↖ Topic Index

Programming Language Support — C++

Description	Keystroke	Function	Note

Editing C++ Files
○ Help & Customization
• cc-mode learn/modify
• C++ code help
○ cc-mode set/help
○ electric keys
• electric-pair mode
• insert new line(s)
○ C++ comments
• Delete whitespace
• Indentation
• non-syntactic indent
• indent rigidly
○ indent lines rigidly
• Open file at point
○ C++ skeletons
○ insert ( )
○ mark function
• show function name
• search support
• highlighting blocks
• navigate in C++ code
  • by xref
  • by call graph
  • by statement
  ○ by block
  ○ by preprocessor
○ Expand pre-processor
• Insert/align eol \
• Hide preprocessor
○ Preview UML
• C++ code search/fix
• Programming help
○ Info on C++
			Emacs supports C++ natively via the built-in **c++-mode.**
			This package extends the Emacs CC Mode built-in package which supports the curly-bracket programming languages like C++.
			• **Supported file extensions**: code files: `.cc, .C, .CC, .cpp, .cxx, .c++, .ii.`
			header files: `.h, .hh, .HH, .hpp, .hxx, .h++, .icc, .inl` The .icc and .inl are added by PEL.
			The content of .h file is analyzed to distinguish between C and C++ and activate the appropriate major mode.
			  • With PEL, you can add more file types by adding the association to the **pel-auto-mode-alist** user option.
			  • PEL adds these to Emacs **auto-mode-alist** user-option on startup.
			🎛️ When **pel-use-speedbar** is set all these extensions are recognized by speedbar, otherwise only the main ones are recognized. See Σ **Speedbar**
			🎛️ Important aspects of C++ source code syntax controlled by the CC Mode are customizable with PEL user option variables.
			PEL customization for C++: Simplifies editing C++ code configuration. (To change, use **pel-cfg-pkg-c++** with **<f12> <f2>**), see below).
			• Emacs customization group: **pel-pkg-for-c++** **See** Σ **Customize**
			  • **pel-c++-indent-width**: Identifies the number of columns used for indentation. Defaults to 3.
			  • **pel-c++-tab-width**: The width of a tab. Defaults to 3. This concept differs from indentation: you can have an indentation of 3 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 3.
			  • 🍏 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-c++-use-tabs**: Whether hard tabs are used in indentation or not: **t**: tabs are used, **nil**: only spaces are used. Default: **nil**.
			  • **pel-c++-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 "stroustrup".
			• Emacs customization group: **pel-pkg-for-cc**. Applies to all CC Mode related modes (like c-mode).
			  • **pel-cc-auto-newline**: Whether automatic newline mode is active on all CC Mode (including c-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 C** , **<f12>** and **M-<f12>**
			The first one is always available. The other two prefixes are only available in c++-mode buffers.
			The **M-<f12>** prefix helps the typing flow when the next key is a Meta key.
			For simplification, the **<f11> SPC C** prefix is normally omitted in the table.

Last updated on:	2025-12-23		🎛️ Tree-Sitter Support 🚧 🔄 when the **pel-use-tree-sitter** user-option is set to **t**, PEL provides tree-sitter support via:
			📦 **tree-sitter** and **tree-sitter-langs** external packages and Emacs built-in support. See **Tree-Sitter (@GitHub)**
			  • Emacs with dynamic module loading, and built with tree-sitter support. tree-sitter library must be installed separately.
			  • See: **How to Get Started with Tree-Sitter**
			  • Emacs must find the tree-sitter language dynamic library files that have a name similar to 'libtree-sitter-python.so' (for Linux) or .dylib (for macOS).
			  • Identify the relevant directory in the **pel-treesit-load-path** . See the docstring of that user-option for further instructions.
			⚠️🚧⚠️ As of Emacs 30 tree-sitter **c++-ts-mode** for C does not provide the flexibility of the **c++-mode** to control the C++ source code style.
			• It's recommended to use **c++-mode** to edit, navigate and manipulate C source code. At least until proper style control is added into c++-ts-mode.

			⌨️ PEL provides the following set of **mode-specific key prefixes**: **<f11> SPC C** as well as **<f12>** and **M-<f12>**
Open this PDF file. See also: Σ **Help/Info**	**<f11> SPC C <f1>** **<f12> <f1>**	(**pel-help-pdf** &optional OPEN-WEB-PAGE)	Open the 𝔓𝔈𝔏 - C++ 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 C++ support	**<f11> SPC C <f2>** **<f12> <f2>**	(**pel-customize-pel** &optional OTHER-WINDOW)	Customize PEL C++ support: cpp. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
Σ **Customize** Emacs C++ support	**<f11> SPC C <f3>** **<f12> <f3>**	(**pel-customize-library** &optional OTHER-WINDOW)	Customize Emacs C++ support: cpp. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
CC Mode Style Management • **Learn style used in current buffer**			Automatic indentation, brace format style and several other C/C++ 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.
Show syntactic information for current line	**C-c C-s**	(**c-show-syntactic-information** ARG)	Show syntactic information for current line. • Display the syntactic information list and highlight the reference position(s) listed as argument to the syntactic list. • Each list starts with a **syntactic symbol** with zero or several reference positions. • 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	**<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 and install it.	**<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.
C++ Code Help			There are several Emacs extension packages that can help writing C/C++ code.
Get man help about C code See: Σ **Help/Info**	• **<f11> ? m** • **M-<f8>** • **⌘-M**	(**man** MAN-ARGS)	Open a Man page inside an Emacs window. See Σ **Help/Info** for more info about man. • Inside a C++ buffer, you can use it to request man help about a C function or structure. • A large amount of information about C library code is available in man form under the various Unix-like platforms.

Description	Keystroke	Function	Note
CC Mode support Behaviour control ☝ Use **<f12> <f4> ?** to display the current state.	Use following commands 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 and what happens when electric characters are typed (when electric mode is activated). • **CC Mode state displayed in the mode line**: ⁑C{…} where: • ⁑ 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		
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-c++-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. Toggle if no ARG or if ARG is 0. • With positive ARG turn on syntactic indentation, turns it off when negative. • With syntactic indentation turned on (the default), the indentation functions and telectric keys indent according to 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.
Toggle Hungry Delete mode †	**<f12> <f4> DEL**	(**c-toggle-hungry-state** &optional ARG)	Toggle hungry-delete-key feature. Affects **<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.
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: toggles how '**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**)
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***. • With auto-newline enabled (indicated by "/la" on mode line after the mode name) newlines are automatically inserted after characters such as brace, comma, semi-colon, and colon.
Change RET key behaviour: select return mode. †	**<f12> <f4> RET**	(**pel-cc-change-newline-mode**)	Change the RET key behaviour 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** : default : uses (**c-context-line-break**) with ability to repeat with an arg. • **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 & optional APPEND**)	Display information about current **CC mode** derivative for the current c++-mode buffer inside a *pel-c++-info* buffer. It has buttons to access important customizable user-options.

