Perl Memoire

[perlrun 3](#_Toc74598146)

[Executing Perl 3](#_Toc74598147)

[Options 3](#_Toc74598148)

[types 5](#_Toc74598149)

[numbers 5](#_Toc74598150)

[hex and binary 5](#_Toc74598151)

[scalar, list, and void context 5](#_Toc74598152)

[boolean truthiness 5](#_Toc74598153)

[undef 5](#_Toc74598154)

[lvalue 5](#_Toc74598155)

[big numbers 5](#_Toc74598156)

[additional types 5](#_Toc74598157)

[one-liner core-util equivalents 7](#_Toc74598158)

[installing modules 8](#_Toc74598159)

[quote operators 9](#_Toc74598160)

[keywords 10](#_Toc74598161)

[built-in functions 11](#_Toc74598162)

[error handling 11](#_Toc74598163)

[flow-control 11](#_Toc74598164)

[binary data functions 12](#_Toc74598165)

[special variables 13](#_Toc74598166)

[global scalars 13](#_Toc74598167)

[global arrays 14](#_Toc74598168)

[global hashes 14](#_Toc74598169)

[global filehandles 14](#_Toc74598170)

[global constants 15](#_Toc74598171)

[format variables 15](#_Toc74598172)

[error variables 15](#_Toc74598173)

[interpreter state variables 16](#_Toc74598174)

[deprecated/removed variables 17](#_Toc74598175)

[input/output 18](#_Toc74598176)

[built-in filehandles 18](#_Toc74598177)

[variable in format string 18](#_Toc74598178)

[diamond operator <> 18](#_Toc74598179)

[input/output functions 18](#_Toc74598180)

[file tests 20](#_Toc74598181)

[error handling 22](#_Toc74598182)

[strings 23](#_Toc74598183)

[string functions 23](#_Toc74598184)

[arrays 24](#_Toc74598185)

[syntax 24](#_Toc74598186)

[array functions 24](#_Toc74598187)

[hashes 26](#_Toc74598188)

[syntax 26](#_Toc74598189)

[hash functions 26](#_Toc74598190)

[regex 28](#_Toc74598191)

[binding operator 28](#_Toc74598192)

[modifiers 28](#_Toc74598193)

[capture groups 28](#_Toc74598194)

[all matches as list 28](#_Toc74598195)

[regex 30](#_Toc74598196)

[regex variables 30](#_Toc74598197)

[files 32](#_Toc74598198)

[file functions 32](#_Toc74598199)

[maths 33](#_Toc74598200)

[math functions 33](#_Toc74598201)

[process management 34](#_Toc74598202)

[sockets 35](#_Toc74598203)

[packages 36](#_Toc74598204)

[Module Data 37](#_Toc74598205)

[Data::Dumper 37](#_Toc74598206)

[Features by version 38](#_Toc74598207)

# perlrun

## Executing Perl

Specify the source for the perl shell command:

1. Line-by-line, as text argument to '-e' or '-E'
2. Contained in the first filename in the command, only if *1)* not specified first
3. Via standard-input, only if neither *1)* or *2)* are given.

## Options

-a Autosplit line mode (*$\_* into *@F* array) (delimiter -F<delim>). Implicitly sets '-n'.

-F<delim> Specify pattern to split on for '-a'.

Use //, '', or "", or '' will be used implicitly. Sets both '-a' and '-n'.

-n Implicit loop over filename arguments, without printing lines.

-p Implicit loop over filename arguments, prints lines. Overrides '-n'.

-e Specify line of program as argument text

-E As per '-e', but enable optional features (see below)

-i[extension] Specifies that files processed by *<>* construct are to be edited in place.

If *extension* is supplied, backup file is made (see below for rules)

-l[octnum] Automatic line-ending processing. When used with '-n' or '-p', *$/* (IRS) is chomped. Sets *$\*

(ORS) to *octnum*, or if not given, to current value of *$/*.

-0[octal/hex] Specify *$/* (IRS) as octal or hexadecimal, or set to null character *\0* if value not given.

-I[directory] Prepend *directory* to module search path (*@INC*)

-s Rudimentary switch parsing

-d Run under debugger

-dt Run under debugger, indicates use of threads.

-h Help, prints summary of options

-D<> Set debugging flags (see options below)

-c Check syntax of program (runs *BEGIN* and *CHECK* blocks).

-C Controls Unicode features (see options below).

-f Disable executing sitecustomize at start-up

-m[module] Execute 'use *module*' at start-up (but do not call module imports)

-M[module] Execute 'use *module*' at start-up

-S Use PATH to search for program unless name contains path seperators

-T Taint, prevent insecure operations

-t Taint, but with warnings instead of fatal errors

-w Enable warnings (sets *$^W*)

-W Force enable warnings

-X Force disable warnings

-u Dump core after compiling program

-U Allow unsafe operations

-v Print version of perl

-V Print summary of perl configuration, and values of *@INC*

-x[directory] Tell perl that program is embedded in unrelated text. If *directory* is specified, switch to said

directory before running.

Implicit loop over filename arguments (non-printing) '-n'

perl -ne '...'

is equivalent to:

while (<>) { ... }

Implicit loop over filename arguments (printing) '-p'

perl -pe '...'

is equivalent to:

while (<>) { ... } continue { die $!\n" unless print $\_; }

Automatic line-end processing '-l'

Inplace editing backup file rules '-i'

Debugging flags '-D'

Rudimentary switch parsing '-s'

Unicode options '-C'

Optional features '-E'

# types

## numbers

<All numbers in perl are handled as doubles?>

## hex and binary

See *pack()/unpack()* for <…>

### hex and binary literals:

0b1111\_1111

0xAf

## scalar, list, and void context

The assignment operator *‘=’* performs list assignment if the LHS is an aggregate, otherwise it performs scalar assignment.

Aggregates include:

Any expression in parentheses *‘()’*

An array, eg: *@var\_array*

An array slice, eg: *@var\_array[...]*

A hash: eg: *%var\_hash*

A hash slice, eg: *%var\_hash{...}*

Any of the above, preceded by *‘my’*, *‘our’*, or *‘local’*

Force *Expr* to be interpreted in a scalar context, returning the result:

scalar(*Expr*)

## boolean truthiness

Perl does not have a specific boolean type.

