↖ Topic Index

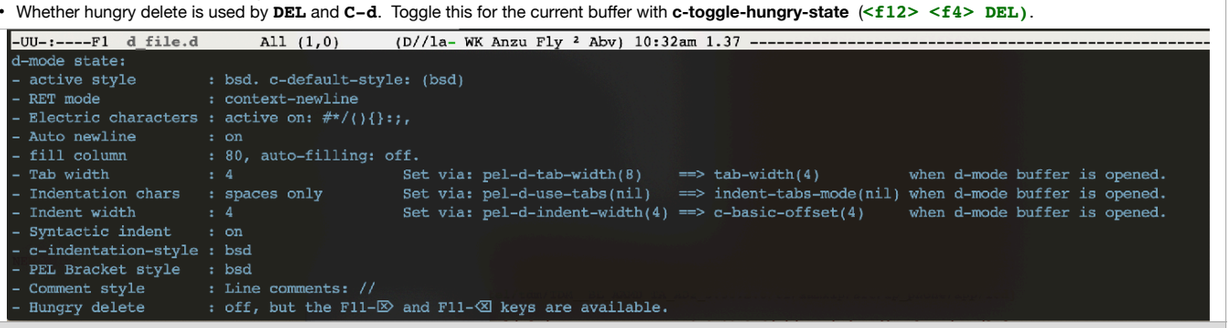
Programming Language Support — D

Description	Keystroke	Function	Note
D programming language support ○ Help & Customization • Set D script shebang • Customize skeleton • Insert file header • Insert parentheses • Control text modes ○ Control D code style • Control text-alignment ○ Show Mode settings • Electric keys ○ Insert new lines • Insert D comments ○ Comment dwim • Hide/show comments • Specialized delete • Indentation ○ Non-syntactic indent • Marking • Show info on code • Highlight blocks ○ Navigate in D ○ Render UML markup Last updated on:	2025-12-23		📦 Emacs supports the D programming language via a the **d-mode** external package. This package extends the Emacs CC Mode built-in package which supports the curly-bracket programming languages like D. Other external packages provide other features for working with D, they are listed in the reference table below. **PEL activates D support** via the customize user option variable **pel-use-d**. It must be set to **t** to activate support for D. 🎛️ Important aspects of D source code syntax controlled by the CC Mode are customizable with PEL user option variables. **PEL customization for D**: Simplifies configuration for editing D source code (To change, execute: `M-x customize-group pel-pkg-for-d`): Emacs customization group: **pel-pkg-for-d** • **pel-d-indent-width**: Identifies the number of columns used for indentation. Defaults to 4. • **pel-d-tab-width**: The width of a tab. Defaults to 4. This concept differs from indentation: you can have an indentation of 4 and tab width of 8: `M-i` will move point to columns that are multiple of 8 `<tab>` will indent to a column that is a multiple of 4. • For most uses it is best to set both values to the width of your needed indentation level. This way you can use commands that use either to control the indentation level. • **pel-d-use-tabs**: Whether hard tabs are used in indentation or not: **t**: tabs are used, **nil**: only spaces are used. Default: **nil**. • **pel-d-backet-style**: The bracket/indentation style supported by the electric keys. One of the values supported by Emacs (also possible to define your own with Elisp code). Default to "bsd". • Emacs customization group: **pel-pkg-for-cc**. Applies to all CC Mode related modes (like d-mode). • **pel-cc-auto-newline**: Whether automatic newline mode is active on all CC Mode (including d-mode). The values for those user option variables can also be stored inside directory local files and even as file local variables. You can also modify them for each buffer and view their current settings using the commands listed in the following set of rows. None of the commands below change PEL default; they change the value for the current buffer only. • PEL provides the following set of mode-specific key prefixes: `<f11> SPC D`, `<f12>` and `M-<f12>` The first one is always available. The other two prefixes are only available in d-mode buffers. The `M-<f12>` prefix helps the typing flow when the next key is a Meta key. For simplification, the `<f11> SPC D` prefix is normally omitted in the table.
Open this PDF file. See also: Σ **Help/Info**	`<f11> SPC D <f1>` `<f12> <f1>`	(**pel-help-pdf** &optional OPEN-WEB-PAGE)	Open the **D - D** local PDF. If the prefix argument (like **C-u** or **M--**) is used, then it opens the remote GitHub hosted raw PDF instead. If the **pel-flip-help-pdf-arg** user-option is set it's the other way around.
Σ **Customize** PEL D support	`<f11> SPC D <f2>` `<f12> <f2>`	(**pel-customize-pel** &optional OTHER-WINDOW)	Customize PEL D support: d-mode. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
Σ **Customize** Emacs D support	`<f11> SPC D <f3>` `<f12> <f3>`	(**pel-customize-library** &optional OTHER-WINDOW)	Customize Emacs D support: d-mode. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
Select d-mode for extension-less D script file ☝The `<f12>` key is available only until a PEL controlled major mode is activated. Then it becomes a buffer prefix key.	`<f12>`	(**pel-as** &optional FORCE)	Inside a fundamental-mode buffer, interactively select major mode for the buffer. Re-do it with arg. ☝see **Create extension-less executable scripts with PEL**. This command is mostly used to set the major mode of a buffer in fundamental-mode', when the `<f12>` key binding is available for it. After being used once in a buffer the major mode is selected and the PEL key binding will not be available when PEL supports the major mode. For D file, select **d.** It will insert a shebang line specified by **pel-d-shebang-line** user option. PEL defines the (as &optional FORCE) alias unless **pel-has-alias-as** user-option is set to nil. You can use `M-x as` to invoke it.
Generic code skeletons • **tempo skeletons**			⚠️ PEL does not yet define skeletons for D; it uses the generic one. See also: Σ **Inserting Text** **T Templates** • Emacs provides the built-in skeleton mechanism and the **tempo skeletons**. • PEL supports both. They are used a little bit differently. PEL provides **generic** tempo skeletons you can use for D until PEL adds D-specific skeletons. • PEL provides key bindings to the tempo skeletons: the generic code templates, accessible via the `<f6>` prefix key, and the language-specific code templates, accessible via the `<f12>` key prefix.
Σ **Customize** PEL Text Insertions control for D code skeletons.	`<f6> <f2>` `<f12> <f12> <f2>`	(**pel-customize-pel** &optional OTHER-WINDOW) (**pel-customize-generic-skels** &optional OTHER-WINDOW)	Open the customization groups that control the format of the various skeletons including the generic skeleton used by the `<f6> h` key and the `<f12><f12> h` key (see below). • If OTHER-WINDOW is non-nil (use **C-u**), display in other window.
Insert generic file module header block — Language agnostic After inserting the template, navigate though areas that must be filled with: • forward: **C-c .** • backward: **C-c ,**	`<f6> h` `<f12> <f12> h`	(**pel-generic-file-header**)	Insert a file header block at the top of the file. Works only for buffer visiting a file. ⚠️ The command key binding `<f6> h` is available only 1 second after Emacs has started. ⚠️ As mentioned above PEL does not yet define D-specific skeletons, this uses the generic one. ☝Specify the format of the header via the user-options in the **pel-pkg-generic-code-style** customization group accessible via `<f6> <f2>` • Inside a **D** buffer, `<f12> <f2>` provides access to the following customization groups: ☝After inserting a template, use **tempo-forward-mark** and **tempo-backward-mark** to move to the beginning of each section that must be filled.
Toggle pel-tempo-mode	`<f6> SPC` `<f12> <f12> SPC`	(**pel-tempo-mode** &optional ARG)	Toggle PEL tempo mode on/off. PEL tempo mode activates `C-c .` and `C-c ,` as well as to `C-c C-.` and `C-c C-,` 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 of keys are only available in graphics mode. ☝The pel-generic-file-header command inserts the text using a tempo skeleton: the PEL tempo mode is automatically activated by typing `<f6> h`.
Expand any tag in template Note: PEL default skeleton does not use tags.	`<f6> <f12>` `<f12> <f12> <f12>`	(**tempo-complete-tag** &optional SILENT)	Look for a tag and expand it. 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. • 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.
Inserting code			Extra text insertion can be done with the following commands.
Insert Parentheses	`M-(`	(**insert-parentheses** &optional ARG)	For D: insert a parenthesis pair '()', leaving point after open-paren. • A positive ARG encloses the following ARG sexps in parenthesis if they are balanced. • A negative ARG encloses the preceding ARG sexps instead. • No argument is equivalent to zero: just insert '()' and leave point between. • PEL makes 'parens-require-spaces' buffer local and set it to nil in D 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. • If region is active, insert enclosing characters at region boundaries. This command assumes point is not in a string or comment.
Search Support			In D mode, the superword mode can be useful since snake_case is often used. Using superword-mode helps searching. PEL activates the superword mode by default in D mode. To change this use the `<f11> t <f2>` to access the customize buffer.
Toggle superword-mode See also: • Σ **Text Modes** • Σ **Search/Replace**	• `<f11> t m p` • `<f12> M-p`	(**superword-mode** &optional ARG)	Toggle superword-mode: a minor mode that treats snake_case as one word. In D '_' are treated as part of words. With prefix argument, enable superword mode if ARG is positive, disable it otherwise. • PEL provides the `<f12> M-p` key for the programming language modes where snake_case is popular (Emacs Lisp, C, C++, Erlang, Python, etc…)
Toggle subword-mode See also: • Σ **Text Modes**	• `<f11> t m b` • `<f12> M-b` • `M-<f12> M-b` • `<f11> SPC D M-b`	(**subword-mode** &optional ARG)	Toggle subword-mode: a minor mode that treats sections of camelCase and PascalCase as distinct words. With prefix argument ARG, enable Subword mode if ARG is positive, disable it otherwise. ☝D naming convention promotes the use of camelCase for functions, enums, constants and variables and PascalCase for types. Using the subword-mode allows you to move into, delete, transpose sections of words with the corresponding word commands.

