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: Optional features '-E')

-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 *$/*.

-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)

-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*)

## variable initial value

When an uninitialized variable is used in a numeric contents, it has the value *0*.

When used in a string context, it has the value *""*.

## 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 <>;

# in-place editing

Specified by *'-i[extension]'* option, or by setting '*$^I'* special variable.

If *extension* is supplied, it is used to create a backup file. If *extension* contains one or more *'\*'* characters, each of them is replaced with the current filename and the result is used as the backup filename, otherwise *extension* is appended to the current filename to create the backup filename. A directory can be included in the backup filename, provided that directory exists.

In-place editing is done (only) by the *<>* construct. The input file is renamed, the output file is opened as the original filename, and any *print()* statements write to the output file.

Note: if lines are not printed during *<>* loop when in-place editing is enabled, resulting 'edited' file will be empty.

## Equivalent behaviour

The one liner:

perl -p -i.bak -e 's/foo/bar/;'

Is equivalent to:

$extension = '.orig';

LINE: while (<>) {

if ($ARGV ne $oldargv) {

if ($extension !~ /\\*/) {

$backup = $ARGV . $extension;

}

else {

($backup = $extension) =~ s/\\*/$ARGV/g;

}

rename($ARGV, $backup);

open(ARGVOUT, ">$ARGV");

select(ARGVOUT);

$oldargv = $ARGV;

}

s/foo/bar/;

}

continue {

print; # this prints to original filename

}

select(STDOUT);

# quote operators

Customary Generic Meaning Interpolates

'' q{} Literal no

"" qq{} Literal yes

`` qx{} Command yes\*

qw{} Word list no

// m{} Pattern match yes\*

qr{} Pattern yes\*

s{}{} Substitution yes\*

tr{}{} Transliteration no (but see below)

y{}{} Transliteration no (but see below)

<<EOF here-doc yes\*

*\* unless the delimiter is ''*

q/STRING/

'STRING'

Single quoted, literal string. A backslash represents a backslash unless followed by the delimiter or another backslash.

qq/STRING/

"String"

Double quoted, interpolated string.

qx/STRING/

`STRING`

String which is interpolated and executed as system command, and the content of standard output from that command is returned (as opposed to *system()*, which returns exit status). Wildcards, pipes, and redirections are honoured. Use *qx''* to avoid interpolation.

qw/STRING/

Evaluates to a list of the words extracted out of *STRING* using whitespace as word delimiters.

qr/STRING/

Regex quote operator. Interpolates unless *'* (single quotes) are used as delimiter.

<<EOF

STRING

EOF

Here-document. *'EOF'* (or another string) is the terminator. Prefix terminator with *'~'* for an indented here-doc. Quotes may be used around the (first) terminator to determine treatment of text. Default (no quotes) is treated the same as double quotes. Single quotes prevent interpolation. Backticks execute *system()* command and substitute result.

\QSTRING\E

Escape version of *quotemeta()*. Characters of *STRING* are treated as <normal> characters, instead of regex characters.

quotemeta(*Expr*)

quotemeta()

Returns the value of *Expr* (if not given, use *$\_*) with all ASCII non-word (that is, all ASCII characters not matching *[A-Za-z\_0-9]* will be preceded by a backslash

## parsing quotes

Perl uses a *do-what-I-mean* approach to pick the most probable interpretation.

When processing a quoted construct, perl first finds the end of that construct, then interprets the contents.

# keywords

continue

last

# <constructs>

//

Logical Defined-Or, as per *||*, but check LHS for definedness instead of truth

## range

Generating a range with a *start*, *end*, and *step*:

list(range(0, 8+1, 2)) # python equivalent

map { 2\* $\_ } 0..4

for (my $i = 0; $i < 8+1; $i += 2) { push @index\_evens, $i; }

Using List::Gen

range(0, 8+1, 2)

except it can't be installed?

# references

Referencing:

my $ref\_var = \@var;

Dereferencing:

@{ $ref\_var }

The braces may be omitted when the item being dereferenced is a scalar, and the variable is a bareword identifier.

The following are equivalent:

${ref\_var}[$y]

ref\_var->[$y]

and:

${$ref\_var}[$y]

$$ref\_var[$y]

$ref\_var->[$y]

<The arrow is not used for array/hash slices>

'Drop the arrow' rule: If an arrow is between <subscript-y kinds of things>, like square brackets, the arrow can be removed. That is, the following are equivalent:

$var[2]->[1]->[0]

$var[2][1][0]

and (note the first arrow cannot be omitted):

$ref\_var->[2]->[1]->[0]

$ref\_var->[2][1][0]

Checking reference types, the following are equivalent:

ref $var\_ref eq 'HASH'

ref $var\_ref eq ref {}

Check reference is hash-like:

croak "Expected hash reference" unless eval { keys %{$var\_hash}; 1 }

Anonymous array: *[]*

Create an array and assign a reference to that array to a variable, that is, the following are equivalent:

my $fruits; { my @temp = ('pineapple', 'papaya', 'mango'); $fruits = \@temp; }

my $fruits = ['pineapple', 'papaya', 'mango'];

Anonymous hash: *{}*

Create a hash and assign a reference to that hash to a variable. The compiler has to guess whether *{}* refers to a block or an anonymous hash. To explicitly specify an anonymous hash, use *+{…}*, to explicitly specify a block, use *{;…}.*

Autovivification

Perl will create an array reference for an unassigned variable when trying to use an unassigned variable as a list, that is, the following are equivalent:

my $var; @$var = (1, 2, 3);

my $var; $var = [ ]; @$var = (1, 2, 3)

Autovivification also works for hash references.

Anonymous subroutine: *sub {}*

# built-in functions

## string/number format conversion

hex(*Expr*)

hex()

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

ord(*Expr*)

ord()

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

chr(Number)

chr()

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

oct(*Expr*)

oct()

Interprets *Expr*, or *$\_* if not given, as an octal string (if it consists of octal digits, and an optional *'0o'* or *'o'* prefix), and return corresponding numerical value. Digits may be preceded by an underscore, which will be ignored. If *Expr* begins with *'0x'* or *'x'*, it is interpreted as a hex string, and if it begins with *'0b'* or *'b'* it is interpreted as a binary string.