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-c++-tab-width** and **pel-c++-use-tabs**.
  - In a c++-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-c++-indent-width** in PEL and whether syntactic indentation mode is active. Shows how it is set and whether it was override by executing the **pel-cc-set-indent-width** command for this buffer (use **<f12> TAB**) for that command.
- 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 "stroustrup", identifying the **Stroustrup C++ style** used by C++ designer, Bjarne Stroustrup. 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.
  - 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**).
- The file search methods and parameters used by **pel-open-at-point** (see sections below).

☝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.

```
----*c++ Control from cpp-code.cpp --- Friday, April 25, 2025 @ 17:57:08 -----
c++-mode state:
- active style        : stroustrup. c-default-style: (stroustrup bsd)
- RET mode            : context-newline, and aligns (comments, assignments, etc...)
- Electric characters : active on: #*/<>(){}:;,
- Auto newline        : on
- fill column         : 80, auto-filling: off.
- Tab width           : 4                  Set via: pel-c++-tab-width(4)    ==> tab-width(4)          when c++-mode buffer is opened.
- Indentation chars   : spaces only        Set via: pel-c++-use-tabs(nil)   ==> indent-tabs-mode(nil) when c++-mode buffer is opened.
- Indent width        : 3                  Set via: pel-c++-indent-width(3) ==> c-basic-offset(3)     when c++-mode buffer is opened.
- Syntactic indent    : on
- c-indentation-style : stroustrup
- PEL Bracket style   : stroustrup
- Comment style       : Line (C++-style) comments: //
- Hungry delete       : off, but the F11-⌦  and F11-⌫  keys are available.
- Project root        : None found, searching for files identified in pel-project-root-identifiers: (.git .hg .projectile .pel-project)
- File finder method  : generic
-  pel-ffind-executable: fd
```

Description	Keystroke	Function	Note
Electric Keys and Keywords	The following **electric C/C++ characters** have special meaning when the electrical state is active in a buffer using c++-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.
()	• **( )**	(**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 'c-cleanup-list'.
<>	• **< >**	(**c-electric-lt-gt** ARG)	If the current language uses angle bracket parens (e.g. template arguments in C++), try to find out if the inserted character is a paren and give it paren syntax if appropriate. • If 'c-electric-flag' and 'c-syntactic-indentation' are both non-nil, the line will be reindented if the inserted character is a paren or if it finishes a C++ style stream operator in C++ mode. Exceptions are when a numeric argument is supplied, or the point is inside a literal.
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(ǀ)		
Toggle electric-pair-mode in current buffer † **Lighter:=** ε(ǀ)	**<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 prefix argument ARG, enable Electric Pair mode if ARG is positive, 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** (PEL default for RET) • **newline** (Emacs default for RET) • **electric-indent-just-newline** • If variable 'pel-newline-does-align' is t, perform 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. 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. • 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.

Description	Keystroke	Function	Note
C++ Comments	2 more characters have electric behaviour: / and * to help support comments in C++. C++ supports 2 types of comments: • Block Comments: **/*** comment ***/** • Line Comments: **//** comment to end of line		
/	**/**	(**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.

Comment/un-comment See also: Σ **Comments** **With PEL**: Comment the current line with **M-0 M-;**	**M-;**	(**pel-c-comment-dwim** ARG)	Comment line or region with **//** or **/* */** style comments depending on the comment style currently used in the buffer. • 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'. • You can configure 'comment-style' to change the way regions are commented: • See **<f12> <f4> M-;** to toggle the comment style. • With numeric argument: comment current line. **M-0 M-;**
	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.

Fill current paragraph
See also:
Σ **Filling/Justification**
	• **M-q**
	• **<f12> F**
	• **M-<f12> F**
	• **<f11> SPC C 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.

Toggle subword-mode See also: • Σ **Text Modes**	• **<f11> t m b** • **<f12> M-b** • **M-<f12> M-b**	(**subword-mode** &optional ARG)	Toggle subword-mode: a minor mode that treats sections of camelCase and PascalCase as distinct words. • With a prefix argument ARG, enable Subword mode if ARG is positive, and disable it otherwise.
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**.

Description	Keystroke	Function	Note

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 ‘//’ 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 C++).
	• Toggle hurry deletion mode of the **DEL** and **C-d** key for the current buffer with **c-toggle-hungry-state** (**<f12> M-DEL**).		

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: on by default, toggled with **<f12> M-i**
			• 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 current line.
			  • Otherwise reindent just the current line.
			  ☝ Hit **<tab>** *anywhere* in the line to adjust the indentation of the line or marked area.
			• 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
			  • Marked region is indented without syntax knowledge 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 also: • 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.

Non Syntactic Indentation	Emacs provides the following command to indent without regards to semantics. More information on indentation is available in the Σ **Indentation** table. 🍏 For most editing scenarios, it’s best to set **pel-c++-tab-width** and **pel-c++-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: • Σ **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-c++-tab-width.** • PEL sets **tab-width** to the value of pel-c++-tab-width for each c++-mode buffer.

Indent/Unindent rigidly
 	
See also:
• Σ **Indentation**
• Σ **Key-Chords**
	• **C-x <tab>**
	• **<f11> <tab> <tab>**
	• **<tab>q**
		(**pel-indent-rigidly &optional N**)	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 the original indent-rigidly.
	indent-rigidly 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-c++-tab-width.**
	  • PEL sets **tab-width** to the value of pel-c++-tab-width for each c++-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>**).		

Description	Keystroke	Function	Note

Indent line(s) rigidly
 	
See also:
• Σ **Indentation**
	• **<f6> <tab>**
	• **<f11> <tab> c**
		(**pel-indent-lines** &optional N)	Indent current or marked lines by N indentation levels controlled by **pel-c++-indent-width**.
			• 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:
• Σ **Indentation**
	• **<backtab>**
	• **<f6> <backtab>**
	• **<f11> <tab> C**
		(**pel-unindent-lines** &optional N)	• Un-indent current line or marked lines by N indentation levels controlled by **pel-c++-indent-width**.
			• 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.

