading and trailing whitespace is accepted and trimmed and dash characters (“-”) are automatically replaced by underscores (“_”) for convenience.</li> <li>If an invalid name is specified it is erased and you are prompted again. Use <b>M-p</b> to bring the old value back.</li> <li>Prompts for function and purpose maintain separate histories. Use <b>M-p</b> and <b>M-n</b> to navigate in the histories at the prompt. You can also use the <b>&lt;up&gt;</b> and <b>&lt;down&gt;</b> keys.</li> </ul> </li> <li>The style of the code inserted is controlled by the user options inside the pel-c-code-style group and the various C style element controls of the CC-mode.</li> <li>Use <b>C-g</b> to cancel at any prompt. See <b>some examples in the PEL manual</b>.</li> </ul>
Toggle pel-tempo-mode	<b>&lt;f12&gt; &lt;f12&gt; SPC</b>	( <b>pel-tempo-mode</b> &optional ARG)	<p>Toggle PEL tempo mode on/off.</p> <p>PEL tempo mode activates <b>C-c .</b> and <b>C-c ,</b> as well as to <b>C-c C-.</b> and <b>C-c C-,</b> key bindings to navigate across tempo mark hot-spots. When pel-tempo-mode is active the pel-tempo-mode lighter (⚡) is shown on the status bar. The second set are only available when Emacs runs in graphics mode.</p> <p>👉 When a skeleton is inserted via the execution of one of the pel-rst-... commands, the pel-tempo-mode is automatically activated.</p>
Jump to next tempo mark	<ul style="list-style-type: none"> <li><b>C-c M-f</b></li> <li><b>C-c .</b></li> <li><b>C-c C-.</b></li> </ul>	( <b>tempo-forward-mark</b> )	<p>Jump to the next mark in ‘tempo-back-mark-list’: the location where code must be updated inside the inserted skeleton.</p> <ul style="list-style-type: none"> <li>These key key bindings are only available when pel-tempo-mode is active.</li> </ul>
Jump to previous tempo mark	<ul style="list-style-type: none"> <li><b>C-c M-b</b></li> <li><b>C-c ,</b></li> <li><b>C-c C-,</b></li> </ul>	( <b>tempo-backward-mark</b> )	<p>Jump to the previous mark in ‘tempo-back-mark-list’: the location where code must be updated inside the inserted skeleton.</p> <ul style="list-style-type: none"> <li>These key binding are only available when pel-tempo-mode is active.</li> </ul>

Description	Keystroke	Function	Note
Tempo Template Tag Insertion	<f12> <f12> <f12>	(tempo-complete-tag &optional SILENT)	<p>Look for a tag and expand it.</p> <p>👉 Instead of using the &lt;f12&gt; &lt;f12&gt; key bindings above, you can type the template name (shown in the title column like “if”, “case”, etc) completely or partially and then hit &lt;f12&gt; &lt;f12&gt; &lt;f12&gt;. A completion buffer opens up if the template name is incomplete (or empty in which case the buffer lists <b>all</b> available template names). Select the template name and hit RET. Emacs expands the template.</p> <ul style="list-style-type: none"> <li>All the tags in the tag lists in ‘tempo-local-tags’ (this includes ‘tempo-tags’) are searched for a match for the text before the point. The way the string to match for is determined can be altered with the variable ‘tempo-match-finder’. If ‘tempo-match-finder’ returns nil, then the results are the same as no match at all.</li> <li>If a single match is found, the corresponding template is expanded in place of the matching string. If a partial completion or no match at all is found, and SILENT is non-nil, the function will give a signal. If a partial completion is found and ‘tempo-show-completion-buffer’ is non-nil, a buffer containing possible completions is displayed.</li> </ul> <p>🔴 Since only one template is available in emacs-lisp-mode, the usefulness of this command is limited here.</p>
Inserting code	Extra text insertion can be done with the following commands. 👉 See also above: <f12> M-e activates electric pair: typing ( inserts the matching )		
Insert Parentheses	M- (	(insert-parentheses &optional ARG)	<p>For C: insert a parenthesis pair ‘()’, leaving point after open-paren.</p> <ul style="list-style-type: none"> <li>A positive ARG encloses the following ARG sexps in parentheses if they are balanced.</li> <li>A negative ARG encloses the preceding ARG sexps instead.</li> <li>No argument is equivalent to zero: just insert ‘()’ and leave point between.</li> <li>PEL makes <b>parens-require-spaces</b> buffer local and set it to nil in C mode buffers, allowing the use of this command to insert the argument parentheses following a function (and without placing a space between the function name and the opening parenthesis.</li> <li>If region is active, insert enclosing characters at region boundaries.</li> <li>This command assumes point is not in a string or comment.</li> </ul>
Marking	Emacs provides the following command to quickly mark the whole content of the current function. More mark commands exists, see the <a href="#">🔗 Marking</a> table.		