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*)

Like the C function *crypt(3)*, provides a (one-way) hashing function. The same values of *Plaintext* and *Salt* will always produces the same result, and there is no simple way to determine *Plaintext* from this result.

lock(*Thing*)

Place an advisory lock on a shared variable or reference object contain in *Thing*, until the lock goes out of scope. If *Thing* is a scalar it is returned, otherwise returns a reference.

prototype(*Function*)

prototype()

Returns the prototype of a function as a string, or *undef* if the function has no prototype. *Function* can be a reference to, or the name of a function, if it is omitted, *$\_* is used. If the name is prefixed with *'CORE::'*, it is taken to be a perl built-in.

reset(*Expr*)

reset()

Used in a continue block to clear variables and reset *'m?pattern?'* searches. *Expr* is interpreted as a list of single characters (hyphens allow for ranges), and any non-lexical variable (scalar, array, and hash) beginning with one of those variables in the current scope is reset. If *Expr* is not given, <only> match searches are reset. Only variables not defined using *my* (non-lexical variables) can be reset. Always returns *1*.

undef(*Expr*)

undef()

Undefines the value of *Expr*, which must be an *lvalue*, if given. Use on scalars, array (using *'@'*), hash (using *'%'*), a subroutine (using *'&'*), or a typeglob (using *'\*'*). Always returns *undef*. Using *undef()* on an element of a hash or list sets the value associated with the element key/index to *undef*, instead of deleting that element (behaviour which is counter-intuitive and should be avoided, instead use *delete()* to delete hash elements – but not to delete list elements, use *splice()* to delete list elements).

## random

rand(*Expr*)

rand()

Return a random fractional number in range *[0, Expr)*. *Expr* should be positive, and defaults to *1* if not given. Automatically calls *srand()* if it has not been called already. Not considered cryptographically secure, more secure alternatives include: *'Data::Entropy'*, *'Crypt::Random'*, *'Math::Random::Secure'*, and *'Math::TrulyRandom'*. To generate a random integer in range *[a, b)*, use:

int( rand( $b-$a+1 ) ) + $a

srand(*Expr*)

srand()

Seed the RNG used by *rand()*, using (integer) *Expr*, or a semi-random value if not given. Alternatively, use *'srand(time())'*. Should not be called more than once.

## date/time

localtime(*Expr*)

localtime()

Convert the epoch, as returned by *time()*, to a 9-element list (with values from C struct *tm*), adjusted to the local timezone. Resulting list contains values: (*sec, min, hour, mday, mon, year, wday, yday, isdst*). *year* gives the number of years since 1900. If *Expr* is not specified, current value of *time()* is used. Use *'POSIX::strftime()'* to get timezone (or custom formatted datetime string), or library *'DateTime'*. High resolution time is available through *'Time::HiRes'*.

gmtime(*Expr*)

gmtime()

Like *localtime()*, but returned values are localized to Greenwich timezone.

time()

Returns the number of non-leap seconds since the epoch.

times()

Returns a four-element list: (*user, system, childuser, childsystem*), giving the user and system times in seconds for this process, and any exited children of this process.

<Note: Or is it (*user, system, childsystem, childuser*), the order from TutorialPoint?>

## 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

## object-orientation

bless()

dbmclose()

dbmopen()

package()

ref(*Expr*)

ref()

Examine the value of *Expr*, or *$\_* if not given, expecting it to be a reference, and return a string giving information about the reference and its type. If *Expr* is not a reference, returns an empty string. If *Expr* is a reference to a blessed object, then the name of the class into which it is blessed will be returned. If it is a reference to an unblessed scalar object, then the return value is one of: *SCALAR*, *VSTRING*, *REF*, *GLOB*, *LVALUE*, or *REGEXP*, or for an unblessed non-scalar, one of: *ARRAY*, *HASH*, *CODE*, *FORMAT*, *IO*. There is (possible) ambiguity between builtin type names and class names for the results of *ref()*. Use *Sclar::Util::blessed()* for information about blessing, and *Scalar::Util::reftype()* for information about physical types.

tie()

tied()

untie()

use()

## scoping

caller()

import()

local()

my()

our()

package()

state()

use()

## modules

do()

import()

no()

package()

require()

## non-function keywords

and

AUTOLOAD

BEGIN

CHECK

cmp

CORE

\_\_DATA\_\_

default

DESTROY

else

elseif

elsif

END

\_\_END\_\_

eq

for

foreach

ge

given

gt

if

INIT

le

lt

ne

not

or

UNITCHECK

unless

until

when

while

x

xor

# [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.

The following lines are all equivalent:

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

while ($\_ = <STDIN>) { print; }

while (<STDIN>) { print; }

for (;<STDIN>;) { print; }

print while defined($\_ = <STDIN>);

print while ($\_ = <STDIN>);

print while <STDIN>;

A filehandle should never be separated from any subsequent arguments with *','*.

## built-in filehandles

STDIN (also *'-'*)

STDOUT (also *'>-'*)

STDERR

DATA

ARGV

ARGVOUT

## filehandles as variables

## 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

open(*Filehandle, Mode, Expr*)

open(*Filehandle, Mode, Expr, List*)

open(*Filehandle, Mode, Reference*)

open(*Filehandle, Expr*)

open(*Filehandle*)

Associates an internal *Filehandle* with the external file specified by *Expr*, which can be used to perform I/O on that file. The special filenames *'-*' and *'>-*' denote standard input/output respectively.

Alternatively, *Expr* can specify an external command, or a *Reference* to a scalar, to open a filehandle on command or in-memory scalar respectively. *Mode* specifies access mode and (optionally) encoding format to use, see 'open-modes'. If *Expr* begins with *'>&'*, then the rest of the string is interpreted as the filename, or file descriptor if numeric. Use *undef* as value for *Expr* to create an anonymous temporary file (which should be used with mode *'+>'*), for which *seek()* will need to be done before reading. An in-memory file is treated as an octet string (and must therefore not contain any code points over *0xFF* unless opened with truncation). Opening a file does not (usually) lock it in perl. Note that *'open'* may also refer to the module of the same name. Returns some defined value upon success, or *undef* on failure and set *$!*. Calls to *open()* should have an accompanying call to *close()*.

Calling *open()* with two arguments, and a pipe character | preceding or following the command of the second argument is a piped-open, as is calling *open()* with three or more arguments where the second argument (mode) is (read filehandle) '-|' or (write filehandle) '|-'. A piped-open launches a concurrent (parallel) child process. See: piped-open.

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.

Do not use a comma after *Filehandle*. Perl will produce a warning when a comma follows a bareword filehandle, but not a <variable> filehandle.