Open file at point			The following command allow opening files from the file name taken at point (the cursor location).
			• In a c++-mode buffer the command is specialized to be more useful for C++ programming and has the extra capability of searching files where header files are stored. The search method is controlled by the following user-options:
			  • **pel-c-file-finder-method** : identifies one of 4 supported method of identifying the header files. See their descriptions below.
			  • **pel-c-file-searched-extra-dir-trees** : List of extra directory trees also searched by the tool identified by **pel-ffind-executable** user-option.
			  • **pel-c-file-finder-ini-tool-name** : The name of a tool chain ***TTT***, to select one of the ***TTT-c-path*** tool-chain key inside the [file-finder] section of the ***pel.ini*** file, a INI-format configuration file. The value mapped to that key identifies the list of directories to search for that tool-chain. The name of the tool chain can be overridden by the value of the environment variable PEL_CC_FIND_TOOLCHAIN, which is read and used when Emacs starts up (or pel-init is executed). Use the command **pel-cc-set-file-finder-ini-tool-name** to change the currently used tool chain name.
			☝ Note that when using the Ido completion mode, it is possible to instruct Ido to use a file name at point as the basis for the file name to open. This Ido behaviour is controlled by the **ido-use-filename-at-point** user-option. With PEL you can control it globally or locally with **<f11> f M-.**

Show active file finder setup for current buffer	• **<f12> <f4> f** • **<M-f12> <f4>**	(**pel-cc-find-show-status** &optional APPEND)	Print C++ specific PEL file finding control user-options and variables info inside a *pel-cc-ffind-status* help-mode buffer. • Prints current state and values of relevant user-options and variables as buttons you can use to get more info and change the values of the user options. • Clear previous buffer content by default. Use prefix arg (like **C-u**) to append instead.
Change Tool search path • (when the pel-ini-file search method is used)	• **<f12> <f4> M-<f6>** • **<f12> <f4> <f54>**	(**pel-cc-set-file-finder-ini-tool-name** &optional TOOL-NAME)	Change activate value of tool-chain name key identified by value of **pel-c++-file-finder-ini-tool-name** user-option. The change is not persistent. • Only used when the **pel-c-file-finder-method** is set to **pel-ini-file.** In that case it effectively select a new set of tool-chain specific directories to search by **pel-open-at-point**. The directories are identified by the corresponding ***TTT-c-path*** *key in the* **[file-finder]** section of the ***pel.ini*** file.

Open file or web-page whose name is at point ★★
 	
☝
🎛
C/C++ Header File finding control ☛
 	
pel-use-ini ☛
 	
Command is also specialized for:
• **reStructuredText**
• **C**
• **Erlang**
• **UNIX Shell**
 	
Generic Delimiting characters
 	
⚠ The complete file detection heuristic is described in the Σ **File mngt** description of the same command.
 	
Select target window ☛
 	
N>20 : open the directory ☛
 	
See function docstring for more info.
	• **M-***
	• **<f11> f .**
	• **6y**
		(**pel-open-at-point** &optional N)	Open the file, library or the URL, named at point, with potential line & column #s.
			• If necessary will search source code files in current project as specified by **pel-filename-at-point-finders** user-option. Type **<f12> <f4> ?** to **show current file search method**.
			🎛 Supports glob characters, partial directory path. When multiple files are found it prompts using the method selected by **pel-prompt-read-method** user-option.
			📦 The **6y** key-chord is available if **pel-use-key-chord** is non-nil. See Σ **Key-Chords**.
			
			This command works generically but is also specialized for C++ major mode: it opens the header file identified by the `#include` statement.
			Aside from generic method described below, the command searches for the header file to open using the method identified by the **pel-c-file-finder-method** and the **pel-c-file-searched-extra-dir-trees** user-options. The first one identifies one of the following search method, the other identifies extra directory tree(s) to search using the search tool identified by the **pel-ffind-executable** user-option:
			• **generic**: the command searches, in current directory and its parents, for a file identifying the parent root directory; a file with a name identified in the **pel-project-root-identifiers** user-option. Something like .git, .hg, .project or .pel-project by default. Then searches for files inside that directory tree.
			• **pel-ini-file**: the command searches inside directories identified by lists defined in the ***pel.ini*** file which PEL identifies for the project like it does for project marker. The ***pel.ini*** file is a **.INI file forma**t. When found, it is opened and information inside the file identifies where to search.
			• The file must contain a **[file-finder]** section with:
			  • The **project-path** key. The value is a list of directories to search recursively.
			  • One or several ***TTT-c-path*** key(s), where ***TTT*** is a tool-chain name. The value is a list of directories to search recursively for that tool-chain.
			  • The currently used tool chain is identified by the following values in order (first one takes priority on startup):
			  • The content of the PEL_CC_FIND_TOOLCHAIN environment variable, if it exists.
			  • The content of the **pel-c-file-finder-ini-tool-name** user-option; which identifies the name of a ***TTT-c-path*** key.
			  • The paths identified in the two lists may use environment variables inside the path strings. Use the $VARNAME format to identify them.
			  • You can modify this tool chain name anytime during an editing session by typing **<f12> <f4> M-<f6>** and specifying another name.
			  • With several ***TTT-c-path*** keys inside the ***pel.ini*** file, you can adjust the include path dynamically for various tool chains.
			• **environment variable name**: the name of an environment variable (like INCLUDE) that holds a list of directory names to search files in.
			  • Directories are not searched recursively for the last 2 options.
			• **explicit lists**: two lists of directory names: one list holds the project directory names, the other hold the tool and library directory names. The lists may identify directory names indirectly via environment variables. The $VARNAME format must be used. Directories are not searched recursively.
			In general the command extracts the file or directory name, and possibly line and column numbers, from text at point and tries to open the file or directory.
			• The generic mode extraction works by identifying the beginning & end of the file/directory/library/URL name string by delimiter characters, one of: tab, newline and: **"`'|()[]{}<>** ‘’“”「」 〔〕 〈〉 《》 【】 〖〗 «»‹›❮❯❬❭ 〘〙 ·。
			
			🎛 When finding several file names, the command lists them and prompts using the method selected by **pel-prompt-read-method** user-option.
			• The default is a very primitive function implemented by PEL. You can select a more powerful **ivy** prompting instead.
			  • With **ivy** selected, PEL will automatically set 📦 **pel-use-ivy** to **t** and **Ivy mode** will be installed automatically when you restart Emacs.
			• Note that the command shows all files found by the specified search method, it does not only use the first one found.
			  • ☝ Use this to detect potential duplication in header file names in large include paths.
			
			The command opens the file in the window selected by the following logic controlled by presence or absence of typed numerical prefix arguments:
			• **Select target window:**
			  • Without argument:
			  • If file or directory is already opened in a window, move point to that window and to the line column coordinates if specified following the file name at point.
			  • If no window holds that file, select the target window according to the number of editable windows in frame: if 1, split that window and use the new window, if 2: use the other window, if 3 or more, use the current window.
			  • With prefix numeric argument N:
			  • N < 0 : create a new window and use that.
			  • (abs N) > 20: then open the **directory** instead of the file. Interpret the window position from the N value adjusted: N-20 (or N+20 if N is negative)
			  • N = 0: use the '*other*' (the next) window.
			  • N = 1, 3, 7or above (excluding 8, 9 and 10): select the target window based on the number of editable windows in frame:
			  • if 1 window: split that window and use the new window,
			  • if 2 windows: use the other window,
			  • if 3 or more windows: use the current window.
			  • N is: 8: up, 2: down, 4:left, 5:current, 6:right.
			  • N is 9: force **opening the file in the OS associated application** (with N=29 or N=-29, open the file's directory with the OS associated application (eg. macOS Finder, Windows Explorer). If this is a URL, open it in the OS default web browser.
			• Selecting Minibuffer, inexistent or dedicated window is not allowed.

