

Perl 5 Constants and Variables

Perl Constants		Perl pragma to declare constants ⚠️ but not read-only! See CPAN modules for defining constants by Neil Bowers and Const::Fast and Attribute::Constant					
Perl Variables Names		Scalar Naming Conventions		Array Naming Conventions		All: 1 st char: underscore or letter. Never use ALLCAPS	
Case sensitive. ASCII by default, UTF-8 if the utf8 pragma is used.		<ul style="list-style-type: none">All variables: words_with_underscoresLocal variables: \$lowercaseGlobal variables: \$Title_CaseConstants: \$UPPER_CASE		Same, but array names should be plural . <ul style="list-style-type: none">@locals@Global_Arrays@CONSTANT_ARRAYS		<ul style="list-style-type: none">Module names are MixedCaseNoUnderscoresConstants are UPPERCASE_WITH_UNDERSCORESPackage wide vars are Mixed_Case_With_UnderscoresFunctions/methods are lowercase_with_underscores	
Scope of variables		global by default		A variable defined without any of the following prefixed keyword is global by default.			
See: Scope of variables in Perl @Perl Maven		my		local, lexical scope, non persistent		Examples: my @values = (42, 36, 99); my (\$v1, \$v2) = (42, 36);	
		state		Local, lexical scope, persistent		Restriction: in <i>Perl</i> < v5.28: array and hashes state cannot be initialized in list context.	
		our		creates a lexical scoped alias to a package variable			
		local		Localizes an existing package variable to the current scope. It's not a declaration. The variable previous value is restored when leaving the scope.			
Perl types Scalar		\$		\$foo		Simple scalar value	
				\$days[28]		29 th element of array @days	
				\$days{'Feb'}		Value associated with the Feb key of hash %days	
				\${days}		Same as \$days, use before alphanumerics.	
Archaic use of single quote: \$Dog'days				\$Dog::days		The \$days variable inside the Dog package.	
				\$#days		Last index of array @days.	
				\$days->[28]		29 th element of array pointed to by reference \$days.	
				\$days[0][2]		Multi-dimensional array	
				\$d{99}{'Feb'}		Multi-dimensional hash	
				\$d{99, 'Feb'}		Multi-dimensional hash emulation	
list and Array		@		@days		Array containing (\$days[0], \$days[1], ... #days[\$#days])	
• 0-based indexed (first index is 0).				@days[3,4,5]		Array slices containing (\$days[3], \$days[4], \$days[5])	
• Last index of array @name is \$#name				@days[3..5]		Array slices containing (\$days[3], \$days[4], \$days[5])	
				<ul style="list-style-type: none">Negative indices used in read access from the end: -1 is last item.Use these negative indices to access from the end. Do not compute index with \$#name -3, if the list size is 2, this will give invalid results.			
• array slices LP␣ Simple explanation				<ul style="list-style-type: none">Use a slice to select multiple elements from a list, array, or hash.Don't use a slice when you know you need exactly one element.An lvalue slice imposes list context on the righthand side.Assign to array slice to update several values. ➡		my @extracted = (6, 2, 8, 4); my @choices = @digits[@extracted] my \$mod_time = (state \$filename)[9]; @extracted[1, 3] = (7, 9);	
						my @digits = (0..9); my @onefive = @digits[1..5]; my @premiers = @digit[1, 2, 3, 5, 7];	
• Anonymous arrays				<ul style="list-style-type: none">What are the advantages of anonymous array? @ StackOverflowPerlref @ Perldoc, Perl reference tutorial @ Perldoc		<ul style="list-style-type: none">Anonymous array := a type of array reference. Use it to build nested data structures.Array reference allows Perl to treat the array as a single item.	
Hash/associative array Hashes @ Perl Maven Note: keys are always strings.		%		%days		Associative array (hash): keys-value pairs. Can be initialized as: <ul style="list-style-type: none">my %days = (Jan => 31, Feb => \$leap? 29 : 28, ...)my %days = ("Jan", 31, 'Feb', \$leap? 29 : 28, ...) Multiple values of a hash can be changed with the following construct:	
hash slice LP␣ ➡				@days{'J','F'}		Hash slice returning a list containing (\$days{'J'}, \$days{'F'}) .	
key-value slices LP␣ ➡				extract/write values:		my scores = @rating{ @names }; @rating { @names } = (45, 55);	
Subroutine		&		&foo		& is needed to create reference to subroutine.	
Typeglob		*		*foo		See: <u>Advanced Perl Programming</u> , 1st Edition Section 3.2	
7 kinds of package variables types:		1. scalar variables \$ 2. array variables @		3. hash variables % 4. subroutine name		5. format names (See <u>write</u> and <u>select</u>) <ul style="list-style-type: none">how to format output in Perl?, Perl-Formats	
						6. file handles 7. directory handles	
References		A reference is a scalar variable whose value is a pointer to another Perl variable. Use it to <u>build more complex data types</u> . Make reference with \ . Stringize it with <u>ref</u>					
Perl references intro Perl reference tutorial Reference purpose IntP␣		my @array = qw(a, b, c); print \$array[1]. # b You can create complex data with references:⚡⚡⚡		my \$array_ref = ['a', 'b', "c\n"]; print \${array_ref}[1]; # b print \$\$array_ref[1]; # b, simpler print \$array_ref->[1]; # b, arrow notation		my %hash = (a=>1, b=>2, c=>3); print \$hash{c}; # 3 ⬅ drop brace around bareword ref. ➡ ⬅ arrow notation is shorter/cleaner ➡	
circumfix dereferencing: <ul style="list-style-type: none">simplify with ->simplify more		my \$data = [0, 1, 2, [40, 50, 60, [100, 200], 70], 8]; print @{{\${\$data}{3}}{3}}{0}, "\n"; #100 print \$data->[3]->[3]->[0], "\n"; # 100 print \$data->[3]->[3]->[0], "\n"; # 100 print \$data->[3][3][0], "\n"; # 100.				<ul style="list-style-type: none">Creale a lexical reference: my \$hash_ref = \%hash;Store a ref to an array or hash into an array: push @array \%hash;Pass array or hash to subroutine: fct(\@a, \%h); Return from sub: return (\@a, \%h); ⬅ Arrows between subscript are optional.	
postfix dereferencing See: cool new Perl feature: postfix dereferencing		(Perl >= v5.20.0) Instead of using a sigil prefix, it uses a postfix sigil and star. sref:ref to scalar, aref:ref to array, href:ref to hash, cref:ref to code, gref:ref to glob					
		\$sref->\$*; # same as \${ \$sref } \$aref->@*; # same as @{\$ \$aref }		\$aref->\$#*; # same as \${ \$aref } #last array idx \$href->%*; # same as %{ \$href }		\$cref->&*; # same as &{ \$cref } \$gref->***; # same as *{ \$gref }	
Reference to subroutine		Store a ref to a subroutine:		my \$fct_ref = \&the_function;		Indirect calls: with the simpler arrow notation:	
						<ul style="list-style-type: none">&{ \$the_function } (arg1, arg2);\$the_function->(arg1, arg2);	
		Using an anonymous subroutine, always calling it indirectly:				my \$op = sub { my \$v1 = shift; my \$v2 = shift; return \$v1 ** \$v2; }; say \$op->(10, 4); # prints 10000	
Closures <ul style="list-style-type: none">Perl closure		A closure binds its environment and keeps it to use it when invoked. <ul style="list-style-type: none">In the example at right, a greeter function is built and returned, remembering how to greet. It is used like this: my \$fr = make_greeting("Bonjour"); my \$it = make_greeting("Buongiorno"); \$fr->('Brigitte'); # prints: "Bonjour, Brigitte!\n" \$it->('Madonna'); # prints: "Buongiorno, Madonna!\n"				sub make_greeting { my \$greet = shift; my \$greet_fct = sub { my \$name = shift; print "\$greet, \$name!\n"; }; return \$greet_fct; # return ref to internal function }	
		A code block defining lexical variable(s) and subroutines consist of a closure too! With the following example, the add_1() subroutine increments the \$count and that's returned by get_count(). The \$count variable cannot be accessed from anywhere else!				{ my \$count; sub add_1 { count += 1; } sub get_count { return count; } }	
Scalar values		Numeric literals examples:		Note: leading 0 work only for literals, not for string-to-number conversions.		Useful related builtin functions	
• numeric:		<ul style="list-style-type: none">integer : using the system's native format.<ul style="list-style-type: none">bigint - transparent big integer support.bignum - transparent big number support.floating-point : using the system's native format.<ul style="list-style-type: none">bigrat - transparent big rational number support.		my \$x = 12345; my \$x = 12345.67; my \$x = 6.02e23; my \$x = 0x1f.0p3; my \$x = 4_294_967_296; my \$x = 0x1234_5678; my \$x = 0377; my \$x = 0o377; my \$x = 0b1100_0010; my \$x = 0xff55;		# integer # floating point # scientific notation # power2 exponent: Perl >= v5.22 # underline for legibility # underline in hex is also OK # octal # octal also Perl >= v5.34 # binary with underlines # hexadecimal	
Note: underline separators can be used inside decimal, hexadecimal and binary literals.		A variable holding an integer can be converted to floating-point if the operation done to it requires it (such as dividing 1 by 2).				<ul style="list-style-type: none">oct - for: binary, octal, hexhexPOSIX::ceilPOSIX::floorabs	