Note that if *print* is followed by '*('*, the closing *')*' will terminate arguments to *print()* <…> - that is, *print()* uses indirect object syntax, the following are not equivalent:

print($fh, List) # prints ($fh, List) to stdout

print $fh List # prints List to $fh

Instead, the following are equivalent:

print $fh List

$fh->print(List)

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.

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(*Expr*)

chdir(*Filehandle*)

chdir(*Dirhandle*)

chdir()

Change the working directory to that specified by argument, or if argument not given to *$ENV{HOME}* (or to *$ENV{'SYS$LOGIN'}* under VMS). On systems that do not support *fchdir(2)*, passing a filehandle or dirhandle raises an exception. Returns true if successful, false otherwise.

chmod(*List*)

Changes the permissions of a list of files. The first element of *List* must be a numeric the numeric mode (as an octal number, not a string, *0644* is ok, *"0644"* is not, use *oct()* to convert a string), while subsequent values are filenames, or (on systems that support *fchmod(2)*), filehandles (which must be passed as globs or glob references, not barewords). The *S\_I\** constants may be useful, these are imported from *'Fcntl'*.

chown(*List*)

Change the owner and group of a list of files. The first and second elements of *List* must be numeric *uid* and *gid* respectively (*-1* to leave unchanged), while subsequent values are filenames, or (on systems that support *fchmod(2)*), filehandles (which must be passed as globs or glob references, not barewords). Most systems only allow superuser to change ownership of files.

chroot(*Filename*)

chroot()

Like system call of same name, makes the named directory the new root directory for all further pathnames that begin with a *'/'* by the process and all its children (note that it does not change the working directory). If *Filename* is omitted, *$\_* is used. It is required for security that after setting *chroot()* to change directory *chdir()* to the new root directory, otherwise the current working directory may be outside the new root.

ioctl(*Filehandle, Function, Scalar*)

Implements the C function *ioctl(2)*. *Scalar* is read and/or written depending on *Function* (if it is a string, a C pointer to that string will be passed as third argument to system *ioctl()* call, whereas a numeric value will be passed directly). *pack()* and *unpack()* may be needed to manipulate the values of structures used by *ioctl()*. Returns *undef* if OS returns *-1*, the string *"0 but true"* if OS returns zero, and the value itself otherwise (therefore returning true on success and false on failure.

fcntl(*Filehandle, Function, Scalar*)

Note that *'fcntl'* may also refer to the module *'Fcntl'*. Implements the *fcntl(2)* (manipulate file descriptor) function. Raises exception if OSdoesn't implement *fcntl(2)*. Argument processing and return value work like *ioctl()*.

glob(*Expr*)

glob()

In a list context, returns a (possibly empty) list of filename expansions on the value of *Expr* such as the standard Unix shell would do. In a scalar context, iterates through these filenames then returns *undef* when list is exhausted. If *Expr* is not given, *$\_* is used. Function splits arguments on whitespace, eg: *glob("\*.c \*.h")* matches all files with either *'.c'* or *'.h'* extension. The expression *glob(".\* \*")* matches all files in the current working directory. To glob filenames that contain whitespace, use extra quotes around the filename to protect it, eg: *glob('"\*e f\*"')* or *glob(q("\*e f\*"))*. If non-empty braces are the only wildcard characters, no filenames are matched but potentially many strings are returned. For example, despite not matching any filenames, the expression below returns the string immediately below it.

glob("{apple,tomato,cherry}={green,yellow,red}")

*apple=greenapple=yellowapple=redtomato=greentomato=yellowtomato=redcherry=greencherry=yellowcherry=red*

link(*Oldfile, Newfile*)

Create a new filename hard-linked to the old filename. Returns true on success, false otherwise.

symlink(*Oldfile, Newfile*)

Create a new filename symbolically linked to the old filename. Return true on success, false otherwise. Raise exception if OS doesn't support symbolic links.

stat(*Filehandle*)

stat(*Expr*)

stat(*Dirhandle*)

stat()

Returns a 13-eleemnt list giving the status info for a file opened by *Filehandle*, or *Dirhandle*, or with a name given by expression *Expr*, or if no argument is given *$\_* is used. Returns an empty list if unsuccessful. See: 'stat-fields' for meaning of returned list. If successful, sets information associated with special filehandle *'\_'*.

lstat(*Filehandle*)

lstat(*Expr*)

lstat(*Dirhandle*)

lstat()

As per *stat()*, but for a symbolic links instead of the file the link points to.

mkdir(*Filename, Mode*)

mkdir(*Filename*)

mkdir()

Create the directory specified by *Filename*, with permissions specified by *Mode* (which defaults to 0777 if not given). Returns true if successful, otherwise returns false and sets *$!*. Use *make\_path()* from *'File::Path'* for creating recursive directory structures.

opendir(*Dirhandle, Expr*)

Opens a directory with name given by *Expr* for processing by *readdir()*, *telldir()*, *seekdir()*, *rewinddir()*, and *closedir()*. Returns true if successful, otherwise <undef/false?>.

readlink(*Expr*)

readlink()

Returns the value of a symbolic link, or *undef* if unsuccessful and set *$!*. Raises an exception if OS does not implement symlinks.

rename(*Oldname, Newname*)

Changes the name of a file, any existing file *Newname* will be erased. Returns true if successful, otherwise returns false and sets *$!*. Behaviour varies by system, it may not work on directories, open files, or pre-existing files. Use *move()* from *'File::Copy'* for platform independent alternative.

rmdir(*Filename*)

rmdir()

Deletes the directory specified by *Filename*, if that directory is empty. Returns true if successful, otherwise returns false and sets *$!*. If *Filename* is not given, *$\_* is used. Use *rmtree()* from *'File::Path'* to delete directory recursively.

sysopen(*Filehandle, Filename, Mode*)

sysopen(*Filehandle, Filename, Mode, Perms*)

Open the file whose name is given by *Filename* and associate it with *Filehandle*. Calls C function *open(2)* with parameters *Filename*, *Mode*, and *Perms*, bypassing *PerlIO* layers. Returns true on success and *undef* otherwise. Values for *Mode* are system-dependent, use constants from *'Fcntl'* (combine with '|' operator), read-only, write-only, and read-write are: *O\_RDONLY*, *O\_WRONLY*, and *O\_RDWR*. If file does not exist, it is created with (octal) permissions *Perms*, or *0666* if not given. Permissions may be modified by *umask()*. A filehandle created by *sysopen()* can be used by buffered or unbuffered IO, that is, it is not limited to use with *sysread()*, *syswrite()*, and *sysseek()*. May fail when file descriptor exceeds 255, if so use *'POSIX::open()*' as an alternative.