Description	Keystroke	Function	Note
CC Mode Style Management • **Learn style used in current buffer**	Automatic indentation, brace format style and several other D stylistic elements are controlled by the CC Mode and the CC mode variables. • You can impose an indentation style by customization. • You can also adjust the style to what is used in the current buffer: Emacs provides the following commands to parse the source code and identify the style it uses. It *learns* the style and sets the style controlling variables from what it detects in the buffer. • ☝Use this to adapt to source code written by others and want to continue using the same style. • ☝For the following commands all commands that use a key binding that ends with an upper case letter install the style.		
Show/Modify syntactic context	**C-c C-o**	(**c-set-offset** SYMBOL OFFSET &optional IGNORED)	Change the value of a syntactic element symbol in 'c-offsets-alist'. • SYMBOL is the syntactic element symbol to change and OFFSET is the new offset for that syntactic element. The optional argument is not used and exists only for compatibility reasons.
Show syntactic information for current line	**C-c C-s**	(**c-show-syntactic-information** ARG)	Show syntactic information for current line. • With universal argument, inserts the analysis as a comment on that line.
Guess the style used in the current buffer, do not install it	**<f12> <f4> g g**	(**c-guess-buffer-no-install** &optional ACCUMULATE)	Guess the style on the whole current buffer; don't install it. • If given a prefix argument (or if the optional argument ACCUMULATE is non-nil) then the previous guess is extended, otherwise a new guess is made from scratch.
Guess the style of the code in the buffer and install it.	**<f12> <f4> g B**	(**c-guess-buffer** &optional ACCUMULATE)	Guess the style on the whole current buffer, and install it. • The style is given a name based on the file's absolute file name. • If given a prefix argument (or if the optional argument ACCUMULATE is non-nil) then the previous guess is extended, otherwise a new guess is made from scratch.
Guess style in the region and install it.	**<f12> <f4> g G**	(**c-guess** &optional ACCUMULATE)	Guess the style using the first '**c-guess-region-max**' bytes of the file, and install it. • The **c-guess-region-max** user-option defaults to 50,000 bytes, nil means all buffer. • The style is given a name based on the file's absolute file name. • If given a prefix argument (or if the optional argument ACCUMULATE is non-nil) then the previous guess is extended, otherwise a new guess is made from scratch.
Guess the style of a region and install it.	**<f12> <f4> g R**	(**c-guess-region** START END &optional ACCUMULATE)	Guess the style on the region and install it. • The style is given a name based on the file's absolute file name. • If given a prefix argument (or if the optional argument ACCUMULATE is non-nil) then the previous guess is extended, otherwise a new guess is made from scratch.
Set buffer style to guessed style.	**<f12> <f4> g I**	(**c-guess-install** &optional STYLE-NAME)	Install the latest guessed style into the current buffer. • This guessed style is a combination of 'c-guess-guessed-basic-offset', 'c-guess-guessed-offsets-alist' and 'c-offsets-alist'. • The style is entered into CC Mode's style system by 'c-add-style'. Its name is either STYLE-NAME, or a name based on the absolute file name of the file if STYLE-NAME is nil.
View Guessed style as a set of Emacs Lisp statements	**<f12> <f4> g ?**	(**c-guess-view** &optional WITH-NAME)	Emit emacs lisp code which defines the last guessed style, so you can put the code into .emacs if you prefer the guessed code. • "STYLE NAME HERE" is used as the name for the style in the emitted code. If WITH-NAME is given, it is used instead. WITH-NAME is expected as a string but if this function called interactively with prefix argument, the value for WITH-NAME is asked to the user.
CC Mode support Behaviour Control	The following commands are CC Mode specific, available for each of the programming languages similar that have a mode derived from CC Mode like D. They can be used to dynamically change the behaviour of important keys such as the return key, delete key, semi-colon, etc.. The CC Mode controls the indentation and bracket style which controls what happens when electric characters are typed (when the electric mode is activated) and provide a better experience when editing C source code. • **CC Mode state displayed in the mode line**: 𝔏C{…} where: • 𝔏 is the CC mode programming language name: C, C++, ObjC, etc… • C is the C comment style: '*' for block command (/* */) and '/' for line comments (//) • {…} are the other electric flags: • '**l**' for electric mode • '**a**' for auto-newline mode • '**h**' for hungry mode • '**w**' for subword mode ☝ Use **<f12> <f4> ?** to display the current state.		
Toggle Electric state †	• **C-c C-l** • **<f12> <f4> e**	(**c-toggle-electric-state** &optional ARG)	Toggle the electric indentation feature done with the electric character keys. • 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.
Set indentation style †	• **C-c .** • **<f12> <f4> s**	(**c-set-style** STYLENAME &optional DONT-OVERRIDE)	Set the bracket/indentation style for the current buffer. • Prompts for the name. • Supports tab completion (so use tab to see the list). Can be one of the values supported by Emacs but you can also add your customized mode with some Emacs Lisp code.
Override indentation width for current buffer †	**<f12> <f4> TAB**	(**pel-cc-set-indent-width** &optional NEW-WIDTH)	Interactively change the Indentation with for current buffer to NEW-WIDTH. • Prompt for new value. • Use 0 to restore value specified by configuration (**pel-d-indent-width**). ☝This can be used to change indentation several times in a file.
Toggle syntactic indentation †	**<f12> <f4> i**	(**c-toggle-syntactic-indentation** &optional ARG)	Toggle syntactic indentation. • Optional numeric ARG, if supplied, turns on syntactic indentation when positive, turns it off when negative, and just toggles it when zero or left out. • When syntactic indentation is turned on (the default), the indentation functions and the electric keys indent according to the syntactic context keys, when applicable. • 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'.
Toggle Comment Style †	• **C-c C-k** • **<f12> <f4> M-;**	(**pel-c-toggle-comment-style** &optional ARG)	Toggle the **C comment style** between block/C-style (/* */) and line/C++-style (//) comments. • With optional numeric ARG, switch to block comment style when positive, to line comment style when negative, and just toggles it when zero or left out. Print newly used style format. ⚠ Only the **//** and **/* */** styles are supported. The **/+ +/** comments are not supported. ☝This is part of CC Mode. Use **<f12> <f4> ?** to display the current state.
Toggle Hungry Delete mode †	**<f12> <f4> DEL**	(**c-toggle-hungry-state** &optional ARG)	Toggle hungry-delete-key feature. Affect **<DEL>** and **C-d** keys. • 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. • 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. ☝This is part of CC Mode. Use **<f12> <f4> ?** to display the current state.
Toggle text alignment on pel-newline-and-indent-below See also: • **Σ Align** • **Σ Indentation** †	**<f11> M-RET**	(**pel-toggle-newline-indent-align**)	Toggle variable ***pel-newline-does-align*** for the local buffer. This toggles the way function 'pel-newline-and-indent-below' operates. • If ***pel-newline-does-align*** is t, it aligns several syntactic element in the current block: the comments, the assignments. • 🎛 Identify modes where ***pel-newline-does-align*** is automatically activated (set to t) by adding the major mode to the list in the **pel-modes-activating-align-on-return** user option. • This affects the behaviour of the following commands: • pel-cc-newline (assigned to **RET** in CC modes like c-mode, c++-mode and d-mode). • pel-newline-and-indent-below (assigned the **M-RET**)