<ul style="list-style-type: none">• string	<ul style="list-style-type: none">• double-quoted strings: perform backslash and variable interpolation of expression that begin with \$ (a scalar) or @ (an array). Hashes cannot be interpolated.• single-quote strings: only perform \ ' and \\ substitution (to ' and \ respectively), nothing else.• Single quote and double quote strings can spread multiple lines: it embeds the newline character on each new line.• \n is only expanded in double quoted strings. In single quote string it is treated as two characters; no substitution is done (as explained above).					
<ul style="list-style-type: none">• Unicode support	Use Unicode literally in a program; add the utf8 pragma : use utf8 ; See: Perl Unicode Tutorial , Perl Unicode Introduction , Perl Unicode Support @ perldoc					
<ul style="list-style-type: none">• Quote constructs	Usual	Generic	Meaning	Interpolates?	Notes	
<p>See:</p> <ul style="list-style-type: none">• Strings in Perl: quoted, interpolated and escaped	' '	q//	Literal string	No	<ul style="list-style-type: none">• Not all characters can be used as the / separator. { }, () and < > can also be used.• You can use whitespace between the quote specifier and its initial bracketing character:<pre>my \$chuck_of_code = q { if (\$condition) { print "Bonjour!"; } };</pre>	
	" "	qq//	Literal string	Yes		
	``	qx//	Command execution	Yes		
	()	qw//	World list	No		
	//	m//	Pattern match	Yes		
	s///	s///	Pattern substitution	Yes		
	tr///	y///	Character translation	No		
	""	qr//	Regular expression	Yes		
	<ul style="list-style-type: none">• It's also possible to write: s<foo>(bar) and tr(a-f) [A-F] as well as separating them on 2 lines:<pre>tr (a-f) [A-F];</pre>• Array variables are interpolated by joining all elements with the separator specified by the \$" special variable (\$LIST_SEPARATOR) .					
	<ul style="list-style-type: none">• Character escapes (only inside double quoted strings)	\a	Alert (bell)	\t		Horizontal tab
	\b	Backspace	\e	ESC character		
	\e	ESC character	\033	ESC in octal	Any Unicode code point, by name:	
	\f	Form feed	\o{33}	ESC in octal	\N{LATIN SMALL LETTER E WITH ACUTE}	é
	\n	Newline (usually LF)	\x7f	DEL in hexadecimal	\N{ U+E9 }	é
	\r	Carriage return (Usually CR)	\cC	Control-C		
<ul style="list-style-type: none">• translation escapes (inside double quoted strings)	\u	Force next character to titlecase	\U	Force all following characters to uppercase. Ends at \E	\E	Ends \U, \L, \F or \Q
	\l	Force next character to lowercase	\L	Force all following characters to lowercase. Ends at \E		
			\F	Force all following characters to Unicode fold case. Ends at \E		
			\Q	Backslash all following non alphanumeric characters. Ends at \E		
<ul style="list-style-type: none">• bareword	In Perl, a <i>bareword</i> refers to a sequence of characters suitable for an identifier. It's not quoted. By default Perl allows barewords to behave like strings. <ul style="list-style-type: none">• This is not allowed when any of use strict; or use strict "subs"; or use v5.12; is specified.					
<ul style="list-style-type: none">• Here documents• Here docs @ Perl maven• Perl here doc @Wikipedia	Perl here-documents are a form of line oriented quoting. There are several forms of here documents, where the identifier (like EOF used below, but can be any word) must be placed at the beginning of the terminating line: <p>Note: They can also be stacked and text can be transformed. See the documentation.</p> <ul style="list-style-type: none">• Default : <<EOF; Supports variable interpolation.• Double quotes: <<"EOF"; Supports variable interpolation. Can also be written with whitespace as in << "EOF";• Single quotes: <<'EOF'; Does not support interpolation. Can also be written with whitespace as in << 'EOF';• backticks: <<'EOF'; Execute commands in a shell and return text printed on stdout. Can also be written with whitespace as in << `EOF`;• indented: <<~EOF; Allows indenting the here-doc string. Can also use the ~ with the other forms: <<~\EOF, <<~"EOF", <<~'EOF', <<~`EOF`					
<ul style="list-style-type: none">• Perl Regexp	Regexp Tutorial , Learn PCRE in X minutes , PCRE cheatsheet , Debuggex regexp tester, regex101 , RegEx Pal					
<ul style="list-style-type: none">• index/substr	\$pos = index (\$page, \$line);	\$last_slash = rindex ("usr/bin/ls", "/");	\$part = substr (\$text, \$pos, \$len)	A value of -1 in pos identifies last character.		
<ul style="list-style-type: none">• Replacement• manipulate strings with substr LPr	my \$pref = "I like awk and erlang"; substr (\$pref, index (\$pref, "awk"), length ("awk")) = "Perl"; substr (\$pref, 0, 0) = "Sally and "; # insert text anywhere		substr (\$pref, -15) =~ s/Perl/Perl5/g; # replace text inside a restricted portion of the string.			