umask(*Expr*)

umask()

Set the *umask* (settings for the permissions of new files) for the process to *Expr*, and return the previous value. If not given *Expr*, return the current value without changing it. Raises an exception if C function *umask(2)* is unavailable. The number *umask* represents disabled permission bits, a call to *sysopen()* with permissions *0777* and a *umask* of *0022* results in files being created with permissions *0755*. Suggested use: Use *0666* when creating files, and *0777* when creating directories, and when protected files are needed, set *umask* to *022*, *027*, or *077*.

unlink(*List*)

unlink()

Deletes a list of files, *List*, or *$\_* if it not given. On success, returns number of files deleted, on failure, return false and set *$!*. Function will not deleted directories unless running as superuser, and *-U* flag has been supplied to perl. Use *rmdir()* to delete directories.

utime(*List*)

Changes the access and modification times on each of a list of files. The first two elements of *List* must be the new numeric access and modification times as epochs, or *undef* to use the current time. Returns the number of files successfully changed. The inode change time of each file is set to the current time. If the files are on a network filesystem, the time of the server is used instead of the time of the local machine. On systems that support C function *futimes(3)*, filehandles may be passed instead of filenames (if they are not barewords, which will be interpreted as filenames), otherwise doing so will cause an exception.

## open-modes

< r read

> w write

>> a append

+>> a+ read and append

+< r+ read and write

+> w+ read and write, file is erased

|- output is piped to command

-| input is received through pipe from command

encoding can be specified as well (after open mode). For example, to open a unicode file for reading:

<:encoding(UTF-8)

To open a binary file for reading:

<:raw :bytes

## stat-fields

0 dev device number of filesystem

1 ino inode number

2 mode file mode (type and permissions)

3 nlink number of (hard) links to the file

4 uid numeric user id of file's owner

5 gid numeric group id of file's owner

6 rdev the device identifier (special files only)

7 size total size of file, in bytes

8 atime last access time as epoch time

9 mtime last modify time as epoch time

10 ctime inode change time as epoch time

11 blksize preferred I/O size in bytes for interacting with file

12 blocks actual number of system-specific blocks on disk (often 512 bytes each)

## encodings

## 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

## printf formatting

<...>

## redirect and restore stdout/stderr

# Make copies of stdout/stderr so we can restore them later

open my $STDOUT\_OLD, '>&', STDOUT or die "Can't save STDOUT\_OLD: $!\n";

open my $STDERR\_OLD, '>&', STDERR or die "Can't save STDERR\_OLD: $!\n";

# Redirect stdout/stderr to files path\_stdout/path\_stderr

open STDOUT, '>', $path\_stdout or die "Can't write to path\_stdout=($path\_stdout): $!\n";

open STDERR, '>', $path\_stderr or die "Can't write to path\_stderr=($path\_stderr): $!\n";

# Restore stdout/stderr

open STDOUT, '>&', $STDOUT\_OLD or die "Can't restore STDOUT\_OLD: $!\n";

open STDERR, '>&', $STDERR\_OLD or die "Can't restore STDERR\_OLD: $!\n";

## piped-open

open DATE, 'date|' or die "cannot pipe from date: $!";

open my $date\_fh, '-|', 'date' or die "cannot pipe from date: $!\n";

Launch command with standard output connected to *DATE* filehandle opened for reading.

open MAIL, '|mail merlyn' or die "cannot pipe to mail: $!";

open my $mail\_fh, '|-', 'mail merlyn' or die "cannot pipe to mail: $!\n";

Launch command with standard input connected to *MAIL* filehandle opened for writing.

## read file example

Read file line by line:

open my $fh, '<', $path\_file or die "Couldn't open file: $!\n";

while (<$fh>) {

print $\_;

}

close $fh or die "Couldn't close file: $!\n";

Read lines to array:

open my $fh, '<', $path\_file or die "Couldn't open file: $!\n";

chomp(my @file\_lines = <$fh>);

close $fh or die "Couldn't close file: $!\n";

## write file example

Note that when printing to a filehandle, said filehandle must be separated from output value(s) by space, not comma.

open my $fh, '>', $path\_file or die "Couldn't open file: $!\n";

print $fh $output\_str;

close $fh or die "Couldn't close file: $!\n";

## read dir example

opendir DIRH, $dir\_path or die "Can't open dir\_path=($dir\_path): $!\n";

foreach (readdir DIRH) {

print $\_;

}

closedir DIRH or die "Can't close dir\_path=($dir\_path): $!\n";

# error handling

handling exceptions

# strings

## string functions

chomp(*Variable*)

chomp(*List*)

chomp()

Safer version of *chop()*, removes any trailing string that corresponds to current value of *$/* (IRS). Returns the number of characters removed from all its arguments. If *List* or *Variable* is not given, *$\_* is used. With default value of *$/* (newline), only newlines are removed. With *$/ = ''* (paragraph mode), all trailing newlines are removed. With *$/ = undef*, function has no effect.

length(*Expr*)

length()

Returns the length in characters of *Expr*, or *$\_* is not given. Returns *undef* if *Expr* is *undef*. To get the length of *Expr* in bytes (when encodes as *UTF-8*), use:

length(Encode::encode("UTF-8", Expr))

substr(*Expr, Offset, Length, Replacement*)

substr(*Expr, Offset, Length*)

substr(*Expr, Offset*)

Extract a substring out of *Expr*, and return (and optionally replace) it. *Offset* is the starting index of this substring (characters in string are zero-indexed), (a negative value specifies number of positions from end). If *Length* is not given, the substring continues to the end of the string, otherwise it is *Length* characters long, or if negative, that many characters are left off the end of the substring. If *Offset* and *Length* specify a substring that is partly outside the search string, the portion that is inside the search string is returned. If they specify a region that is entirely outside the search string, *undef* is returned with a warning, or if the function is used as an *lvalue*, an exception. Function may be used as *lvalue* if *Replacement* is not given, in which case, assignment will be made as if RHS had been passed as *Replacement*, that is the following are equivalent:

substr(*Expr, Offset, Length, Replacement*)

substr(*Expr, Offset, Length*) = Replacement

The value returned by the three-argument form of *substr()* remembers which part of the string it is assigned to each time it is modified, <when it is used in an alias context, i.e: foreach loop> (remembering from the end with a negative *Offset*). See the example below:

$x = '1234567890';

for (substr($x, 1, 2)) {

$\_= 'a'; say "x=($x)"; # 1a4567890

$\_ = 'abc'; say "x=($x)"; # 1abc4567890

$\_ = 'abcdef'; say "x=($x)"; # 1abcdef4567890

}

uc(*Expr*)

uc()

Returns an uppercased version of *Expr*, or *$\_* if not given. This function implements *'\U'* escape in double quoted strings. Behaviour is altered by pragmas *'bytes'*, *'locale'*, and *'unicode\_strings'*, as well as UTF8 flag of *Expr*.

ucfirst(*Expr*)

ucfirst()

Returns *Expr*, or *$\_* if not given, with first character in uppercase. This function implements *'\u'* escape in double quoted strings. Behaviour is altered by pragmas *'bytes'*, *'locale'*, and *'unicode\_strings'*, as well as UTF8 flag of *Expr*.

lc(*Expr*)

lc()

Return a lowercase version of *Expr*, or *$\_* if not given. This function implements *'\L'* escape in double quoted strings. Behaviour is altered by pragmas *'bytes'*, *'locale'*, and *'unicode\_strings'*, as well as UTF8 flag of *Expr*.

lcfirst(*Expr*)

lcfirst()

Returns *Expr*, or *$\_* if not given with first character in lowercase. This function implements *'\l'* escape in double quoted strings. Behaviour is altered by pragmas *'bytes'*, *'locale'*, and *'unicode\_strings'*, as well as UTF8 flag of *Expr*.

chop(*Variable*)

chop(*List*)

chop()

Chops off the last character of a string *Variable*, the elements of *List*, the values (not the keys) of a hash, or *$\_* if not given,and returns the character chopped.

index(*Str, Substr, Position*)

index(*Str, Substr*)

Searches for one string, within another, without wildcard-like behaviour of regex patten match. Returns the position of the first occurrence of *Substr* in *Str*, at or after index *Position*, or *-1* if not found. If *Position* is before the start, or after the end of *Str*, it is treated as the start/end.

rindex(*Str, Substr, Position*)

rindex(*Str, Substr*)

Like *index()*, but return the position of the last occurrence of *Substr* in *Str*, at or after index *Position*, or *-1* if not found.

sprint(*Format, List*)

Return a string formatted by the *printf* conventions of the C function *sprint()*. Unlike *printf()*, if given an array for *Format*, this array is given in a scalar context, which is almost never useful as a format string.

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. If *Pattern* is a single space in single or double quotes (but not */ /*), splitting is done on whitespace (as if */\s+/* was used). 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.

reverse(*List*)

In a list context, returns a list consisting of the elements of *List* in reverse order, or *$\_* if not given. In scalar context, elements of *List* are concatenated into a string, which is then reversed and returned. Can be used to invert keys and values of a hash, though if there are any duplicate values only one can be kept as a key.

fc(*Expr*)

fc()

Returns the casefolded version of *Expr*, or *$\_* if not given. This function implements *'\F'* escape in double quoted strings. Casefolding is a more aggressive form of to-lowercase *lc()*. Used to compare equality of strings regardless of case.

## escape sequences

Those denoted *'Not in []'* are not usable within bracketed character classes.

\000 Octal escape sequence. See also \o{}.

\1 Absolute backreference. Not in [].

\a Alarm or bell.

\A Beginning of string. Not in [].

\b{}, \b Boundary. (\b is a backspace in []).

\B{}, \B Not a boundary. Not in [].

\cX Control-X.

\d Match any digit character.

\D Match any character that isn't a digit.

\e Escape character.

\E Turn off \Q, \L and \U processing. Not in [].

\f Form feed.

\F Foldcase till \E. Not in [].

\g{}, \g1 Named, absolute or relative backreference. Not in [].

\G Pos assertion. Not in [].

\h Match any horizontal whitespace character.

\H Match any character that isn't horizontal whitespace.

\k{}, \k<>, \k'' Named backreference. Not in [].

\K Keep the stuff left of \K. Not in [].

\l Lowercase next character. Not in [].

\L Lowercase till \E. Not in [].

\n (Logical) newline character.

\N Match any character but newline. Not in [].

\N{} Named or numbered (Unicode) character or sequence.

\o{} Octal escape sequence.

\p{}, \pP Match any character with the given Unicode property.

\P{}, \PP Match any character without the given property.

\Q Quote (disable) pattern metacharacters till \E. Not in [].

\r Return character.

\R Generic new line. Not in [].

\s Match any whitespace character.

\S Match any character that isn't a whitespace.

\t Tab character.

\u Titlecase next character. Not in [].

\U Uppercase till \E. Not in [].

\v Match any vertical whitespace character.

\V Match any character that isn't vertical whitespace

\w Match any word character.

\W Match any character that isn't a word character.

\x{}, \x00 Hexadecimal escape sequence.

\X Unicode "extended grapheme cluster". Not in [].

\z End of string. Not in [].

\Z End of string. Not in [].

# arrays

## syntax

A perl array returns its length when used in a scalar context

Check if 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.

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]}";

### Key exists:

exists($var\_hash{$var\_key})

## 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/// *or* y///

To declare regex as variable, quote with:

qr//

## 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 boundary

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 continue 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 boundary

o evaluate expression only once

s allows '.' to match newline

x Allows whitespace in the expression for clarity

g Replace all occurrences of the found expression

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

r Leave original 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 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.

## regex functions

pos(*Scalar*)

pos()

Returns the offset of where the last *m//g* search left off for *Scalar*, or *$\_* if not given. Offset is in characters, unless *'use bytes'* (which is not recommended) is in effect, in which case it is in bytes. If match has not been run on *Scalar*, or has run but failed, *undef* is returned. Assigning a value to *pos()* will change the offset used by the regex engine. Using *'\G'* in a regex indicates that matches should only be made after position given by *pos()*.

## regex symbols

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

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

\* 0 or more occurrences

+ 1 or more occurrences

? 0 or 1 occurrences

{n} exactly n occurrences

{n,} n or more occurrences

{,m} at most m occurrences

{n,m} n to m occurrences

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

\Q Begin metaquoting, escape all characters not matching *[A-Za-z0-9]*

\E End metaquoting

Non-greedy quantifiers

?? 0 or 1

\*? 0 or more

+? 1 or more

{m,n}? specific number

## matching literal string

Escape all regex symbols (characters not matching *[A-Za-z0-9]*) in string with *quotemeta()*, or, use *\Q$string\_var\E* modifiers.