Open file with alternate extension Supports: • Σ **File-mngt** • **C**	**M-<f12> M-f**	(**pel-open-file-alternate**)	Open a file with same name but an alternate extension. • The new extension depends on the current file extension. • The list of alternate extensions is currently very limited and restricted to C and C++. If the alternate file is not found, save the file basename in the kill ring and prompt for the file name to open.

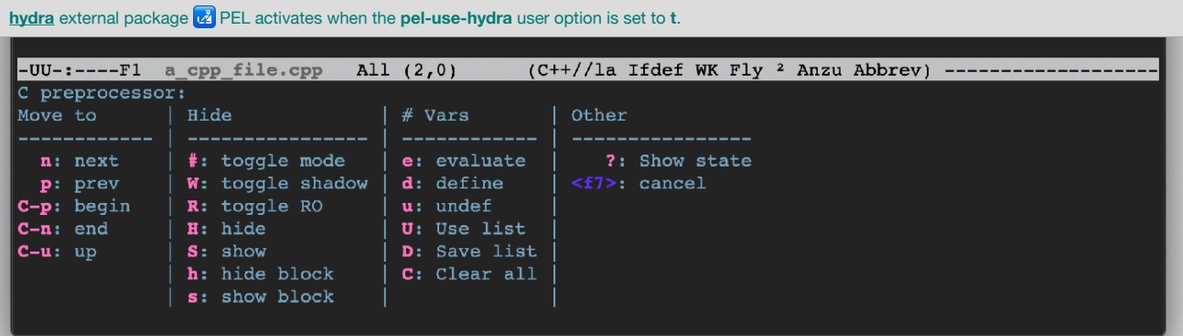
Description	Keystroke	Function	Note
Tempo skeletons for C++ See also: • **C Code Templates** 📘 as they also mostly similar to the templates for C++, although the C++ templates are separate/ independent from the C templates, the principles are the same. • Σ **Inserting Text** for more info and information about tempo skeleton and yasnippet template-based text insertion			PEL provides support for flexible text template insertion through the Emacs built-in **tempo skeleton** mechanism. • PEL creates key bindings to invoke the skeletons in the supported major modes, using the same key prefix sequence for each mode: **<f12> <f12>**, with the same key bindings for equivalent concepts (such as file header block) as much as possible. • 🎛 Several aspects of the PEL Emacs Lisp Source Code Style is controlled by the user options inside the **pel-c++-code-style** group. This group can be edited with **<f12> <f12> <f2>** from a C++ mode buffer and include the following options: • **pel-c++-skel-module-header-block-style** : allows selecting a user-define module-header comment block. • **pel-c++-skel-insert-file-timestamp** : set whether an automatically updated timestamp is inserted in the file header block. • **pel-c++-skel-use-separators** : set whether blocks use horizontal separator lines. • **pel-c++-skel-doc-markup** 🚧 : identifies the documentation markup supported by the templates. Not yet implemented. • **pel-c++-skel-cfile-section-titles** : identifies documentation section titles inserted in code files. • **pel-c++-skel-hfile-section-titles** : 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. • **pel-c++-skel-insert-function-sections** : set whether C++ function templates are inserted in the function description comment. • **pel-c++-skell-function-section-titles** : identifies title of the C++ function templates sections inserted when pel-c++-skel-insert-function-sections is **t**. • **pel-c++-skel-function-define-style** : select the C++ function comment block style. Several styles are provided: - no special comment - a basic, free-format style to describe the function above its code. - a Man-page style comment block with the sections identified by pel-c++-skell-function-section-titles - a user defined tempo skeleton loaded from a user specified file name. See the **source code example.** • **pel-c++-skel-function-name-on-first-column** : identifies whether return type is located on the same line as function name or just above. • **pel-c++-skel-with-license** : specify whether copy right and code license is specified. An option provide ability to insert open source software license text controlled by lice. • **pel-c++-use-include-guards** : specify which type of include guard is inserted in header files. The available choices are: - no include guard - use #pragma once statement - use classic #ifdef/#define/#endif block using symbol created from file name - use a #ifdef/#define/#endif block using symbol created from file name and UUID for its uniqueness. ☝Emacs user options by default take effect globally. But by using file and directory variables (see Σ **File/Directory Variables**) 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. • 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 *tempo-marks)* with the standard tempo-mode keys **C-c M-f** and **C-c M-b** or some other keys like **C-c .** and **C-c ,**.
Σ **Customize** PEL C++ Skeletons layout	**<f12> <f12> <f2>**	(**pel-customize-pel** &optional OTHER-WINDOW)	Customize PEL C++ skeleton layout. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
Insert a file header	**<f12> <f12> h**	(**pel-elisp-file-header**)	Insert a file description block. Distinguish between code files and header files. • Prompts for the file purpose. • For header files, include guard is inserted if requested by customization. • 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. • See examples of generated code located in the **example/templates/cpp** repo directory. • Access the customization buffer by typing: **<f12> <f12> <f2>**
Insert a function definition with comment block	**<f12> <f12> f**	(**pel-c++-function**)	Insert a C++ function definition code and comment template. • The command prompts for the function name and its purpose. • 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. • 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. • If an invalid name is specified it is erased and you are prompted again. Use **M-p** to bring the old value back. • Prompts for function and purpose maintain separate histories. Use **M-p** and **M-n** to navigate in the histories at the prompt. You can also use the **<up>** and **<down>** keys. • 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. • Use **C-g** to cancel at any prompt.
Insert a class definition	**<f12> <f12> c**	(**pel-c++-class**)	Insert a C++ definition code template. • Prompts for the class name. Replaces dash by underscores. • When **pel-c++-has-doc-block** is **t**, prompts for the purpose of the class. Capitalize the first letter and appends a period if there is none. • The layout of the class definition is controlled by the following user-options: • **pel-c++-has-doc-block** • **pel-c++class-doc-section-titles** • **pel-c++class-members-sections** : this identifies the member sections, their access (public/ protected/private) and code/comment lines. The strings may contain the following markers: • **$$** : identify the location of a tempo mark (see the navigation commands below) • **$class-name** : replaced by the name of the class.
Insert #define	**<f12> <f12> d**	(**pel-c-define**)	Insert a C pre-processor **#define** statement. • 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).
Insert #include <.h>	**<f12> <f12> i**	(**pel-c-include-lib**)	Insert a C pre-processor **#include <>** statement to include a library file. • 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. • If there is text after point, insert a new line to place that text on the next line. • 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 **C-c .** move point there).
Insert #include "h"	**<f12> <f12> I**	(**pel-c-include-local**)	Insert a C pre-processor **#include ""** statement to include a local file. • 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. • If there is text after point, insert a new line to place that text on the next line. • 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 **C-c .** move point there).
Toggle pel-tempo-mode	**<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 are only available when Emacs runs in graphics mode. ☝When a skeleton is inserted via the execution of one of the pel-rst-… commands, the pel-tempo-mode is automatically activated.
Jump to next tempo mark	• **C-c M-f** • **C-c .** • **C-c C-.**	(**tempo-forward-mark**)	Jump to the next mark in 'tempo-back-mark-list': the location where code must be updated inside the inserted skeleton. • These key key bindings are only available when pel-tempo-mode is active.
Jump to previous tempo mark	• **C-c M-b** • **C-c ,** • **C-c C-,**	(**tempo-backward-mark**)	Jump to the previous mark in 'tempo-back-mark-list': the location where code must be updated inside the inserted skeleton. • These key binding are only available when pel-tempo-mode is active.