Perl 5 Special Literal and Variables

Special Literals					
	<ul style="list-style-type: none"><code>__FILE__</code> : current file name<code>__LINE__</code> : current line number		<ul style="list-style-type: none"><code>__PACKAGE__</code> : current package name<code>__SUB__</code> : reference to current subroutine		<ul style="list-style-type: none"><code>__END__</code> : use to indicate logical end of script<code>__DATA__</code> : same, but supports reading text
Perl Special Variables <ul style="list-style-type: none">Perl Variables	<ul style="list-style-type: none">👉 To get information about a Perl special variable from the command line use the perldoc -v command.To get information about <code>\$<</code> use: perldoc -v '\$<'				
<ul style="list-style-type: none">Deprecated and removed variables:	<code>\$#</code>	<code>\$*</code>	<code>\$_[</code>	<code>\${^ENCODING}</code>	<code>\${^WIN32_SLOPPY_STAT}</code>
<ul style="list-style-type: none">General variables	Note that the <code>\$</code> , <code>@</code> and <code>%</code> prefixes are the sigil that identify the scalar, array and hash access context. The name of the variable is placed after that character.				
default input and pattern searching space	<ul style="list-style-type: none"><code>\$ARG</code><code>\$_</code>		subroutine parameters	<ul style="list-style-type: none"><code>@ARG</code><code>@_</code>	
list separator	<ul style="list-style-type: none"><code>\$LIST_SEPARATOR</code><code>\$"</code>		Subscript separator for multidimensional array emulation	<ul style="list-style-type: none"><code>\$\$SUBSCRIPT_SEPARATOR</code><code>\$\$SUBSEP</code><code>\$;</code>	
Name of executed program	<ul style="list-style-type: none"><code>\$PROGRAM_NAME</code><code>\$0</code>		Name used to execute the current copy of Perl	<ul style="list-style-type: none"><code>\$EXECUTABLE_NAME</code><code>\$^X</code>	
Perl process ID	<ul style="list-style-type: none"><code>\$PROCESS_ID</code><code>\$PID</code><code>\$\$</code>	Process real GID	<ul style="list-style-type: none"><code>\$REAL_GROUP_ID</code><code>\$GID</code><code>\$(</code>	Process effective GID	<ul style="list-style-type: none"><code>\$EFFECTIVE_GROUP_ID</code><code>\$EGID</code><code>\$)</code>
Process real UID	<ul style="list-style-type: none"><code>\$REAL_USER_ID</code><code>\$UID</code><code>\$<</code>		Process effective UID	<ul style="list-style-type: none"><code>\$EFFECTIVE_USER_ID\$</code><code>\$EUID</code><code>\$></code>	
Special variables in sort	<ul style="list-style-type: none"><code>\$a</code><code>\$b</code>	The Perl sort function uses global variables <code>\$a</code> and <code>\$b</code> . sort sorts strings. Pass a sorting function that uses the <code><=></code> equality operator to force numerical comparisons: <code>@sorted = sort { \$a <=> \$b } @unsorted;</code>			
Current environment	<code>%ENV</code>		Environment variable accessed as an associative array (a hash). <ul style="list-style-type: none">See: Perl: How to access shell environment variables through Perl associative arrays.		
Perl interpreter revision, version and subversion	<ul style="list-style-type: none"><code>\$OLD_PERL_VERSION</code><code>\$]</code>		Perl interpreter revision, version and subversion	<ul style="list-style-type: none"><code>\$PERL_VERSION</code><code>\$^V</code>	
Maximum file descriptor	<ul style="list-style-type: none"><code>\$\$SYSTEM_FD_MAX</code><code>\$^F</code>		Fields of each line when auto-split mode is on.	<code>@F</code>	
Include Directories	<code>@INC</code>		Included filenames	<code>%INC</code>	Hook localization (?) <code>\$INC</code>
inplace-edit extension value	<ul style="list-style-type: none"><code>\$_INPLACE_EDIT</code><code>\$^I</code>		Package's class parent classes	<code>@ISA</code>	Emergency memory pool <code>\$^M</code>
Maximum block nesting	<code>\${^MAX_NESTED_EVAL_BEGIN_BLOCKS}</code>			Time when program began running	<ul style="list-style-type: none"><code>\$BASETIME</code><code>\$^T</code>
Name of OS where this Perl was built	<ul style="list-style-type: none"><code>\$OSNAME</code><code>\$^O</code>	Signal handlers	<code>%SIG</code>	Coderefs for various perl keywords	<code>%{^HOOK}</code>
<ul style="list-style-type: none">Regexp Variables					
captured sub-patterns	<code>\$<digit>(\$1, \$2, ...)</code>		Capture buffer content	<code>@{^CAPTURE}</code>	
String matched	<ul style="list-style-type: none"><code>\$MATCH</code><code>\$&</code>		String matched (compiled regexp)	<code>\$_{^MATCH}</code>	
String preceding match	<ul style="list-style-type: none"><code>\$PREMATCH</code><code>\$'</code>		String preceding match (compiled regexp)	<code>\$_{^PREMATCH}</code>	