Description	Keystroke	Function	Note
Toggle auto-newline insertion mode 🗡	• **C-c C-a** • **<f12> <f4> M-RET**	(**c-toggle-auto-newline** &optional ARG)	Toggle **auto-newline** feature. • 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. • Turning on auto-newline automatically enables ***electric indentation***. • 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. 💻 Emacs allows customizing the style and how automatic newlines are used. See the CC Mode Manual section: Customizing Auto-newlines.
Change RET key behaviour: select return mode. 🗡	**<f12> <f4> RET**	(pel-cc-change-newline-mode)	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: • context-newline : the default : uses (**c-context-line-break**) with the extra ability to repeat its execution with an argument. • newline-and-indent: uses (**newline** ARG t) to insert newline and indent. • just-newline-no-indent: uses (**electric-indent-just-newline** ARG) ☛ 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 **pel-initial-c-newline-mode** can be set to change the default for c-mode.		
Display current Mode settings	**<f12> <f4> ?**	(pel-cc-mode-info)	Display information about current **CC mode** derivative for the current d-mode buffer.

	The information displayed in specialized help buffer includes the following:
	• CC mode style currently active, along with a list of styles associated with current mode. Change it for the current buffer with **C-c .** or **<f12> <f4> s**. The Emacs the **c-default-style** user option defines associations between major modes and the style to use. PEL provides the **pel-c-backet-style** that is used to set the style for c-mode. Use **<f12> <f2>** from a c-mode buffer to access the customization buffer to change it.
	• Return key behaviour:
	  • RET (return key) mode. Change with pel-cc-change-newline-mode (**<f12> <f4> RET**).
	  • Whether return performs alignment. Change that with pel-toggle-indent-align (**<f11> M-RET**).
	• State of **electric C characters** (toggle it on/off with c-toggle-electric-state (**C-c C-l** or **<f12> <f4> e**):
	  • whether it is active or not, and when active what character(s) exhibit electric behaviour.
	  • if auto-newline on some characters (';' and some other based on style) is active. Toggle this with **C-c C-a** or **<f12> <f4> M-RET**.
	• The fill column: the column where force line wrap is done when the auto-fill-mode is active. Toggle auto fill mode with **<f11> RET**.
	• Tab width and whether hard tabs are used. These are set by the user options **pel-d-tab-width** and **pel-d-use-tabs**. In a d-mode buffer use **<f12> <f2>** to open the appropriate customization buffer to change them.
	  • ☝ Remember that tab width does **not** 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.
	• Indentation width, controlled by **c-basic-offset** normally set by **pel-d-indent-width** in PEL, and whether syntactic indentation mode is active.
	• 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 **Allman style**. You can dynamically change for the current buffer with c-set-style command (**C-c .** or **<f12> <f4> s**).
	  • ☝ 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.
	• The comment style. Supports C-style (**/* */**) and C++-style (**//**) comments, but unfortunately not the D **/+ +/** comments.
	  • This can be changed dynamically for the current buffer with the c-toggle-comment-style command (**C-c C-k** or **<f12> <f4> M-;**). 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.
	• Whether hungry delete is used by **DEL** and **C-d**. Toggle this for the current buffer with **c-toggle-hungry-state** (**<f12> <f4> DEL**).