The following evaluate to *false* (anything else evaluates to *true*):

An empty string

Numerical zero

An empty list

The literal string *"0"* *(note that "0.0", "00", "+0", ect. are true as strings)*

An undefined value

An object with an overloaded Boolean operator that evaluates to one of the above

A magical variable that evaluates to one of the above

## undef

The perl equivalent of *null*.

## lvalue

An *lvalue* is an expression which can be written on the LHS of an assignment statement, that is, an expression that defines a specific memory address of a variable.

## big numbers

Math::BigInt

Math::BigRat

Math::BigFloat

## additional types

Math::String

Math::FixedPrecision

Math::Currency

Bit::Vector

Math::BigIntFast

Math::Pari

Math::Cephes

Math::Cephes::Fraction

Math::GMP

Math::GMPz

Math::GMPq

Math::GMPf

# one-liner core-util equivalents

cat:

print <>;

sort:

print sort <>;

# installing modules

cpan <module>

# quote operators

q/STRING/

'STRING'

qq/STRING/

"String"

qx/STRING/

`STRING`

qw/STRING/

<<EOF

# keywords

continue

last

# built-in functions

hex(*Expr*)

Return the numeric value of *Expr* as a hex string. If *Expr* is not given, use *$\_*.

ord(*Expr*)

Return the numeric value of the first character of *Expr*, or zero for an empty string. If *Expr* is not given, use *$\_*.

chr(Number)

Return the character represented by *Number* in either ASCII or Unicode. If *Number* is not given, use *$\_*.

vec(Expr, Offset, Bits)

Treats the string *Expr* as a bit vector with elements of width *Bits* (which must be a power of 2), and returns the value of the element specified by *Offset* as an unsigned integer. If *Bits* is 4 or less, then the bits of each bytes are broken into *8/Bits* groups in little-endian order.

defined(*&func*)

defined(*Expr*)

Return a boolean indicating whether *Expr* has value other than *undef*, or check whether a function *func* has ever been defined.

scalar(*Expr*)

Forces *Expr* to be interpreted in scalar context and return result.

crypt(*Plaintext, Salt*)

## date/time

gmtime()

localtime()

time()

times()

## error handling

die(*List*)

Raises an exception. Inside an *eval*, the exception is stuffed into *$@* and *eval* is terminated with *undef* value. If raised outside all enclosing *eval*-s, the uncaught exception is printed to standard-error, and perl exits with an exit code indicating failure. <Propagating exceptions (with *$@*)>. See <handling exceptions/error handling>.

warn(*List*)

Emits warning, typically printing it to *STDERR*. Interprets *List* and uses *$@* in the same way as *die()*.

## flow-control

next(*Label*)

next(*Expr*)

next()

Like C *'continue'*, start next iteration of loop (after executing the *continue* block, if present). If *Label* or *Expr* is omitted, the command refers to the innermost loop, otherwise to the loop corresponding to the given label. A block is considered to be a loop that executes only once (and therefore *next()* will exit such a block early). Should not be used to exit *grep()* or *map()* operations. A block that returns a value will not do so if exited with *next()*.

last(*Label*)

last(*Expr*)

last()

Like C *'break'*, immediately exit the loop in question (the *continue* block, if any, is not executed). If *Label* or *Expr* is omitted, the command refers to the innermost loop, otherwise to the loop corresponding to the given label. A block is considered to be a loop that executes only once (and therefore *last()* will exit such a block early). Should not be used to exit *grep()* or *map()* operations. A block that returns a values will not do so if exited with *last()*.

redo(*Label*)

redo(*Expr*)

redo()

Restart the loop in question without evaluating the conditional again (the *continue* block, if any, is not executed). If *Label* or *Expr* is omitted, the command refers to the innermost loop, otherwise to the loop corresponding to the given label. A block is considered to be a loop that executes only once (and therefore *redo()* will re-execute the block, effectively turning it into a loop). Should not be used to exit *grep()* or *map()* operations.

## binary data functions

pack(*Template, List*)

Used for binary-to-string conversion. Convert values of *List* into a string, using rules given by *Template*. This string is the concatenation of the converted values.

Example usage, get smiling emoji as string from byte representation:

$foo = pack("U4", 0xf0, 0x9f, 0x98, 0x80);

unpack(*Template, Expr*)

unpack(*Template*)

Used for string-to-binary conversion. Convert string *Expr* (use *$\_* by default if not given) into list of values, using rules given by *Template*.

### hex-string to byte-array

my $emoji\_hex\_str = unpack("H\*", '😀');

# convert to byte-array:

@emoji\_bytes = map({ pack('C', hex($\_)) } ($emoji\_hex\_str =~ /(..)/g));

### pack/unpack template rules

# [special variables](https://perldoc.perl.org/perlvar)

Enable long variable names with:

use English;

Behaviour of the default variable – the following pairs are all equivalent:

while (<>) {...}

while (defined($\_ = <>)) {...}

/REGEX/

$\_ =~ /REGEX/

tr/a-z/A-Z/

$\_ =~ tr/a-z/A-Z/

chomp

chomp($\_)

## global scalars

$\_ $ARG

The default input and pattern-searching space.

@\_ @ARG

In a subroutine, contains parameters passed to said subroutine, and is the default array for *pop()* and *shift()*

$" $LIST\_SEPARATOR

When an array or array slice is interpolated, its elements are separated by this value.

$ARGV

Name of current file when reading from <>

$, $OFS $OUTPUT\_FIELD\_SEPERATOR

If defined, value is printed between each argument given to *print()*. Default is *undef*.

$. $NR $INPUT\_LINE\_NUMBER

Current line-number for the last filehandle accessed

$/ $RS $INPUT\_RECORD\_SEPARATOR

Influences perl's idea of what a 'line' is. Default is *'\n'*. Set to *undef* to read to the end of the file as a line. Set to *""* to read until blank line (and treat consecutive empty lines as a single empty line) as a line. Set to reference to number to read that many bytes as a line. Value is a string, not a regex.

$\ $ORS $OUTPUT\_RECORD\_SEPERATOR

If defined, value is printed after last argument given to *print()*. Default is *undef*.

$$ $PID $PROCESS\_ID

The process number of the perl running this script. Can be set (but doing so is discouraged)

$0 $PROGRAM\_NAME

Contains the name of the program being executed

$( $GID $REAL\_GROUP\_ID

The real gid of this process. If membership of multiple groups is supported, is a space separated list.