## substitute and assign

(my $newstr = $oldstr) =~ s/foo/bar/

my $newstr = $oldstr =~ s/foo/bar/r

# maths

## math functions

exp(*Expr*)

exp()

Return *e* to the power of *Expr*, or of *$\_* if not given.

sqrt(*Expr*)

sqrt()

Return the square root of *Expr*, or of *$\_* if not given. Only works for negative values if *'Math::Complex'* is loaded.

log(*Expr*)

log()

Returns the natural logarithm of *Expr*, or of *$\_* if not given.

int(*Expr*)

int()

Returns the integer portion of *Expr*, or of *$\_* if not given. Truncates value instead of rounding.

sin(*Expr*)

sin()

Returns the sine of *Expr* (in radians), or of *$\_* if not given. Inverse sine is given by *'Math::Trig::asin'*.

cos(*Expr*)

cos()

Returns the cosine of *Expr* (in radians), or of *$\_* if not given. Inverse cosine is given by *'Math::Trig::acos'*.

atan2(*Y, X*)

Returns the arctangent of *Y/X* in the range *[-pi, pi]*. Return value for *atan(0, 0)* is OS specific. Regular tan is given by *'Math::Trig::tan'*.

abs(*Expr*)

abs()

Return the absolute value of *Expr*, or of *$\_* if not given.

# file tests

-e Existence

-r Readable by this (effective) user or group

-w Writeable by this (effective) user or group

-x Executable by this (effective) user or group

-o Ownership by this (effective) user or group

-R Readable by this real user or group

-W Writeable by this real user or group

-X Executable by this real user or group

-O Ownership by this real user or group

-z Exists and has zero size (false for directories)

-s Exists and has non-zero size (returns size in bytes)

-f Plain file

-d Directory

-l Symbolic link

-S Socket

-p Named pipe

-b Block file (i.e: Disk)

-c Character-special file (i.e: I/O device)

-u setuid

-g setgid

-k Sticky bit set

-t TTY (as reported by *isatty()*)

-T Looks like a text file

-B Looks like a binary file

-M Modification age (in days)

-A Access age (in days)

-C Inode modification (in days)

# globbing

To match all items (dotfiles are given by *".\*"*, non-dotfiles by *"\*"*) in the current directory:

glob ".\* \*";

Use either *glob()* function, or *<>* operator, that is, the following are equivalent:

my @all\_files = <\*>;

my @all\_files = glob "\*";

# directory handles

Iterate over all items in a given directory:

opendir my $dir\_h, $selected\_dir or die "Cannot open selected\_dir=($selected\_dir)";

foreach (readdir $dir\_h) {

say "$\_";

}

closedir $dir\_h;

<Dirhandles are exempt from the 'no commas after filehandles' rule?>

# Cwd

Import with:

use Cwd;

Functions:

getcwd()

Returns the current working directory. On error, returns *undef* and set *$!*.

cwd()

fastcwd()

A faster, less safe version of *getcwd()*

fastgetcwd()

Synonym for *cwd()*

getdcwd()

(Windows) Get current working directory for a specific drive

abs\_path(*file*)

abs\_path()

Must be imported explicitly. Get absolute pathname for a given *file*, or if not given, the current working directory. On error, returns *undef* and set *$!*.

realpath()

Must be imported explicitly. Synonym for *abs\_path()*.

fast\_abs\_path()

Must be imported explicitly. A faster, less safe version of *abs\_path()*.

# [File::Spec](https://perldoc.perl.org/File::Spec)

Comprised of OS specific modules, the appropriate of which is loaded on import of *File::Spec*:

File::Spec::Unix

File::Spec::Mac

File::Spec::OS2

File::Spec::Win32

File::Spec::VMS

Module is object-oriented, subroutines should therefore be called as class methods:

File::Spec->catfile(*'a', 'b'*)

Methods:

canonpath(*$path*)

Logical clean-up of *path*

catdir(*@directories*)

Concatenate two or more directory names to form a complete path ending with a directory (with trailing slash removed).

catfile(*@directories, $filename*)

Concatenate one or more directory names and a filename to form a complete path ending in the filename.

curdir()

Return a string representation of the current directory.

devnull()

Return a string representation of the null device.

rootdir()

Return a string representation of the root directory.

tmpdir()

Return a string representation of the first writable directory from a list of possible temporary directories, or the current directory if no such result is found.

updir()

Return a string representation of the parent directory

no\_upwards(*@directories*)

Given a list of files in a directory, strip out *'.'* and *'..'* (note paths such as *'../../file'* are not removed).

case\_tolerant()

Return true/false indicating whether alphabetic case is significant when comparing file specifications.

file\_name\_is\_absolute(*$path*)

Return true if *path* is an absolute path.

path()

Return *PATH* as a list.

join(*@directories, $filename*)

Same as *catfile()*.

*($volume, $directories, $file) =* splitpath(*$path[, $no\_file]*)

Split a path into volume, directory, and filename portions. On Unix, returns *''* for volume.

splitdir(*$directories*)

Opposite of *catdir()*. <>

catpath(*$volume, $directory, $file*)

Take *volume*, *directory*, and *file*, and returns an entire path. On Unix, *volume* is ignored.

abs2rel(*$path[, $base]*)

Take a destination path and an optional base path, and return a relative path from the base path to the destination path.

rel2abs(*$path[, $base]*)

Convert a relative path to an absolute path. If *$base* is not given, *Cwd::cwd()* is used.

# [Storable](https://perldoc.perl.org/Storable)

Note that reading storable documents constitutes a security risk.

store(*data, file*)

Writes variable *data* to *file*, returns *undef* upon error.

retrieve(*file*)

Recreates object previously stored in *file*, and return a reference to this object. Returns *undef* upon error.

<>

# process management

alarm()

exec()

Execute a system command and never returns (use *system()* to execute command and return). Returns false if command does not exist and it is being executed directly instead of by system shell.

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(*List*)

system(*Program, List*)

Executes a command, and returns exit status of command. Command runs in environment inherited from Perl, the hash *%ENV*, which can be modified.

If there is more than one argument in *List*, or *List* is an array with more than one value, start the program given by the first element, with the arguments given by the remaining elements (which is a somewhat more secure approach). If there is only one argument, it is given to the system shell (*/bin/sh -c* by default) for parsing. Standard output of command is not captured (use backticks or *qx//* to do so). *SIGINT* and *SIGQUIT* are ignored during execution of *system()*.