Mark the complete function body  See also: <a href="#">🔗 Marking</a>	C-M-h	(c-mark-function)	<p>Mark complete function.</p> <ul style="list-style-type: none"> <li>Put mark at end of the current top-level declaration or macro, point at beginning.</li> <li>If point is not inside any then the closest following one is chosen. Each successive call of this command extends the marked region by one function.</li> <li>A mark is left where the command started, unless the region is already active (in Transient Mark mode).</li> <li>As opposed to C-M-a and C-M-e, this function does not require the declaration to contain a brace block.</li> </ul>
Getting Syntactic Information	Use the following commands to extract syntactic information from the source code.		
Display name of current function	<ul style="list-style-type: none"> <li>C-c C-z</li> <li>&lt;f12&gt; f</li> <li>&lt;M-f12&gt; f</li> </ul>	(c-display-defun-name &optional ARG)	<p>Display the name of the current CC mode defun and the position in it.</p> <ul style="list-style-type: none"> <li>With a prefix arg, push the name onto the kill ring too.</li> </ul>
Search Support	In C mode, the superword mode can be useful since <a href="#">snake_case</a> is often used. Using superword-mode helps searching. PEL activates the superword mode by default in C mode. To change this use the <f11> t <f2> to access the customize buffer.		
Toggle superword-mode See also: • <a href="#">🔗 Text Modes</a> • <a href="#">🔗 Search/Replace</a>	<ul style="list-style-type: none"> <li>&lt;f11&gt; t m p</li> <li>&lt;f12&gt; M-p</li> </ul>	(superword-mode &optional ARG)	<p>Toggle superword-mode: a minor mode that treats <a href="#">snake_case</a> as one word. In C ‘_’ are treated as part of words.</p> <ul style="list-style-type: none"> <li>With a prefix argument ARG, enable superword mode if ARG is positive, and disable it otherwise.</li> </ul>
Highlighting blocks	The following commands can be used to activate or toggle useful modes to highlight blocks of (), [], and []. • <a href="#">show-paren-mode</a> , which highlights the parens that matches the one before or after point. • <a href="#">rainbow-delimiters</a> mode, where matching nested parens are highlighted with the same colour.		
Toggle show-paren mode on/off  See also: <a href="#">🔗 Highlight</a>	<ul style="list-style-type: none"> <li>&lt;f12&gt; M-9</li> <li>&lt;M-f12&gt; M-9</li> </ul>	(show-paren-mode &optional ARG)	<p>Toggle visualization of matching parens (Show Paren mode).</p> <ul style="list-style-type: none"> <li>With prefix argument ARG, enable Show Paren mode if ARG is positive, disable it otherwise.</li> <li>Show Paren mode is a global minor mode. When enabled, any matching parenthesis is highlighted in ‘show-paren-style’ after ‘show-paren-delay’ seconds of Emacs idle time.</li> </ul>
Enable/Disable coloured highlight of nested blocks (),[],[] See also: <a href="#">🔗 Highlight</a>	<ul style="list-style-type: none"> <li>&lt;f12&gt; M-r</li> <li>&lt;M-f12&gt; M-r</li> </ul>		
	<ul style="list-style-type: none"> <li>&lt;f11&gt; h R</li> </ul>	(rainbow-delimiters-mode &optional ARG)	<p>Highlight nested parentheses, brackets, and braces with colours according to their depth.</p> <ul style="list-style-type: none"> <li>Customize the depth and colours with <b>M-x customize-group rainbow-delimiters</b></li> </ul> <p>📦 Requires: <a href="#">rainbow-delimiters.el</a></p> <p>🔗 PEL activates this when the <b>pel-use-rainbow-delimiters</b> user option is set to <b>t</b>.</p>
Navigation in C	This current list below describe the specialized commands only. See the others inside <a href="#">🔗 Navigation</a>		
• By definitions	Move to the definition of function or type at point. See <a href="#">🔗 Xref</a> for more information to activate the various engines that support cross referencing for C code.		