$) $EGID $EFFECTIVE\_GROUP\_ID

The effective gid of this process. If membership of multiple groups is supported, is a space separated list.

$< $UID $REAL\_USER\_ID

The real uid of this process

$> $EUID $EFFECTIVE\_USER\_ID

The effective uid of this process

$; $SUBSEP $SUBSCRIPT\_SEPERATOR

The subscript separator for multidimensional array emulation

$a $b

Special variables used by *sort()*

$] $OLD\_PERL\_VERSION

Version of perl interpreter, represented as a decimal *5.XXXYYY*, where *XXX* and *YYY* are the version/subversion

$^F $SYSTEM\_FD\_MAX

The maximum (currently used) system file descriptor (typically 2)

$^I $INPLACE\_EDIT

Current value of the inplace-edit '-i' option. Set to *undef* to disable inplace editing.

$^M

Emergency memory pool

$^O $OSNAME

The name of the OS under which this copy of perl was built

$^T $BASETIME

The time (in seconds since epoch) at which program began running

$^V $PERL\_VERSION

The revision, version, and subversion of the perl interpreter, as a version object (use *$]* for decimal representation)

$^X $EXECUTABLE\_NAME

The (relative or absolute) path of the perl executable, or alternatively the string used to invoke perl

$| $OUTPUT\_AUTOFLUSH

If nonzero, forces a flush after every write or print for the currently selected output channel. Default is 0 (note that a value of zero doesn't necessarily mean output channel is buffered). Does not effect input buffering

${^LAST\_FH}

Reference to the last read filehandle. Set by *<HANDLE>*, *readline()*, *tell()*, *eof()*, and *seek()*.

## global arrays

@F

Contains the fields of each line when autosplit '-a' is enabled

@INC

List of places that *do*, *require*, and *use* operators look for their library files. Consists of any arguments given with -I, followed by default perl library.

@ARGV

Command line arguments intended for script. Note that *$ARGV[0]* is the first argument, not the program name.

## global hashes

%ENV

Contains current shell environment. Any changes are inherited by child processes created by *fork()*

%INC

Entries for each filename included via *do*, *require*, or *use* operators, with the filenames as keys, and path of corresponding file as values. For files loaded via a hook, the <name of the?> hook is used as the key instead.

%SIG

Contains signal handlers

## global filehandles

ARGV

Filehandle that iterates over command line arguments in *@ARGV*

ARGVOUT

Filehandle that points to the currently open output file when doing inplace-editing '-i'

STDIN

STDOUT

STDERR

Filehandles for standard input/output/error

DATA

Filehandle for anything following *\_\_END\_\_* or *\_\_DATA\_\_*

\_ (underscore)

Filehandle used to cache the information from the last *stat*, *lstat*, or file test operator

## global constants

\_\_END\_\_

Indicates logical end of program. Any following text is ignored, but readable via DATA filehandle.

\_\_FILE\_\_

Represent the filename at the point in program where it is used. Not interpolated into strings.

\_\_LINE\_\_

Represents the current line number. Not interpolated into strings.

\_\_PACKAGE\_\_

Represents the current package name at compile time, or undefined if there is no current package. Not interpolated into strings.

## format variables

$^A $ACCUMULATOR

Current value of the *write()* accumulator for *format()* lines

$^L $FORMAT\_FORMFEED

What formats output as a form feed. Default is '\f'.

$% $FORMAT\_PAGE\_NUMBER

The current page number of currently selected output channel

$- $FORMAT\_LINES\_LEFT

Number of lines left of the page of the currently selected output channel

$: $FORMAT\_LINE\_BREAK\_CHARACTERS

The set of characters after which a string may be broken to fill continuation fields (starting with *^*) in a format. Default is *' \n-'* (space, newline, hyphen).

$= $FORMAT\_LINES\_PER\_PAGE

The current page length (printable lines) of the currently selected output channel. Default is 60.

$^ $FORMAT\_TOP\_NAME

The name of the current top-of-page format for the currently selected output channel. Default is the name of the filehandle with *'\_TOP'* appended.

$~ $FORMAT\_NAME

The name of the current report format for the currently selected output channel. Default is the name of the filehandle.

## error variables

$! $ERRNO $OS\_ERROR

Current value of the C *errno* integer value. When referenced as a string, yields the message corresponding to *errno*.

%! %ERRNO %OS\_ERROR

Each element of *%!* Is true only if *$!* is set to that value

$? $CHILD\_ERROR

The status returned by the last pipe close, backtick command, successful call to *wait()* or *waitpid()*, or from *system()*. The exit value of the subprocess is really (*$? >> 8*), and *$? & 127* gives which signal, if any, the process died from, and *$? & 128* reports whether there was a core dump.

$@ $EVAL\_ERROR

The perl error from the last *eval* operator, i.e: the last exception that was caught.

${^CHILD\_ERROR\_NATIVE}

The native status returned by the last pipe close, backtick command, successful call to *wait()* or *waitpid()*, or from *system()*.

$^E $EXTENDED\_OS\_ERROR

Error information specific to the current OS

$^S $EXCEPTIONS\_BEING\_CAUGHT

Current state of the interpreter

$^W $WARNING

The current value of the warning switch, true if '-w' was used, modifiable

${^WARNING\_BITS}

The current set of checks enabled by the *use warnings* pragma

## interpreter state variables

$^C $COMPILING

Current value of flag associated with '-c'

$^D $DEBUGGING

Current value of debugging flags '-D'

${^GLOBAL\_PHASE}

Current pause of the perl interpreter. Possible values are: CONSTRUCT, START, CHECK, INIT, RUN, END, DESTRUCT.

$^H

For internal use. Contains compile-time hints for the perl interpreter.

%^H

Provides the same scoping semantics as *$^H* – can be used for implementation of lexically scoped pragmas

${^OPEN}

For internal use. String, describing input and output layers, separated by *\0* byte.

$^P $PERLDB

For internal use. Variable for debugging support.

${^TAINT}

One if taint mode '-T' is on, zero if it off, minus-one for only taint warnings '-t'

${^SAFE\_LOCALES}

One if safe locale operations are available, zero otherwise

${^UNICODE}

Perl Unicode settings '-C'

${^UTF8CACHE}

State of internal UTF-8 offset caching code. One if on (default), zero if off, minus-one for debugging.