Description	Keystroke	Function	Note
Tempo Template Tag Insertion	**<f12> <f12> <f12>**	(**tempo-complete-tag** &optional SILENT)	Look for a tag and expand it.
	☝Instead of using the **<f12> <f12>** key bindings above, you can type the template name (shown in the title column like "if", "case", etc) completely or partially and then hit **<f12> <f12> <f12>**. A completion buffer opens up if the template name is incomplete (or empty in which case the buffer lists **all** available template names). Select the template name and hit RET. Emacs expands the template. • 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			
Insert Parentheses	**M-(**	(**insert-parentheses** &optional ARG)	For C++: 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 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. • If region is active, insert enclosing characters at region boundaries. • This command assumes point is not in a string or comment.
Marking	Emacs provides the following command to quickly mark the whole content of the current function. More mark commands exists, see the Σ **Marking** table.		
Mark the complete function body See also: Σ **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.
Search Support	In C++ 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 C++ 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 C++ '_' are treated as part of words. • With a prefix argument ARG, enable superword mode if ARG is positive, and 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…)
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: Σ **Highlight**	**<f12> h (** **<f11> h (**	(**show-paren-mode** &optional ARG)	Toggle visualization of matching parens (Show Paren mode). • With prefix argument ARG, enable Show Paren mode if ARG is positive, 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: Σ **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**.
Navigation in C++	This current list below describe the specialized commands only. See the others inside Σ **Navigation**		
• **By definitions**	Move to the definition of function or type at point. See Σ **Xref** for more information to activate the various engines that support cross referencing for C code.		
Find definition of identifier at point See also: Σ **Xref**	**M-.**	(**xref-find-definitions** IDENTIFIER)	Grab symbol at point and move cursor to its definition. • If there are more than one match, prompt in the *xref* buffer. • To search for a symbol entered manually, type **C-u M-.** • With dumb-jump this performs a search using ag, ripgrep or git grep if available.
Go back to where M-. was last issued	**M-,**	(**xref-pop-marker-stack**)	• Pop back to where M-. was last invoked. • Marker depth is controlled by the **xref-marker-ring-length** user option.
• **By call graph**	Use the call-graph external package to build a call-graph of a C++ function. Uses either GNU Global or Git grep as backend.		
Build call-graph of function at point/region	**<f12> M-g**	(**call-graph** &optional FUNC)	Generate 'call-graph' for FUNC / func-at-point / func-in-active-region. With prefix argument, discard cached data and re-generate reference data. ⚠ Preliminary support: validity of the generated graph needs to be investigated. 📦 Requires external **call-graph** package, activated by PEL when **pel-use-call-graph** is t.
By C pre-processor	Move across **C preprocessor conditional inclusion statements** #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)	Move point forward to matching #endif • If point on a #if|#ifdef|#ifndef statement moves to the matching endif • With **C-u** or numerical arg: move forward to matching #else|#elif • On success, push the original position on the mark ring and return the new position. • On error, issue user error on mismatch. Shift marking is available with **C-M-<right>**
Move point backward to matching #if |#ifdef | #ifndef • **or matching #else | #elif**	**<f6> <left>**	(**pel-c-preproc-backward-conditional** &optional TO-ELSE)	Move point backward to matching beginning of #if|#ifdef|#ifndef conditional. • With **C-u** or numerical arg: move backward to matching #else|#elif • On success, push the original position on the mark ring and return the new position. • On error, issue user error on mismatch. Shift marking is available with **C-M-<left>**
Move outward forward to matching #endif	**<f6> <down>**	(**pel-c-preproc-outward-forward-conditional** &optional NEST-COUNT)	Move point forward, outward to end of current #if|#ifdef|#ifndef statement. • By default move 1 nest level outward. A larger count can be specified with optional NEST-COUNT numeric argument. • On success, push the original position on the mark ring and return the new position. • On error, issue user error on mismatch.
Move outward backward to matching #if | #ifdef | #ifndef	**<f6> <up>**	(**pel-c-prepcroc-outward-backward-conditional** &optional NEST-COUNT)	Move point backward, outward to beginning of current #if|#ifdef|#ifndef statement. • By default move 1 nest level outward. A larger count can be specified with optional NEST-COUNT numeric argument. • On success, push the original position on the mark ring and return the new position. On error, issue user error on mismatch.
Show all C pre-processor conditional statements inside an occur buffer	**<f6> o**	(**pel-c-preproc-conditionals-occur** &optional NLINES)	Show C pre-processor conditional statements inside an occur buffer. • Each line is shown with NLINES before and after, or -NLINES before if NLINES is negative. • NLINES defaults to **list-matching-lines-default-context-lines** user-option value. • If a region is defined the search is restricted to the region. See **occur search**.

Description	Keystroke	Function	Note
Show all C pre-processor conditional statements of project buffers inside an occur buffer	**<f6> <f8> o**	(**pel-c-preproc-conditionals-multi-occur** &optional NLINES)	Show C pre-processor conditional statements of current project **buffers** inside an occur buffer. • Each line is shown with NLINES before and after, or -NLINES before if NLINES is negative. • NLINES defaults to **list-matching-lines-default-context-lines** user-option value. See **occur search**. • This command uses **Projectile**. You must have pel-use-projectile user-option set and projectile active (use **<f11> <f8> <f8>** to activate it.)

• **By blocks**
  • **functions**
  • **structures**
	• Move to beginning /end of function definition blocks or structure definition blocks.
	• Move across C++ statements and C++ scope blocks, or any group of ( ), [ ], { } or < > 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.
	• ☝ Both **<f12>** and **<M-f12>** key prefixes are available for several bindings to ease typing some sequences. The one easier to type is identified in **bold**.		