Find definition of identifier at point  See also: <a href="#">🔗 Xref</a>	M- .	(xref-find-definitions IDENTIFIER)	<p>Grab symbol at point and move cursor to its definition.</p> <ul style="list-style-type: none"> <li>If there are more than one match, prompt in the “xref” buffer.</li> <li>To search for a symbol entered manually, type C-u M- .</li> <li>With dumb-jump this performs a search using ag, ripgrep or git grep if available.</li> </ul>
Go back to where M-. was last issued	M- ,	(xref-pop-marker-stack)	<ul style="list-style-type: none"> <li>Pop back to where M-. was last invoked.</li> <li>Marker depth is controlled by the <b>xref-marker-ring-length</b> user option.</li> </ul>
By C pre-processor	Move across <a href="#">C preprocessor conditional inclusion statements</a> #if #ifdef #ifndef   #else #elif   #endif 🚧 Does not yet support C++23 #elifdef and #elifndef		
Move point forward to matching #endif • or matching #else   #elif	<f6> <right>	(pel-c-preproc-forward-conditional &optional TO-ELSE)	<p>Move point forward to matching #endif</p> <ul style="list-style-type: none"> <li>If point on a #if#ifdef#ifndef statement moves to the matching endif</li> <li>With C-u or numerical arg: move forward to matching #else#elif</li> <li>On success, push the original position on the mark ring and return the new position.</li> <li>On error, issue user error on mismatch. Shift marking is available with <b>C-M-&lt;right&gt;</b></li> </ul>
Move point backward to matching #if  #ifdef   #ifndef • or matching #else   #elif	<f6> <left>	(pel-c-preproc-backward-conditional &optional TO-ELSE)	<p>Move point backward to matching beginning of #if#ifdef#ifndef conditional.</p> <ul style="list-style-type: none"> <li>With C-u or numerical arg: move backward to matching #else#elif</li> <li>On success, push the original position on the mark ring and return the new position.</li> <li>On error, issue user error on mismatch. Shift marking is available with <b>C-M-&lt;left&gt;</b></li> </ul>
Move outward forward to matching #endif	<f6> <down>	(pel-c-preproc-outward-forward-conditional &optional NEST-COUNT)	<p>Move point forward, outward to end of current #if#ifdef#ifndef statement.</p> <ul style="list-style-type: none"> <li>By default move 1 nest level outward. A larger count can be specified with optional NEST-COUNT numeric argument.</li> <li>On success, push the original position on the mark ring and return the new position.</li> <li>On error, issue user error on mismatch.</li> </ul>
Move outward backward to matching #if   #ifdef   #ifndef	<f6> <up>	(pel-c-preproc-outward-backward-conditional &optional NEST-COUNT)	<p>Move point backward, outward to beginning of current #if#ifdef#ifndef statement.</p> <ul style="list-style-type: none"> <li>By default move 1 nest level outward. A larger count can be specified with optional NEST-COUNT numeric argument.</li> <li>On success, push the original position on the mark ring and return the new position. On error, issue user error on mismatch.</li> </ul>
Show all C pre-processor conditional statements inside an occur buffer	<f6> o	(pel-c-preproc-conditionals-occur &optional NLINES)	<p>Show C pre-processor conditional statements inside an occur buffer.</p> <ul style="list-style-type: none"> <li>Each line is shown with NLINES before and after, or -NLINES before if NLINES is negative.</li> <li>NLINES defaults to ‘list-matching-lines-default-context-lines’.</li> <li>If a region is defined the search is restricted to the region.</li> </ul>




Description	Keystroke	Function	Note
<ul style="list-style-type: none"> <li>By functions</li> <li>By structures</li> </ul>			<ul style="list-style-type: none"> <li>Move to beginning /end of function definition blocks or structure definition blocks.    Jump over comments.</li> <li>👉 When point is located before opening brace or right after closing brace and <b>show-paren-mode</b> is on, the matching parentheses are highlighted.</li> </ul>
Forward to start of next top level function or struct	<f12> <down>	(pel-beginning-of-next-defun &optional SILENT DONT-PUSH_MARK)	Move forward to the beginning of the next function or type definition. <ul style="list-style-type: none"> <li>Move point before the function type or the struct or typedef keyword.</li> <li>Beeps if does not find beginning of next function unless SILENT is non-nil.</li> <li>If the beginning of next function is found, push the start location to the mark ring unless DONT-PUSH_MARK is non-nil.               <ul style="list-style-type: none"> <li>Move back to previous position with <b>M-`</b> or <b>&lt;f6&gt;&lt;f6&gt;</b>.</li> </ul> </li> <li>➡️Shift marking is available. With <b>&lt;f6&gt;</b> and <b>&lt;f12&gt;</b> hit Shift after function key, before cursor key.</li> <li>👉 This command complements what end-of-defun does.</li> <li>It moves forward but not to the end of the function definition (like end-of-defun) but to the beginning of the function definition, which is often what users of other editors expect.</li> </ul>
Forward to end of current top-level function or struct.	C-M-e	(c-end-of-defun &optional ARG)	Move forward to the end of a top level declaration. <ul style="list-style-type: none"> <li>With argument, do it that many times. Negative argument -N means move back to Nth preceding end.</li> </ul>
	C-M-<end> <f12> <right>	(end-of-defun &optional ARG)	Move forward to the end of next function or type definition. With argument, do it that many times. Negative argument -N means move back to Nth preceding end of defun. <ul style="list-style-type: none"> <li>➡️Shift marking is available. With <b>&lt;f6&gt;</b> and <b>&lt;f12&gt;</b> hit Shift after function key, before cursor key.</li> <li>⚠️ This command moves to the end of the next <b>top-level</b> function. It skips nested functions.</li> </ul>