String following match	• \$POSTMATCH • \$'		String following match (compiled regexp)	{^POSTMATCH}	
Last capture group	• \$LAST_PAREN_MATCH • \$+		Most recently closed capture group	• \$LAST_SUBMATCH_RESULT • \$^N	
Match capture key values	• %{^CAPTURE} • %LAST_PAREN_MATCH • %+		Maximum regexp nested group	\${^RE_COMPILE_RECURSION_LIMIT}	
Match start offsets	• @LAST_MATCH_START • @-	Match ends offsets	• @LAST_MATCH_END • @+	Named captured groups	• %{^CAPTURE_ALL} • %-
Last successful pattern	\${^LAST_SUCESSFUL_PATTERN}		Result of last successful regexp assertion		\$^R • \$LAST_REGEXP_CODE_RESULT
regexp debug flag	\${^RE_DEBUG_FLAG}		regexp internal optimization/memory	\${^RE_TRIE_MAXBUF}	
• Format Variables	The format mechanism is use to generate printed layouts. It's an old Perl feature but still useful in various places.				
Current value of the write() accumulator for format() lines.	• \$ACCUMULATOR • \$^A				
Form feed format. defaults to \f	• IO::Handle->format_formfeed(EXPR) • \$FORMAT_FORMFEED • \$^L		Set of characters after which a string may be broken to fill continuation fields	• IO::Handle->format_line_break_characters EXPR • \$FORMAT_LINE_BREAK_CHARACTERS • \$:	
Number of lines left on the page on currently selected output channel	• HANDLE->format_lines_left(EXPR) • \$FORMAT_LINES_LEFT • \$-		Current page length of current output channel	• HANDLE->format_lines_per_page(EXPR) • \$FORMAT_LINES_PER_PAGE • \$=	
Name of current top-page format of output channel	• HANDLE->format_top_name(EXPR) • \$FORMAT_TOP_NAME • \$^		Report format name of output channel	• HANDLE->format_name(EXPR) • \$FORMAT_NAME • \$~	
• Error Variables	The variables \$@ , \$! , \$^E , and \$? contain information about different types of error conditions that may appear during execution of a Perl program. They correspond to errors detected by the Perl interpreter, C library, operating system, or an external program, respectively.				
Perl error from the last eval operator	• \$EVAL_ERROR • \$@		Current state of interpreter	• \$EXCEPTIONS_BEING_CAUGHT • \$^S	
Current value of C errno integer variable	• \$OS_ERROR • \$ERRNO • \$!	\$! returns the system variable errno when used in a numeric context, but returns the string from pererror() when used in string context.	Hash of error names to 0 or 1, set to 1 if current error is this error.	• %OS_ERROR • %ERRNO • %!	
OS detected error	• \$EXTENDED_OS_ERROR • \$^E				
Status returned by last pipe close, backtick command, wait, waited, or system() call.	• \$CHILD_ERROR • \$?		native status returned by last pipe close , backtick command, wait() or waitpid() or system() call	\${^CHILD_ERROR_NATIVE}	
Current value of warning switch	• \$WARNING • \$^W		Current set of warning checks enabled by the use warnings pragma	\${^WARNING_BITS}	
• Variables related to the interpreter state	These variables provide information about the current interpreter state.				
Flag associated with the -c switch	• \$COMPILING • \$^C		The current value of the debugging flags	• \$DEBUGGING • \$^D	
Current phase of the perl interpreter	\${^GLOBAL_PHASE}		Debugging support. Internal variable.	• \$PERLDB • \$^P	
Compile-time hints for the perl interpreter. Internal use only	\$^H		Values of compiled statements	%^H	
Taint mode	\${^TAINT}		Safe locale operations availability	\${^SAFE_LOCALES}	
Input/Output Layers. Internal use by PerlIO only.	\${^OPEN}		Unicode Settings of Perl	\${^UNICODE}	
Internal UTF-8 offset caching code state	\${^UTF8CACHE}		State of UTF-8 locale detected by perl at startup.	\${^UTF8LOCALE}	
• File handle Variables	See also: Perl File Handles The following variables are used in the Input/Output handling as well as program arguments.				
Name of current file read from <>	\$ARGV	Command line arguments of the script ← See diamond operator <>. →	@ARGV	Number of arguments minus one	\$#ARGV
Special file handle that iterates over command-line filenames in @ARGV	ARGV	Special file handle that points to currently open output file when doing edit-in-place processing	ARGVOUT		
Output field separator for the print operator	• IO::Handle->output_field_separator(EXPR) • \$OUTPUT_FIELD_SEPARATOR • \$OFS • \$,		Current line number for the last file handled accessed	• HANDLE->input_line_number(EXPR) • \$INPUT_LINE_NUMBER • \$NR • \$.	
Input record separator (newline by default)	• IO::Handle->input_record_separator(EXPR) • \$INPUT_RECORD_SEPARATOR • \$RS • \$/		Output record separator	• IO::Handle->output_record_separator(EXPR) • \$OUTPUT_RECORD_SEPARATOR • \$ORS • \$\	
Auto-flush control • order of output @ Perl Maven • Suffering from Buffering?	• HANDLE->autoflush(EXPR) • \$OUTPUT_AUTOFLUSH • \$!	Perl activates file buffering by default. Assign 1 to \$! to activate auto-flush.	Last read file handle	\${^LAST_FH}	