<Has same special behaviour re: *','* between arguments as *print()*>

Return value is two octets, the 'high' octet has the exit value of the program (*>> 8* to access), the 'low' octet <>.

times()

wait()

waitpid()

# sockets

accept()

bind()

connect()

getpeername()

getsockname()

getsockopt()

listen()

recv()

send()

setsockopt()

shutdown()

socket()

socketpair()

# [POSIX](https://perldoc.perl.org/POSIX)

Importing everything from *'POSIX'* is strongly discouraged. Use *'use POSIX ()'* and use fully qualified names, i.e: *'POSIX::SEEK\_END'*, or give explicit import list.

## functions

strftime()

# signals

Signal processing example:

# Signal handlers

sub my\_hup\_handler { state $n = 0; say "Caught HUP: $n"; ++$n; }

sub my\_usr1\_handler { state $n = 0; say "Caught USR1: $n"; ++$n; }

sub my\_usr2\_handler { state $n = 0; say "Caught USR2: $n"; ++$n; }

sub my\_int\_handler { say "Caught INT., Exiting"; exit; }

say "Process id \$\$=($$)";

# Register signal handlers

foreach my $signal ( qw(int hup usr1 usr2) ) {

$SIG{ uc $signal } = "my\_${signal}\_handler";

}

# Wait for the arival of signal(s)

while (1) {

sleep 1;

}

Send signal to process with process id *pid* with:

kill -*signal* *pid*

or

perl -E 'kill *signal* => *pid*'

Unix signals: view with *'kill -l'*

<…>

# user and group info

endgrent()

endhostent()

endnetent()

endpwent()

getgrent()

getgrgid()

getgrnam()

getlogin()

getpwent()

getpwnam()

getpwuid()

setgrent()

setpwent()

# network info

endprotoent()

endservent()

gethostbyaddr()

gethostbyname()

gethostent()

getnetbyaddr()

getnetbyname()

getnetent()

getprotobyname()

getprotobynumber()

getprotoent()

getservbyname()

getservbyport()

getservent()

sethostent()

setnetent()

setprotoent()

setservent()

# modules

## installing module

cpan *<module>*

Get hash of modules (and their version) included with a given version of perl:

my $check\_version = $];

my %modules\_included = %{ $Module::CoreList::version{$check\_version} };

List module search path directories

perl -E '$"="\n"; say "@INC"'

Add directory to module search path (at compile time)

use lib '<dir>'

# packages

@ISA

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

# [DateTime](https://metacpan.org/pod/DateTime)

Epoch to datetime:

my $dt = DateTime->from\_epoch(epoch => $epoch\_value, time\_zone => 'local');

Datetime to epoch:

my $epoch = $dt->epoch;

Current datetime (local timezone):

my $dt\_now = DateTime->today(time\_zone=>'local');

Add offset to datetime

my $dt\_new = $dt->clone->add( DateTime::Duration->new( days => 1 ) );

Note: Use *clone* to return result and leave original datetime unmodified.

## constructor methods

DateTime->new(*parms*)

Does not parse datetimes. Parameters:

year

month

day

hour

minute

second

nanosecond

locale

time\_zone

formatter

DateTime->from\_epoch(*epoch => $epoch, …*)

DateTime->now(*…*)

DateTime->today(*…*)

DateTime->last\_day\_of\_month(*…*)

DateTime->from\_day\_of\_year(*…*)

DateTime->from\_day\_of\_year(*…*)

DateTime->from\_object(*object => $object, …*)

$dt->clone

## get methods

$dt->year

$dt->ce\_year

$dt->era\_name

$dt->era\_abbr

$dt->christian\_era

$dt->secular\_ear

$dt->year\_with\_era

$dt->year\_with\_christian\_era

$dt->year\_with\_secular\_ear

$dt->month

$dt->month\_name

$dt->month\_abbr

$dt->day

$dt->day\_of\_week

$dt->local\_day\_of\_week

$dt->day\_name

$dt->day\_abbr

$dt->day\_of\_year

$dt->quarter

$dt->quarter\_name

$dt->quarter\_abbr

$dt->day\_of\_quarter

$dt->weekday\_of\_month

$dt->ymd(*…*), $dt->mdy(*…*), $dt->dmy(*…*)

$dt->hour

$dt->hour\_1

$dt->hour\_12

$dt->hour\_12\_0

$dt->am\_or\_pm

$dt->minute

$dt->second

$dt->fractional\_second

$dt->millisecond

$dt->microsecond

$dt->nanosecond

$dt->hms(*…*)

$dt->datetime(*…*)

$dt->rcf3339

$dt->stringify

$dt->is\_leap\_year

$dt->is\_last\_day\_of\_month

$dt->is\_last\_day\_of\_quarter

$dt->is\_last\_day\_of\_year

$dt->month\_length

$dt->quarter\_length

$dt->year\_length

$dt->week

$dt->week\_year

$dt->week\_number

$dt->week\_of\_month

$dt->jd, $dt->mjd

$dt->time\_zone

$dt->offset

$dt->is\_dst

$dt->time\_zone\_long\_name

$dt->time\_zone\_short\_name

$dt->strftime(*$format, …*)

$dt->format\_cldr(*$format, …*)

$dt->epoch

$dt->hires\_epoch

$dt->is\_finite, $dt->is\_infinite

$dt->utc\_rd\_values

$dt->local\_rd\_values

$dt->leap\_seconds

$dt->utc\_rd\_as\_seconds

$dt->locale

$dt->formatter

## set methods

DateTime set methods (unless otherwise specified) return the object itself (making method chaining possible).

$dt->set(*…*)

Accepts same parameters as *new*, except *local* and *time\_zone* (use *set\_locale()* and *set\_time\_zone()* instead).

$dt->set\_year

$dt->set\_month

$dt->set\_day

$dt->set\_hour

$dt->set\_minute

$dt->set\_second

$dt->set\_nanosecond

$dt->truncate(*to => …*)

$dt->set\_locale(*$locale*)

$dt->set\_time\_zone(*$tz*)

$dt->set\_formatter(*$formatter*)

## math methods

DateTime maths methods return the object itself (making method chaining possible).

$dt->duration\_class

$dt->add\_duration(*$duration\_object*)

$dt->add(*$duration\_object*)

$dt->add(*…*)

Creates a new *DateTime::Duration* object using given parameters, and call *$dt->add\_duration()* with that object.

