

## Perl

### The Practical Extraction Report Language

Created by Larry Wall

Basically a scripting language

Gnu Software - Runs on almost all platforms

Contains extensive string manipulation and pattern matching primitives

Has automatic memory management

### The language grew from less-structured roots

Some idiosyncrasies retained for backward compatability

Some "features" designed to simplify easy tasks although they are less useful for more complicated tasks

### Advantages

Can run arbitrary programs on computer

Good for operating on text files

Portable across many platforms

### Disadvantages

Programs can be hard to understand or maintain

Some program "features" may give suprising results

## Perl Comments

Larry Wall was a sys admin and spent his time solving problems NOW

### Perl is one of his arsenal of tools

"But Wall and Perl are all about wiggle room, about messy imperfection and fuzzy creativity. After all, duct tape is valuable not because it offers a perfect solution to your plumbing problems, but because it gets the job done. Perl, to some eyes, may not seem elegant. But that's not Wall's concern. His humble goal is to be useful, to help people do what they need to do..." – The Joy of Perl by Andrew Leonard

### Eric's Raymond's Criticism

"The design of the language is showing its age and significant signs of bloat," says Raymond. "Perl was never a pretty or elegant language; its lure lay in its brute usefulness. Over time, the problems consequent from Larry's early choices seem to me to be accumulating and reinforcing one another just a little faster than the benefits pile up."

– The Joy of Perl by Andrew Leonard

### Larry Wall's Response

"'It is amazing,' wrote Tolstoy, 'how complete is the delusion that beauty is goodness.'"

## The Hello World Program

Here is a perl program that prints "Hello world"

As a command

```
swm> perl -e 'print "Hello World\n";'
```

-e switch executes command line argument

As a program

```
#!/usr/local/bin/perl
print "Hello World\n";
```

Stored in a text file with the execute file permission

The line `#!/usr/local/bin/perl` tells the command shell what program to run

# are generally shell comments but the convention is that the `#!` as the first line of a text file indicates the program to be run with the rest of the file as standard input

The path `/usr/local/bin/perl` is where the perl program is installed on your machine

## Switches with the perl command

see "man perlrun" for a full list

(these switches are not part of the perl program)

-a turn on autosplit mode for splitting on whitespace

-c Check syntax without execution

-d turn on perl debugger

-D turn on debugging flags

-D14 watch execution of script

-e <command> put program in command line

-F <pattern> specify pattern for splitting

-h print command line options

-iextension modify file inplace with <>

-mmodule use module before executing script

-n loop over input printing specified lines

-v print version information

-w warn on questionable use

## Some Simple Examples

```
print every line containing "RIT" in file "data"
#!/usr/local/bin/perl -w
```

```
use strict; # check for bad code
```

```
while(<>) { # for every line
    print if /RIT/; # print it if it contains
    "RIT"
}
```

Always use the -w flag and the use strict; statement

### Sort a file

```
#!/usr/local/bin/perl -w
```

```
use strict; # check for bad code
my @lines = <>;
@lines = sort(@lines);
print @lines;
```

### What does this do?

```
#!/usr/local/bin/perl -w
$file = `2cities.txt`;
open( INFO, $file );
$lineNumber = 1;
while ( $line = <INFO> ) {
    if ( ( $line =~ s/(.)\1/ \($1$1)/ ) ) {
        print "$lineNumber $line";
        $lineNumber++;
    } else {
        print "$line";
    }
}
close (INFO);
```

## An example of a real script

```
Mail Grades
#!/usr/local/bin/perl -w
use strict;
use IO::File;
my $mail = \*STDOUT;
my $db = new IO::File "Sec3Grades.txt"
    or die "can't open grades.txt";
while (<$db>) {
    chomp;
    s{\r$}{};
    next if /^#/;
    my
($login,$name,$x1,$q1,$q2,$p1,$p2,$lab,$final,$bonus,$grade,$letter) = split /\t/;
    my $out = new IO::File "|usr/lib/sendmail -t"
        or die;
    select($out);
    print "To: $login\n";
    print "Cc: swm\n";
    # print "To: swm\n";
    print "Subject: CS2: Grade of Final Exam\n";
    print "\n";
    print "$name,\n\n";
    print "Your quiz grades were $q1 and $q2
percent\n";
    print "Your project grades were $p1 and $p2\n";
    print "Your lab grade average was $lab\n";
    print "You were awarded a bonus point\n" if
$bonus;
    print "Your final exam grade was $final\n";
    print "Final grade was $grade or letter
$letter\n";
    print "\n\n";
    print "\n--Sidney Marshall\n";
    print "(Message was automatically
generated.)\n";
    $out->close;
}
```

## Perl Language Principles

All variables (almost) must indicate their type with a character in front of them (\$, @, %, &, \*)

The special character indicated the type of the expression NOT the type of the variable

For simple variables the special character indicates the type of the variable

There are several kinds of variables in perl

\$ for scalars (strings, numbers, and references)

\$x \$y \$z \$myVar

@ for arrays (arrays always contain scalars)

@names @pages

% for hashes (perl's hash tables)

%weekdays %linkfiles

& for subroutine names

&getPage

\* for typeglobs (symbol table entries - more later)

Note, e.g., that an array @a has elements \$a[0] because the type of an array element is a scalar

All of these namespaces are distinct - there is no relation between @a and \$a - these are two totally different variables

variables are passed by reference so a subroutine can modify the value of a variable that is passed to the subroutine (unlike C++ and java)

comments start with # and end with the end of the line

## Syntax of Perl

A perl program is a sequence of statements

Semicolons separate statements - not required after the last statement in a file or block

Simple statements can be followed by modifier

```
if EXPR
unless EXPR
```

```
while EXPR
until EXPR
foreach EXPR
```

In the last three loop constructs, for each value in EXPR, it aliases the variable \$\_ to the value and executes the statement.

```
do { ... } while/until EXPR
```

Declarations

```
my $x;      # lexically scoped local variable
our $x;     # global variable (the default)
local $x;   # dynamically scoped local variable
sub mysub;  # forward declaration
```

## Compound Statements

Compound statements (must use {} for the BLOCKs)

```
LABEL: BLOCK
LABEL: BLOCK continue BLOCK
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
LABEL: while (EXPR) BLOCK
LABEL: while (EXPR) BLOCK continue BLOCK
LABEL: for (EXPR; EXPR; EXPR) BLOCK
LABEL: foreach VAR (LIST) BLOCK
LABEL: foreach VAR (LIST) BLOCK continue BLOCK
```

LABEL is optional

VAR is optional - defaults to \$\_

For loops

```
for( S1; S2; S3 ) {}
```

means

```
S1; while(S2) {} continue {S3}
```

example

```
$on_a_tty = -t STDIN && -t STDOUT;
sub prompt { print "yes? " if $on_a_tty }
for ( prompt(); <STDIN>; prompt() ) {
    # do something
}
```

Foreach loops

```
foreach my $var (@elements) { $var += 1 }
```

```
for $count (10,9,8,7,6,5,4,3,2,1,'BOOM') { }
```

```
for (1..15) { print "Merry Christmas\n"; }
```

```
foreach $item (split(/:[\\n:]*/, $ENV{TERMCAP}))
{
    print "Item: $item\n";
}
```

## Basic BLOCKs and Switch Statements

a BLOCK is one or more statements separated by semicolons and enclosed in braces

```
{ S1; S2 }
```

A BLOCK by itself (labeled or not) is semantically equivalent to a loop that executes once so you can use any loop control statements inside it to leave or restart the block.

```
SWITCH: {
    if (/^abc/) { $abc = 1; last SWITCH; }
    if (/^def/) { $def = 1; last SWITCH; }
    if (/^xyz/) { $xyz = 1; last SWITCH; }
    $nothing = 1;
}
```

(Note that there is no switch statement in perl)

Basic BLOCKs can have a continue BLOCK even if they are not loops

## Loop control

`next [LABEL]`

The next command is like the continue statement in C; it starts the next iteration of the loop:

next cannot be used to exit a block which returns a value such as `eval {}`, `sub {}` or `do {}`, and should not be used to exit a `grep()` or `map()` operation.

Note that a block by itself is semantically identical to a loop that executes once. Thus next will exit such a block early.

`last [LABEL]`

The last command is like the break statement in C (as used in loops); it immediately exits the loop in question. If the LABEL is omitted, the command refers to the innermost enclosing loop. The continue block, if any, is not executed:

Note that a block by itself is semantically identical to a loop that executes once. Thus last can be used to effect an early exit out of such a block.

`redo [LABEL]`

The redo command restarts the loop block without evaluating the conditional again. The continue block, if any, is not executed. If the LABEL is omitted, the command refers to the innermost enclosing loop.

redo cannot be used to retry a block which returns a value such as `eval {}`, `sub {}` or `do {}`, and should not be used to exit a `grep()` or `map()` operation.

Note that a block by itself is semantically identical to a loop that executes once. Thus redo inside such a block will effectively turn it into a looping construct.

## goto

`goto LABEL`

`goto EXPR`

`goto &NAME`

The `goto-LABEL` form finds the statement labeled with LABEL and resumes execution there. It may not be used to go into any construct that requires initialization, such as a subroutine or a `foreach` loop. It also can't be used to go into a construct that is optimized away, or to get out of a block or subroutine given to `sort`. It can be used to go almost anywhere else within the dynamic scope, including out of subroutines, but it's usually better to use some other construct such as `last` or `die`.

The author of Perl has never felt the need to use this form of `goto` in Perl.

The `goto-EXPR` form expects a label name, whose scope will be resolved dynamically, but isn't necessarily recommended if you're optimizing for maintainability:

```
goto ("FOO", "BAR", "GLARCH")[$i];
```

The `goto-&NAME` form exits the current subroutine and immediately calls in its place the named subroutine using the current value of `@_`. This is used by AUTOLOAD subroutines that wish to load another subroutine and then pretend that the other subroutine had been called in the first place.

NAME needn't be the name of a subroutine; it can be a scalar variable containing a code reference, or a block which evaluates to a code reference.