Move block forward See also: • **Σ Navigation** ☝ • Use this to move to end of next syntax element or to end of block when already **outside** the block. • Use **C-M-u** to exit a block **(see below)**.	• **<f12> <right>** • <M-f12> <right> • **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.
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 backward See also: • **Σ Navigation**	• **<f12> <left>** • <M-f12> <left> • **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.
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.
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.
	• **<f12> <up>** • <M-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.
	C-M-<home>		⚠ 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.
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.
	• **<f12> <down>** • <M-f12> <down>	(**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.
	C-M-<end>		⚠ This command moves to the end of the next **top-level** function. It skips nested functions.
Backward to end of previous top level function or struct	• <f12> <M-up> • **<M-f12> <M-up>**	(**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. 🚧
Forward to start of next top level function or struct ☝ • Use this to move from the top of the file to the first block.	• <f12> <M-down> • **<M-f12> <M-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.
• **By class visibility**	• Move across C++ class visibility statements : public, protected, private		
To next visibility statement	**<f12> s v**	(**pel-move-down-to-class-visibility**)	Move point to the next C++ class member visibility statement (public, protected or private) • Does not move point to comments or when nothing found. Move back with **<f6> <f6>**
To previous visibility statement	**<f12> s V**	(**pel-move-up-to-class-visibility**)	Move point to the previous C++ class member visibility statement (public, protected or private) • Does not move point to comments or when nothing found. Move back with **<f6> <f6>**

Description	Keystroke	Function	Note
• 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. • Also work on other parentheses-like expressions defined by the current language mode. • With ARG, do it that many times. Negative arg means move backward but to a less deep spot.
Down/inside sexp/block See also: • Σ 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. • Also work on other parentheses-like expressions defined by the current language mode. • With ARG, do it that many times. Negative arg mans 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 or comment.		
Go to beginning of statement (backward)	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 (forward)	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.
• switch statement	Move to beginning /end of enclosing switch statement		
Go to start of current switch statement	<f6> t w s	(pel-cc-to-switch-begin)	Move point to the start { of current switch statement, if any. • If point is inside switch statement, mark position before moving point. Move it back with M-`. • If point is not inside a switch statement, issue a user error.
Go to end of current switch statement	<f6> t w e	(pel-cc-to-switch-end)	Move point just past the end } of current switch statement, if any • If point is inside switch statement, mark position before moving point. Move it back with M-`. • If point is not inside a switch statement, issue a user error.
Go to start of enum definition block	<f6> t e s	(pel-cc-to-enum-begin)	Move point to the start { of current enum definition block, if any. • If point is inside such a block, mark position before moving point. Move it back with M-`. • If point is not inside an appropriate block statement, issue a user error.
Go to end of enum definition block	<f6> t e e	(pel-cc-to-enum-end)	Move point just past the end } of current enum definition block, if any. • If point is inside such a block, mark position before moving point. Move it back with M-`. • If point is not inside an appropriate block statement, issue a user error.
Go to start of class/ struct definition block	• <f6> t s s • <f6> t c s	• (pel-cc-to-struct-begin) • (pel-cc-to-class-begin)	Move point to the start { of current class or struct definition block, if any. • If point is inside such a block, mark position before moving point. Move it back with M-`. • If point is not inside an appropriate block statement, issue a user error. The 2 commands are the same (the second one is an alias to the first)
Go to end of class/ struct definition block	• <f6> t s e • <f6> t c e	• (pel-cc-to-struct-end) • (pel-cc-to-class-end)	Move point just past the end } of current class or struct definition block, if any. • If point is inside such a block, mark position before moving point. Move it back with M-`. • If point is not inside an appropriate block statement, issue a user error. The 2 commands are the same (the second one is an alias to the first)
Go to start of union definition block	<f6> t u s	(pel-cc-to-union-begin)	Move point to the start { of current union definition block, if any. • If point is inside such a block, mark position before moving point. Move it back with M-`. • If point is not inside an appropriate block statement, issue a user error.
Go to end of union definition block	<f6> t u e	(pel-cc-to-union-end)	Move point just past the end } of current union definition block, if any. • If point is inside such a block, mark position before moving point. Move it back with M-`. • If point is not inside an appropriate block statement, issue a user error.
C Preprocessor	Emacs supports navigation through C preprocessor conditional statements, allow expansion of preprocessor macros, hiding pre-processor statements that would not be executed with the Hide-ifdef mode. There are also external packages that provide extra support. All commands provided by Emacs and external packages are listed below. They can be used for editing C and C++ source code. PEL provides a key hydra to help navigate trough preprocessor directives and to hide/show code areas based on preprocessor logic and defined variables. ✱ The key sequences that start with <f12> <f7> open the pel-Σc-preproc Hydra allowing further hydra keys to be typed without any prefix. 📦 Requires the hydra external package PEL activates when the pel-use-hydra user option is set to t.		
Open the C preprocessor hydra with <f12> <f7> followed by on of the hydra keys:			
Move across preprocessor conditionals	The following commands move point across the #if, #else, #elif and #endif C pre-processor conditional statements.		
Move to previous preprocessor directive	• <f12> # p ✱ <f12> <f7> p	(pel-pp-prev-directive)	Move point to previous preprocessor directive.
Move to next preprocessor directive	• <f12> # n ✱ <f12> <f7> n	(pel-pp-next-directive)	Move point to next preprocessor directive.
Move up in the pre-processor conditional block	• C-c C-u ✱ <f12> <f7> C-u	(c-up-conditional COUNT)	Move back to the containing preprocessor conditional, leaving mark behind. • A prefix argument acts as a repeat count. With a negative argument, move forward to the end of the containing preprocessor conditional. • "#elif" is treated like "#else" followed by "#if", so the function stops at them when going backward, but not when going forward.
Move to the previous pre-processor conditional block	• C-c C-p ✱ <f12> <f7> C-p	(c-backward-conditional COUNT &optional TARGET-DEPTH WITH-ELSE)	Move back across a preprocessor conditional, leaving mark behind. • A prefix argument acts as a repeat count. • With a negative argument, move forward across a preprocessor conditional.