Backward to beginning of current top-level function or struct	C-M-a	(c-beginning-of-defun &optional ARG)	Move backward to the beginning of a function or type definition. <ul style="list-style-type: none"> <li>With a positive argument, move backward that many functions or structures. A negative argument -N means move forward to the Nth following beginning.</li> </ul>
	C-M-<home>	(beginning-of-defun &optional ARG)	Move backward to the beginning of function or type definition. <ul style="list-style-type: none"> <li>Move point before the function type or the struct or typedef keyword.</li> <li>With ARG, do it that many times. Negative ARG means move forward to the ARGth following beginning of defun.</li> <li>➡️Shift marking is available. With <b>&lt;f6&gt;</b> and <b>&lt;f12&gt;</b> hit Shift after function key, before cursor key.</li> <li>⚠️ This command moves to the beginning go the next function or of the same nesting level of the current location. It skips the functions that are more deeply nested.</li> </ul>
	<f12> <up>		
Backward to end of previous top level function or struct	<f12> <left>	(pel-end-of-previous-defun &optional SILENT DONT-PUSH_MARK)	Move backwards to the end of the previous function or type definition. <ul style="list-style-type: none"> <li>Beeps if does not find end of previous function unless SILENT is non-nil.</li> <li>If the end of previous function is found, push the start location to the mark ring unless DONT-PUSH_MARK is non-nil.               <ul style="list-style-type: none"> <li>Move back to previous position with <b>M-`</b> or <b>&lt;f6&gt;&lt;f6&gt;</b>.</li> </ul> </li> <li>➡️Shift marking is available. With <b>&lt;f6&gt;</b> and <b>&lt;f12&gt;</b> hit Shift after function key, before cursor key.</li> <li>⚠️ In some cases it fails to detect the end of the previous block and fails. 🐛</li> </ul>
By blocks	Move across C statements and C scope blocks, or any group of (), [], {} or <> blocks.		
By List element	Move to the end or the beginning of a block		
Backward block/list  See also: 📖 Navigation	C-M-p	(backward-list &optional ARG)	Move backward across one balanced group of parentheses. <ul style="list-style-type: none"> <li>This command will also work on other parentheses-like expressions defined by the current language mode.</li> <li>With ARG, do it that many times.</li> <li>Negative arg -N means move forward across N groups of parentheses.</li> <li>This command assumes point is not in a string or comment.</li> <li><b>C-M-p</b> : ➡️ Shift marking is available in graphics mode, <b>not in terminal mode</b>.</li> </ul>
Move block backward  See also: • 📖 Navigation	C-M-b C-M-<left> C-[ C-b Esc C-b Esc C-<left>    ⚠️	(backward-sexp &optional ARG)	Move backward across one balanced expression (sexp). <ul style="list-style-type: none"> <li>With ARG, do it that many times. Negative arg -N means move forward across N balanced expressions. This command assumes point is not in a string or comment.</li> <li><b>C-M-b</b> : ➡️ Shift marking is available in graphics mode, <b>not in terminal mode</b>.</li> <li><b>C-M-&lt;left&gt;</b> : ➡️ Shift marking works with this command.</li> <li>⚠️ With PEL: if you want to use <b>Esc C-&lt;left&gt;</b> binding you must ensure that <b>pel-windmove-on-esc-cursor</b> user option is set to nil, otherwise it does something else.</li> <li>❖ <b>C-M-&lt;left&gt;</b> does not work on Windows, but <b>H-&lt;left&gt;</b> works.</li> <li>🐧 Several Linux distros map <b>C-M-&lt;left&gt;</b> to desktop workspace operation. In that case you can either use another key binding or change Linux key binding in Systems-&gt;settings-&gt;keyboard-&gt;shortcuts to prevent it from using that key sequence.</li> </ul>
Forward block/list  See also: 📖 Navigation	C-M-n	(forward-list &optional ARG)	Move forward across one balanced group of parentheses. <ul style="list-style-type: none"> <li>This command will also work on other parentheses-like expressions defined by the current language mode.</li> <li>With ARG, do it that many times.</li> <li>Negative arg -N means move backward across N groups of parentheses.</li> <li>This command assumes point is not in a string or comment.</li> <li><b>C-M-n</b> : ➡️ Shift marking is available in graphics mode, <b>not in terminal mode</b>.</li> </ul>
Move block forward  See also: • 📖 Navigation	C-M-f C-M-<right> C-[ C-f Esc C-f Esc C-<right>    ⚠️	(forward-sexp &optional ARG)	Move forward across one balanced expression (sexp). <ul style="list-style-type: none"> <li>With ARG, do it that many times. Negative arg -N means move backward across N balanced expressions. This command assumes point is not in a string or comment.</li> <li><b>C-M-f</b> : ➡️ Shift marking is available in graphics mode, <b>not in terminal mode</b>.</li> <li><b>C-M-&lt;right&gt;</b> : ➡️ Shift marking works with this command.</li> <li>⚠️ With PEL: if you want to use <b>Esc C-&lt;right&gt;</b> binding you must ensure that <b>pel-windmove-on-esc-cursor</b> user option is set to nil, otherwise it does something else.</li> <li>❖ <b>C-M-&lt;right&gt;</b> does not work on Windows, but <b>H-&lt;right&gt;</b> does.</li> <li>🐧 Several Linux distros map <b>C-M-&lt;right&gt;</b> to desktop workspace operation. In that case you can either use another key binding or change Linux key binding in Systems-&gt;settings-&gt;keyboard-&gt;shortcuts to prevent it from using that key sequence.</li> </ul>
in/out of blocks	Move in or out of C scope blocks, or any group of (), [], {} or <> blocks.		