## Variables

There are several kinds of variables in perl

`$`scalars

`@`arrays

`%`hashes

`&`subroutine

`*`symbolTableEntry (or typeglobs)

## Scalar Values - \$var

Numbers

represented as doubles

perl will convert strings to numbers if needed

Strings

sequence of characters

there is support for multibyte characters

References

Are like pointers or addresses

All references to a value share that value and changes to that value by one reference are seen by all references

## Lists and Arrays

Expressions can have a list type although no variable has a list type

A list can be assigned to an array and lists can be interpolated into lists

Evaluating a list in a scalar context returns the last element of the list

Evaluating an array in a scalar context returns the number of elements in the array (one more than the last index)

Lists are constructed by ( , , )

In some contexts the parentheses are optional

The value of a list in a scalar context is last value of the list

May assign to a list for destructuring

may have undef to throw away values

last LHS element may be an array or hash to suck up all remaining list elements

The value of list assignment in scalar context is length of RHS

A pattern match in list context returns list of all matching parts of string

Variables always store arrays - never lists

## Array values - @arrays

Highest index is \$#list

Subscripting syntax is: \$list[\$index]

arrays in scalar context returns #elements (= \$#array + 1)

lists in scalar context return last element

subscripts

negative subscripts fetch from end of array

slice of empty list is empty list

slices

[3..5]

[3,5,4,6]

```
foreach (@array[ 4 .. 10 ]) { s/peter/paul/ }
```

Referencing a nonexistent element automatically extends the array

## Hash values - %hashes

### \$hash{\$key}

may use => for , in list denotation - useful for making hashes

exists \$hash{\$key} is true if key in hash

delete \$hash{\$key} removes key from hash

a hash can be made from a list of [key, value] pairs

The syntax (key1=>value1, key2=>value2) creates a list that can be assigned to a hash variable

the => operator is like the , operator except it forces the operand on its left to be interpreted as a string

The syntax {key1=>value1, key2=>value2} creates a reference to a hash

{ } creates a reference to an empty hash

## Subroutine values - &subroutine

### name(\$arg)

Subroutines are generally called and not manipulated as a value but this is possible by using references, e.g., passing a reference to the subroutine value into another subroutine

## \*symbolTableEntry or typeglobs

Represents a symbol table entry

Refers to all of the values referred to by a variable name

SCALAR, ARRAY, HASH, CODE, IO, GLOB

Used to be used for IO handles and other references before references were added to the language

## Context

All expressions in perl are evaluated in one of three contexts

There may be no relationship between the values in the list context and the scalar context

scalar context

```
$var = exp;
```

list context

```
@var = exp;
```

void context

```
exp;
```

the scalar operator can force execution in a scalar context

```
@var = (@exp1, scalar @exp2, @exp3);
```

list values return their last element in a scalar context

array values return their size in a scalar context

hashes return the fraction of buckets used in a scalar context

hashes return a reference to the hash in a list context

## More on references

References are made by placing \ in front of a variable

```
$scalarref = \$foo;
```

```
$arrayref = \@ARGV;
```

```
$hashref = \%ENV;
```

```
$coderef = \&handler;
```

```
$globref = \*foo;
```

or by assignment

```
$arrayRef = [1, 2, 3]; # array reference
```

```
%hashRef = {'1'=>'true', '2'=>'false'}; # hash reference
```

{ \$arrayRef } is the name of an array so

@{ \$arrayRef } or @\$arrayRef is the array

\${ \$arrayRef }[1] is an element of the array

Can also do \$aref->[1] or \$ahash->{key}

-> is the dereferencing operator

ref() function tests for references and returns the empty string if not a reference and one of SCALAR, ARRAY, HASH, CODE, REF, GLOB, LVALUE, if it is

A reference can also be created by using a special syntax, lovingly known as the \*foo{THING} syntax. \*foo{THING} returns a reference to the THING slot in \*foo (which is the symbol table entry which holds everything known as foo).

```
$scalarref = *foo{SCALAR}; $
```

```
$arrayref = *ARGV{ARRAY}; @
```

```
$hashref = *ENV{HASH}; %
```

```
$coderef = *handler{CODE}; &
```

```
$ioref = *STDIN{IO};
```

```
$globref = *foo{GLOB}; *
```

Strings used as references refer to the variable with the string name

## undef

undef() or undef returns the undefined literal

can be used to set the value of a variable to the undefined value

can be used in a list LHS to throw away a value

undef(\$a[\$i]) returns true if \$a[\$i] is undefined

## Interpolation

List values never contain lists

lists interpolate sublists

A list that contains list values effectively flattens them into one long list

hashes interpolate as list of key value pairs

"xxx" strings have \$x and @x expanded

You can use \${x} or @{x} to explicitly indicate where the identifier ends if necessary

'xxx' do not interpolate

## Subroutines

To declare subroutines:

```
sub NAME; # A "forward" declaration.
```

```
sub NAME (PROTO); # ditto, but with prototypes
```

```
sub NAME BLOCK # A declaration and a definition.
```

```
sub NAME (PROTO) BLOCK # same with prototypes
```

To define an anonymous subroutine at runtime:

```
$_subref = sub BLOCK; # no proto
```

```
$_subref = sub (PROTO) BLOCK; # with proto
```

To import subroutines:

```
use MODULE qw(NAME1 NAME2 NAME3);
```

To call subroutines:

```
NAME (LIST); # & is optional with parentheses.
```

```
NAME LIST; # Parens optional if predeclared/imported.
```

```
&NAME (LIST); # Circumvent prototypes.
```

```
&NAME; # Makes current @_ visible to called subroutine.
```

Subroutines accept @\_ and produce a list value

arguments are aliases of parameters and can be modified

return value evaluated in context of caller (scalar, list, void)

can do my(\$x, \$y) = @\_ or use shift

example

```
($v3, $v4) = upcase($v1, $v2); # no change to $v1, $v2
```

```

sub upcase {
    return unless defined wantarray; # void context
    my @parms = @_ ;
    for (@parms) { tr/a-z/A-Z/ }
    return wantarray ? @parms : $parms[0];
}

```

Last expression in subroutine is the return value unless an earlier explicit return statement

## Variable Scoping

my declares lexically scoped local variable

local declares dynamically scoped local variable

Certain variables are always "global" and cannot be declared with "my"

local declarations must be used with \$\_ \$ARGV and others of this ilk

saves old value (somewhere) and restores this value on exit can localize typeglobs too

local declaration limits scope of global defaulted variables

our declares variables as their global value

Does nothing except prevent warnings in strict modes

## Scopes

The scope of a package declaration, my(), and local() operators is from the declaration itself through the end of the enclosing block, eval, or file, whichever comes first

Unqualified dynamic identifiers will be in this namespace

Except that the following default to the main package

```

_ ARGV ARGVOUT ENV INC SIG STDIN STDOUT STDERR

```

## Variable Names

Technical Note on the Syntax of Variable Names

Usually, variables must begin with a letter or underscore, in which case they can be arbitrarily long and may contain letters, digits, underscores, or the special sequence :: or '. In this case, the part before the last :: or ' is taken to be a package qualifier.

Perl variable names may also be a sequence of digits or a single punctuation or control character. These names are all reserved for special uses by Perl. Perl has a special syntax for the single-control-character names: It understands ^X (caret X) to mean the control-X character.

Perl identifiers that begin with digits, control characters, or punctuation characters are always forced to be in package main. A few other names are also exempt:

```

_ ARGV ARGVOUT ENV INC SIG STDIN STDOUT STDERR

```

## Regular Expressions

operators =~ and !~

Bind a pattern operation to a variable, i.e., pattern operation operates on the bound variable

If not bound then operators generally operate on \$\_

Result is true if something matched

commands

m// - match

// also works if it is unambiguous

s// - substitute

modifiers

i - case insensitive

m - multiple lines s ^ and \$ match in middle of string

s - single line so . matches newline

x - permit whitespace in patterns

## In Regular Expressions

Ordinary characters match themselves

## \ quote next metacharacter or sequence

\t	tab	(HT, TAB)
\n	newline	(LF, NL)
\r	return	(CR)
\f	form feed	(FF)
\a	alarm (bell)	(BEL)
\e	escape (think troff)	(ESC)
\033	octal char (think of a PDP-11)	
\x1B	hex char	
\x{263a}	wide hex char	(Unicode SMILEY)
\c[	control char	
\N{name}	named char	
\l	lowercase next char (think vi)	
\u	uppercase next char (think vi)	
\L	lowercase till \E (think vi)	
\U	uppercase till \E (think vi)	
\E	end case modification (think vi)	
\Q	quote (disable) pattern metacharacters till \E	
\w	Match a "word" character (alphanumeric plus "_")	
\W	Match a non-"word" character	
\s	Match a whitespace character	
\S	Match a non-whitespace character	
\d	Match a digit character	
\D	Match a non-digit character	
\pP	Match P, named property. Use \p{Prop} for longer names.	
\PP	Match non-P	
\X	Match eXtended Unicode "combining character sequence", equivalent to (?:\PM\pM*)	
\C	Match a single C char (octet) even under utf8.	
\b	Match a word boundary	
\B	Match a non-(word boundary)	
\A	Match only at beginning of string	
\Z	Match only at end of string, or before newline at the end	
\z	Match only at end of string	
\G	Match only at pos() (e.g. at the end-of-match position of prior m//g)	

## Special Characters in Regular Expressions

^ matches beginning of line

. matches any character

\$ matches end of line

| alternation - match either expression

() grouping

Postfix operators

\* 0 or more times

+ 1 or more times

? 0 or 1 time

{n}? exactly n times

{n,}? at least n times

{n,m}? match between n to m times

## Character Classes

[] character class

[abc] matches either a, b, or c

[a-z] is a range of characters

[abA-Z] Match these characters or range

^ means not in a character class

[^a] matches everything but a

[^abA-Z] Match all but these characters

[:class:]

alpha

alnum

ascii

blank

cntrl

digit \d

graph

Any alphanumeric or punctuation (special) character.

lower

print

Any alphanumeric or punctuation (special) character or space.

punct

Any punctuation (special) character.

space \s

upper

word \w

xdigit

Any hexadecimal digit. Though this may feel silly ([0-9A-Fa-f] would work just fine) it is included for completeness.

## Bracketting and capturing partial Matches

The bracketing construct ( ... ) creates capture buffers. To refer to the digit'th buffer use digit within the match. Outside the match use "\$" instead of "\\".