Description	Keystroke	Function	Note
Move to the next pre-processor conditional block	**C-c C-n** ✱ **<f12> <f7> C-n**	(**c-forward-conditional** COUNT &optional TARGET-DEPTH WITH-ELSE)	Move forward across a preprocessor conditional, leaving mark behind. • A prefix argument acts as a repeat count. • With a negative argument, move backward across a preprocessor conditional. • If there aren't enough conditionals after (or before) point, an error is signaled. • "#elif" is treated like "#else" followed by "#if", except that the nesting level isn't changed when tracking subconditionals.
Expand Pre-Processor	• **C-c C-e** • **<f12> # #** • **M-<12> # #**	(**c-macro-expand** START END SUBST)	Expand C macros in the region, using the C preprocessor. • Normally display output in temp buffer, but prefix arg means replace the region with it.
	🎛 Customizations: **'c-macro-preprocessor'** specifies the preprocessor to use. If the user option **'c-macro-prompt-flag'** is non-nil prompt for arguments to the preprocessor (e.g. '-DDEBUG -I ./include'), otherwise use **'c-macro-cppflags'.**		
Insert/align or delete end-of-line backslash	**C-c C-**	(**c-backslash-region** FROM TO DELETE-FLAG &optional LINE-MODE)	Insert, align, or delete end-of-line backslashes on the lines in the region. • With no argument, inserts backslashes and aligns existing backslashes. • With an argument, deletes the backslashes.
	• This function does not modify blank lines at the start of the region. If the region ends at the start of a line and the macro doesn't continue below it, the backslash (if any) at the end of the previous line is deleted. • You can put the region around an entire macro definition and use this command to conveniently insert and align the necessary backslashes. 🎛 Customizations: The backslash alignment is done according to: '**c-backslash-column**', '**c-backslash-max-column**' and '**c-auto-align-backslashes**'.		
Hide-ifdef Mode Use **<f12> # ?** to show the value of **hide-ifdef-env** and **hide-ifdef-define-alist**	The Hide-ifdef mode can hide portion of the C pre-processor blocks. • 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 Emacs variable **hide-ifdef-env** association list . Use **<f1> v** or, with PEL, **<f12> # ?** to see the content of these variables. See **Help/Info**. ☝ With PEL, the commands listed below are bound to the **<f12>** prefix keys and also to the **M-<f12>** prefix keys. 🎛 Several customize user option variables affect how the hiding is done (to change, execute: **M-x customize-group hide-ifdef**): • '**hide-ifdef-env**' An association list of defined symbols for the current project. Initially, the global value of 'hide-ifdef-env' is used. This variable was a buffer-local variable, which limits hideif to parse only one C/C++ file at a time. We've extended hideif to support parsing a C/C++ project containing multiple C/C++ source files opened simultaneously in different buffers. Therefore 'hide-ifdef-env' can no longer be buffer local but must be global. • (SYMBOL) is used when the SYMBOL is defined (but without explicit value) • (SYMBOL . VALUE) when the symbol is defined with an explicit value. • '**hide-ifdef-define-alist**' 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'. • '**hide-ifdef-lines**' Set to non-nil to not show #if, #ifdef, #ifndef, #else, and #endif lines when hiding. • '**hide-ifdef-initially**' Indicates whether 'hide-ifdefs' should be called when Hide-Ifdef mode is activated. • '**hide-ifdef-read-only**' Set to non-nil if you want to make buffers read only while hiding. After 'show-ifdefs', read-only status is restored to previous value. Key Prefixes: the **<f12>** , **M-<f12>** and **<f11> SPC C** key prefixes are available for all the following commands, although not all shown below.		
Show state preprocessor modes	• **<f12> # ?** ✱ **<f12> <f7> ?**	(**pel-pp-show-state**)	Show state of C preprocessor control modes on the echo area. • Also displays the **hide-ifdef-env** and the **hide-ifdef-define-alist** variables by the Hide-ifdef mode (see next page) ⚠ If too long, see the information in the *Messages* buffer.
Toggle the Hide-Ifdef mode	• **<f12> M-#** • **M-<f12> M-#** ✱ **<f12> <f7> #** • **<f11> SPC c M-#**	(**hide-ifdef-mode** &optional ARG)	Toggle features to hide/show #ifdef blocks (Hide-Ifdef mode). • With a prefix argument ARG, enable Hide-Ifdef mode if ARG is positive, and disable it otherwise. • Hide-Ifdef mode is a buffer-local minor mode for use with C and C-like major modes. When enabled, code within `#ifdef` constructs that the C preprocessor would eliminate may be hidden from view.
Toggle read-only mode when text is hidden	• **C-c @ C-q** • **<f12> # r** ✱ **<f12> <f7> R**	(**hide-ifdef-toggle-read-only**)	Toggle read-only: toggle 'hide-ifdef-read-only'. • Note that you can make the file read only by default when hide-ifdef is hiding text, by setting the '**hide-ifdef-read-only**' user option to **t**.
Toggle shadowing of hidden text.	• **C-c @ C-w** • **<f12> # w** ✱ **<f12> <f7> W**	(**hide-ifdef-toggle-shadowing**)	Toggle shadowing. When shadowing is on, text that would be hidden is "shadowed" instead: it is displayed with the shadow face (normally something dim, all depending of the theme used).
Hide content of all #ifdef statements that would not be included	• **C-c @ h** • **<f12> # H** • **M-<f12> # H** ✱ **<f12> <f7> H** • **<f11> SPC c # H**	(**hide-ifdefs** &optional NOMSG)	Hide the contents of some #ifdefs. • Assume that defined symbols have been added to 'hide-ifdef-env'. • 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. • With prefix command presents it will also hide the #ifdefs themselves. • Turn off hiding by calling '**show-ifdefs**'.
Restore all hidden into view	• **C-c @ s** • **<f12> # S** ✱ **<f12> <f7> S**	(**show-ifdefs**)	Cancel the effects of 'hide-ifdef': show the contents of all #ifdefs.
Hide part of current block that would not be included	• **C-c @ C-d** • **<f12> # h** ✱ **<f12> <f7> h**	(**hide-ifdef-block** &optional ARG START END)	Hide the ifdef block (true or false part) enclosing or before the cursor. • With optional prefix argument ARG, also hide the #ifdefs themselves.
Show all parts of the current #ifdef block	• **C-c @ C-s** • **<f12> # s** ✱ **<f12> <f7> s**	(**show-ifdef-block** &optional START END)	Show the ifdef block (true or false part) enclosing or before the cursor.
Set a variable to a specific value	• **C-c @ d** • **<f12> # d** ✱ **<f12> <f7> d**	(**hide-ifdef-define** VAR &optional VAL)	Define a VAR to VAL (default 1) in 'hide-ifdef-env'. • This allows hiding the block inside **#ifndef VAR** (or the equivalent) by executing the command hide-ifdefs.
Undefine a variable	• **C-c @ u** • **<f12> # u** ✱ **<f12> <f7> u**	(**hide-ifdef-undef** START END)	Undefine a VAR • This allows hiding the blocks inside **#ifdef VAR** (or the equivalent) by executing the command hide-ifdefs.
Save the symbol environment list into a named list	• **C-c @ D** • **<f12> # D** ✱ **<f12> <f7> D**	(**hide-ifdef-set-define-alist** NAME)	Save the state of the current hide-ifdev-env to a list with the specified NAME for later re-use. The value is saved inside the '**hide-ifdef-define-alist**' variable. ⚠ 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.
Use a named symbol environment list	• **C-c @ U** • **<f12> # U** ✱ **<f12> <f7> U**	(**hide-ifdef-use-define-alist** NAME)	Use an already saved symbol list with the specified NAME and store it inside the 'hide-ifdef-env' to be used in the editing session. Set 'hide-ifdef-env' to the define list specified by NAME.
Clear the complete list of #define'd symbols inside 'hide-ifdef-env'	• **C-c @ C** • **<f12> # C** ✱ **<f12> <f7> C**	(**hif-clear-all-ifdef-defined**)	Clears all symbols defined in 'hide-ifdef-env'. • It will backup this variable to 'hide-ifdef-env-backup' before clearing to prevent accidental clearance.
Evaluate pre-processor macro	• **C-c @ e** • **<f12> # e** ✱ **<f12> <f7> e**	(**hif-evaluate-macro** RSTART REND)	Evaluate the macro expansion result for the active region. • If no region active, find the current #ifdefs and evaluate the result. • Currently it supports only math calculations, strings or argumented macros can not be expanded.