Backward Up/outside sexp hierarchy  See also: • 📖 Navigation	C-M-u C-M-<up> C-[ C-u Esc C-u Esc C-<up>    ⚠️	(backward-up-list &optional ARG ESCAPE-STRINGS NO-SYNTAX-CROSSING)	Move backward out of one level of parentheses or nested blocks. <ul style="list-style-type: none"> <li>This command will also work on other parentheses-like expressions defined by the current language mode. With ARG, do this that many times. A negative argument means move forward but still to a less deep spot.</li> <li>⚠️ With PEL: if you want to use <b>Esc C-&lt;up&gt;</b> binding you must ensure that <b>pel-windmove-on-esc-cursor</b> user option is set to nil.</li> <li><b>C-M-u</b> : ➡️ Shift marking is available in graphics mode, <b>not in terminal mode</b>.</li> <li><b>C-M-&lt;up&gt;</b> : ➡️ Shift marking works with this command.</li> <li>❖ <b>C-M-&lt;up&gt;</b> does not work on Windows, but <b>H-&lt;up&gt;</b> does.</li> </ul>
Forward Up/outside sexp hierarchy  See also: 📖 Navigation	C-M-]	(up-list &optional ARG ESCAPE-STRINGS NO-SYNTAX-CROSSING)	Move forward out of one level of parentheses or nested blocks. <ul style="list-style-type: none"> <li>This command will also work on other parentheses-like expressions defined by the current language mode.</li> <li>With ARG, do this that many times. A negative argument means move backward but still to a less deep spot.</li> </ul>



Description	Keystroke	Function	Note
<b>Hide-ifdef Mode</b> <ul style="list-style-type: none"> <li>hide/show code controlled by C-preprocessor</li> </ul>	<p>Hide-ifdef mode suppresses (hides or shadows) the display of code that the C preprocessor wouldn't pass through.</p> <ul style="list-style-type: none"> <li>It supports complete C/C++ expression and precedence.</li> <li>It scans for new #define symbols and macros. <ul style="list-style-type: none"> <li>It hides blocks of code that would not be include in the expanded file according to the state of pre-processor symbols that are maintained inside the Hide-ifdef environment: the <b>hide-ifdef-env</b> association list Emacs variable (use <b>&lt;f1&gt; v</b> to see the content of Emacs variables). See <a href="#">Help/Info</a>.</li> </ul> </li> <li>When hiding code, the hidden code is marked by ellipses (...). <ul style="list-style-type: none"> <li>⚠ Be cautious when editing near ellipses, since the hidden text is still in the buffer, and you can move the point into it and modify text unawares. <ul style="list-style-type: none"> <li>You can make your buffer read-only while hide-ifdef-hiding by setting <b>hide-ifdef-read-only</b> user-option to a non-nil value. <ul style="list-style-type: none"> <li>Access it hide-ifdef customization group with <b>&lt;f12&gt; # &lt;f3&gt;</b></li> </ul> </li> <li>You can toggle this variable with hide-ifdef-toggle-read-only (with <b>C-c @ C-q</b> or with <b>&lt;f12&gt; # r</b> or <b>&lt;f12&gt; &lt;f7&gt; R</b>.</li> </ul> </li> </ul> </li> </ul> <p>📖 With PEL, the commands are reachable via the <b>&lt;f12&gt;</b> prefix keys can also be reached via the <b>&lt;M-f12&gt;</b> and the <b>&lt;f11&gt; SPC c</b> prefix keys.</p> <ul style="list-style-type: none"> <li>✳ The key sequences that start with <b>&lt;f12&gt; &lt;f7&gt;</b> open the pel-<a href="#">c-preproc Hydra</a> allowing further hydra keys to be typed without any prefix.</li> <li>✳ 📦 Requires the <a href="#">hydra</a> external package  PEL activates when the <b>pel-use-hydra</b> user option is set to <b>t</b>.</li> </ul> <p>🔧 Several customize user option variables affect how the hiding is done:</p> <ul style="list-style-type: none"> <li>to change, execute: <b>M-x customize-group hide-ifdef</b> or type <b>&lt;f12&gt; # &lt;f3&gt;</b></li> <li>'hide-ifdef-env' <ul style="list-style-type: none"> <li>An association list of defined symbols for the current project. The list holds the following forms: <ul style="list-style-type: none"> <li>(SYMBOL) is used when the SYMBOL is defined (but without explicit value)</li> <li>(SYMBOL . VALUE) when the symbol is defined with an explicit value.