### Special Variables

\1 \2 etc. refer to the nth bracketed construct in the current regular expression

The regular expression /(.\*\1)/ matches any string that repeats twice like "abcabc"

\$1 \$2 etc. refer to the nth bracketed construct after the regular expression match completes

### Examples:

```
s/^( [^ ]* ) * ( [^ ]* ) /$2 $1/; # swap first two words
```

```
if (/(\.)\1/) { # find first doubled char
    print "'$1' is the first doubled character\n";
}
```

```
if (/Time: (..):(..):(..)/) { # parse out values
    $hours = $1;
    $minutes = $2;
    $seconds = $3;
}
```

### Removing C++ comments

```
s#/\[^\]*\*\++([\^/*][^\]*\*\++)*|//[^\n]*|("(\.|\[^\]
\])*"|\'(\.|\[^\']\])*')\.[^\/'"\n\]*)#$2#gs;
```

## Operators

### Operator Precedence and Associativity

Operator precedence and associativity work in Perl more or less like they do in mathematics.

Operator precedence means some operators are evaluated before others. For example, in 2 + 4 \* 5, the multiplication has higher precedence so 4 \* 5 is evaluated first yielding 2 + 20 == 22 and not 6 \* 5 == 30.

Operator associativity defines what happens if a sequence of the same operators is used one after another: whether the evaluator will evaluate the left operations first or the right. For example, in 8 - 4 - 2, subtraction is left associative so Perl evaluates the expression left to right. 8 - 4 is evaluated first making the expression 4 - 2 == 2 and not 8 - 2 == 6.

## Precedence

Perl operators have the following associativity and precedence, listed from highest precedence to lowest. Operators borrowed from C keep the same precedence relationship with each other, even where C's precedence is slightly screwy. (This makes learning Perl easier for C folks.) With very few exceptions, these all operate on scalar values only, not array values.

left	terms and list operators
(leftward)	
left	->
nonassoc	++ --
right	**
right	! ~ \ and unary + and -
left	=~ !~
left	* / % x
left	+ - .
left	<< >>
nonassoc	named unary operators
nonassoc	< > <= >= lt gt le ge
nonassoc	== != <=> eq ne cmp
left	&
left	^
left	&&
left	
nonassoc	.. ...
right	?:
right	= += -= *= etc.
left	, =>
nonassoc	list operators (rightward)
right	not
left	and
left	or xor

## Terms and List Operators (Leftward)

A TERM has the highest precedence in Perl. They include variables, quote and quote-like operators, any expression in parentheses, and any function whose arguments are parenthesized. Actually, there aren't really functions in this sense, just list operators and unary operators behaving as functions because you put parentheses around the arguments.

If any list operator (print(), etc.) or any unary operator (chdir(), etc.) is followed by a left parenthesis as the next token, the operator and arguments within parentheses are taken to be of highest precedence, just like a normal function call.

In the absence of parentheses, the precedence of list operators such as print, sort, or chmod is either very high or very low depending on whether you are looking at the left side or the right side of the operator. For example, in

```
@ary = (1, 3, sort 4, 2);
print @ary; # prints 1324
```

the commas on the right of the sort are evaluated before the sort, but the commas on the left are evaluated after. In other words, list operators tend to gobble up all arguments that follow, and then act like a simple TERM with regard to the preceding expression.

## Be careful with parentheses:

```
# These evaluate exit before doing the print:
print($foo, exit); # Obviously not what you want.
print $foo, exit; # Nor is this.
```

```
# These do the print before evaluating exit:
(print $foo), exit; # This is what you want.
print($foo), exit; # Or this.
print ($foo), exit; # Or even this.
```

Also note that

```
print ($foo & 255) + 1, "\n";
```

probably doesn't do what you expect at first glance. The parentheses enclose the argument list for print which is evaluated (printing the result of `$foo & 255`). Then one is added to the return value of print (usually 1). The result is something like this:

```
1 + 1, "\n"; # Obviously not what you meant.
```

To do what you meant properly, you must write:

```
print (($foo & 255) + 1, "\n");
```

Also parsed as terms are the `do {}` and `eval {}` constructs, as well as subroutine and method calls, and the anonymous constructors `[]` and `{}`.

## The Arrow Operator

`"->"` is an infix dereference operator, just as it is in C and C++. If the right side is either a `[...]`, `{...}`, or a `(...)` subscript, then the left side must be either a hard or symbolic reference to an array, a hash, or a subroutine respectively. (Or technically speaking, a location capable of holding a hard reference, if it's an array or hash reference being used for assignment.)

Otherwise, the right side is a method name or a simple scalar variable containing either the method name or a subroutine reference, and the left side must be either an object (a blessed reference) or a class name (that is, a package name).

## Auto-increment and Auto-decrement

`"++"` and `"--"` work as in C. That is, if placed before a variable, they increment or decrement the variable by one before returning the value, and if placed after, increment or decrement after returning the value.

```
$i = 0; $j = 0;
print $i++; # prints 0
print ++$j; # prints 1
```

The auto-increment operator has a little extra builtin magic to it. If you increment a variable that is numeric, or that has ever been used in a numeric context, you get a normal increment. If, however, the variable has been used in only string contexts since it was set, and has a value that is not the empty string and matches the pattern  `/^[a-zA-Z]*[0-9]*z/` , the increment is done as a string, preserving each character within its range, with carry:

```
print ++($foo = '99'); # prints '100'
print ++($foo = 'a0'); # prints 'a1'
print ++($foo = 'Az'); # prints 'Ba'
print ++($foo = 'zz'); # prints 'aaa'
```

`undef` is always treated as numeric, and in particular is changed to 0 before incrementing (so that a post-increment of an `undef` value will return 0 rather than `undef`).

The auto-decrement operator is not magical.

## Exponentiation

Binary `"**"` is the exponentiation operator.

It binds even more tightly than unary minus:

```
-2**4 is -(2**4), not (-2)**4
```

## Symbolic Unary Operators

Unary `"!"` performs logical negation or "not"

See also `"not"` for a lower precedence version of `"!"`.

Unary `"-"` performs arithmetic negation if the operand is numeric

If the operand is an identifier, a string consisting of a minus sign concatenated with the identifier is returned

Otherwise, if the string starts with a plus or minus, a string starting with the opposite sign is returned

One effect of these rules is that `-bareword` is equivalent to `"-bareword"`.

Unary `"~"` performs bitwise negation or 1's complement

Note that the width of the result is platform-dependent: `~0` is 32 bits wide on a 32-bit platform, but 64 bits wide on a 64-bit platform

Unary `"+"` has no effect whatsoever, even on strings

It is useful syntactically for separating a function name from a parenthesized expression that would otherwise be interpreted as the complete list of function arguments.

Unary `"\"` creates a reference to whatever follows it

Do not confuse this behavior with the behavior of backslash within a string

## Binding Operators

Binary `"=~"` binds a scalar expression to a pattern match

Certain operations search or modify the string `$_` by default

This operator makes that kind of operation work on some other string

The right argument is a search pattern, substitution, or transliteration

The left argument is what is supposed to be searched, substituted, or transliterated instead of the default `$_`

When used in scalar context, the return value generally indicates the success of the operation

Behavior in list context depends on the particular operator

If the right argument is an expression rather than a search pattern, substitution, or transliteration, it is interpreted as a search pattern at run time

Binary `"!~"` is just like `"=~"` except the return value is negated in the logical sense

## Multiplicative Operators

Binary `"**"` multiplies two numbers.

Binary `"/"` divides two numbers.

Binary `"%"` computes the modulus of two numbers

Binary `"x"` is the repetition operator

In scalar context or if the left operand is not enclosed in parentheses, it returns a string consisting of the left operand repeated the number of times specified by the right operand

In list context, if the left operand is enclosed in parentheses, it repeats the list

If the right operand is zero or negative, it returns an empty string or an empty list, depending on the context

```
print '-' x 80; # print row of dashes
```

```
print "\t" x ($tab/8), ' ' x ($tab%8); # tab over
```

```
@ones = (1) x 80; # a list of 80 1's
```

```
@ones = (5) x @ones; # set all elements to 5
```

## Additive Operators

Binary `"+"` returns the sum of two numbers.

Binary `"-"` returns the difference of two numbers.

Binary `"."` concatenates two strings.

## Shift Operators

Binary `"<<"` returns the value of its left argument shifted left by the number of bits specified by the right argument

Arguments should be integers.

Binary `">>"` returns the value of its left argument shifted right by the number of bits specified by the right argument

Arguments should be integers.

Note that both "<<" and ">>" in Perl are implemented directly using "<<" and ">>" in C

The result of overflowing the range of the integers is undefined because it is undefined also in C

Shifting by a negative number of bits is also undefined

## Named Unary Operators

The various named unary operators are treated as functions with one argument, with optional parentheses

Regarding precedence, the filetest operators, like -f, -M, etc. are treated like named unary operators, but they don't follow this functional parenthesis rule

That means, for example, that -f(\$file)."bak" is equivalent to -f "\$file.bak".