${^UTF8LOCALE}

Whether UTF-8 locale was detected by perl at startup '-CL'

## deprecated/removed variables

$#

Format printed numbers Deprecated in Perl 5, removed in 5.10.

$\*

Enable multiline matching. Deprecated in Perl 5, removed in 5.10.

$[

What is the index of the first element of an array (0 for zero indexing, 1 for one indexing. Deprecated in perl 5.12, has no effect as of 5.30.

# input/output

Evaluating *<STDIN>* (or other line-input operator) in a scalar context returns the next line of input.

Standard use in a loop is:

while (<STDIN>) { ... }

Which is equivalent to:

while (defined($\_ = <STDIN>)) { ... }

Evaluating line-input operator in list context gives all remaining lines as a list.

## built-in filehandles

STDIN (also '-')

STDOUT

STDERR

DATA

ARGV

ARGVOUT

## variable in format string

Specify a variable in a string that is otherwise surrounded with *${var}*, for example, a printf format string with a variable width:

%${field\_width}s

## diamond operator <>

The diamond operator reads line-by-line from each of the files specified as arguments to the perl program, that is, from each value of *@ARGV*, or from standard-input if no such files are specified. Like all line-input operators, it returns a single line at a time in a scalar context, or a list of all lines in a list context.

The diamond operator treats filename '-' as standard input.

The current file being read by the diamond operator is given by *$ARGV*. When reading multiple files with the diamond operator, the line number *$.* is not reset for each file. This can be fixed by closing each file, *ARGV*, upon reaching *eof*:

while (<>) { ... } continue { close ARGV if eof; }

Generally, using the diamond operator in more than one place is indicative of a mistake.

Double diamond '*<<>>'*: Functions like diamond operator, but prevents perl performing 'pipe-open' should a filename contain the special character '|'. Note that double-diamond does not interpret '-' as standard input.

## input/output functions

binmode(*Filehandle[, Layer]*)

Arrange for *Filehandle* to be read/written in binary or text mode as per *Layer*. If *Filehandle* is an expression, the result is taken as the name of the filehandle. Returns true on success, otherwise return *undef* and set *$!*. Defaults to binary mode if *Layer* is omitted. Options for *Layer* are called I/O layers, and include: *':raw'* (binary) *':bytes'*, *':utf8'*, and others.

close(*Filehandle*)

Closes the file or pipe associated with the filehandle, flushes IO buffers, reset the line counter *$.*, and close system file descriptor. Returns true on success, or upon error false and set errno *$!*. Closing a pipe waits for processing executing on pipe to exit, unless it is open in another thread, in which case function will return true without waiting for the child process to terminate. Closing read end of pipe before writing is complete results in writer receiving a *SIGPIPE*.

closedir(*Dirhandle*)

Close a directory opened by *opendir()* and return the success of said operation.

bdmclose(*Hash*)

Break the binding between a BDM file and a hash. Superseded by function *untie()*.

bdmopen(*Hash, Dbname, Mask*)

Bind a DBM, NDBM, SDBM, GDBM, or Berkely-DB file to a hash. Superseded by function *tie()*.

eof(*Filehandle*)

Return 1 if the next read on *Filehandle* will return end-of-file, or if it is not open. If *Filehandle* is not given and parenthesis not used, that is *'eof'*, use the last file read. For empty parentheses, that is, *'eof()*'*,* refer to the pseudofile accessed via the diamond operator, *'<>'*. If *'eof()'* is called after *'<>'* has returned end-of-file, it will default to (new) *@ARGV* list, and should that be empty, therefore to standard-input. In a *'while (<>) { ... }'* loop, *'eof'*, or *'eof(ARGV)'* can detect the end of each file, whereas *'eof()'* will detect the end of the last file only.

fileno(*Dirhandle*)

fileno(*Filehandle*)

Return the file descriptor number (as used by C and POSIX) for a file or directory handle, or undefined for an unopen filehandle. If there is no real file descriptor at OS level (such as filehandle connected to memory object via *open()* with a reference for third argument), minus-one is returned. If *Filehandle* is an expression, the value is taken as an indirect filehandle. Returns *undef* on systems without file descriptor support (and sets *$!*).

flock(*Filehandle, Operation*)

Call Unix file-lock *flock()* on *Filehandle*, returning true or false depending on success. Function may wait indefinitely Note that the *flock* file-lock is an advisory lock – programs that do not also use it may modify files locked with it. Filehandle is flushed before being either locked or unlocked.

Format

Declare a picture format for use by the write function.

getc(*Filehandle*)

Return the next character from the input file attached to *Filehandle*, or *undef* for end-of-file or error (setting *$!* in the latter case). If *Filehandle* is not given, use standard-input. Can be used to fetch single characters from user without waiting for an Enter.

print(*Filehandle, List*)

print(*Filehandle*)

print(*List*)

print()

Print a string or a list of strings. *Filehandle* may be a scalar variable containing the name of a reference to the filehandle. If *Filehandle* is omitted, default to last selected (with *select()*) filehandle, defaults to standard-output. If *List* is omitted, default to *$\_* (if *Filehandle* is a bareword, not indirect, filehandle, like *'FH'*, and not like *'$fh'*, in the latter case *List* is required). The current value of *$,* (OFS) is printed between each list item, the current value of *$\* (ORS) is printed after the entire list. Printing to closed pipe or socket will generate *SIGPIPE* signal.

printf(*Filehandle, Format, List*)

printf(*Filehandle*)

printf(*Format, List*)

printf()

Prints string resolved by *sprint(Format, List)*, without *$\* (ORS) as *print()* does. *Format* is a printf format string, and *List* contains variables to be resolved in said string.

read(*Filehandle, Scalar, Length, Offset*)

read(*Filehandle, Scalar, Length*)

Attempt to read *Length* characters of data into variable *Scalar* from the specified *Filehandle*. Returns number of characters read, zero for end-of-file, or *undef* (and set *$!*) in case of an error. *Scalar* will be grown or shrunk so that the last character read is the last character of *Scalar*. If given *Offset*, read data at position after beginning. A negative *Offset* specifies placement at that many characters from the end of the string. A 'character' is a byte by default, but if *Filehandle* is open with *':utf8'* layer, UTF8-encoded Unicode characters are used.

readdir(*Dirhandle*)

In a scalar context, return the next directory entry for a directory opened by *opendir()*, or *undef* if there are no more. In a list context, returns all the remaining entries in the directory, or an empty list if there are no more. If *Dirhandle* is not the current directory, the directory path will need to be prefixed to the entries before they are used in most operations.