Perl 5 Input/Output 🚧🚧

References	<div><div>• open @ perldoc browser</div><div>• Writing to files with Perl @ Perl Maven</div><div>• open file in-memory @ stackOverflow</div></div> <div><div>• Stupid open() tricks @Perl.com:</div><div>• No explicit filename</div><div>• create an anonymous temporary file</div></div> <div><div>• print to a string</div><div>• read lines from a string</div></div>			
print, printf, sprintf	print, printf, sprintf (which describes the format) . Note: print , a list operator, is more efficient than printf . print and printf output to stdout by default, but accept a file handle as the first argument if it is NOT followed by a separating comma! (a <code>'</code> , puts it in the list to print!)			
say	use <code>feature qw(say);</code> or use <code>v5.10;</code> (or higher). Like print, but implicitly appends a newline at the end of the list.			
diamond operator <>	Both <> and <<>> operators read the content of files listed on the command line via @ARGV. Nothing or - on the command line identifies stdin. The <> operator supports shell redirection and pipe operations which <<>> does not allow (for security reasons).			
The double diamond, a more secure <> (Perl >= v5.22)	<code>print <>;</code>	⬅ Simple implementation of /bin/cat	<code>print <<>>;</code>	⬅ safer one
	<code>print sort <>;</code>	⬅ Simple implementation of /bin/sort	<code>print sort <<>>;</code>	⬅ safer one
 In-place-editing ⚠ The <> operator tries to duplicate the original file's permission and ownership.	Set \$^I to a backup file extension (such as Emacs "~" or ".bak") to change the behaviour of the <> and <<>> operators and print. In a <code>while (<>) { ... }</code> loop, when \$^I is not undef (its default), Perl: <ul style="list-style-type: none">renames currently processed file with the specified extension added,opens a new file with the original nameprints into the new file.Any modification goes into the new file: in-place-editing it!			
perl -i cmdline option	It's also possible to do this on the command line!		For example: <code>perl -p -i~ -w -e 's/something/Something else/g' data*.dat</code>	
Special filehandle names Also See: <ul style="list-style-type: none">File handle Variables section above.<code>open</code>	ARGV	The special filehandle that iterates over command-line filenames in @ARGV. Usually written as the null filehandle in the angle operator <> (or <<>>)		
	ARGVOUT	The special filehandle that points to the currently open output file when doing edit-in-place processing with -i. <ul style="list-style-type: none">Useful when you have to do a lot of inserting and don't want to keep modifying \$_		
	STDIN	<STDIN> : line input operator for the STDIN filehandle (for the standard input). <ul style="list-style-type: none">Each time <STDIN> is used in scalar context, Perl reads 1 complete line of the standard input and uses it as the value of <STDIN>.The string includes a line termination character. Use the chomp built-in function to strip it off the variable.If <STDIN> is read in list context, it returns all lines inside a list! For example, <code>foreach (<STDIN>) { ... }</code> reads the entire stdin in 1 step: \$_ holds it all!		
		<pre>while (<STDIN>) { # print all print; # lines of # stdin }</pre>	<pre>while (defined(\$_ = <STDIN>)) { print \$_; }</pre>	The code in the left-most cell is the shortest form. It is equivalent to the code beside it; each line of stdin is stored in the default variable \$_ and the loop stops on end at which time <STDIN> returns undef.
	STDOUT	standard output		
	STDERR	standard error Note: generally STDERR is not buffered, while STDOUT is buffered by default. Text sent on STDERR may show up before STDOUT. <ul style="list-style-type: none">Print a new line on STDOUT to help flushing it or assign 1 to \$ to activate auto-flush.		
	DATA			

Perl 5 Built-in Functions 🚧🚧

Perl Functions Perl syntax	👉 To get information about a Perl function from the command line use the perldoc -f command. <ul style="list-style-type: none">To get information about print use: perldoc -f print
⚠ Cautionary notes	Some of the Perl functions exhibit various limitations and the vary over Perl versions. This section describes the ones I am aware and the proposed alternatives.
<ul style="list-style-type: none">each keyword is brokenUse Var::Pairs instead.	Do NOT use the built-in each . It is broken, as described by Damian Conway in his Modern Perl Best Practice O'Reilly course , section control structure. <ul style="list-style-type: none">each is not re-entrant:<ul style="list-style-type: none">nested loops of each over the same hash does not work as expected and will create infinite loop since the nested loop each juts iterates from where the first loop each left it.Exiting the loop leaves the state of the each internal pointer at the current location.<ul style="list-style-type: none">If you use each on the same hash later it will resume from where it left, it will not start form the beginning.

Perl 5 Statements 🚧🚧

Loop control	See perlsyn for more information on Perl syntax which includes declarations, blocks, loops, labels, subroutines, etc...		
👉 Use the last and redo inside a naked block of code to control looping.	loop control keywords: <ul style="list-style-type: none">last ⚠: exits the loop.next ⚠: starts the next iteration of the loop.redo ⚠: restarts the loop block without evaluating the condition again.	The last , next , and redo loop control keywords work in the following constructs: <ul style="list-style-type: none"><code>while (condition) { ... }</code><code>until (condition) { ... }</code><code>for (init; condition; continue) { ... }</code><code>foreach array { ... }</code>naked block: <code>{ ... }</code>	Notes: <ul style="list-style-type: none">The while and foreach loops may have a continue block: executed before evaluating condition again, which corresponds to the 3rd part of a for loop statement. See this @ stackOverflow.Blocks can be labelled ⚠ as targets to last, next, and redo
Statement modifiers	<ul style="list-style-type: none"><code>if EXPR</code><code>unless EXPR</code><code>while EXPR</code><code>until EXPR</code><code>for LIST</code><code>foreach LIST</code><code>when EXPR</code>	The for and foreach statements impose a list context ; the complete list is processed. Therefore a loop like the following trying to stop on a line that has <code>"__END__"</code> on it will not work since it reads all of STDIN: <pre>foreach (<STDIN>) { last if ?__END__/?; ...; }</pre>	The while statement imposes a scalar context ; it takes one line at a time from <code><STDIN></code> and the following code works properly: <pre>while (<STDIN>) { last if /__END__/?; ...; }</pre>
do block	<ul style="list-style-type: none">The do block is *very useful* to set a value based on several conditions, just as the <code>? : conditional operator</code> but with an explicit block that may use scoped variables.Takes advantage of a block value is the value of the last expression executed inside the block. Do <code>"not"</code> return from the block.The last, next and redo cannot be used inside do blocks.		<pre>my \$next_step = do { my (\$perl_nirvana, \$emacs_nirvana) = check-nirvana-levels(); if (\$perl_nirvana < 5 && \$emacs_nirvana < 8) { 'study-Perl' } elsif (some_other_cond()) { 'time-to-cook' } elsif (\$emacs_nirvana < 7) { 'look-into-eieio' } else { \$isit_winter? 'go-skiing' : 'go-canoeing' } }</pre>
Compound statements			
<code>if, elsif, else</code>			
<code>unless</code>			
<code>? : conditional operator</code>			

Perl 5 Subroutines🚧

Perl subroutines			
subroutine &	<ul style="list-style-type: none">Why we teach the subroutine ampersandWhy should I use the & to call a Perl subroutine? @ StackOverflow		Another point of view: Subroutines and Ampersands
Subroutine Prototypes	An older Perl feature. Clashes with subroutine signatures as of Perl v5.20. In <i>Perl >= v5.20</i> put the :prototype attribute before subroutine prototype parenthesis.		
Subroutine signatures <ul style="list-style-type: none"><i>Perl >=5.36</i>: Stable<i>Perl >= 5.20</i>: Experimental See: Use v5.20 subroutine signatures	Exactly zero arguments	()	Zero or 1 argument, no default, unnamed: (\$=)
	Zero or 1 argument, no default, named	(\$val=)	Zero or 1 argument, named, with default (\$val=1)
	exactly 1 named argument:	(\$val)	Exactly 2 arguments (\$v1, \$v2)
	2, 3 or 4 arguments no defaults:	(\$v1, \$v2, \$=, \$=)	2,3 or 4 arguments, 1 default: (\$v1, \$v2, \$v3='a', \$=)
	Two or more, any number of arguments.	(\$v1, \$v2, @)	Two or more arguments, remainders into a named array: (\$v1, \$v2, @rest)
	Two or more arguments: an even number	(\$v1, \$v2, %)	Two or more arguments, remainders into a named hash: (\$v1, \$v2, %rest)
	Class method	(\$class, ...)	Object method (\$self, ...)
Returned value	<ul style="list-style-type: none">The result of the last evaluated expression is implicitly returnedThe return operator can be used but it's not required unless used to change execution flow (return immediately from the subroutine).The subroutine can return a scalar in scalar context or a list if called in list context.<ul style="list-style-type: none">Inside the subroutine, use the wantarray function to determine the context of the subroutine call.		