## Relational Operators

Binary "<" returns true if the left argument is numerically less than the right argument

Binary ">" returns true if the left argument is numerically greater than the right argument

Binary "<=" returns true if the left argument is numerically less than or equal to the right argument

Binary ">=" returns true if the left argument is numerically greater than or equal to the right argument

Binary "lt" returns true if the left argument is stringwise less than the right argument

Binary "gt" returns true if the left argument is stringwise greater than the right argument

Binary "le" returns true if the left argument is stringwise less than or equal to the right argument

Binary "ge" returns true if the left argument is stringwise greater than or equal to the right argument

## Equality Operators

Binary "==" returns true if the left argument is numerically equal to the right argument

Binary "!=" returns true if the left argument is numerically not equal to the right argument

Binary "<=>" returns -1, 0, or 1 depending on whether the left argument is numerically less than, equal to, or greater than the right argument

If your platform supports NaNs (not-a-numbers) as numeric values, using them with "<=>" returns undef  
NaN is not "<", "==", ">", "<=" or ">=" anything (even NaN), so those 5 return false. NaN != NaN returns true, as does NaN != anything else

If your platform doesn't support NaNs then NaN is just a string with numeric value 0.

```
perl -le '$a = NaN; print "No NaN support" if $a == $a'
```

```
perl -le '$a = NaN; print "NaN support" if $a != $a'
```

Binary "eq" returns true if the left argument is stringwise equal to the right argument

Binary "ne" returns true if the left argument is stringwise not equal to the right argument

Binary "cmp" returns -1, 0, or 1 depending on whether the left argument is stringwise less than, equal to, or greater than the right argument

"lt", "le", "ge", "gt" and "cmp" use the collation (sort) order specified by the current locale if use locale is in effect

## Bitwise operators

### Bitwise And

Binary "&" returns its operands ANDed together bit by bit

Note that "&" has lower priority than relational operators, so for example the brackets are essential in a test like

```
print "Even\n" if ($x & 1) == 0;
```

### Bitwise Or and Exclusive Or

Binary "|" returns its operands ORed together bit by bit

Binary "^" returns its operands XORed together bit by bit

Note that "|" and "^" have lower priority than relational operators, so for example the brackets are essential in a test like

```
print "false\n" if (8 | 2) != 10;
```

## Logical Operators

### C-style Logical And

Binary "&&" performs a short-circuit logical AND operation

Scalar or list context propagates down to the right operand if it is evaluated

### C-style Logical Or

Binary "||" performs a short-circuit logical OR operation

Scalar or list context propagates down to the right operand if it is evaluated

The || and && operators return the last value evaluated (unlike C's || and &&, which return 0 or 1)

Thus, a reasonably portable way to find out the home directory might be:

```
$home = $ENV{'HOME'} || $ENV{'LOGDIR'} ||  
(getpwuid($<))[7] || die "You're homeless!\n";
```

In particular, this means that you shouldn't use this for selecting between two aggregates for assignment:

```
@a = @b || @c;           # this is wrong  
@a = scalar(@b) || @c;   # really meant this  
@a = @b ? @b : @c;       # this works fine, though
```

As more readable alternatives to && and || when used for control flow, Perl provides and and or operators (see below)

The short-circuit behavior is identical

The precedence of "and" and "or" is much lower, however, so that you can safely use them after a list operator without the need for parentheses:

```
unlink "alpha", "beta", "gamma"  
or gripe(), next LINE;
```

With the C-style operators that would have been written like this:

```
unlink("alpha", "beta", "gamma")  
|| (gripe(), next LINE);
```

Using "or" for assignment is unlikely to do what you want; see below

## Range Operators

Binary ".." is the range operator

In list context, it returns a list of values counting (up by ones) from the left value to the right value

If the left value is greater than the right value then it returns the empty list

The range operator is useful for writing foreach (1..10) loops and for doing slice operations on arrays.

The range operator also works on strings, using the magical auto-increment (see below)

In scalar context, ".." returns a boolean value

The operator is bistable, like a flip-flop, and emulates the line-range (comma) operator of sed, awk, and various editors

Each ".." operator maintains its own boolean state

It is false as long as its left operand is false

Once the left operand is true, the range operator stays true until the right operand is true, AFTER which the range operator becomes false again

The right operand is not evaluated while the operator is in the "false" state, and the left operand is not evaluated while the operator is in the "true" state

The precedence is a little lower than || and &&

The value returned is either the empty string for false, or a sequence number (beginning with 1) for true

The sequence number is reset for each range encountered

The final sequence number in a range has the string "E0" appended to it, which doesn't affect its numeric

value, but gives you something to search for if you want to exclude the endpoint

You can exclude the beginning point by waiting for the sequence number to be greater than 1

If either operand of scalar "." is a constant expression, that operand is considered true if it is equal (==) to the current input line number (the \$. variable).

## Examples:

As a scalar operator:

```
if (101 .. 200) { print; } # print 2nd hundred lines
```

short for

```
if ($. == 101 .. $. == 200) ...
```

```
next line if (1 .. /^$/); # skip header lines
```

short for

```
... if ($. == 1 .. /^$/);
```

```
s/^/> / if (/^$/ .. eof()); # quote body
```

parse mail messages

```
while (<>) {
    $in_header = 1 .. /^$/;
    $in_body = /^$/ .. eof;
    if ($in_header) {
        # ...
    } else { # in body
        # ...
    }
} continue {
    close ARGV if eof; # reset $. each file
}
```

## Example to illustrate the difference between the two range operators:

```
@lines = (" - Foo",
          "01 - Bar",
          "1 - Baz",
          " - Quux");
```

```
foreach(@lines)
{
    if (/0/ .. /1/)
    {
        print "$_\n";
    }
}
```

This program will print only the line containing "Bar"

If the range operator is changed to ..., it will also print the "Baz" line

## And now some examples as a list operator:

```
for (101 .. 200) { print; } # print $_ 100 times
@foo = @foo[0 .. $#foo]; # an expensive no-op
@foo = @foo[$#foo-4 .. $#foo]; # slice last 5 items
```

The range operator (in list context) makes use of the magical auto-increment algorithm if the operands are strings. You can say

```
@alphabet = ('A' .. 'Z');
```

to get all normal letters of the English alphabet, or

```
$hexdigit = (0 .. 9, 'a' .. 'f')[$num & 15];
```

to get a hexadecimal digit, or

```
@z2 = ('01' .. '31'); print $z2[$mday];
```

to get dates with leading zeros. If the final value specified is not in the sequence that the magical increment would produce, the sequence goes until the next value would be longer than the final value specified.

Because each operand is evaluated in integer form, 2.18 .. 3.14 will return two elements in list context.

```
@list = (2.18 .. 3.14); # same as @list = (2 .. 3);
```

## Conditional Operator

Ternary "?:" is the conditional operator, just as in C

For example:

```
printf "I have %d dog%s.\n", $n,
      ($n == 1) ? ' ' : "s";
```

Scalar or list context propagates downward into the 2nd or 3rd argument, whichever is selected:

```
$a = $ok ? $b : $c; # get a scalar
@a = $ok ? @b : @c; # get an array
$a = $ok ? @b : @c; # oops, that's just a count!
```

The operator may be assigned to if both the 2nd and 3rd arguments are legal lvalues:

```
($a_or_b ? $a : $b) = $c;
```

Because this operator produces an assignable result, using assignments without parentheses will get you in trouble.

For example, this:

```
$a % 2 ? $a += 10 : $a += 2
```

Really means this:

```
((($a % 2) ? ($a += 10) : $a) += 2)
```

Rather than this:

```
($a % 2) ? ($a += 10) : ($a += 2)
```

That should probably be written more simply as:

```
$a += ($a % 2) ? 10 : 2;
```

## Assignment Operators

"=" is the ordinary assignment operator.

Assignment operators work as in C. That is,

```
$a += 2;
```

is equivalent to

```
$a = $a + 2;
```

although without duplicating any side effects that dereferencing the lvalue might trigger, such as from tie(). Other assignment operators work similarly. The following are recognized:

**=	+=	*=	&=	<<=	&&=
	-=	/=	=	>>=	=
	.	%=	^=		
		x=			

Although these are grouped by family, they all have the precedence of assignment.

Unlike in C, the scalar assignment operator produces a valid lvalue. Modifying an assignment is equivalent to doing the assignment and then modifying the variable that was assigned to. This is useful for modifying a copy of something, like this:

```
($tmp = $global) =~ tr [A-Z] [a-z];
```

Likewise,

```
($a += 2) *= 3;
```

is equivalent to

```
$a += 2;
```

```
$a *= 3;
```

Similarly, a list assignment in list context produces the list of lvalues assigned to, and a list assignment in scalar context returns the number of elements produced by the expression on the right hand side of the assignment.

## Comma Operator

Binary ""," is the comma operator

In scalar context it evaluates its left argument, throws that value away, then evaluates its right argument and returns that value

In list context, it's just the list argument separator, and inserts both its arguments into the list

The => operator is a synonym for the comma, but forces any word to its left to be interpreted as a string (as of 5.001)

It is helpful in documenting the correspondence between keys and values in hashes, and other paired elements in lists

## List Operators (Rightward)

On the right side of a list operator, it has very low precedence, such that it controls all comma-separated expressions found there

The only operators with lower precedence are the logical operators "and", "or", and "not", which may be used to evaluate calls to list operators without the need for extra parentheses:

```
open HANDLE, "filename"
  or die "Can't open: $!\n";
```

## Logical words

### Logical Not

Unary "not" returns the logical negation of the expression to its right - It's the equivalent of "!" except for the very low precedence

### Logical And

Binary "and" returns the logical conjunction of the two surrounding expressions - It's equivalent to && except for the very low precedence

This means that it short-circuits: i.e., the right expression is evaluated only if the left expression is true

### Logical or and Exclusive Or

Binary "or" returns the logical disjunction of the two surrounding expressions - It's equivalent to || except for the very low precedence

This makes it useful for control flow

```
print FH $data or die "Can't write to FH: $!";
```

It also short-circuits: i.e., the right expression is evaluated only if the left expression is false

Due to its precedence, you should probably avoid using this for assignment, only for control flow.

```
$a = $b or $c;      # bug: this is wrong
($a = $b) or $c;   # really means this
$a = $b || $c;     # better written this way
```

However, when it's a list-context assignment and you're trying to use "||" for control flow, you probably need "or" so that the assignment takes higher precedence:

```
@info = stat($file) || die; # wrong, stat is
scalar
@info = stat($file) or die; # better, list value
```

Then again, you could always use parentheses

Binary "xor" returns the exclusive-OR of the two surrounding expressions

It cannot short circuit, of course.

## C Operators Missing From Perl

Here is what C has that Perl doesn't:

unary &

Address-of operator (but see the "&" operator for taking a reference)

unary \*

Dereference-address operator. (Perl's prefix dereferencing operators are typed: \$, @, %, and &)

(TYPE)

Type-casting operator

## Quote and Quote-like Operators

While we usually think of quotes as literal values, in Perl they function as operators, providing various kinds of interpolating and pattern matching capabilities

Perl provides customary quote characters for these behaviors, but also provides a way for you to choose your quote character for any of them. In the following table, a {} represents any pair of delimiters you choose.

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?
<<EOF		here-doc	yes*

\* unless the delimiter is ''.