readline(*Expr*)

readline()

Reads from the filehandle whose typeglob is contained in *Expr* (or *\*ARGV* if *Expr* is not provided). In a scalar context, reads and returns the next line until end-of-file is reached, whereupon the next call returns *undef*. In a list context, reads until end-of-file is reached and returns a list of lines. The definition of a line is given by *$/* (IRS). If *$/* is *undef*, the entire file is read (Slurp mode). This is the function which implements the line-input operator, that is, the following are equivalent:

my $line = <STDIN>;

my $line = readline(STDIN);

rewinddir(*Dirhandle*)

Set the current position of *Dirhandle* to the beginning of the directory for the *readdir()* routine

say(*Filehandle, List*)

say(*Filehandle*)

say(*List*)

say()

Like *print()*, but include newline at end of values of *List*, irrespective of value of *$\* (ORS). If *List* is not given, *$\_* is used (in which case *Filehandle* must be a bareword filehandle like *'FH'*, not an indirect one like *'$fh'*. Must be enabled with *'use feature 'say''* or *'use v5.10'* (or higher).

seek(*Filehandle, Position, Whence*)

Sets *Filehandle*-s position, like the C function *'fseek()'*. *Filehandle* may be an expression whose value gives the name of the filehandle. Values of *Whence:* 0 *'SEEK\_SET'* to set new position in bytes to *Position*, 1 *'SEEK\_CUR'* to set new position to current position plus *Position*, and 2 *'SEEK\_END'* to set it to end-of-file plus *Position* (typically negative). These constants are from module *'Fcntl'*. Returns 1 if successful, otherwise false. This function operates on bytes, regardless of encoding being used. If positioning file for *sysread()* or *syswrite()*, use *sysseek()* instead of *seek()*. Some systems require a *seek()* call whenever switching between reading/writing. Function may have effect of calling C function *clearerr()*.

seekdir(*Dirhandle, Pos*)

Set the current position of *Dirhandle* for the *readdir()* routine. *Pos* must be a value returned by *telldir()*.

select(*Filehandle*)

select()

Return the currently selected filehandle. If *Filehandle* is supplied, set it as the new default output filehandle (used as output by *write()* or *print()* when called without specifying filehandle). *Filehandle* may be an expression whose value gives the name of the actual filehandle.

select(*Rbits, Wbits, Ebits, Timeout*)

Calls C function *select(2)* (use for synchronous IO multiplexing) with the bitmasks specified. *'IO::Select'* provides a simpler interface than *select()* for the same task.

<...>.

On error, returns *-1* and sets *$!*. Avoid mixing buffered IO (like *read()*, or *readline()*) with *select()*, use *sysread()* instead.

Usual idiom:

my ($nfound, $timeleft) = select(my $rout = $rin, my $wout = $win, my $eout = $ein, $timeout);

or, without timeout (block until something becomes ready):

my $nfound = select(my $rout = $rin, my $wout = $win, my $eout = $ein, undef);

syscall(*Number, List*)

Calls the system call specified as the first element of List, passing the remaining elements as argument to said call. Note: only up to 14 arguments to function *syscall()* are supported. Returns whatever value is returned by the system call. *-1* is returned on error, and *$!* is set (note that some syscalls return *-1* without having failed, to handle this, set *$! = 0* before calling function, then check *$!* upon return value of *-1*).

<...>.

*syscall(SYS\_pipe())* returns the number of the read end of the pipe, but cannot retrieve the file number of the other end – use *pipe()* instead.

sysread(*Filehandle, Scalar, Length, Offset*)

sysread(*Filehandle, Scalar, Length*)

Attempts to read *Length* bytes into variable *Scalar* from specified *Filehandle*, using C function *read(2)*, bypassing usual *PerlIO* layers. Avoid mixing with calls to other reads: *print()*, *write()*, *seek()*, *tell()*, *eof()*. Return number of bytes actually read, 0 at end-of-file, or *undef* for an error and set *$!*. *Scalar* is grown or shrunk so that the last byte read is the last byte of *Scalar* after reading. If *Offset* is specified, begin reading after that many bytes (or that many bytes counting backwards from end if negative). A positive *Offset* greater than length of *Scalar* results in *Scalar* being padded with *'\0'* bytes before appending read result. If *Filehandle* is marked *':utf8'*, an exception is thrown. There is no *syseof()* – use *sysread()* and check for return value of *0*.

sysseek(*Filehandle, Position, Whence*)

Set *Filehandle*-s system position in bytes using C function *lseek(2)*, bypassing the usual *PerlIO* layers. Values of *Whence:* 0 *'SEEK\_SET'* to set new position in bytes to *Position*, 1 *'SEEK\_CUR'* to set new position to current position plus *Position*, and 2 *'SEEK\_END'* to set it to end-of-file plus *Position* (typically negative). These constants are from module *'Fcntl'*. Returns new position, or undefined on failure (position *0* is returned as *"0 but true"*). Avoid mixing with calls to read functions other than *sysread()*.

syswrite(*Filehandle, Scalar, Length, Offset*)

syswrite(*Filehandle, Scalar, Length*)

syswrite(*Filehandle, Scalar*)

Attempts to write *Length* bytes of data from variable *Scalar* to the specified *Filehandle*, using C function *write(2)*, bypassing usual *PerlIO* layers. If *Length* is not specified, writer entire *Scalar*. avoid mixing with reads, other than *sysread()*, and with *print()*, *write()*, *seek()*, *tell()*, or *eof()*. Returns the number of bytes actually written, or *undef* if there was an error (and set *$!*). If *Length* is greater than the number of bytes in *Scalar* after *Offset*, only as much data as is available will be written. If *Offset* is specified, begin after that many bytes of *Scalar* (or that many bytes counting backwards from end if negative). Will raise exception if filehandle is marked *':utf8'*, or upon attempting to write charaters with code points over *255*.

tell(*Filehandle*)

tell()

Returns the current position in bytes for *Filehandle*, or *-1* on error. *Filehandle* may be an expression whose value gives the name of the actual filehandle. If *Filehandle* is omitted, assume the file last read. Behaviour for standard streams (like *STDIN*) is OS dependent.Do not use on a filehandle that has been manipulated by *sysread()*, *syswrite()*, or *sysseek()*. There is no *systell()*, use *sysseek($fh, 0, 1)* for that.