Perl 5 Modules🚧

Perl Modules		
Perl core modules	<ul style="list-style-type: none">How to detect where a module is installed : <code>perldoc -l Module</code>How to check if a module is part of Perl core : <code>corelist</code> Module (Perl >= v5.9.2)	
Access to Modules	Provide access to modules in your code with one of the following: <code>do</code> , <code>require</code> or <code>use</code>	
Modules @perltutorial Modules Using simple modules ⚡	<code>do</code>	Looks for the module file by searching the <code>@INC</code> path. Performed at run time (and therefore can be done conditionally). <ul style="list-style-type: none">If Perl finds the file, it places the code inside the calling program and executes it. Otherwise, Perl will skip the <code>do</code> statement silently.👉 The "included" code does not have access to the lexical variables from the main program.Skip the <code>@INC</code> path lookup if given a file path starting with <code>./</code>, <code>../</code>, or <code>/</code>
	<code>require</code>	Loads the module file once, also searching the <code>@INC</code> path. Performed at run time (and therefore can be done conditionally). <ul style="list-style-type: none">If the <code>require</code> for the same file appears twice, Perl ignores it. Perl will issue an error message if it cannot find the file (as opposed to <code>do</code>).Skip the <code>@INC</code> path lookup if given a file path starting with <code>./</code>, <code>../</code>, or <code>/</code>
	<code>use</code>	Similar to <code>require</code> except that Perl applies it before the program starts: it's done at compile time . Modify it dynamically in a <code>BEGIN</code> block. See <code>IntP⚡</code> . <ul style="list-style-type: none">Therefore the <code>use</code> statement cannot be invoked inside conditional statements such as if-else. Used often to include a module in a program. That imports the defaults as defined by the module's code. Select what to import with one of the two equivalent forms: (See <code>IntP⚡</code>): <ul style="list-style-type: none"><code>use Module::Name ('function_a', 'function_b');</code><code>use Module::Name qw(function_a function_b);</code><code>use Module::Name ();</code> # import nothing. All accesses to the module must be done with <code>Module::Name::something</code>
Error handling for: Can't locate in @INC <ul style="list-style-type: none">How to fix that	For the above statements to work Perl must be able to identify the location of the requested module(s). <ul style="list-style-type: none">Perl looks for a module code inside the directories identified by the <code>@INC</code> array. if you have: <code>use The::Module;</code> inside your code, Perl looks for a sub-directory named 'The' containing a file named 'Module.pm' inside each <code>@INC</code> directory. If Perl does not find it, there are multiple ways to solve the problem : <ul style="list-style-type: none">Add the required directory to the list of directories identified in the ':' separated list in the PERL5LIB environment variable. (use ';' as separators in Windows).Add a <code>use lib 'path/to/the/directory';</code> statement inside your Perl file to add the required directory when executing a specific piece of Perl code, at compile time.Run Perl with the -I (capital i) option to run the code with the extra directory added to <code>@INC</code> array. To List the directories used by Perl from one of the following equivalent command lines: <ul style="list-style-type: none"><code>perl -e 'print join("\n", @INC), "\n";'</code><code>perl -le 'print for INC;'</code> <div>You can also get more information with <code>perl -V</code></div>	
See Also: <code>IntP⚡</code>		
<ul style="list-style-type: none">See: <code>show-perl-inc @ USRHOME</code>		
Specially Named Blocks	5 specially named blocks are run at the beginning or end of a running program: BEGIN , UNITCHECK , CHECK , INIT and END . See: <code>BEGIN</code> block - running code during compilation. Note the security risk warnings . The <code>BEGIN</code> block is used to implement other Perl functionality.	
Declare packages	In Perl a package can span several files and one file may contain the code of several packages. The package starts with the <code>package</code> keyword. The special <code>__PACKAGE__</code> literal contains the name of the current package.	

Topic: Data Introspection🚧

Data Introspection				
Using Perl Debugger • Debugger Tutorial	Debug a program:		<code>perl -d</code> program_name program_args	
	Debug interactive session:		<code>perl -d -e 0</code>	
Debugger commands	<code>q</code>	Quit debugger	<code>s</code>	single step
	<code>h</code>	help. List all available commands.	<code>x</code>	evaluate expression
Modules for Data introspection	Data::Dumper <i>(Perl >= 5.005)</i>		The module provides the Dumper function that prints strings that can be used by eval to rebuild the data. <ul style="list-style-type: none">It is similar to the x command of the debugger.Pass reference to the variables , otherwise it extends them to list and show each entry as its own variable. <ul style="list-style-type: none"><code>print Dumper(\@array);</code><code>print Dumper \%hash;</code>	
	Data::Dump <i>(Requires Perl >= v5.6.0)</i>		Provides a dump function that has nicer output, but is not eval compatible. <ul style="list-style-type: none">dump() prints on the stdout. No need to use print. <code>use Data::Dump qw(dump);</code> <code>dump(\@array);</code> <code>dump(\%hash);</code>	
	Data::Printer		A nicer data dumper, not eval compatible. <ul style="list-style-type: none">It provides the p subroutine that does not require a reference to the variable as it inspects it first.p() prints on the stdout. No need to use print. <code>use Data::Printer;</code> <code>p(@array);</code> <code>p(%hash);</code>	
Modules for Data Marshalling • Data Serialization in Perl	There are several modules, either part of Perl core or outside, that provides mechanism to marshall/serialize and unmarshall/de-serialize data. <ul style="list-style-type: none">See the links at left for more info.			