Non-bracketing delimiters use the same character fore and aft, but the four sorts of brackets (round, angle, square, curly) will all nest, which means that

```
q{foo{bar}baz}
```

is the same as

```
'foo{bar}baz'
```

There can be whitespace between the operator and the quoting characters, except when # is being used as the quoting character

q#foo# is parsed as the string foo

q #foo# is the operator q followed by a comment

Its argument will be taken from the next line

This allows you to write:

```
s {foo} # Replace foo
  {bar} # with bar.
```

## Escape Sequences In Transliterations

The following escape sequences are available in constructs that interpolate and in transliterations.

\t	tab	(HT, TAB)
\n	newline	(NL)
\r	return	(CR)
\f	form feed	(FF)
\b	backspace	(BS)
\a	alarm (bell)	(BEL)
\e	escape	(ESC)
\033	octal char	(ESC)
\x1b	hex char	(ESC)
\x{263a}	wide hex char	(SMILEY)
\c{ }	control char	(ESC)
\N{name}	named Unicode character	

The following escape sequences are available in constructs that interpolate but not in transliterations:

\l	lowercase next char
\u	uppercase next char
\L	lowercase till \E
\U	uppercase till \E
\E	end case modification
\Q	quote non-word characters till \E

All systems use the virtual "\n" to represent a line terminator, called a "newline"

There is no such thing as an unvarying, physical newline character.

## Interpolation

For constructs that do interpolate, variables beginning with "\$" or "@" are interpolated

Subscripted variables such as \$a[3] or \$href->{key}[0] are also interpolated, as are array and hash slices

But method calls such as \$obj->meth are not.

Interpolating an array or slice interpolates the elements in order, separated by the value of "\$", so is equivalent to interpolating join "\$", @array

"Punctuation" arrays such as @+ are only interpolated if the name is enclosed in braces @{+}

You cannot include a literal \$ or @ within a \Q sequence

An unescaped \$ or @ interpolates the corresponding variable, while escaping will cause the literal string \\$ to be inserted

You'll need to write something like m\Quser\E@\Qhost/

Patterns are subject to an additional level of interpretation as a regular expression

This is done as a second pass, after variables are interpolated, so that regular expressions may be incorporated into the pattern from the variables

If this is not what you want, use `\Q` to interpolate a variable literally

Apart from the behavior described above, Perl does not expand multiple levels of interpolation

In particular, contrary to the expectations of shell programmers, back-quotes do NOT interpolate within double quotes, nor do single quotes impede evaluation of variables when used within double quotes

## Regular Expressions

A regular expression is an expression that gives a rule for matching strings.

The simplest regular expression is a simple string that matches itself

Classically, a regular expression is created by the following rules

The empty string is a regular expression

Any character is a regular expression

Two regular expressions can be combined with the "concatenation" operator which matches anything which is the concatenation of something matching the first regular expression immediately followed by something matching the second regular expression

Two regular expressions can be combined with the "alternation" operator which matches if either alternative matches

A regular expression followed by the "Kleene star" operator `"*"` which match anything that is zero or more concatenated strings each of which match the operand regular expression

Sometimes the following operators are added

"not" - anything that does not match the regular expression

"and" - anything that simultaneously matches both operand regular expressions

## Regexp Quote-Like Operators

`?PATTERN?`

This is just like the `/pattern/` search, except that it matches only once between calls to the `reset()` operator.

## Match operator

`m/PATTERN/cgimosx`

`/PATTERN/cgimosx`

Searches a string for a pattern match, and in scalar context returns true if it succeeds, false if it fails. If no string is specified via the `=~` or `!~` operator, the `$_` string is searched. (The string specified with `=~` need not be an lvalue—it may be the result of an expression evaluation, but remember the `=~` binds rather tightly.)

Options are:

- `c` Do not reset search position on a failed match when `/g` is in effect.
- `g` Match globally, i.e., find all occurrences.
- `i` Do case-insensitive pattern matching.
- `m` Treat string as multiple lines.
- `o` Compile pattern only once.
- `s` Treat string as single line.
- `x` Use extended regular expressions.

If `"/"` is the delimiter then the initial `m` is optional. With the `m` you can use any pair of non-alphanumeric, non-whitespace characters as delimiters.

This is particularly useful for matching path names that contain `"/"`, to avoid LTS (leaning toothpick syndrome).

If `"?"` is the delimiter, then the match-only-once rule of `?PATTERN?` applies. If `""` is the delimiter, no interpolation is performed on the `PATTERN`.

`PATTERN` may contain variables, which will be interpolated (and the pattern recompiled) every time the pattern search is evaluated, except for when the delimiter is a single quote. (Note that `$(, $)`, and `$|` are not interpolated because they look like end-of-string tests.)

If the `PATTERN` evaluates to the empty string, the last successfully matched regular expression is used instead.

If the `/g` option is not used, `m//` in list context returns a list consisting of the subexpressions matched by the parentheses in the pattern, i.e., `($1, $2, $3...)`. (Note that here `$1` etc. are also set, and that this differs from Perl 4's behavior.) When there are no parentheses in the pattern, the return value is the list `(1)` for success. With or without parentheses, an empty list is returned upon failure.

## m// Examples:

```
open(TTY, '/dev/tty');
<TTY> =~ /^y/i && foo(); # do foo if desired

if (/Version: *([0-9.]*)/) { $version = $1; }

next if m#^/usr/spool/uucp#;

# poor man's grep
$arg = shift;
while (<>) {
    print if /$arg/o; # compile only once
}

if (($F1, $F2, $Etc) = ($foo =~
/^(\\S+)\\s+(\\S+)\\s*(.*)/))
```

This last example splits `$foo` into the first two words and the remainder of the line, and assigns those three fields to `$F1`, `$F2`, and `$Etc`. The conditional is true if any variables were assigned, i.e., if the pattern matched.

## The /g modifier

The `/g` modifier specifies global pattern matching—that is, matching as many times as possible within the string. How it behaves depends on the context. In list context, it returns a list of the substrings matched by any capturing parentheses in the regular expression. If there are no parentheses, it returns a list of all the matched strings, as if there were parentheses around the whole pattern.

In scalar context, each execution of `m//g` finds the next match, returning true if it matches, and false if there is no further match. The position after the last match can be read or set using the `pos()` function. A failed match normally resets the search position to the beginning of the string, but you can avoid that by adding the `/c` modifier (e.g. `m//gc`). Modifying the target string also resets the search position.

Examples:

```
# list context
($one,$five,$fifteen) = (`uptime` =~
/(\d+\.\d+)/g);

# scalar context
$/ = "";
while (defined($paragraph = <>)) {
    while ($paragraph =~ /[a-z]
[""]*[.!?]+[""]*\s/g) {
        $sentences++;
    }
}
print "$sentences\n";
```

## Quoting operators

q/STRING/

'STRING'

A single-quoted, literal string. A backslash represents a backslash unless followed by the delimiter or another backslash, in which case the delimiter or backslash is interpolated.

```
$foo = q!I said, "You said, 'She said it.'!";
$bar = q('This is it.');
```

```
$baz = '\n'; # a two-character string
```

qq/STRING/

"STRING"

A double-quoted, interpolated string.

```
$_ .= qq
```

```
(The previous line contains the naughty word
"$1".\n)
```

```
if /\b(tcl|java|python)\b/i; # :-)
```

```
$baz = "\n"; # a one-character string
```

## Quoting Regular Expressions

qr/STRING/imosx

This operator quotes (and possibly compiles) its STRING as a regular expression. STRING is interpolated the same way as PATTERN in m/PATTERN/. If ' is used as the delimiter, no interpolation is done. Returns a Perl value which may be used instead of the corresponding /STRING/imosx expression.

For example,

```
$rex = qr/my.STRING/is;
s/$rex/foo/;
```

is equivalent to

```
s/my.STRING/foo/is;
```

The result may be used as a subpattern in a match:

```
$re = qr/$pattern/;
```

```
$string =~ /foo${re}bar/; # can be interpolated
```

```
$string =~ $re; # or used standalone
```

```
$string =~ /$re/; # or this way
```

Options are:

- i Do case-insensitive pattern matching.
- m Treat string as multiple lines.
- o Compile pattern only once.
- s Treat string as single line.
- x Use extended regular expressions.

## Quoting Shell Commands

qx/STRING/

`STRING`

A string which is (possibly) interpolated and then executed as a system command with /bin/sh or its equivalent. Shell wildcards, pipes, and redirections will be honored. The collected standard output of the command is returned; standard error is unaffected. In scalar context, it comes back as a single (potentially multi-line) string, or undef if the command failed. In list context, returns a list of lines (however you've defined lines with \$/ or \$INPUT\_RECORD\_SEPARATOR), or an empty list if the command failed.

Because backticks do not affect standard error, use shell file descriptor syntax (assuming the shell supports this) if you care to address this. To capture a command's STDERR and STDOUT together:

```
$output = `cmd 2>&1`;
```

To capture a command's STDOUT but discard its STDERR:

```
$output = `cmd 2>/dev/null`;
```

To capture a command's STDERR but discard its STDOUT (ordering is important here):

```
$output = `cmd 2>&1 1>/dev/null`;
```

Using single-quote as a delimiter protects the command from Perl's double-quote interpolation, passing it on to the shell instead:

```
$perl_info = qx(ps $$); # that's Perl's $$
```

```
$shell_info = qx'ps $$'; # that's the new shell's $$
```

## Quote Words

qw/STRING/

Evaluates to a list of the words extracted out of STRING, using embedded whitespace as the word delimiters. It can be understood as being roughly equivalent to:

```
split(' ', q/STRING/);
```

the differences being that it generates a real list at compile time, and in scalar context it returns the last element in the list. So this expression:

```
qw(foo bar baz)
```

is semantically equivalent to the list:

```
'foo', 'bar', 'baz'
```

Some frequently seen examples:

```
use POSIX qw( setlocale localeconv )
```

```
@EXPORT = qw( foo bar baz );
```

A common mistake is to try to separate the words with comma or to put comments into a multi-line qw-string. For this reason, the use warnings pragma and the -w switch (that is, the \$^W variable) produces warnings if the STRING contains the "," or the "#" character.