☝Notice the name of the PEL user-options that set the significant feature controlling Emacs variables in the message ☛ 🎛 More info is shown in that buffer as buttons that provide access to more help and ability to customize the values.			
Electric Keys	The following **electric D characters** have special meaning when the electrical state is active in a buffer using d-mode. • Toggle electric behaviour in the current buffer with: with c-toggle-electric-state (**C-c C-l** or **<f12> <f4> e**).		
#	**#**	(**c-electric-pound** ARG)	Insert a "#".
	• 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. • D does not use the pound character much. It only uses it for #line statements.		
()	• **(** • **)**	(**c-electric-paren** ARG)	Insert a parenthesis.
	• 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. • Whitespace between a function name and the parenthesis may get added or removed; see the variable 'c-cleanup-list'. • 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'.		
{}	• **{** • **}**	(**c-electric-brace** ARG)	Insert a brace.

	• 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:
	  a) 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'.
	  b) Any auto-newlines are indented. The original line is also reindented unless 'c-syntactic-indentation' is nil.
	  c) If auto-newline is turned on, various newline cleanups based on the settings of 'c-cleanup-list' are done.		

:	**:**	(**c-electric-colon** ARG)	Insert a colon.

	• 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:
	  a) 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'.
	  b) Any auto-newlines are indented. The original line is also reindented unless 'c-syntactic-indentation' is nil.
	  c) 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'.		

:,	• **;** • **,**	(**c-electric-semi&comma** ARG)	Insert a comma or semicolon.

	• 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:
	  a) 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&comma-criteria' for how newline insertion is determined.
	  b) Any auto-newlines are indented. The original line is also reindented unless 'c-syntactic-indentation' is nil.
	  c) 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'.		

Description	Keystroke	Function	Note
Electric pairs	It is also possible to control the insertion of character pairs by activating the **electric-pair-mode** in the buffer. • Type the first of a pair to insert this one and its matching character for (), [], {}, "" and ''. • 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 ε(1)		
Toggle electric-pair-mode in current buffer **Lighter:=** ε(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 a prefix argument ARG, enable Electric Pair mode if ARG is positive, and disable it otherwise. • 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.		
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. • With PEL the default behaviour can be selected by customization and modified dynamically for the current buffer with the **pel-cc-change-newline-mode** command (bound to **<F12> M-RET**) see the CC-Mode behaviour control section above. • The pel-cc-newline command also aligns comments and assignment in the code block if the **pel-modes-activating-align-on-return** user option list includes the current major mode. The state for the current buffer can also be modified by the **pel-cc-change-newline-mode** command (**<f11> M-RET**).		

Insert a new line and operate according to the currently active selected return mode.
 	
With PEL, modify behaviour with <F12> M-RET.
	RET	(**pel-cc-newline** &optional N)	Insert a newline and perhaps align.
			• With argument N repeat N times.
			• 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):
			  • c-context-line-break
			  • newline
			  • electric-indent-just-newline
			• If the variable 'pel-newline-does-align' is t, then perform the text alignment done by the function 'align'.

	Use : (**c-context-line-break**) : Do a line break suitable to the context. • 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. • 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. • 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. • When point is inside a string, only insert a backslash when it is also inside a preprocessor directive.		
	Use: (**newline** &optional ARG INTERACTIVE): Insert a newline, and move to left margin of the new line if it's blank. • With ARG, insert that many newlines. • If option 'use-hard-newlines' is non-nil, the newline is marked with the text-property 'hard'. • If 'electric-indent-mode' is enabled, this indents the final new line that it adds, and reindents the preceding line. • To just insert a newline, use M-x electric-indent-just-newline. 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. • With ARG, insert that many newlines.		

Insert an indented line below unbroken current line
See also:
Σ **Indentation**
	• **M-RET**
	• **<f11> <tab> RET**
		(**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.
			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.
			• If ***pel-newline-does-align*** is t, it aligns several syntactic element in the current block: the comments, the assignments.
			  • You can toggle this on/off with **<f11> M-RET**.
			  • Identify modes where ***pel-newline-does-align*** is automatically activated (set to t) by adding the c-mode to the list in the **pel-modes-activating-align-on-return** user option.

Insert a newline	**C-j**	(**electric-newline-and-maybe-indent**)	Insert a newline.
			• If 'electric-indent-mode' is enabled, that's that, but if it is *disabled* then additionally indent according to major mode.
			  • Indentation is done using the value of 'indent-line-function'.
			  • In programming language modes, this is the same as TAB.
			  • In some text modes, where TAB inserts a tab, this command indents to the column specified by the function 'current-left-margin'.

Open New Line in Context See also: • Σ **Whitespace**	**C-o**	(**c-context-open-line**)	Insert a line break suitable to the context and leave point before it. • This is the '**c-context-line-break**' equivalent to '**open-line**', which is normally bound to **C-o**. See 'c-context-line-break' for the details. ☝ Normally C-o is bound to open-line. PEL rebinds it to c-context-open-line for the CC modes. If you want to open the line without indenting the next use open-line via **<f12> C-o**
Open new line	• **<f12> C-o** • **M-<f12> C-o**	(**open-line** N)	Insert a newline and leave point before it. • If there is a fill prefix and/or a 'left-margin', insert them on the new line if the line would have been blank. • With arg N, insert N newlines.
D comments	2 more characters have electric behaviour: / and * to help support comments in D. D supports the following types of comments (only the first 2 are explicitly supported by Emacs): • Block Comments: **/*** comment ***/** • Line Comments: **//** comment to end of line • Nesting Block Comments: **/+** nesting /+comments +/ can span multiple lines and surround // style comment **+/** • Documentation Comments: Use several prefixes: **///** , **/**** multi-line documentation */ , and **/++** multi-line documentation **+/**		

/	**/**	(**c-electric-slash** ARG)	Insert a slash character.
			• If the slash is inserted immediately after the comment prefix in a c-style comment, the comment might get closed by removing whitespace and possibly inserting a "*". See the variable 'c-cleanup-list'.
			• Indent the line as a comment, if:
			  1. The slash is second of a "//" line oriented comment introducing token and we are on a comment-only-line, or
			  2. The slash is part of a "*/" token that closes a block oriented comment.
			• If a numeric ARG is supplied, point is inside a literal, or 'c-syntactic-indentation' is nil or 'c-electric-flag' is nil, indentation is inhibited.