</li> </ul> </li> </ul> </li> <li>'hide-ifdef-define-alist' <ul style="list-style-type: none"> <li>An association list of pre-defined symbol lists. Use 'hide-ifdef-set-define-alist' to save the current 'hide-ifdef-env' and 'hide-ifdef-use-define-alist' to set the current 'hide-ifdef-env' from one of the lists in 'hide-ifdef-define-alist'.</li> </ul> </li> <li>'hide-ifdef-lines' <ul style="list-style-type: none"> <li>Set to non-nil to not show #if, #ifdef, #ifndef, #else, and #endif lines when hiding.</li> </ul> </li> <li>'hide-ifdef-initially' <ul style="list-style-type: none"> <li>Indicates whether 'hide-ifdefs' should be called when Hide-Ifdef mode is activated.</li> </ul> </li> <li>'hide-ifdef-read-only' <ul style="list-style-type: none"> <li>Set to non-nil if you want to make buffers read only while hiding.</li> <li>After 'show-ifdefs', read-only status is restored to previous value.</li> </ul> </li> </ul>		
<b>Toggle the Hide-Ifdef mode :</b> <ul style="list-style-type: none"> <li>hide/show code suppressed by C preprocessor</li> </ul>	<ul style="list-style-type: none"> <li><b>&lt;f12&gt; M-#</b></li> <li><b>&lt;M-f12&gt; M-#</b></li> <li>✳ <b>&lt;f12&gt; &lt;f7&gt; #</b></li> </ul> <hr/> <ul style="list-style-type: none"> <li><b>&lt;f11&gt; SPC c M-#</b></li> </ul>	(hide-ifdef-mode &optional ARG)	Toggle features to hide/show ifdef blocks (Hide-Ifdef mode). <ul style="list-style-type: none"> <li>With a prefix argument, enable Hide-Ifdef mode if ARG is positive, and disable it otherwise.</li> <li>Hide-Ifdef mode is a buffer-local minor mode for use with C and C-like major modes.</li> </ul> When enabled, code within #ifdef constructs that the C preprocessor would eliminate may be hidden from view.
<b>Toggle read-only mode when text is hidden</b>	<ul style="list-style-type: none"> <li><b>C-c @ C-q</b></li> <li><b>&lt;f12&gt; # r</b></li> <li>✳ <b>&lt;f12&gt; &lt;f7&gt; R</b></li> </ul>	(hide-ifdef-toggle-read-only)	Toggle read-only: toggle 'hide-ifdef-read-only'. <ul style="list-style-type: none"> <li>Note that you can make the file read only by default when hide-ifdef is hiding text, by setting the <b>'hide-ifdef-read-only'</b> user option to <b>t</b>.</li> </ul>
<b>Toggle shadowing of hidden text.</b>	<ul style="list-style-type: none"> <li><b>C-c @ C-w</b></li> <li><b>&lt;f12&gt; # w</b></li> <li>✳ <b>&lt;f12&gt; &lt;f7&gt; W</b></li> </ul>	(hide-ifdef-toggle-shadowing)	Toggle shadowing. <ul style="list-style-type: none"> <li>When shadowing is on, text that would be hidden is “shadowed” instead: it is displayed with the <b>shadow face</b> (normally something dim, all depending of the theme used).</li> </ul>
<b>Hide code suppressed by C preprocessor</b>	<ul style="list-style-type: none"> <li><b>C-c @ h</b></li> <li><b>&lt;f12&gt; # H</b></li> <li><b>&lt;M-f12&gt; # H</b></li> <li>✳ <b>&lt;f12&gt; &lt;f7&gt; H</b></li> </ul> <hr/> <ul style="list-style-type: none"> <li><b>&lt;f11&gt; SPC c # H</b></li> </ul>	(hide-ifdefs &optional NOMSG)	Hide the contents of some #ifdefs. <ul style="list-style-type: none"> <li>Assume that defined symbols have been added to 'hide-ifdef-env'.</li> <li>The text hidden is the text that would not be included by the C preprocessor if it were given the file with those symbols defined.</li> <li>With prefix command presents it will also hide the #ifdefs themselves.</li> </ul> Turn off hiding by calling <b>'show-ifdefs'</b> .
<b>Restore all hidden into view</b>	<ul style="list-style-type: none"> <li><b>C-c @ s</b></li> <li><b>&lt;f12&gt; # S</b></li> <li>✳ <b>&lt;f12&gt; &lt;f7&gt; S</b></li> </ul>	(show-ifdefs)	Cancel the effects of 'hide-ifdef': show the contents of all #ifdefs.