telldir(*Dirhandle*)

Return the current position of the *readdir()* routines on *Dirhandle*. Result may be given to *seekdir()*. <Has same caveats about possible directory compaction as corresponding system routine>.

truncate(*Filehandle, Length*)

truncate(*Expr, Length*)

Truncate (reduce size of) the file open on *Filehandle*, or with the name given by *Expr*, to the specified *Length*. Raises exception if truncate is unimplemented by OS. Returns true if successful, *undef* on error. Behaviour is undefined if *Length* is greater than length of file. Position in file of *Filehandle* is unchanged.

write(*Filehandle*)

write(*Expr*)

write()

Writes a formatted record to the specified *Filehandle*, or filehandle with name given by *Expr*, using format associated with said file. A format can be associated with current output channel by assigning the name of that format to *$~*. If *Filehandle* and *Expr* are not given, uses default output channel.

chdir()

chmod()

chown()

chroot()

fcntl()

glob()

ioctl()

link()

lstat()

mkdir()

open()

opendir()

readlink()

rename()

rmdir()

stat()

symlink()

sysopen()

umask()

unlink()

utime()

## file tests

-X(*Filehandle*)

-X(*Expr*)

-X(*Dirhandle*)

-X()

A file test, where *X* is one of the letters below. Tests associated file (given by argument, or *$\_* if argument not given, except for *-t* where default is standard input), returns 1 if true, or '' otherwise, unless file doesn't exist (or can't be examined) in which case *undef* is returned and *$!* is set. All tests, except *-l* follow symbolic links.

-r File is readable by effective uid/gid

-w File is writable by effective uid/gid

-x File is executable by effective uid/gid

-o File is owned by effective uid/gid

-R File is readable by real uid/gid

-W File is writable by real uid/gid

-X File is executable by by real uid/gid

-O File is owned by real uid

-e File exists

-z File has zero size

-s File has nonzero size (returns size in bytes)

-f File is a plain file

-d File is a directory

-l File is a symbolic link (fails if symlinks aren't supported by file-system)

-p File is a named pipe (FIFO), or Filehandle is a pipe

-S File is a socket

-b File is a block special file

-c File is a character special file

-t File is opened to a tty

-u File has setuid bit set

-g File has setgid bit set

-k File has sticky bit set

-T File is an ASCII or UTF-8 text file (heuristic guess)

-B File is a binary file (opposite of -T)

-M Script start time minus file modification time, in days

-A Script start time minus file access time, in days

-C Script start time minus inode change time, in days

# error handling

handling exceptions

# strings

## string functions

chomp()

length()

substr()

uc()

ucfirst()

lc()

lcfirst()

chr()

chop()

index()

rindex()

sprint()

ord()

quotemeta()

split()

reverse()

# arrays

## syntax

Length – array returns its length when used in a scalar context

Is Empty:

if (!@var\_array)

if (@var\_array == 0)

if (scalar @var\_array == 0)

## array functions

push(*Array, List*)

Pushes the values in *List* onto the end of *Array* (in-place). Returns the number of elements in resulting array.

pop(*Array*)

Remove the last element of *Array* (in-place) and return it, or return *undef* if *Array* is empty.

shift(*Array*)

Remove the first element of *Array* (in-place) and return it, or return *undef* if *Array* is empty. Remaining elements are all shifted left.

unshift(*Array, List*)

Places the values of *List* onto the beginning of *Array* (in-place), shifting existing values right. Returns number of elements in resulting array.

sort(*List*)

sort(*Subroutine, List*)

sort(*Block, List*)

Sort *List* (not in-place) and return result. Sorting order may be specified by *Subroutine*, or *Block.* Behaviour is undefined in scalar context. *Subroutine* must be a subroutine (or a reference to a subroutine) that returns greater-than zero, less-than zero, or zero, given two arguments, according to how those arguments are to be ordered.

Examples of *Block*:

{ $a cmp $b } # alphabetical

{ $a <=> $b } # numeric

wantarray()

Returns *true* if called within function that was called in list context. Returns *false* if called within function that was called in scalar context. Returns *undef* if called within function that was called in void context.

exists(*Expr*)

If *Expr* specifies an element of a hash, return *true* if the specified element has ever been initialized (even if corresponding value if undefined). Use of *exists()* on list elements is (strongly) discouraged due to often surprising behaviour (use *defined()* instead).

defined(*Var*)

Returns *true* if variable *Var* has value other than *undef*.

grep(*Expression, @Array*)

Extract any elements from the given *Array* which evaluate *true* given the regex *Expression*. Returns list of results in list context, or number of results in scalar context.

split(*Pattern*)

split(*Pattern, Expr*)

split(*Pattern, Expr, Limit*)

Split string *Expr* (use *$\_* if not given) into a list of strings, using *Pattern*. If *Pattern* is an empty string, *Expr* is split between characters. Returns resulting list in list context, or size of this list in scalar context. If *Limit* is given, it specifies the maximum number of fields into which *Expr* may be split (which is 1 greater than the number of splits that will be made). If *Limit* is negative, as many fields as possible will be produced. If it is unspecified, or zero, trailing empty fields are stripped. When assigning result of *split()* to list of variables, *Limit* is implicitly 1 greater than the number of variables in said list.

join(*Delim, List*)

Join the separate strings of *List* into a single string, with fields separated by *Delim*, and return it.

reverse(*List*)

Reverses values in *List* (not in-place). Returns reversed list in list context, or in scalar context, concatenated string of reversed list with characters of each element also reversed.

splice(*Array, Offset, Length, List*)

splice(*Array, Offset, Length*)

splice(*Array, Offset*)

splice(*Array*)

Removes elements designated by *Offset* and *Length* from *Array*, and replace them with elements of *List*, if any, (in-place). In list context, returns elements removed from the array. In scalar context, returns last element removed, or *undef* if no element is removed. If *Offset* is omitted, remove all elements. If *Offset* is negative, use negative indexing. If *Length* is omitted, remove everything following value given by *Offset*. If *Length* is negative, remove from value given by *Offset* except for the last *Length* values of the array.