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 **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.

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 wit h better styled-code that does not explicitly uses the keyword.		
Problematic code	Problem: C++ code that compare pointer against NULL and value against TRUE, true, FALSE, and false.		
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
 	
• Use **mk-diffo** to check if anything changed!
	<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:
			• `if (pointer == NULL)` by `if (!pointer)`
			• `if (value == TRUE)` by `if (value)`
			• `if (value == FALSE)` by `if (!value)`
			• `if (value == true)` by `if (value)`
			• `if (value == false)` by `if (!value)`
			• `if (pointer != NULL)` by `if (pointer)`
			• `if (value != TRUE)` by `if (!value)`
			• `if (value != FALSE)` by `if (value)`
			• `if (value != true)` by `if (!value)`
			• `if (value != false)` by `if (value)`
			
			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.
			
			☝The command can detect and reformat a large number of expressions but not all of them.
			⚠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 *there should be **no** change in the generated assembler code.*
			• The **best way to check if the reformatting is correct** is to compare the generated assembler code for the file before and after the code reformatting done by this command.
			  • With GCC toolchain, use the **`objdump -- disassemble`** command on the object file to generate the assembler files. The LLVM toolchain provide the **llvm-objdump** equivalent.
			  • I have written a very useful utility for this: **mk-diffo**, part of my **USRHOME project**. **mk-diffo** generates assembler from the object file and compares the generated assembler of the current version of the source with a previous version, stored in a tree of backup copies.

Problematic code	Problem: C pre-processor conditionals that compare a symbol without checking if it is defined. This may cause unexpected result. • Instead of: **`#if VAR`** write **`#if ((defined(VAR) && (VAR != 0))`** • Instead of: **`#if VAR == 0`** write **`#if (!defined(VAR) || (VAR == 0))`** • Instead of: **`#if VAR == 1`** write **`#if (defined(VAR) && (VAR ==1))`**		
Search for poor pre-processor conditional #if VAR	**<f12> s #**	(**pel-c-search-preproc-if**)	Move point to the end of the next **`#if VAR`** expression.
Search for poor pre-process conditional #if VAR==0 #if VAR==1	**<f12> s 0**	(**pel-c-search-preproc-if-set**)	Move point to the end of the next **`#if VAR == 0`** expression or **`#if VAR == 1`** expression.
Improve C/C++ code: remove explicit comparisons against NULL, TRUE, FALSE, true and false	**<f12> s C-p**	(**pel-c-fix-preproc-if-problems**)	Inside current buffer, replace all instances of problematic C pre-processor conditional code listed below with the corresponding safer code. • Instead of: **`#if VAR`** it writes **`#if ((defined(VAR) && (VAR != 0))`** • Instead of: **`#if VAR == 0`** it writes **`#if (!defined(VAR) || (VAR == 0))`** • Instead of: **`#if VAR == 1`** it writes **`#if (defined(VAR) && (VAR == 1))`**
Programming Help	PEL has bindings for the following commands that are useful when editing source code, markup files or any file that has a mode that supports imenu.		
Show what completion mode is currently used.	• **<f11> ? M-c** • **<f11> M-c ?**	(**pel-show-active-completion-mode**)	Display the completion mode currently used, and the Ido prompt geometry when appropriate. • Show key bindings for changing other aspects of input completion.
Show function at point	**<f11> ? F**	(**pel-show-function**)	Display the name of the current "*function*" at point in the mini-buffer.

Toggle which-function-mode to display name of current function at point
See also:
• **Menus**
• **Mode Line**
☝The concept of "*function*" is major mode specific. For example, in C++ mode, if point is inside a class definition it shows the name of the class.
	• **<f11> ? f**
	• **<f11> M-d f**
		(**which-function-mode** &optional ARG)	Toggle mode line display of current function (Which Function mode).
			• With a prefix argument ARG, enable Which Function mode if ARG is positive, and disable it otherwise.
			
			• The **which-function-mode** is a global minor mode. When enabled, the current function name is continuously displayed in the mode line.
			⚠ Detection of functions and variables depend on the **imenu** functionality. If you modify the content of a buffer, you need to force a menu rescan to get proper results. You can force a rescan with **pel-imenu-rescan**, bound to **<f11> <f10> r**.
			👀 Identify major modes that automatically active the mode with **which-function-mode** user-option.
			• Use **`M-x customize-option which-function-mode`** to open the relevant customization buffer.
			• With PEL you can use:
			  • **<f11> ? <f3>** to access the which-func customization group. It will provide access to the customization group even when the feature has not yet been loaded, something that Emacs does not do by default.
			  • **<f11> <f2> o `which-function-mode` RET** to access the user-option directly.

Emacs & C++— References

Document	Notes
Emacs Support for C++	
GNU emacs - CC Mode Manual	
GNU Emacs Manual - Styles	
Emacs BSD/Allman Style with 4 Space Tabs?	
Emacs: Linux Kernel Style but with Allman/BSD Style Braces?	
Emacs Wiki - Indenting C	
Indent preprocessor directives as C code in emacs	Does not fully address the way I want to have multi-indentations for pre-processor
elisp code - ppindent.el	Implements pre-processor indentation with the # always in the first column. Not yet exactly what I want.
Demystify C++ Metaprograms using Emacs	
Programming in C++, Rules and Recommendations	ellemtel style
company-mode ; Modular in-buffer completion framework for Emacs	
C++	
C++ @ Wikipedia	See also these Wikipedia pages • **Criticism of C++** • **C++23** , **C++20** , **C++17** , **C++14** , **C++11** , **C++03** • **C and C++ operators**
C++ Standard @ ISO C++	
JTC1/SC22/WG21 - The C++ Standard Committee ISOCPP	See also: **C++ Standard Draft Sources @ GitHub**
C++ Reference @ cppreference.com	
C++ Core Guidelines @ GitHub	
CppCon The C++ Conference	
C++ Annotations	
PC-lint Plus from Gimpel	Strongly recommended static analyzer for C and C++. Will improve your knowledge of C++. Best used when you instrument your code with some directives. For serious C++ development, as it requires some time investment.
Edison Design Group C++	The Edison Design Group provides C++ parsing and tools to several C++ tool vendors. So it's a good thing to know what version of C++ EDG supports. They also provide a good source of links for C++ standard features in forms of Google Sheets: • **C++ 20 features** • **C++17 features** • **C++14 features** • **C++11 features**