<b>Hide part of current block that would not be included</b>	<ul style="list-style-type: none"> <li><b>C-c @ C-d</b></li> <li><b>&lt;f12&gt; # h</b></li> <li>✳ <b>&lt;f12&gt; &lt;f7&gt; h</b></li> </ul>	(hide-ifdef-block &optional ARG START END)	Hide the ifdef block (true or false part) enclosing or before the cursor. <ul style="list-style-type: none"> <li>With optional prefix argument ARG, also hide the #ifdefs themselves.</li> </ul>
<b>Show all parts of the current #ifdef block</b>	<ul style="list-style-type: none"> <li><b>C-c @ C-s</b></li> <li><b>&lt;f12&gt; # s</b></li> <li>✳ <b>&lt;f12&gt; &lt;f7&gt; s</b></li> </ul>	(show-ifdef-block &optional START END)	Show the ifdef block (true or false part) enclosing or before the cursor.
<b>Set a variable to a specific value</b>	<ul style="list-style-type: none"> <li><b>C-c @ d</b></li> <li><b>&lt;f12&gt; # d</b></li> <li>✳ <b>&lt;f12&gt; &lt;f7&gt; d</b></li> </ul>	(hide-ifdef-define VAR &optional VAL)	Define a VAR to VAL (default 1) in 'hide-ifdef-env'. <ul style="list-style-type: none"> <li>This allows hiding the block inside <b>#ifdef VAR</b> by executing the command hide-ifdefs.</li> </ul>
<b>Undefine a variable</b>	<ul style="list-style-type: none"> <li><b>C-c @ u</b></li> <li><b>&lt;f12&gt; # u</b></li> <li>✳ <b>&lt;f12&gt; &lt;f7&gt; u</b></li> </ul>	(hide-ifdef-undef START END)	Undefine a VAR <ul style="list-style-type: none"> <li>This allows hiding the blocks inside <b>#ifdef VAR</b> by executing the command hide-ifdefs.</li> </ul>
<b>Save the symbol environment list into a named list</b>	<ul style="list-style-type: none"> <li><b>C-c @ D</b></li> <li><b>&lt;f12&gt; # D</b></li> <li>✳ <b>&lt;f12&gt; &lt;f7&gt; D</b></li> </ul>	(hide-ifdef-set-define-alist NAME)	Save the state of the current <b>hide-ifdev-env</b> to a list with the specified NAME for later re-use. The value is saved inside the <b>hide-ifdef-define-alist</b> variable. <ul style="list-style-type: none"> <li>⚠ The list is not saved to disk. You may want to pre-create the value for a given project and store it inside your local directory variables for example.</li> </ul>
<b>Use a named symbol environment list</b>	<ul style="list-style-type: none"> <li><b>C-c @ U</b></li> <li><b>&lt;f12&gt; # U</b></li> <li>✳ <b>&lt;f12&gt; &lt;f7&gt; U</b></li> </ul>	(hide-ifdef-use-define-alist NAME)	Set <b>'hide-ifdef-env'</b> to the already saved symbol list with the specified NAME. <ul style="list-style-type: none"> <li>Takes the value from the <b>hide-ifdef-define-alist</b>.</li> </ul>
<b>Clear the complete list of #define'd symbols inside 'hide-ifdef-env'</b>	<ul style="list-style-type: none"> <li><b>C-c @ C</b></li> <li><b>&lt;f12&gt; # C</b></li> <li>✳ <b>&lt;f12&gt; &lt;f7&gt; C</b></li> </ul>	(hif-clear-all-ifdef-defined)	Clears all symbols defined in <b>'hide-ifdef-env'</b> . <ul style="list-style-type: none"> <li>It first backup this variable to <b>'hide-ifdef-env-backup'</b> before clearing to prevent accidental clearance.</li> </ul>
<b>Evaluate pre-processor macro</b>	<ul style="list-style-type: none"> <li><b>C-c @ e</b></li> <li><b>&lt;f12&gt; # e</b></li> <li>✳ <b>&lt;f12&gt; &lt;f7&gt; e</b></li> </ul>	(hif-evaluate-macro RSTART REND)	Evaluate the macro expansion result for the active region. <ul style="list-style-type: none"> <li>If no region active, find the current #ifdefs and evaluate the result.</li> <li>⚠ Currently it supports only math calculations; strings or argumented macros can not be expanded.</li> </ul>



Description	Keystroke	Function	Note
Rendering markup embedded in comments	The following commands are used to create images from specific markup code embedded inside C source code comments. This can be useful when using these markup languages to describe UML diagrams or finite-state machines for example.  You can also use Graphviz, see <a href="#">M Graphviz Dot</a>		
Preview UML diagram from plantUML source in current plantUML region of commented source code  See also: <a href="#">M PlantUML</a>	<f12> u	(pel-render-commented-plantuml PREFIX &optional POS)	Render the PlantUML markup embedded in current mode comment. <ul style="list-style-type: none"> <li>Use region if identified otherwise use PlantUML block at point.</li> <li>Uses prefix (as PREFIX) to choose where to display it: <ul style="list-style-type: none"> <li>4 (when prefixing the command with <b>C-u</b>) -&gt; new window</li> <li>16 (when prefixing the command with <b>C-u C-u</b>) -&gt; new frame.</li> <li>else -&gt; new buffer</li> </ul> </li> <li>This can be used inside buffer using <b>any</b> major mode, when PlantUML markup is embedded inside source code comment.</li> </ul> <p>👉 Use this in source code to describe your code architecture with PlantUML markup, then generate the UML rendering by moving point inside the PlantUML block and issuing this command.</p> <p>📦 Requires the <b>plantuml-mode</b> external package,  activated by <b>pel-use-plantuml</b> user option being non-nil.</p>
C Specific search and replace	The following PEL commands are specialized search and replace functions used to detect and fix code that explicitly compare a pointer to NULL and a boolean value to true or false. Comparing against these symbols is poor C or C++ code and should be replaced. The following commands help locating such code and replacing it with better styled-code that does not explicitly uses the keyword.		