Assuming *$#a >= $i*, the following are equivalent:

splice(@a, @a, 0, ($x, $y)) push(@a, ($x, $y))

splice(@a, -1) pop(@a)

splice(@a, 0, 1) shift(@a)

splice(@a, 0, 0, ($x, $y)) unshift(@a, ($x, $y))

splice(@a, $i, 1, $y) $a[$i] = $y

grep(*Block, List*)

grep(*Expr, List*)

Evaluate *Block* or *Expr* for each element of *List*, (locally setting *$\_* to each element). In a scalar context, returns the number of times expression was true. In a list context, returns aliases into the original list for those items that return true – that is, modifying values in the list returned by *grep()* also modifies values in the original list (and as this is not obvious, it is best avoided).

map(*Block, List*)

map(*Expr, List*)

Evaluate *Block* or *Expr* for each element of *List*, (locally setting *$\_* to each element). In a scalar context, returns the length of the resulting list. In a list context, returns resulting list. If an expression returns an empty list, it is excluded from the resulting list.

# hashes

Sets of key-value pairs. Order may be altered by insertions and deletions, but remains the same for an unmodified hash instance. A hash evaluates to a list in list context, or to length (number of pairs) in scalar context. The functions *values()*, *keys()*, and *each()* have the same order for a given hash. The iterator used by *each()* is attached to the hash (or array), and a call to *values()* or *keys()* with the same hash resets it, as does <referencing> the hash in a list context.

## syntax

### Initialising:

%var\_hash = ('Welcome' => 10, 'to' => 20, 'Geeks' => 80);

or

%var\_hash = ('Welcome', 10, 'to', 20, 'Geeks', 80);

### Add/set element:

$var\_hash{'abc'} = 123;

### Access element:

print "$var\_hash{'Welcome'}"

### Access slice:

print "@var\_hash{'Welcome', 'to'}"

### Nested hash – access/set element:

print "$var\_hash{'key\_outer'}{'key\_inner'}"

$var\_hash{'key\_outer'}{'key\_inner'} = $value

### Iterate over kv pairs:

foreach my $k (keys %hash\_var) { my $v = $hash\_var{$k}; ... }

while ( my($k, $v) = each(%hash\_var) ) { ... }

The function *each()* should be avoided, use *keys()* as in the first example instead.

### Copy of hash:

%new\_hash = %var\_hash

### Hash to array:

@var\_list = %var\_hash

### Printing a hash:

use Data::Dumper;

print Dumper(\%hash\_var);

or

print "@{[%hash\_var]}";

## hash functions

values(*Array*) # Perl > 5.12

values(*Hash*)

In a list context, returns a list of all the values of *Hash* (or elements of *Array*). In a scalar context, returns number of values. Has the side effect of resetting *Hash* internal iterator. Modifying *$\_* values as they are returned also modifies value in hash. The following equivalent examples sets all values in a hash to 23:

for (values(%var\_hash)) { $\_ = 23; }

for (%var\_hash{keys $var\_hash}) { $\_ = 23; }

keys(*Array*) # Perl > 5.12

keys(*Hash*)

In a list context, returns a list of all the keys of *Hash* (or indices of *Array*). In scalar context, returns number of keys. Has the side effect of resetting *Hash* internal iterator. Modifying *$\_* values as they are returned does not modify keys in hash.

each(*Array*) # Perl > 5.12

each(*Hash*)

In a list context, returns a 2-element (key, value) list for the next element in *Hash* (or index and value of next array element). In scalar context, returns only next key (or index). After all entries from *Hash* have been returned, the next call returns an empty list in list context, or *undef* in scalar context, and the call after that restarts the iteration. Adding or deleting elements while iterating over *Hash* can cause undefined behaviour, except for deleting the most recently returned item. *keys()* should be preferred to *each()* for iterating over hash.

delete(*Expr*)

Given an expression *Expr* that specifies an element or slice of a hash, delete the specified elements from that hash, so that *exists()* no longer returns true for that element. In a list context, returns value(s) deleted. In scalar context, returns last element deleted. Deleting a value that does not exist results in *undef* being returned in the corresponding position. When passed a key/value-slice, two items (key-value pairs) are returned for each item deleted. May be called for array values, but this is strongly discouraged. Deleting array elements does not change the indices of existing values. Example usage:

delete(%var\_hash{'Welcome', 'to'});

exists(*Expr*)

Given an expression *Expr* that specifies an element of a hash, return true if the specified element has ever been initialized (even if current value is *undef*). May be called for array values, but this is strongly discouraged. If *Expr* specifies the name of a subroutine, return true if the specified subroutine has ever been declared. If testing for an element of a nested hash, any intervening elements will be created, for example: calling *exists($ref->{‘A’}->{‘B’})* will create *$ref->{‘A’}* (but not *‘B’*). Example usage:

exists($var\_hash{'Welcome'});

# regex

match operator

m//

substitute operator

s///

transliterate

tr///

## binding operator

By default, these regex operators are matched against *$\_* the default variable. That is:

m/regex/

is equivalent to:

$\_ =~ m/regex/

And:

!m/regex/

is equivalent to:

$\_ !~ m/regex/

When a variable other than *$\_* is used, the regex is matched against that instead.

The match operator does not modify the variable being compared. The substitution and transliterate operators however store the operation result in the same variable being compared.

## modifiers

match

i case insensitive

m match ^$ against newlines instead of against string boundry

o evaluate expression only once

s allows '.' to match newline

x Allows whitespace in the expression for clarity

g globally find all matches

cg allow search to contine even after a global match fails

a use only ASCII versions of character classes

c don't reset pos on failed matches when using /g

substitute

i case insensitive

m match ^$ against newlines instead of against string boundry

o evaluate expression only once

s allows '.' to match newline

x Allows whitespace in the expression for clarity

g Replace all occurences of the found expression

e Evaluate the replacement as a perl statement, and use the return value as replacement text

r Leave origional string alone and return modified copy

transliterate

c Complements SEARCHLIST

d Delete found-but-unreplaced characters

s Squashes duplicate replaced characters

## capture groups

Named capture group

(?<name>REGEX)

(?P<name>REGEX)

Backreference to group (use name *n* where *n* is the number of the group for unnamed groups, or just *\n*)

\g{name}

\k<name>

(?P=name)

## all matches as list

Result of match operator, with *g* (match all) modifier, when evaluated in list context.

## regex

[...] any single character in ...

[^...] any single character not in ...