$dt->subtract\_duration(*$duration\_object*)

$dt->subtract(*DateTime::Duration->new-parameters*)

$dt->subtract(*$duration\_object*)

$dt->subtract\_datetime(*$datetime*)

$dt->delta\_md(*$datetime*)

$dt->delta\_days(*$datetime*)

$dt->delta\_ms(*$datetime*)

$dt->subtract\_datetime\_absolute(*$datetime*)

$dt->is\_between(*$lower, $upper*)

## class methods

DateTime->DefaultLocale(*$locale*)

DateTime->compare(*$dt1, $dt2*)

DateTime->compare\_ignore\_floating(*$dt1, $dt2*)

Recommendations:

Use floating timezone where <>

Do not use timezones for far-future (*year >= 5000*) dates.

# [DateTime::Format::Strptime](https://metacpan.org/pod/DateTime::Format::Strptime)

## methods

DateTime::Format::Strptime->new(*%args*)

$strptime->parse\_datetime(*$string*)

$strptime->format\_datetime(*$datetime*)

$strptime->locale

$strptime->pattern

$strptime->time\_zone

$strptime->errmsg

## exports

strptime($strptime\_pattern, $string)

strftime($strftime\_pattern, $datetime)

# [DateTime::Duration](https://metacpan.org/pod/DateTime::Duration)

## methods

DateTime::Duration->new(*…*)

$dur->clone

$dur->in\_units(*…*)

$dur->is\_positive, $dur->is\_zero, $dur->is\_negative

$dur->is\_wrap\_mode, $dur->is\_limit\_mode, $dur->is\_preserve\_mode

$dur->end\_of\_month\_mode

$dur->calendar\_duration

$dur->clock\_duration

$dur->inverse(*…*)

$dur->add\_duration(*$duration\_object*), $dur->subtract\_duration(*$duration\_object*)

$dur->add(*…*), $dur->subtract(*…*)

$dur->multiply(*$number*)

DateTime::Duration->compare(*$duration1, $duration2, $base\_datetime*)

$dur->delta\_months, $dur->delta\_days, $dur->delta\_minutes, $dur->delta\_seconds, $dur->delta\_nanoseconds

$dur->deltas

$dur->years, $dur->months, $dur->weeks, $dur->days, $dur->hours, $dur->minutes, $dur->seconds, $dur->nanoseconds

# Time::Local

# List::Util

# File::HomeDir

Get users home directory:

my $user\_dir = File::HomeDir->my\_home;

# File::Basename

dirname()

basename()

*($name, $path, $suffix) =* fileparse(*$fullname, @suffixlist*)

Divides a file path into its *$dirs*, *$filename* and (optionally) the filename *$suffix*. The concatenation of these three items will denote the same location as *$fullname*.

# [File::Find](https://perldoc.perl.org/File::Find)

find(*\&wanted, @directories*)

find(*\%options, @directories*)

Does a depth-first search over the given *directories* in the order they are given. For each file or directory found, it calls *wanted*. Additionally, for each directory found, it will *chdir()* into that directory and continue the search recursively and invoking *wanted* on each item found.

finddepth(*\&wanted, @directories*)

finddepth(*\%options, @directories*)

Like *find()*, except *wanted* is invoked for a directory after it is invoked for the directory's contents, that is it does a postorder traversal instead of a preorder traversal, working from the bottom of the directory tree up (whereas *find()* works from the top of the tree down).

*%options*

A hash reference may be given instead of a function reference as first argument to *find()* and *finddepth()*, describing the operations to be performed for each file. The possible keys for the hash are:

*wanted*

Function reference (mandatory)

*bydepth*

Reports the name of aa directory only after all its entries have been reported

preprocess

Function reference, used to preprocess the current directory. Called with a list of strings (file/dir names) and expected to return a list of strings <file/dir names?>

postprocess

Function reference, invoked before leaving current directory.

follow

Causes symlinks to be followed

follow\_fast

Similar to *follow*, but may report some files more than once.

follow\_skip

If *1* (default), all files which are neither directories or symlinks are not processed a second time. If *0*, *die* is called if any file is about to be processed a second time. If *2*, ignore duplicate files and directories.

dangling\_symlinks

If false, silently ignore dangling symlinks. If true and a function reference, called with dangling symlink and containing directory as arguments, if true and not a function reference, warnings are issued for dangling symlinks.

no\_chdir

Does not *chdir()* as it recurses, in which case *$\_* will be the same as *$File::Find::name*.

untaint

If used in taint mode, internal directory names have to be untainted before the can be passed to *chdir()*.

untaint\_pattern

Pattern used when *untaint* is set. Default is *qr|^([-+@\w./]+)$|*

untaint\_skip

If set, a directory which fails *untaint\_pattern* is skipped, in which case default behaviour is to *die*.

*&wanted*

Function called for each file and directory. Takes no arguments, and any return value is ignored. Within the scope of the function, there are the following variables:

$File::Find::dir

Current directory name

$\_

Current filename

$File::Find::name

Complete path to file

$File::Find::prune

Set to *1* to prevent recursion into current directory (if not using *finddir()*).

# Data

Data::Dumper()

Able to print nested reference list/hash structures.

Data::Dump::dump()

Data::Printer()

# IO

## IO:Handle

## IO::File

## IO::Tee

## IO::Scalar

## IO::Pipe

## IO::Null

## IO::Interactive

# JSON

to\_json(*$perl\_scalar[, $optional\_hashref]*)

Convert the given perl datastructure to Unicode string and return it. Hashes, lists, <other datastructures> should be passed as references. Croaks on error.

from\_json(*$json\_text[, $optional\_hashref]*)

Opposite of *to\_json()*, parse given *json\_text*, and return a reference to the resulting <hash?>. Croaks on error.

encode\_json(*$perl\_scalar*)

Convert the given perl data structure to UTF-8 encoded binary string and return it. Hashes, lists, <other datastructures> should be passed as references. Croaks on error.

decode\_json(*$json\_text*)

Opposite of *to\_json()*, parse given *json\_text*, and return a reference to the resulting <hash?>. Croaks on error.

JSON::is\_bool()

<>

## Options:

<>

## Conversion rules:

JSON to Perl

JSON object becomes a reference to a perl hash, order of object keys is not preserved.

JSON array becomes a reference to a perl array.

JSON strings and numbers are converted <as expected>. Json *null* becomes perl *undef*.

<>

Perl to JSON

Hash reference becomes a JSON object, order of hash keys is not preserved.

Array reference becomes JSON array.

<Scalar values>

Other references are generally not allowed.

<>

## serialization/deserialization

<>

# 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