Search for poor code using comparison against NULL	<f12> s n	(pel-c-search-equal_NULL)	Move point to the next expression like if (ptr == NULL) or if (NULL == ptr)
	<f12> s N	(pel-c-search-not-equal_NULL)	Move point to the next expression like if (ptr != NULL) or if (NULL != ptr)
Search for poor code using comparison against false or FALSE	<f12> s f	(pel-c-search-equal_false)	Move point to the next expression like if (boolean == false) or if (false == boolean). Also search for FALSE.
	<f12> s F	(pel-c-search-not-equal_false)	Move point to the next expression like if (boolean != false) or if (false != boolean). Also search for FALSE.
Search for poor code using comparison against true or TRUE	<f12> s t	(pel-c-search-equal_true)	Move point to the next expression like if (boolean == true) or if (true != boolean). Also search for TRUE
	<f12> s T	(pel-c-search-not-equal_true)	Move point to the next expression like if (boolean != true) or if (true != boolean). Also search for TRUE
Search for any of the poor code listed in the previous 6 commands	<f12> s *	(pel-c-search-any-comparison-problem)	Move point to the next instance of any of the expressions searched by the 6 commands above.
Improve C/C++ code: remove explicit comparisons against NULL, TRUE, FALSE, true and false	<f12> s C-f	(pel-c-fix-comparison-problems)	Replace all instances of C/C++ code that explicitly compares a pointer against NULL or a boolean value against true, false, TRUE and FALSE by the logically equivalent expression that does not use the keyword: For example this replaces: <ul style="list-style-type: none"> <li>if (pointer == NULL) by if (!pointer)</li> <li>if (value == TRUE) by if (value)</li> <li>if (value == FALSE) by if (!value)</li> <li>if (value == true) by if (value)</li> <li>if (value == false) by if (!value)</li> <li>if (pointer != NULL) by if (pointer)</li> <li>if (value != TRUE) by if (!value)</li> <li>if (value != FALSE) by if (value)</li> <li>if (value != true) by if (!value)</li> <li>if (value != false) by if (value)</li> </ul> It handles more complex expressions for ‘pointer’ and ‘value’ and also supports the expressions where the variable is placed on the right hand side of the comparison. <ul style="list-style-type: none"> <li>👉 The command can detect and reformat a large number of expressions but not all of them.</li> <li>⚠️ Therefore it’s a good idea to backup the original code and check the difference after executing this command to detect potential errors. If the translation is correct <i>there should be no change in the generated assembler code.</i></li> <li>The <b>best way to check if the reformatting is correct</b> is to compare the generated assembler code for the file before and after the code reformatting done by this command. <ul style="list-style-type: none"> <li>With GCC toolchain, use the <b>objdump -- disassemble</b> command on the object file to generate the assembler files. The LLVM toolchain provide the <b>llvm-objdump</b> equivalent.</li> </ul> </li> </ul>

## Emacs & C— References

Document	Notes
<a href="#">GNU emacs - CC Mode Manual</a>	
<a href="#">GNU Emacs Manual - Styles</a>	
<a href="#">Emacs BSD/Allman Style with 4 Space Tabs?</a>	
<a href="#">Emacs: Linux Kernel Style but with Allman/BSD Style Braces?</a>	
<a href="#">Emacs Wiki - Indenting C</a>	
<a href="#">Indent preprocessor directives as C code in emacs</a>	Does not fully address the way I want to have multi-indentations for pre-processor
<a href="#">elisp code - ppindent.el</a>	Implements pre-processor indentation with the # always in the first column. Not yet exactly what I want.
<a href="#">company-mode ; Modular in-buffer completion framework for Emacs</a>	