Topic: Directory Operations 🚧

Directory Operations	In Books: LPo		
Opening Files	All file open operations are relative to the <i>current working directory</i> (for relative file names)		<code>open my \$filehandle, '<:utf8', 'a_relative/path.txt'</code>
Creating temporary files	File::Temp (Perl >= v5.6.1). Using <code>File::Temp</code> <ul style="list-style-type: none">Also see IO::File		
Built-in Functions	Related Functions/Packages / Descriptions	Notes	
Getting file names by: <ul style="list-style-type: none">Globbering :<ul style="list-style-type: none">with globwith the glob operator <code><></code>	File::Glob (Perl >= v5.6.0) - provides more control.	Example:	<code>my @all_files = glob '*';</code> <code>my @perl_files = glob '*.pm *.pl'; # 2 globs, space-separated</code>
	The <> operator is identifying: <ul style="list-style-type: none">a filehandle, when: the item inside <> is a Perl identifier or an indirect file handle read scalar,a glob expression otherwise.	Glob examples:	<code>my @all_files = <'*>;</code> <code>my @all_files = <*>; # 1 glob: no space, no need for string</code> <code>my @perl_files = <*.pm *.pl*>; # 2 globs, space-separated</code>
	See: readline		<code>my \$etc_dir = '/etc';</code> <code>my @etc_dir_files = <\$etc_dir/* \$etc_dir/*.>;</code>
		Filehandle examples:	<code>my @files = <LARRY/*>; # a glob</code> <code>my @this_lines = <LARRY>; # a filehandle read</code> <code>my \$name = 'LARRY';</code> <code>my @this_lines = <\$name>; # indirect filehandle read of LARRY handle</code> <code>my @same_lines = readline LARRY; # another way to write above</code> <code>my @same_lines = readline \$name;</code>
<ul style="list-style-type: none">with a directory handle LPo	<ul style="list-style-type: none">opendir : open a directory: get a directory handlereaddir : read the directory handle. But see this.closedir : close the directory handle.DirHandle (Perl <= 5.5)File::Spec::Functions (Perl >= v5.5.4)Path::Class	Example: iterate explicitly over a list of file names extracted from the directory using these 3 functions.	<code>my \$dir = '/usr/bin';</code> <code>opendir my \$dh, \$dir or die "Failed opening \$dir: \$!";</code> <code>foreach \$file (readdir \$dh) {</code> <code> print "File \$file is inside \$dir\n"; # ⚠️ no path in name!</code> <code>}</code> <code>closedir \$dh;</code>
Creating directory	<ul style="list-style-type: none">mkdir	Example:	<code>mkdir \$dir_name, oct(\$permissions); # octal for permissions</code> <code>mkdir \$dir_name, 0700; # do not use "0700", it's 700 decimal!</code>
Removing directory	<ul style="list-style-type: none">rmdir Removes an empty directory.File::Path remove_tree , rmtree remove dir & files (Perl >= v5.0.1)		
Removing files	<ul style="list-style-type: none">unlink a list or \$_		<code>unlink 'file1.txt', 'file2.txt';</code> <code>unlink qw(file1.txt file2.txt);</code> <code>unlink glob 'file?.txt'</code>
Renaming files	<ul style="list-style-type: none">rename an old file name to a new one.<ul style="list-style-type: none">The fat comma operator is sometimes used to highlight what is the old and the new name.	As in here:	<code>rename 'old_name' , 'new_name';</code> <code>rename old_name => 'new_name'; # use fat comma to quote word left of it.</code>
Changing permissions	<ul style="list-style-type: none">chmod changes file permissions		
Changing ownership	<ul style="list-style-type: none">chown changes file ownership		
Creating Hard link	<ul style="list-style-type: none">link to create a hard link		
Creating symbolic link	<ul style="list-style-type: none">symlink to create a symbolic link		
chdir Change current working directory	<ul style="list-style-type: none">File::chdirFile::HomeDir	<ul style="list-style-type: none">Change the current working directory.chdir without argument attempt to change to user home directory using the <code>\$ENV{HOME}</code> and <code>\$ENV{LOGDIR}</code> environment values if ⚠️ they are set. The File::HomeDir module helps in setting them.The built-in chdir is global ⚠️ for the entire program. Use File::chdir facilities for localized operations.	
Modules	Functions	Extra Information	
	Legend: Exported by default, exported on request, Win32 specific		
Cwd	<ul style="list-style-type: none">getcwd, cwd, fastcwd, fastgetcwd, getdcwdabs_path, realpath, fast_abs_path		<code>use Cwd;</code> <code>my \$curdir = getcwd;</code> <code>print "cwd is \$curdir\n";</code>
File::Basename	<ul style="list-style-type: none">fileparse, basename, dirname.		
File::Spec File::Spec::Functions	<ul style="list-style-type: none">functional interface to methods: canonpath, catdir, catfile, curdir, rootdir, updir, no_upwards, file_name_is_absolute, path. <code>devnul</code>, <code>tmpdir</code>, <code>case_tolerant</code>, <code>splitpath</code>, <code>splitdir</code>, <code>catpath</code>, <code>abs2rel</code>, <code>rel2abs</code>. All can be imported by using the <code>:ALL</code> tag.		
File::Find : Traverse a directory tree. See: File::Find::Closures	find , finddepth , %options . In wanted : File::Find::dir , File::Find::name Note that \$_ gets the base name of the file (no path). It is used to perform filetest operations in the example here (as explicit argument to -s, and implicit argument to -d and -f). This traverses the entire tree.	<code>use File::Find;</code> <code>find(sub {printf("- %-10s : %4d, %s\n", \$_, -s \$_, File::Find::name)</code> <code> if (-d or -f) and (\$_ ne "."); }, '.');</code> # in the above it lists the names of files inside all directories not showing the directory name	

Topic: List Operations 🚧

List Operators			
Sorting lists	sort	Sort a list	<code>my @sorted = sort @unsorted_list;</code> in place: <code>my @data = sort @data;</code>
	reverse	Sort a list in reverse order	<code>my @rsorted = reverse @unsorted_list;</code> in place: <code>my @data = reverse @data;</code>
Filtering list with grep	<code>my @adult_ages = grep \$_ > 18, @ages;</code>	<code>my @lucky_ages = grep /7\$/, @ages; # all that end with 7</code>	<code>my @read_ages = grep { \$_ >= 7 && \$_ <= 77 } @ages;</code>
Counting matches	<code>my \$count = grep \$_ > 18, @ages;</code>		
	An expression, subroutine or block with trailing boolean can be used as the grep criteria. Each item in the list is identified inside grep by \$_ <ul style="list-style-type: none">The block is an anonymous subroutine. 🐘 Return a boolean from the subroutine, but fall-off, do not return, from a block!		
Transform a list with map			