## Substitution

s/PATTERN/REPLACEMENT/egimosx

Searches a string for a pattern, and if found, replaces that pattern with the replacement text and returns the number of substitutions made. Otherwise it returns false (specifically, the empty string).

If no string is specified via the =~ or !~ operator, the \$\_ variable is searched and modified. (The string specified with =~ must be scalar variable, an array element, a hash element, or an assignment to one of those, i.e., an lvalue.)

If the delimiter chosen is a single quote, no interpolation is done on either the PATTERN or the REPLACEMENT. Otherwise, if the PATTERN contains a \$ that looks like a variable rather than an end-of-string test, the variable will be interpolated into the pattern at run-time. If you want the pattern compiled only once the first time the variable is interpolated, use the /o option. If the pattern evaluates to the empty string, the last successfully executed regular expression is used instead.

Options are:

- e Evaluate the right side as an expression.
- g Replace globally, i.e., all occurrences.
- i Do case-insensitive pattern matching.
- m Treat string as multiple lines.
- o Compile pattern only once.
- s Treat string as single line.
- x Use extended regular expressions.

Any non-alphanumeric, non-whitespace delimiter may replace the slashes. If single quotes are used, no interpretation is done on the replacement string (the /e modifier overrides this, however). Unlike Perl 4, Perl 5 treats backticks as normal delimiters; the replacement text is not evaluated as a command. If the PATTERN is delimited by bracketing quotes, the REPLACEMENT has its own pair of quotes, which may or may not be bracketing quotes, e.g., s(foo)(bar) or s<foo>/bar/. A /e will cause the replacement portion to be treated as a full-fledged Perl expression and evaluated right then and there. It is, however, syntax checked at compile-time. A second e modifier will cause the replacement portion to be evaluated before being run as a Perl expression.

## Examples:

```
s/\bgreen\b/mauve/g; # don't change wintergreen

$paths = ~ s|usr/bin|usr/local/bin|;

s/Login: $foo/Login: $bar/; # run-time pattern

($foo = $bar) = ~ s/this/that/; # copy, then
change

$count = ($paragraph = ~ s/Mister\b/Mr./g);
        # get change-count

$_ = 'abc123xyz';
s/\d+/$&*2/e;           # yields 'abc246xyz'
s/\d+/sprintf("%5d", $&)/e; # yields 'abc 246xyz'
s/\w/$& x 2/eg; # yields 'aabbcc 224466xxxyzz'

# Add one to the value of any numbers in the
string
s/(\d+)/1 + $1/eg;

s/([ ]*) *([ ]*)/$2 $1/; # reverse 1st two
fields
```

Note the use of \$ instead of \ in the last example. Unlike sed, we use the \<digit> form in only the left hand side. Anywhere else it's \$<digit>.

## Transliteration

```
tr/SEARCHLIST/REPLACEMENTLIST/cds
y/SEARCHLIST/REPLACEMENTLIST/cds
```

Transliterates all occurrences of the characters found in the search list with the corresponding character in the replacement list. It returns the number of characters replaced or deleted. If no string is specified via the =~ or !~ operator, the \$\_ string is transliterated. (The string specified with =~ must be a scalar variable, an array element, a hash element, or an assignment to one of those, i.e., an lvalue.)

A character range may be specified with a hyphen, so tr/A-J/0-9/ does the same replacement as tr/ACEGIBDFHJ/0246813579/. If the SEARCHLIST is delimited by bracketing quotes, the REPLACEMENTLIST has its own pair of quotes, which may or may not be bracketing quotes, e.g., tr[A-Z][a-z] or tr(+~\*)/ABCD/.

Note also that the whole range idea is rather unportable between character sets--and even within character sets they may cause results you probably didn't expect. A sound principle is to use only ranges that begin from and end at either alphabets of equal case (a-e, A-E), or digits (0-4). Anything else is unsafe. If in doubt, spell out the character sets in full.

### Options:

```
c Complement the SEARCHLIST.
d Delete found but unreplaced characters.
s Squash duplicate replaced characters.
```

If the /c modifier is specified, the SEARCHLIST character set is complemented. If the /d modifier is specified, any characters specified by SEARCHLIST not found in REPLACEMENTLIST are deleted. (Note that this is slightly more flexible than the behavior of some tr programs, which delete anything they find in the SEARCHLIST, period.) If the /s modifier is specified, sequences of characters that were transliterated to the same character are squashed down to a single instance of the character.

If the /d modifier is used, the REPLACEMENTLIST is always interpreted exactly as specified. Otherwise, if the REPLACEMENTLIST is shorter than the SEARCHLIST, the final character is replicated till it is long enough. If the REPLACEMENTLIST is empty, the SEARCHLIST is replicated. This latter is useful for counting characters in a class or for squashing character sequences in a class.

## Examples:

```
$ARGV[1] =~ tr/A-Z/a-z/; # make all lower case

$count = tr/*/*/;           # count the stars in $_

$count = $sky =~ tr/*/*/;  # count the stars in $sky

$count = tr/0-9//;         # count the digits in $_

tr/a-zA-Z//s;              # bookkeeper -> bokeper

($HOST = $host) =~ tr/a-zA-Z/;

tr/a-zA-Z/ /cs; # change non-alphas to single
space

tr [\200-\377
 [\000-\177];           # delete 8th bit
```

If multiple transliterations are given for a character, only the first one is used:

```
tr/AAA/XYZ/
```

will transliterate any A to X.

Because the transliteration table is built at compile time, neither the SEARCHLIST nor the REPLACEMENTLIST are subjected to double quote interpolation. That means that if you want to use variables, you must use an eval():

```
eval "tr/$oldlist/$newlist/";
die $$@ if $$@;
```

```
eval "tr/$oldlist/$newlist/, 1" or die $$@;
```

## line-oriented Quoting

<<EOF

Following a << you specify a string to terminate the quoted material, and all lines following the current line down to the terminating string are the value of the item. The terminating string may be either an identifier (a word), or some quoted text. An unquoted identifier works like double quotes. There must be no space between the << and the identifier, unless the identifier is quoted. The terminating string must appear by itself (unquoted and with no surrounding whitespace) on the terminating line.

```
print <<EOF;
The price is $Price.
EOF
```

```
print << "EOF"; # same as above
The price is $Price.
EOF
```

```
print << `EOC`; # execute commands
echo hi there
echo lo there
EOC
```

```
print <<"foo", <<"bar"; # you can stack
them
I said foo.
foo
I said bar.
bar
```

```
myfunc(<< "THIS", 23, <<'THAT');
Here's a line
or two.
THIS
and here's another.
THAT
```

Just don't forget that you have to put a semicolon on the end to finish the statement.

## Gory details of parsing quoted constructs

### I/O Operators

There are several I/O operators you should know about.

In scalar context, evaluating a filehandle in angle brackets yields the next line from that file (the newline, if any, included), or undef at end-of-file or on error. When \$/ is set to undef (sometimes known as file-slurp mode) and the file is empty, it returns "" the first time, followed by undef subsequently.

Ordinarily you must assign the returned value to a variable, but there is one situation where an automatic assignment happens. If and only if the input symbol is the only thing inside the conditional of a while statement (even if disguised as a for(;;) loop), the value is automatically assigned to the global variable \$\_, destroying whatever was there previously. The \$\_ variable is not implicitly localized. You'll have to put a local \$\_; before the loop if you want that to happen.

The following lines are equivalent:

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

This also behaves similarly, but avoids \$\_ :

```
while (my $line = <STDIN>) { print $line }
```

In these loop constructs, the assigned value (whether assignment is automatic or explicit) is then tested to see whether it is defined. The defined test avoids problems where line has a string value that would be treated as false by Perl.

The filehandles STDIN, STDOUT, and STDERR are predefined.

If a <FILEHANDLE> is used in a context that is looking for a list, a list comprising all input lines is returned, one line per list element.

<FILEHANDLE> may also be spelled `readline(*FILEHANDLE)`.

### The null filehandle

The null filehandle <> is special. Input from <> comes either from standard input, or from each file listed on the command line. Here's how it works: the first time <> is evaluated, the @ARGV array is checked, and if it is empty, \$ARGV[0] is set to "", which when opened gives you standard input. The @ARGV array is then processed as a list of filenames. The loop

```
while (<>) {
    ... # code for each line
}
```

is almost equivalent to the following Perl-like pseudo code:

```
unshift(@ARGV, '-') unless @ARGV;
while ($ARGV = shift) {
    open(ARGV, $ARGV);
    while (<ARGV>) {
        ... # code for each line
    }
}
```

It really does shift the @ARGV array and put the current filename into the \$ARGV variable. It also uses filehandle ARGV internally--<> is just a synonym for <ARGV>, which is magical.

You can modify @ARGV before the first <> as long as the array ends up containing the list of filenames you really want. Line numbers (\$) continue as though the input were one big happy file.

If you want to set @ARGV to your own list of files, go right ahead. This sets @ARGV to all plain text files if no @ARGV was given:

```
@ARGV = grep { -f && -T } glob('*') unless @ARGV;
```

You can even set them to pipe commands. For example, this automatically filters compressed arguments through gzip:

```
@ARGV = map { /\. (gz|Z)$/ ? "gzip -dc < $_ |" : $_ } @ARGV;
```

The <> symbol will return undef for end-of-file only once. If you call it again after this, it will assume you are processing another @ARGV list, and if you haven't set @ARGV, will read input from STDIN.

If what the angle brackets contain is a simple scalar variable (e.g., <\$foo>), then that variable contains the name of the filehandle to input from, or its typeglob, or a reference to the same. For example:

```
$fh = \*STDIN;
$line = <$fh>;
```

If what's within the angle brackets is neither a filehandle nor a simple scalar variable containing a filehandle name, typeglob, or typeglob reference, it is interpreted as a filename pattern to be globbed, and either a list of filenames or the next filename in the list is returned, depending on context. This distinction is determined on syntactic grounds alone. That means <\$x> is always a readline() from an indirect handle, but <\${hash{key}}> is always a glob(). That's because \$x is a simple scalar variable, but \${hash{key}} is not--it's a hash element.