\* 0 or more occurences

+ 1 or more occurences

? 0 or 1 occurences

{n} exactly n occurences

{n,} n or more occurences

{,m} at most m occurences

{n,m} n to m occurences

a|b a or b

\w word character

\W non-word character

\s whitespace [\t\n\r\f]

\S non-whitespace

\d digit [0-9]

\D non-digit

^ or \A match beginning of string

$ or \Z match end of string (before newline)

\z end of string

\b{} match at Unicode boundary of specified type

\B{} Match where corresponding \b{} doesn't match

\b word bountry when outside brackets, backspace (0x08) inside brackets

\B non word boundry

\G match only after '*pos*' position of last match

\n newline

\t tab

\1 ... \9 n-th grouped subexpression

\10 10th grouped subexpression if matched, otherwise octal representation of char code

Non-greedy quanitifiers

?? 0 or 1

\*? 0 or more

+? 1 or more

{m,n}? specific number

## regex variables

$<digit>

Subpattern from the n-th set of capturing parentheses from the last successful pattern match

@{^CAPTURE}

Contents of the capture buffers (if any) of the last successful pattern match. 0-th element is equivalent to $1, ect.

$& $MATCH

String matched by the last successful pattern match (excluding matches hidden within a BLOCK or *eval()*)

${^MATCH}

Same as $MATCH, without associated performance penalty

$` $PREMATCH

String preceding whatever was matched by the last successful pattern match (excluding matches hidden within a BLOCK or *eval()*)

${^PREMATCH}

Same as $PREMATCH, without associated performance penalty

$' $POSTMATCH

String following whatever was matched by the last successful pattern match (excluding matches hidden within a BLOCK or *eval()*)

${^POSTMATCH}

Same as $POSTMATCH, without associated performance penalty

$+ $LAST\_PAREN\_MATCH

The text matched by the highest used capture group of the last successful search pattern, equal to the highest numbered captured variable with a defined value

$^N $LAST\_SUBMATCH\_RESULT

The text matched by the used group most recently closed (the group with the rightmost closing parenthesis) of the last successful search pattern

$#+

Number of subgroups in last successful match

@+ @LAST\_MATCH\_END

Offsets of the ends of the last successful submatches in the currently active dynamic scope. $+[0] is the offset of the end of the entire match (what *pos()* returns when called on variable that was matched against), $+[1] is the offset past where $1 ends, ect.

@- @LAST\_MATCH\_START

As per @LAST\_MATCH\_END, but with the offsets of the beginnings of the last successful submatches

%+ ${^CAPTURE} $LAST\_PAREN\_MATCH

Named capture groups in the last successful match in the currently active dynamic scope. Keys list only the names of groups that have been captured, value is text matched by capture group. If multiple capture groups have the same name, the leftmost group is kept.

%- %{^CAPTURE\_ALL}

Named capture groups in the last successful match in the currently active dynamic scope. Keys are the names of the capture groups (whether they were found or not), and corresponding values are a list of all matches from groups of said name.

$^R $LAST\_REGEXP\_CODE\_RESULT

The result of evaluation of the last successful *(?{code})* regex assertion.

${^RE\_COMPILE\_RECURSION\_LIMIT}

The maximum number of open but unclosed parenthetical groups there may be at any point in a regex. Default is 1000.

${^RE\_DEBUG\_FLAGS}

Regex debugging flags

${^RE\_TRIE\_MAXBUF}

Controls memory usage of regex optimisations. Default is 65536 which corresponds to 512kB. Higher value trades memory for speed, negative value conserves as much memory as possible.

# files

## file functions

open()

glob()

tell()

getc()

reverse()

rename()

print()

# maths

## math functions

exp()

srand()

sqrt()

rand()

log()

int()

sin()

cos()

atan2()

abs()

# process management

alarm()

exec()

fork()

Perform *fork(2)* system call to create new process that is a duplicate of the current program. Returns child *pid* to the parent process, zero to the child process, or *undef* if unsuccessful. Perl attempts to flush all open output files before forking. File descriptors can inherit file descriptors, these may need to be reopened to */dev/null* by child for parent to complete. On Windows (or other OS-s where system *fork()* is unavailable) some versions of perl will emulate this call. See *'perlfork'*.

getpgrp()

getppid()

getpriority()

kill()

pipe()

readpipe()

setpgrp()

setpriority()

sleep()

system()

times()

wait()

waitpid()

# sockets

accept()

bind()

connect()

getpeername()

getsockname()

getsockopt()

listen()

recv()

send()

setsockopt()

shutdown()

socket()

socketpair()

# packages

@ISA

Array containing list of the package (class) parent classes (if any)

# Module Data

## Data::Dumper

# Features by version

v5.10

State (persistent) variables

Named capture groups

:default indirect multidimensional

bareword\_filehandles

:5.10 bareword\_filehandles indirect

multidimensional say state switch

:5.12 bareword\_filehandles indirect

multidimensional say state switch

unicode\_strings

:5.14 bareword\_filehandles indirect

multidimensional say state switch

unicode\_strings

:5.16 bareword\_filehandles current\_sub evalbytes

fc indirect multidimensional say state

switch unicode\_eval unicode\_strings

:5.18 bareword\_filehandles current\_sub evalbytes

fc indirect multidimensional say state

switch unicode\_eval unicode\_strings

:5.20 bareword\_filehandles current\_sub evalbytes

fc indirect multidimensional say state

switch unicode\_eval unicode\_strings

:5.22 bareword\_filehandles current\_sub evalbytes

fc indirect multidimensional say state

switch unicode\_eval unicode\_strings

:5.24 bareword\_filehandles current\_sub evalbytes

fc indirect multidimensional postderef\_qq

say state switch unicode\_eval

unicode\_strings

:5.26 bareword\_filehandles current\_sub evalbytes

fc indirect multidimensional postderef\_qq

say state switch unicode\_eval

unicode\_strings

:5.28 bareword\_filehandles bitwise current\_sub

evalbytes fc indirect multidimensional

postderef\_qq say state switch unicode\_eval

unicode\_strings

:5.30 bareword\_filehandles bitwise current\_sub

evalbytes fc indirect multidimensional

postderef\_qq say state switch unicode\_eval

unicode\_strings

:5.32 bareword\_filehandles bitwise current\_sub

evalbytes fc indirect multidimensional

postderef\_qq say state switch unicode\_eval

unicode\_strings

:5.34 bareword\_filehandles bitwise current\_sub

evalbytes fc indirect multidimensional

postderef\_qq say state switch unicode\_eval

unicode\_strings