*	*	(**c-electric-star** ARG)	Insert a star character.
			• If 'c-electric-flag' and 'c-syntactic-indentation' are both non-nil, and the star is the second character of a C style comment starter on a comment-only-line, indent the line as a comment.
			• If a numeric ARG is supplied, point is inside a literal, or 'c-syntactic-indentation' is nil, this indentation is inhibited.
			With this key it becomes easy to type the following two styles of multi-line block comment:
			
			`/* Two star`
			`** continuation`
			`** prefix for`
			`** multi-line`
			`** C comment.`
			`*/`
			
			`/* Single star`
			`  * prefix for`
			`  * multi-line`
			`  * C comment.`
			`  */`
			When typing the '*' at the beginning of the line, it indents automatically. If another '*' is typed, indentation is set to allow a two-star continuation, otherwise it is placed for a single star continuation.

Description	Keystroke	Function	Note
Comment/un-comment See also: Σ **Comments** **With PEL**: Comment the current line with **M-0 M-;**	**M-;**	(**comment-dwim** ARG)	Comment line or region with **//** or **/* */** style comments depending on the comment style currently used in the buffer.
		(**pel-comment-dwim** ARG)	Same as **comment-dwim** but comments the current line with a numeric ARG or 0.

	• When no marked region and no comment:
	  • On empty line: insert comment starter at the proper indentation level. Typed again: move it toward end of line.
	  • On line with code: insert comment starter after the code for an end-of-line comment
	• With marked un-commented region: Comment region (each line is commented)
	• With marked commented region: Removes the comment.
	• Call the comment command you want (Do What I Mean).
	  • If the region is active and 'transient-mark-mode' is on, call 'comment-region' (unless it only consists of comments, in which case it calls 'uncomment-region'). Else, if the current line is empty, call 'comment-insert-comment-function' if it is defined, otherwise insert a comment and indent it. Else if a prefix ARG is specified, call 'comment-kill'. Else, call 'comment-indent'.
	• With numeric argument: comment current line. **M-0 M-;**
	• You can configure 'comment-style' to change the way regions are commented: see **<F12> M-;** to toggle the comment style.		

	C-c C-c	(**comment-region** BEG END &optional ARG)	Comment or uncomment each line in the region.
			  • With just **C-u** prefix arg, uncomment each line in region BEG .. END.
			  • Numeric prefix ARG means use ARG comment characters.
			  • If ARG is negative, delete that many comment characters instead.
			• The strings used as comment starts are built from '**comment-start**' and '**comment-padding**'; the strings used as comment ends are built from '**comment-end**' and 'comment-padding'.
			• By default, the '**comment-start**' markers are inserted at the current indentation of the region, and comments are terminated on each line (even for syntaxes in which newline does not end the comment and blank lines do not get comments). This can be changed with '**comment-style**'.
			☝ If you try this when no region is marked and the **/* */** style comments is active, the comment ends on the next space, which is probably not what you want. The command comment-dwim works better.

Hide/Show comments See also: Σ **Comments**	**<f11> ; ;**	(**hide/show-comments-toggle** &optional START END)	Toggle hiding/showing of comments in the active region or whole buffer. • If the region is active then toggle in the region. Otherwise, in the whole buffer. 📦 This requires the **hide-comnt.el** package (see Σ **Comments**). PEL activates it when the **pel-use-hide-comnt** user option is **t**.

Fill current paragraph
See also:
Σ **Filling/Justification**
	• **M-q**
	• **<f12> F**
	• **M-<f12> F**
	• **<f11> SPC D F**
		(**c-fill-paragraph** &optional ARG)	Like **<f11> t f p** but handles **//** and **/* */** style comments.
			• If any of the current line is a comment or within a comment, fill the comment or the paragraph of it that point is in, preserving the comment indentation or line-starting decorations (see the 'c-comment-prefix-regexp' and 'c-block-comment-prefix' variables for details).
			• If point is inside multiline string literal, fill it. This currently does not respect escaped newlines, except for the special case when it is the very first thing in the string. The intended use for this rule is in situations like the following:
			
			`char description[] = "\`
			`A very long description of something that you want to fill to make nicely formatted output.";`
			
			• If point is in any other situation, i.e. in normal code, do nothing.
			• Optional prefix ARG means justify paragraph as well.

Hungry Deletion of Whitespace	The CC mode provides two commands that can perform "hungry whitespace deletion" that can also be used in every mode.
	• ☝ PEL provides the convenient keys with the **<f11>** prefix keys for those 2 commands, available in **all** modes.
	• 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 **<DEL>** and **C-d**, to perform hungry deletions. That's not currently supported in other modes.
	  • When the Hungry Delete Mode is on, the mode-line displays a 'h' to the right of the '//l' indication of electric mode.
	  • The Hungry Mode also activates the key prefixes below that start with **C-c**. 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).
	  • 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 **<f12> M-DEL** key for those modes (like D).		

Delete preceding char or all preceding whitespace. See also: • Σ **Cut & Paste**	• **C-c DEL** • **C-c ⌫** • **C-c C-⌫** • **C-c C-<backspace>** • **C-c C-DEL** • **<f11> ⌫ ⌫** • **<f11> DEL DEL**	(**c-hungry-delete-backwards**)	Delete the preceding character or all preceding whitespace back to the previous non-whitespace character. ☞ In terminal mode, even though **C-⌫**, **C-<backspace>** and **C-DEL** 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. ☝ With PEL, the **<f11> ⌫ ⌫** 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.
Delete next char or all following whitespace. See also: • Σ **Cut & Paste**	• **C-c C-d** • **C-c ⌦** • **C-c C-⌦** • **C-c C-<delete>** • **<f11> ⌦**	(**c-hungry-delete-forward**)	Delete the following character or all following whitespace up to the next non-whitespace character. ☞ In terminal mode, even though **C-⌦** and **C-<delete>** 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. ☝ With PEL, the **<f11> ⌦** 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.
Indentation	All syntactic indentation control for D is controlled by the CC-Mode logic and provided commands listed below. • Rigid indentation commands are also available and listed at the end of this list. They are also listed in the Σ **Indentation** table.		
Indent current line or region See also: • Σ **Indentation**	**<tab>**	(**c-indent-line-or-region** &optional ARG REGION)	Indent active region, current line, or block starting on this line.

	• Behaviour depends on syntactic-indentation mode (enabled by default but can be toggled on/off with the **<f12> M-i** key):
	  • With syntactic-indentation on (the default):
	  • In Transient Mark mode, when the region is active, reindent the region.
	  • Otherwise, with a prefix argument, rigidly reindent the expression starting on the current line. Otherwise reindent just the current line.
	  ☝ This might seem strange for new Emacs users, but it ends up being very useful. You can type **<tab>** anywhere in the line to adjust its indentation or everything in the marked area if a block is marked.
	  • With syntactic-indentation off:
	  • <tab> always indent current line by one level
	  • **C-u - <tab>** or **M- <tab>** always un-indent current line by one level
	  • Indenting marked region is done without syntax knowledge and at the same level as previous line.
	• ☝ If you want to indent rigidly you can use:
	• (**pel-indent-rigidly &optional N**) (bound to **C-x <tab>** and to **<f11> <tab><tab>**) to indent the line or region rigidly.
	• (**tab-to-tab-stop**), bound to **M-i** to insert spaces to the next tab stop column.		