One level of double-quote interpretation is done first, but you can't say <\$foo> because that's an indirect filehandle as explained in the previous paragraph. (In older versions of Perl, programmers would insert curly brackets to force interpretation as a filename glob: <\${foo}>. These days, it's considered cleaner to call the internal function directly as glob(\$foo), which is probably the right way to have done it in the first place.) For example:

```
while (<*.c>) {
    chmod 0644, $_;
}
```

is the same as

```
chmod 0644, <*.c>;
($file) = <blurch*>;
```

## Functions and reserved words

### Language

**bless** REF,CLASSNAME

**bless** REF

This function tells the thingy referenced by REF that it is now an object in the CLASSNAME package. If CLASSNAME is omitted, the current package is used. Because a bless is often the last thing in a constructor, it returns the reference for convenience.

**continue** BLOCK

Actually a flow control statement rather than a function. If there is a continue BLOCK attached to a BLOCK (typically in a while or foreach), it is always executed just before the conditional is about to be evaluated again, just like the third part of a for loop in C. Thus it can be used to increment a loop variable, even when the loop has been continued via the next statement (which is similar to the C continue statement).

last, next, or redo may appear within a continue block. last and redo will behave as if they had been executed within the main block. So will next, but since it will execute a continue block, it may be more entertaining.

```
while (EXPR) {  
    ### redo always comes here  
    do_something;  
} continue {  
    ### next always comes here  
    do_something_else;  
    # then back the top to re-check EXPR  
}  
### last always comes here
```

Omitting the continue section is semantically equivalent to using an empty one, logically enough. In that case, next goes directly back to check the condition at the top of the loop.

**defined** EXPR

**defined**

Returns a Boolean value telling whether EXPR has a value other than the undefined value undef. If EXPR is not present, \$\_ will be checked.

**delete** EXPR

Given an expression that specifies a hash element, array element, hash slice, or array slice, deletes the specified element(s) from the hash or array. In the case of an array, if the array elements happen to be at the end, the size of the array will shrink to the highest element that tests true for exists() (or 0 if no such element exists).

Returns a list with the same number of elements as the number of elements for which deletion was attempted. Each element of that list consists of either the value of the element deleted, or the undefined value. In scalar context, this means that you get the value of the last element deleted (or the undefined value if that element did not exist).

Deleting an array element effectively returns that position of the array to its initial, uninitialized state. Subsequently testing for the same element with exists() will return false.

**die** LIST

Outside an eval, prints the value of LIST to STDERR and exits with the current value of \$! (errno). If \$! is 0, exits with the value of (\$? >> 8) (backtick `command` status). If (\$? >> 8) is 0, exits with 255. Inside an eval(), the error message is stuffed into \$@ and the eval is terminated with the undefined value. This makes die the way to raise an exception.

If the last element of LIST does not end in a newline, the current script line number and input line number (if any) are also printed, and a newline is supplied. Note that the "input line number" (also known as "chunk") is subject to whatever notion of "line" happens to be currently in effect, and is also available as the special variable \$.

**do** BLOCK

Not really a function. Returns the value of the last command in the sequence of commands indicated by BLOCK. When modified by a loop modifier, executes the BLOCK once before testing the loop condition. (On other statements the loop modifiers test the conditional first.)

do BLOCK does not count as a loop, so the loop control statements next, last, or redo cannot be used to leave or restart the block.

**do** SUBROUTINE(LIST)

A deprecated form of subroutine call.

**do** EXPR

Uses the value of EXPR as a filename and executes the contents of the file as a Perl script. Its primary use is to include subroutines from a Perl subroutine library.

**each** HASH

When called in list context, returns a 2-element list consisting of the key and value for the next element of a hash, so that you can iterate over it. When called in scalar context, returns only the key for the next element in the hash.

Entries are returned in an apparently random order. The actual random order is subject to change in future versions of perl, but it is guaranteed to be in the same order as either the keys or values function would produce on the same (unmodified) hash. Since Perl 5.8.1 the ordering is different even between different runs of Perl for security reasons.

When the hash is entirely read, a null array is returned in list context (which when assigned produces a false (0) value), and undef in scalar context. The next call to each after that will start iterating again. There is a single iterator for each hash, shared by all each, keys, and values function calls in the program; it can be reset by reading all the elements from the hash, or by evaluating keys HASH or values HASH. If you add or delete elements of a hash while you're iterating over it, you may get entries skipped or duplicated, so don't. Exception: It is always safe to delete the item most recently returned by each().

**eval** EXPR

**eval** BLOCK

In the first form, the return value of EXPR is parsed and executed as if it were a little Perl program. The value of the expression (which is itself determined within scalar context) is first parsed, and if there weren't any errors, executed in the lexical context of the current Perl program, so that any variable settings or subroutine and format definitions remain afterwards. Note that the value is parsed every time the eval executes. If EXPR is omitted, evaluates \$\_. This form is typically used to delay parsing and subsequent execution of the text of EXPR until run time.

In the second form, the code within the BLOCK is parsed only once--at the same time the code surrounding the eval itself was parsed--and executed within the context of the current Perl program. This form is typically used to trap exceptions more efficiently than the first (see below), while also providing the benefit of checking the code within BLOCK at compile time.

In both forms, the value returned is the value of the last expression evaluated inside the mini-program; a return statement may be also used, just as with subroutines. The expression providing the return value is evaluated in void, scalar, or list context, depending on the context of the eval itself.

If there is a syntax error or runtime error, or a die statement is executed, an undefined value is returned by eval, and \$@ is set to the error message. If there was no error, \$@ is guaranteed to be a null string.

Note that, because `eval` traps otherwise-fatal errors, it is useful for determining whether a particular feature (such as `socket` or `symlink`) is implemented. It is also Perl's exception trapping mechanism, where the `die` operator is used to raise exceptions.

If the code to be executed doesn't vary, you may use the `eval-BLOCK` form to trap run-time errors without incurring the penalty of recompiling each time. The error, if any, is still returned in `$@`.

#### **exists EXPR**

Given an expression that specifies a hash element or array element, returns true if the specified element in the hash or array has ever been initialized, even if the corresponding value is undefined. The element is not autovivified if it doesn't exist.

#### **glob EXPR**

##### **glob**

In list context, returns a (possibly empty) list of filename expansions on the value of `EXPR` such as the standard Unix shell `/bin/csh` would do. In scalar context, `glob` iterates through such filename expansions, returning undef when the list is exhausted. This is the internal function implementing the `<*.c>` operator, but you can use it directly. If `EXPR` is omitted, `$_` is used.

#### **goto LABEL**

##### **goto EXPR**

##### **goto &NAME**

The `goto-LABEL` form finds the statement labeled with `LABEL` and resumes execution there.

The `goto-EXPR` form expects a label name, whose scope will be resolved dynamically.

The `goto-&NAME` form is quite different from the other forms of `goto`. In fact, it isn't a `goto` in the normal sense at all, and doesn't have the stigma associated with other `gotos`. Instead, it exits the current subroutine (losing any changes set by `local()`) and immediately calls in its place the named subroutine using the current value of `@_`. This is used by `AUTOLOAD` subroutines that wish to load another subroutine and then pretend that the other subroutine had been called in the first place (except that any modifications to `@_` in the current subroutine are propagated to the other subroutine.) After the `goto`, not even caller will be able to tell that this routine was called first.

`NAME` needn't be the name of a subroutine; it can be a scalar variable containing a code reference, or a block which evaluates to a code reference.

#### **grep BLOCK LIST**

##### **grep EXPR,LIST**

This is similar in spirit to, but not the same as, `grep(1)` and its relatives. In particular, it is not limited to using regular expressions.

Evaluates the `BLOCK` or `EXPR` for each element of `LIST` (locally setting `$_` to each element) and returns the list value consisting of those elements for which the expression evaluated to true. In scalar context, returns the number of times the expression was true.

#### **keys HASH**

Returns a list consisting of all the keys of the named hash. (In scalar context, returns the number of keys.)

#### **last LABEL**

##### **last**

The `last` command is like the `break` statement in C (as used in loops); it immediately exits the loop in question. If the `LABEL` is omitted, the command refers to the innermost enclosing loop. The `continue` block, if any, is not executed:

Note that a block by itself is semantically identical to a loop that executes once. Thus `last` can be used to effect an early exit out of such a block.

#### **local EXPR**

A `local` modifies the listed variables to be local to the enclosing block, file, or `eval`. If more than one value is listed, the list must be placed in parentheses.

#### **lock THING**

This function places an advisory lock on a shared variable, or referenced object contained in `THING` until the lock goes out of scope.

#### **map BLOCK LIST**

##### **map EXPR,LIST**

Evaluates the `BLOCK` or `EXPR` for each element of `LIST` (locally setting `$_` to each element) and returns the list value composed of the results of each such evaluation. In scalar context, returns the total number of elements so generated. Evaluates `BLOCK` or `EXPR` in list context, so each element of `LIST` may produce zero, one, or more elements in the returned value.

#### **my EXPR**

##### **my TYPE EXPR**

A `my` declares the listed variables to be local (lexically) to the enclosing block, file, or `eval`. If more than one value is listed, the list must be placed in parentheses.

#### **next LABEL**

##### **next**

The `next` command is like the `continue` statement in C; it starts the next iteration of the loop:

`next` cannot be used to exit a block which returns a value such as `eval {}`, `sub {}` or `do {}`, and should not be used to exit a `grep()` or `map()` operation.

Note that a block by itself is semantically identical to a loop that executes once. Thus `next` will exit such a block early.

#### **our EXPR**

##### **our EXPR TYPE**

An `our` declares the listed variables to be valid globals within the enclosing block, file, or `eval`. That is, it has the same scoping rules as a "my" declaration, but does not create a local variable. If more than one value is listed, the list must be placed in parentheses. The `our` declaration has no semantic effect unless "use strict vars" is in effect, in which case it lets you use the declared global variable without qualifying it with a package name. (But only within the lexical scope of the `our` declaration. In this it differs from "use vars", which is package scoped.)

#### **pop ARRAY**

##### **pop**

Pops and returns the last value of the array, shortening the array by one element.

If there are no elements in the array, returns the undefined value (although this may happen at other times as well). If `ARRAY` is omitted, pops the `@ARGV` array in the main program, and the `@_` array in subroutines, just like `shift`.