Topic: Process control ⚠️

Process Control	In Books: LPα		Important security information: perldoc perlsec	
Environment Variables	Inside the %ENV hash.		Perl %Config hash: Perl configuration information. For example, whether it support threads, what are path separators, etc... <ul style="list-style-type: none">To use it: <code>use Config;</code>	
Built-in Functions	Example	Description/ Notes		
system (2 functions) <ul style="list-style-type: none">using the shell<ul style="list-style-type: none">security risk?avoiding the shell<ul style="list-style-type: none">other syntax	system 'ls -l \$HOME';		Run child process asynchronously using parent's stdin, stdout and stderr, using the OS native command shell.	
	system "cd \$project; make &";		Use the Unix shell to execute a long running build asynchronously. 🙌 However: avoid using the shell like this . <ul style="list-style-type: none">Using the shell to build commands from unvalidated user input data may lead to security issues.	
	system 'tar', 'cvf', \$tarfile, @directories;		No shell invoked when more than 1 argument is passed to system. No shell interpretation, piping, re-direction done.	
	system ('tar', @arguments);		0 means success: <code>unless (system 'tar', arguments) { print "tar command success\n"; }</code>	
	system ({ \$prog }, \$arg0, @args);			
	👉 Note that if the string contain no shell metacharacters it is executed directly (not through a shell).			
system return value: <ul style="list-style-type: none">A value of 0 usually means all was OK.	2 bytes:	MSByte: child program exit code. LSByte: system-specific information bits: <ul style="list-style-type: none">0x80 : set on core dump.0x7f : signal number	<code>my \$retval = system(...);</code> <code>my \$childp_exitcode = \$retval >> 8;</code> <code>my \$had_core_dump = (\$retval & 0x80) == 0x80? 1 : 0;</code> <code>my signal_number = \$retval & 0x7f;</code>	← shift most significant byte ← use least significant byte
exec	Unlike system, exec does not return to the parent Perl process. Use: <code>exec 'the_program' or die "Could not run: \$!"; #or warn or exit</code>			
backquotes ``	Use backquotes to capture the stdout of a program. That's the main point of using it. <ul style="list-style-type: none">The trailing newline is not filtered out; it can be filter by chomp.			<code>chomp(my \$current_date = `date`);</code>
	<ul style="list-style-type: none">The value inside the backquotes is treated like the single double quote string argument of system: it will invoke the shell if there are any shell meta-characters and supports interpolation.<ul style="list-style-type: none">The following example builds a dictionary (hash) of topics with the text extracted from perldoc.Note that ``...` is also written as qx/ ... /backquote operation in scalar context returns 1 string. In list context it returns a list of strings (1 per line).			<code>my @topics = qw(die warn exit);</code> <code>my %info;</code> <code>foreach (@topics) {</code> <code> \$info{\$_} = `perldoc -t -f \$_`;</code> <code>}</code>
Modules				
Capture streams	<ul style="list-style-type: none">Capture::Tiny	Can be used to capture the stdout and stderr streams for various ways if executing other programs		
Inter-process support	<ul style="list-style-type: none">IPC::System::Simple	Can also be used to capture streams and provide more inter-process support. <ul style="list-style-type: none">It provides systemx which never uses the shell, along with other useful functions.		
Processes as filehandles	In Books: LPα			
Perl ← program	Launching a process that pipes into the Perl process	<code>open DATE, 'date ' or die "Cannot pipe from date: \$!";</code>		Use a bare word to define the DATE file handle.
		<code>open my \$date_fh, ' ', 'date' or die "Cannot pipe from date: \$!";</code>		This one and the others define a local file handle variable. The file handle variable can later be used to read, as the above one, but is not global.
		<code>open my \$ps_fh, ' ', 'ps', 'aux' or die "Cannot pipe from ps: \$!";</code>		
		<code>open my \$find_fh, ' -', 'find', qw(. -name ".p[lm]" -print) or die "Cannot pipe from find: \$!";</code>		
Perl → program	Launching a process that the Perl process pipes into.	<code>open my \$dispatcher_fh, ' -', 'dispatcher', qw('—to-perl-groups' 'Help!') or die "Cannot pipe to the dispatcher: \$!";</code>		
Forking	In Books: LPα . See also: Linux fork(2) system call, QA: Why do we need fort to create new processes? Why fork woks the way it does?			
fork with exec and waitpid See also: <ul style="list-style-type: none">Other IPC functionsPerl IPC	<ul style="list-style-type: none">fork the process into parent and child.in the child process start the program with execIn the parent process wait for the program termination with waitpid	<code>defined(my \$process_id = fork) or die "Fork failed: \$!";</code> <code>unless (\$process_id) {</code> <code> # Inside the child process (created by fork)</code> <code> exec 'long_running_process' or die "Failed starting long_running_process: \$!";</code> <code>}</code> <code># Inside the parent process, wait for completion of long_running_process.</code> <code>waitpid(\$process_id, 0);</code>		
Signals	In Books: LPα			
kill	Sends a signal to a list of processes. <ul style="list-style-type: none">The signal may be identified by number or name (string), which is more portable.The %Config{sign_name} provides the supported signal names.			<code>kill 'INT', \$pid or die "Can't signal \$pid with SIGINT: \$!";</code>
	<ul style="list-style-type: none">Note that the <i>fat comma</i> operator (=>) can be used to automatically quote signal name:			<code>kill INT => \$pid or die "Can't signal \$pid with SIGINT: \$!";</code>
	<ul style="list-style-type: none">If the signal is 0 or "ZERO" no signal is sent to the process; instead Perl checks if it's possible to send a signal to the process: ie: if the process exists.			<code>unless (kill 0, \$process_id) {</code> <code> warn "Process \$process_id is no longer running!";</code> <code>}</code>
	<ul style="list-style-type: none">If the signal is a negative number or a string that starts with '-' the signal is sent to the process group identified by the process scalar argument.			<ul style="list-style-type: none"><code>kill '-KILL', \$process_group</code><code>kill -9, \$process_group</code>
Signal handlers	<ul style="list-style-type: none">Set the signal handler by setting %SIG for the signal name (with no 'SIG' prefix) to a string holding the name of the subroutine.			<code>\$SIG{ 'INT' } = 'dispatcher_int_handler';</code>

PerlTidy formatting control ⚠️

perltidy option	Option	Impact
indentation style	<ul style="list-style-type: none">-bl,--opening-brace-on-new-line--brace-left	<ul style="list-style-type: none">Without this option (the default) the code indentation style selected is K&R style.With this option, the indentation style is Allman/BSD style.