Indent lines of list after point See: Σ **Indentation**	**C-M-q**	(**indent-pp-sexp** &optional ARG)	Indent each line of the list starting just after point, or pretty-print it. • A prefix argument (**C-u**) specifies pretty-printing. Pretty-printing essentially uses more lines as it places the beginning of each list on a new line.
Indent current function or class	**C-c C-q**	(**c-indent-defun**)	Indent the content of the current top-level function or class. Leaves point unchanged.

Indent a region	**C-M-**	(**indent-region** START END &optional COLUMN)	Indent each nonblank line in the region.
			• A numeric prefix argument specifies a column: indent each line to that column.
			• With no prefix argument, the command chooses one of these methods and indents all the lines with it:
			  1. 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.
			  2. If 'indent-region-function' is non-nil, call that function to indent the region.
			  3. Indent each line via 'indent-according-to-mode'.

	☝ When a region is marked you can also use the simple **<tab>** to do the same when syntactic-indentation is active.		

Description	Keystroke	Function	Note
Non Syntactic Indentation	Emacs provides the following command to indent without regards to semantics. More information on indentation is available in the ∑ **Indentation** table. For most editing scenarios, it's best to set **pel-d-tab-width** and **pel-d-indent-width** 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.		
Insert spaces or tabs to next defined tab-stop column See also: • ∑ **Indentation**	**M-i**	(**tab-to-tab-stop**)	Insert spaces or tabs to next defined tab-stop column. • The exact location of the next tab stop is identified by the value of the **tab-stop-list** and **tab-width** for the current buffer. • With PEL, the tab-stop interval is controlled by the value of **pel-d-tab-width.** • PEL sets **tab-width** to the value of pel-d-tab-width for each d-mode buffer.

Indent/Unindent rigidly
 	
See also:
• ∑ **Indentation**
• ∑ **Key-Chords**
	• **C-x <tab>**
	• **<f11> <tab> <tab>**
	• **<tab>q**
		(**pel-indent-rigidly &optional N**)
		————————————
		PEL uses the above instead of the standard:
		
		(**indent-rigidly** START END ARG &optional INTERACTIVE)
			Indent rigidly the marked region or current line N times.
			• **If a region is marked**, it uses 'indent-rigidly' and provides the same prompts to control indentation changes.
			• **If no region is marked**, it operates on current line(s) identified by the numeric argument N (or if not specified N=1):
			  • N = [-1, 0, 1] : operate on current line
			  • N > 1 : operate on the current line and N-1 lines below.
			  • N < -1 : operate on the current line and (abs N) -1 lines above.
			PEL rebinds this key, but it extends the functionality: pel-indent-rigidly uses indent-rigidly, described below the dashed line.
			————————————————————————————
			Indent all lines starting in the region.
			• If called interactively with no prefix argument, activate a transient mode in which the indentation can be adjusted interactively by typing **<left>**, **<right>**, **S-<left>**, or **S-<right>**.
			————————————————————————————
			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 *indent-rigidly-map* with by default provides:
			  • **S-<right>** indent-rigidly-right-to-tab-stop
			  • **S-<left>** indent-rigidly-left-to-tab-stop
			  • **<right>** indent-rigidly-right
			  • **<left>** indent-rigidly-left
			Typing any other key deactivates the transient mode.
			• The **S-<right>** and **S-<left>** keys indent/de-indent to the next tab-stop position, which is controlled by the **tab-width** user option.
			• With PEL, the tab-stop interval is controlled by the value of **pel-d-tab-width.**
			  • PEL sets **tab-width** to the value of pel-d-tab-width for each d-mode buffer.

	If you use the cua-mode: the cua-mode uses **C-x**, to invoke this command when cua-mode is active, type it really fast or type **C-x C-x <tab>** (or use the PEL binding **<f11> <tab> <tab>**).		

Indent line(s) rigidly
 	
See also:
• ∑ **Indentation**
	• **<f6> <tab>**
	• **<f11> <tab> c**
		(**pel-indent-lines** &optional N)	Indent current or marked lines by N indentation levels.
			• Works with point anywhere on the line.
			• All lines touched by the region are indented.
			• A special argument N can specify more than one indentation level. It defaults to 1.
			• If a negative number is specified, 'pel-unindent-lines' is used.
			• If a region is marked, the function does not deactivate it to allow repeated execution of the command. It also modifies the region to include all characters in all affected lines.
			• Use **C-g** to de-activate the region.
			• Handles presence of hard tabs:
			  • If indent-tabs-mode is non-nil the indentation is created with a mix of hard-tabs and space characters.
			  • If indent-tabs-mode is nil, any hard tab in the indentation of the marked lines is replaced by the proper number of spaces. Hard tabs after first non-whitespace character on the line are left.

Un-indent line(s) rigidly
 	
See also:
• ∑ **Indentation**
	• **<backtab>**
	• **<f6> <backtab>**
	• **<f11> <tab> C**
		(**pel-unindent-lines** &optional N)	• Un-indent current line or marked lines by N indentation levels.
			• Works with point is anywhere on the line.
			• All lines touched by the region are un-indented.
			• If region was marked, the function does not deactivate it to allow repeated execution of the command.
			• If a region was marked, the function does not deactivate it to allow repeated execution of the command. It also modifies the region to include all characters in all affected lines
			• Use **C-g** to de-activate the region.
			• Handles presence of hard tabs:
			  • If indent-tabs-mode is non-nil the indentation is created with a mix of hard-tabs and space characters.
			  • If indent-tabs-mode is nil, any hard tab in the indentation of the marked lines is replaced by the proper number of spaces. Hard tabs after first non-whitespace character on the line are left.

Marking	Emacs provides the following command to quickly mark the whole content of the current function. More mark commands exists, see the ∑ **Marking** table.		
Mark the complete function body See also: ∑ **Marking**	**C-M-h**	(**c-mark-function**)	Mark complete function. • Put mark at end of the current top-level declaration or macro, point at beginning. • 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. • A mark is left where the command started, unless the region is already active (in Transient Mark mode). • As opposed to C-M-a and C-M-e, this function does not require the declaration to contain a brace block.
Getting Syntactic Information	Use the following commands to extract syntactic information from the source code.		
Display name of current function	• **C-c C-z** • **<f12> f** • **M-<f12> f**	(**c-display-defun-name** &optional ARG)	Display the name of the current CC mode defun and the position in it. • With a prefix arg, push the name onto the kill ring too.
Highlighting blocks	The following commands can be used to activate or toggle useful modes to highlight blocks of (), {}, and []. • show-paren-mode, which highlights the parens that matches the one before or after point. • rainbow delimiters mode, where matching nested parens are highlighted with the same colour.		
Toggle show-paren mode on/off See also: ∑ **Highlight**	**<f12> h (** **<f11> h (**	(**show-paren-mode** &optional ARG)	Toggle visualization of matching parens (Show Paren mode). • With a prefix argument ARG, enable Show Paren mode if ARG is positive, and disable it otherwise. • 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.
Enable/Disable coloured highlight of nested blocks (),{},[] See also: ∑ **Highlight**	**<f12> h)** **<f11> h)**	(**rainbow-delimiters-mode** &optional ARG)	Highlight nested parentheses, brackets, and braces with different colours according to their depth. • Customize the depth and colours with **M-x customize-group rainbow-delimiters** **Requires:** rainbow-delimiters.el PEL activates this when the **pel-use-rainbow-delimiters** user option is set to **t**.

Description	Keystroke	Function	Note
Navigation in D See also: Σ **Navigation**	Emacs provides commands to navigate across source of curly bracket programming languages like D. Most commands are specialization of the normal navigation commands which are described in the table Σ **Navigation**, along with the other commands that are also available. The list below describe the specialized commands only. See the others inside Σ **Navigation**, like the navigation by blocks, very useful in D.		
• **By functions** • **By structures**	• Move to beginning /end of function definition blocks or structure definition blocks. • Jump over comments. • ☝ When point is located before opening brace or right after closing brace and **show-paren-mode** is on, the matching parentheses are highlighted.		
Forward to start of next top level function or struct	**<f6> <down>** **<f12> <down>**	(**pel-beginning-of-next-defun** &optional SILENT DONT-PUSH_MARK)	Move forward to the beginning of the next function or type definition. • Move point before the function type or the struct or typedef keyword. • Beeps if does not find beginning of next function unless SILENT is non-nil. • If the beginning of next function is found, push the start location to the mark ring unless DONT-PUSH_MARK is non-nil. • Move back to previous position with **M-`** or **<f6><f6>**. ☛ Shift marking is available. With **<f6>** and **<f12>** hit Shift after function key, before cursor key. ☝ This command complements what end-of-defun does. • 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.
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. • With argument, do it that many times. Negative argument -N means move back to Nth preceding end.
	• **C-M-<end>** • **<f6> <right>** **<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. ☛ Shift marking is available. With **<f6>** and **<f12>** hit Shift after function key, before cursor key. ⚠ This command moves to the end of the next **top-level** function. It skips nested functions.
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. • With a positive argument, move backward that many functions or structures. A negative argument -N means move forward to the Nth following beginning.
	• **C-M-<home>** • **<f6> <up>** **<f12> <up>**	(**beginning-of-defun** &optional ARG)	Move backward to the beginning of function or type definition. • Move point before the function type or the struct or typedef keyword. • With ARG, do it that many times. Negative ARG means move forward to the ARGth following beginning of defun. ☛ Shift marking is available. With **<f6>** and **<f12>** hit Shift after function key, before cursor key. ⚠ 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.
Backward to end of previous top level function or struct	**<f6> <left>** **<f12> <left>**	(**pel-end-of-previous-defun** &optional SILENT DONT-PUSH_MARK)	Move backwards to the end of the previous function or type definition. • Beeps if does not find end of previous function unless SILENT is non-nil. • If the end of previous function is found, push the start location to the mark ring unless DONT-PUSH_MARK is non-nil. • Move back to previous position with **M-`** or **<f6><f6>**. ☛ Shift marking is available. With **<f6>** and **<f12>** hit Shift after function key, before cursor key. ⚠ In some cases it fails to detect the end of the previous block and fails. 🚧
• **By blocks**	• Move across D statements and D 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: Σ **Navigation**	**C-M-p**	(**backward-list** &optional ARG)	Move backward across one balanced group of parentheses. • This command will also work on other parentheses-like expressions defined by the current language mode. • With ARG, do it that many times. • Negative arg -N means move forward across N groups of parentheses. • This command assumes point is not in a string or comment. • **C-M-p** : ☛ Shift marking is available in graphics mode, not in terminal mode.
Move block backward See also: • Σ **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). • 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. • **C-M-b** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **C-M-<left>** : ☛ Shift marking works with this command. • ⚠ With PEL: if you want to use **Esc C-<left>** binding you must ensure that **pel-windmove-on-esc-cursor** user option is set to nil, otherwise it does something else. ❖ **C-M-<left>** does not work on Windows, but **H-<left>** works. 🐧 Several Linux distros map **C-M-<left>** to desktop workspace operation. In that case you can either use another key binding or change Linux key binding in Systems->settings->keyboard->shortcuts to prevent it from using that key sequence.
Forward block/list See also: Σ **Navigation**	**C-M-n**	(**forward-list** &optional ARG)	Move forward across one balanced group of parentheses. • This command will also work on other parentheses-like expressions defined by the current language mode. • With ARG, do it that many times. • Negative arg -N means move backward across N groups of parentheses. • This command assumes point is not in a string or comment. • **C-M-n** : ☛ Shift marking is available in graphics mode, not in terminal mode.
Move block forward See also: • Σ **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). • 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. • **C-M-f** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **C-M-<right>** : ☛ Shift marking works with this command. • ⚠ With PEL: if you want to use **Esc C-<right>** binding you must ensure that **pel-windmove-on-esc-cursor** user option is set to nil, otherwise it does something else. ❖ **C-M-<right>** does not work on Windows, but **H-<right>** does. 🐧 Several Linux distros map **C-M-<right>** to desktop workspace operation. In that case you can either use another key binding or change Linux key binding in Systems->settings->keyboard->shortcuts to prevent it from using that key sequence.
• **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: • Σ **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. • 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. • ⚠ With PEL: if you want to use **Esc C-<up>** binding you must ensure that **pel-windmove-on-esc-cursor** user option is set to nil. • **C-M-u** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **C-M-<up>** : ☛ Shift marking works with this command. ❖ **C-M-<up>** does not work on Windows, but **H-<up>** does.
Forward Up/outside sexp hierarchy See also: Σ **Navigation**	**C-M-]**	(**up-list** &optional ARG ESCAPE-STRINGS NO-SYNTAX-CROSSING)	Move forward out of one level of parentheses or nested blocks. • 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 backward but still to a less deep spot.

Description	Keystroke	Function	Note
Down/inside sexp/block See also: • **Navigation**	• **C-M-d** • **C-M-<down>** • **C-[C-d** • **Esc C-d** • **Esc C-<down>** ⚠	(**down-list** &optional ARG)	Move forward down one level of parentheses. • 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 backward but still go down a level. • This command assumes point is not in a string or comment. • ⚠ With PEL: if you want to use **Esc C-<down>** binding you must ensure that **pel-windmove-on-esc-cursor** user option is set to nil. • **C-M-d** : Shift marking is available in graphics mode, not in terminal mode. • **C-M-<down>** : Shift marking works with this command. ❖ **C-M-<down>** does not work on Windows, but **H-<down>** does.
• **By statements**	Move to beginning /end of statement of comment sentence.		
Go to beginning of statement	**M-a**	(**c-beginning-of-statement** &optional COUNT LIM SENTENCE-FLAG)	Go to the beginning of the innermost statement. • With prefix arg, go back N - 1 statements. • If already at the beginning of a statement then go to the beginning of the closest preceding one, moving into nested blocks if necessary (use **C-M-b** to skip over a block). If within or next to a comment or multiline string, move by sentences instead of statements.
Go to the end of statement	**M-e**	(**c-end-of-statement** &optional COUNT LIM SENTENCE-FLAG)	Go to the end of the innermost statement. • With prefix arg, go forward N - 1 statements. • Move forward to the end of the next statement if already at end, and move into nested blocks (use **C-M-f** to skip over a block). If within or next to a comment or multiline string, move by sentences instead of statements.
Rendering markup embedded in comments	The following commands are used to create images from specific markup code embedded inside D 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 **Graphviz Dot**		

Preview UML diagram from plantUML source in current plantUML region of commented source code
 	
See also: **PlantUML**
	<f12> u	(**pel-render-commented-plantuml** PREFIX &optional POS)	Render the PlantUML markup embedded in current mode comment.
			• Use region if identified otherwise use PlantUML block at point.
			• Uses prefix (as PREFIX) to choose where to display it:
			  • 4 (when prefixing the command with **C-u**) -> new window
			  • 16 (when prefixing the command with **C-u C-u**) -> new frame.
			  • else -> new buffer
			• This can be used inside buffer using **any** major mode, when PlantUML markup is embedded inside source code comment.
			☝ 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.
			📦 Requires the **plantuml-mode** external package, activated by **pel-use-plantuml** user option being non-nil.

Emacs & D— References

Document	Notes
The D Programming Language	
D (programming language) - Wikipedia	Overview of D
D Home Page	
D Home Page - Documentation	Links to the Language Reference, Library Reference, Command-line Reference (macOS), Feature Overview and Articles.
DUB - The D Package Registry	Browsable/searchable list of packages

The D Style - D Code Guideline	This document provides a set of style conventions promoted by the D community. Several items in this guideline identify stylistic aspects that can be configured in Emacs. Some of them are listed here:
	• **Indentation**:
	  • spaces instead of tabs ➢
	  • indentation level: 4 columns ➢ c-basic-offset = 4
	• **Line Length** : soft limit of 80, hard limit of 120. They can exceed 80 columns but never 120.
	• **Brackets style**:
	  • Use the Allman style (also called BSD style) where each brace is on their own line. ➢ add: (d-mode . "bsd") to c-default-style
	• **Whitespace in statements**:
	  • 1 space after **for**, **foreach**, **if**, **while** and **version** keyword and the opening parenthesis: `if (x) { ... }`
	  • 1 space between binary operators, assignments, casts, lambdas.
	  • No space between unary operators, after assert, function calls, function definition name.
	• **Naming Conventions**:
	  • Constant, enums, variable and function names should be camelCased.
	  • User defined type names should be PascalCased.

The Next Big Programming Language You've Never Heard Of | WIRED - 2014	D is a very nice language, unfortunately it never got the attention could have got if it had some big corporate backup. Interview with Andrei Alexandrescu discussing his encounter with Walter Bright and the D language.

Emacs Support for D	Support for D for Emacs is based on:
	• Emacs D Mode
	• Code completion support that uses:
	  • A completion front end, either:
	  • Auto-Complete based using ac-dcd.
	  • Company based using company-dcd.
	  • Both of these depend on flycheck-dmd-dub, which uses DCD, the D Completion Daemon, written in D.
	  • Both require/use flycheck
	• D Unit test support: flycheck-d-unittest

Emacs D Mode	The main support for D. Available on MELPA as d-mode. The d-mode is based on cc-mode.
ac-dcd : Auto Complete D Code Completion via DCD backend	Available on MELPA as ac-dcd. • This project also recommend using yasnippet and popwin.
Company-DCD - Company D Code Completion via DCD backend • See also: **Customize**	Available on MELPA as company-dcd. • DCD is the D Completion Daemon (DCD @ Github). • For Emacs customization of company-dcd, see the **company-dcd** Emacs customization group (use **<f11> <f2> g company-dcd**)
flycheck-dmd-dub	Available from melba as flycheck-dmd-dub. • Flycheck support for D: reads D library dependency information from DUB (the D Package Registry). • To use it you must install DCD separately: see next row.
DCD: D Completion Daemon	• DCD instruction installation on the DCD Github page. • See also the DUB DCD page which has the same info as GitHub but also has internal documentation of the D code interfaces down to the source code. • On macOS, the dcd-client and dcd-server commands can be installed with Homebrew.
D Unit Test support: flycheck-d-unittest	Available on MELPA as flycheck-d-unittest. • Runs D unit test with "dmd -uninttest and -main options". • Takes advantage that D has built-in syntax and dmd support for unit test builds and runs. • The project has a wiki page, "Start D with Emacs", describing how to install Emacs support for D (but only describes d-mode and flycheck-d-unittest)
yasnippets for D	I have found the following: • Per Nordlöw snippets for D

Document	Notes
Emacs Support for Curly Bracket Programming Languages	The d-mode is based on the CC Mode. The CC Mode, a collection of libraries, provides support for several curly-bracket languages like C, C++, Java, Objective-C, Pike, AWK and it also applies to D. Several features of the CC Mode are used for the D support, so it's useful to be aware of them.
GNU Emacs CC Mode Manual	D is a curly-bracket programming language and therefore supported by Emacs CC Mode. It controls: • whether hard tabs or spaces are used: indent-tabs-mode • the number of columns per tab: tab-width • the indentation style (see indentation style meanings): c-default-style a-list with an entry for D PEL provides user options to activate the use of D in Emacs (pel-use-D) and user options for the tab, style to use and what CC modes are activated by default.
GNU Emacs Manual - C and Related Modes	The main Emacs manual also provides information on the support for C and similar programming languages, and these apply to the D programming language as well. The sections include: • Motion in C • Electric C • Hungry Delete • Other C Commands