#### **push ARRAY,LIST**

Treats `ARRAY` as a stack, and pushes the values of `LIST` onto the end of `ARRAY`. The length of `ARRAY` increases by the length of `LIST`. Returns the new number of elements in the array.

#### **redo LABEL**

##### **redo**

The `redo` command restarts the loop block without evaluating the conditional again. The `continue` block, if any, is not executed. If the `LABEL` is omitted, the command refers to the innermost enclosing loop.

`redo` cannot be used to retry a block which returns a value such as `eval {}`, `sub {}` or `do {}`, and should not be used to exit a `grep()` or `map()` operation.

Note that a block by itself is semantically identical to a loop that executes once. Thus `redo` inside such a block will effectively turn it into a looping construct.

ref EXPR

ref

Returns a non-empty string if EXPR is a reference, the empty string otherwise. If EXPR is not specified, \$\_ will be used. The value returned depends on the type of thing the reference is a reference to. Builtin types include:

SCALAR  
ARRAY  
HASH  
CODE  
REF  
GLOB  
LVALUE

If the referenced object has been blessed into a package, then that package name is returned instead.

reset EXPR

reset

Generally used in a continue block at the end of a loop to clear variables and reset ?? searches so that they work again. The expression is interpreted as a list of single characters (hyphens allowed for ranges). All variables and arrays beginning with one of those letters are reset to their pristine state. If the expression is omitted, one-match searches (?pattern?) are reset to match again. Resets only variables or searches in the current package. Always returns 1.

return EXPR

return

Returns from a subroutine, eval, or do FILE with the value given in EXPR. Evaluation of EXPR may be in list, scalar, or void context, depending on how the return value will be used, and the context may vary from one execution to the next. If no EXPR is given, returns an empty list in list context, the undefined value in scalar context, and nothing at all in a void context.

reverse LIST

In list context, returns a list value consisting of the elements of LIST in the opposite order. In scalar context, concatenates the elements of LIST and returns a string value with all characters in the opposite order.

Used without arguments in scalar context, reverse() reverses \$\_.

shift ARRAY

shift

Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down. If there are no elements in the array, returns the undefined value. If ARRAY is omitted, shifts the @\_ array within the lexical scope of subroutines and formats, and the @ARGV array at file scopes or within the lexical scopes established by the eval "", BEGIN {}, INIT {}, CHECK {}, and END {} constructs.

sort SUBNAME LIST

sort BLOCK LIST

sort LIST

In list context, this sorts the LIST and returns the sorted list value. In scalar context, the behaviour of sort() is undefined.

If SUBNAME or BLOCK is omitted, sorts in standard string comparison order. If SUBNAME is specified, it gives the name of a subroutine that returns an integer less than, equal to, or greater than 0, depending on how the elements of the list are to be ordered. (The <=> and cmp operators are extremely useful in such routines.) SUBNAME may be a scalar variable name (unsubscripted), in which case the value provides the name of (or a reference to) the actual subroutine to use. In place of a SUBNAME, you can provide a BLOCK as an anonymous, in-line sort subroutine.

If the subroutine's prototype is (\$\$), the elements to be compared are passed by reference in @\_, as for a normal subroutine. This is slower than unprototyped subroutines, where the elements to be compared are passed into the subroutine as the package global variables \$a and \$b (see example below). Note that in the latter case, it is usually counter-productive to declare \$a and \$b as lexicals.

In either case, the subroutine may not be recursive. The values to be compared are always passed by reference, so don't modify them.

splice ARRAY,OFFSET,LENGTH,LIST

splice ARRAY,OFFSET,LENGTH

splice ARRAY,OFFSET

splice ARRAY

Removes the elements designated by OFFSET and LENGTH from an array, and replaces them with the elements of LIST, if any. In list context, returns the elements removed from the array. In scalar context, returns the last element removed, or undef if no elements are removed. The array grows or shrinks as necessary. If OFFSET is negative then it starts that far from the end of the array. If LENGTH is omitted, removes everything from OFFSET onward. If LENGTH is negative, removes the elements from OFFSET onward except for -LENGTH elements at the end of the array. If both OFFSET and LENGTH are omitted, removes everything. If OFFSET is past the end of the array, perl issues a warning, and splices at the end of the array.

sub NAME BLOCK

sub NAME (PROTO) BLOCK

This is subroutine definition, not a real function per se. Without a BLOCK it's just a forward declaration. Without a NAME, it's an anonymous function declaration, and does actually return a value: the CODE ref of the closure you just created.

undef EXPR

undef

Undefines the value of EXPR, which must be an lvalue. Use only on a scalar value, an array (using @), a hash (using %), a subroutine (using &), or a typeglob (using \*). Always returns the undefined value. You can omit the EXPR, in which case nothing is undefined, but you still get an undefined value that you could, for instance, return from a subroutine, assign to a variable or pass as a parameter.

unshift ARRAY,LIST

Does the opposite of a shift. Or the opposite of a push, depending on how you look at it. Prepends list to the front of the array, and returns the new number of elements in the array.

values HASH

Returns a list consisting of all the values of the named hash. (In a scalar context, returns the number of values.)

wantarray

Returns true if the context of the currently executing subroutine or eval() block is looking for a list value. Returns false if the context is looking for a scalar. Returns the undefined value if the context is looking for no value (void context).

warn LIST

Produces a message on STDERR just like die, but doesn't exit or throw an exception.

If LIST is empty and \$@ already contains a value (typically from a previous eval) that value is used after appending "\t...caught" to \$@. This is useful for staying almost, but not entirely similar to die.

If \$@ is empty then the string "Warning: Something's wrong" is used.

## String Operations

chomp VARIABLE

chomp( LIST )

chomp

This safer version of "chop" removes any trailing string that corresponds to the current value of \$/. It returns the total number of characters removed from all its arguments. It's often used to remove the newline from the end of an input record when you're worried that the final record may be missing its newline. When in paragraph mode (\$/ = ""), it removes all trailing newlines from the string. When in slurp mode (\$/ = undef) or fixed-length record mode (\$/ is a reference to an integer or the like) chomp() won't remove anything. If VARIABLE is omitted, it chops \$.\_.

If VARIABLE is a hash, it chops the hash's values, but not its keys.

You can actually chomp anything that's an lvalue, including an assignment:

```
chomp($cwd = `pwd`);  
chomp($answer = <STDIN>);
```

If you chop a list, each element is chopped, and the total number of characters removed is returned.

chop VARIABLE

chop( LIST )

chop

Chops off the last character of a string and returns the character chopped. If VARIABLE is omitted, chops \$.\_. If VARIABLE is a hash, it chops the hash's values, but not its keys.

If you chop a list, each element is chopped. Only the value of the last chop is returned.

join EXPR,LIST

Joins the separate strings of LIST into a single string with fields separated by the value of EXPR, and returns that new string.

lc EXPR

lc

Returns a lowercased version of EXPR. This is the internal function implementing the \L escape in double-quoted strings.

If EXPR is omitted, uses \$.\_.

lcfirst EXPR

lcfirst

Returns the value of EXPR with the first character lowercased.

If EXPR is omitted, uses \$.\_.

length EXPR

length

Returns the length in characters of the value of EXPR. If EXPR is omitted, returns length of \$.\_. Note that this cannot be used on an entire array or hash to find out how many elements these have. For that, use scalar @array and scalar keys %hash respectively.

m//cgimosx

The match operator.

c Do not reset search position on a failed match when /g is in effect.

g Match globally, i.e., find all occurrences.

i Do case-insensitive pattern matching.

m Treat string as multiple lines.

o Compile pattern only once.

s Treat string as single line.

x Use extended regular expressions.

pos SCALAR

pos

Returns the offset of where the last m//g search left off for the variable in question (\$.\_ is used when the variable is not specified). May be modified to change that offset.

q/STRING/ same as 'STRING'

qq/STRING/ same as "STRING"

qr/STRING/ returns a regular expression matcher

qx/STRING/ same as `STRING`

qw/STRING/ list of 'STRING' separated by whitespace

Generalized quotes.

quotemeta EXPR

quotemeta

Returns the value of EXPR with all non-"word" characters backslashed.

If EXPR is omitted, uses \$.\_.

rindex STR,SUBSTR,POSITION

rindex STR,SUBSTR

Works just like index() except that it returns the position of the LAST occurrence of SUBSTR in STR. If POSITION is specified, returns the last occurrence at or before that position.

s/PATTERN/REPLACEMENT/egimosx

The substitution operator.

e Evaluate the right side as an expression.

g Replace globally, i.e., all occurrences.

i Do case-insensitive pattern matching.

m Treat string as multiple lines.

o Compile pattern only once.

s Treat string as single line.

x Use extended regular expressions.

scalar EXPR

Forces EXPR to be interpreted in scalar context and returns the value of EXPR.

There is no equivalent operator to force an expression to be interpolated in list context because in practice, this is never needed. If you really wanted to do so, however, you could use the construction @{{ (some expression) }}, but usually a simple (some expression) suffices.

split /PATTERN/,EXPR,LIMIT

split /PATTERN/,EXPR

split /PATTERN/

split

Splits the string EXPR into a list of strings and returns that list. By default, empty leading fields are preserved, and empty trailing ones are deleted. (If all fields are empty, they are considered to be trailing.)

In scalar context, returns the number of fields found and splits into the @\_ array. Use of split in scalar context is deprecated, however, because it clobbers your subroutine arguments.

If EXPR is omitted, splits the \$.\_ string. If PATTERN is also omitted, splits on whitespace (after skipping any leading whitespace). Anything matching PATTERN is taken to be a delimiter separating the fields. (Note that the delimiter may be longer than one character.)

If LIMIT is specified and positive, it represents the maximum number of fields the EXPR will be split into, though the actual number of fields returned depends on the number of times PATTERN matches within EXPR. If LIMIT is unspecified or zero, trailing null fields are stripped (which potential users of pop would do well to remember). If LIMIT is negative, it is treated as if an arbitrarily large LIMIT had been specified. Note that splitting an EXPR that evaluates to the empty string always returns the empty list, regardless of the LIMIT specified.

A pattern matching the null string (not to be confused with a null pattern //, which is just one member of the set of patterns matching a null string) will split the value of EXPR into separate characters at each point it matches that way. For example:

`substr EXPR,OFFSET,LENGTH,REPLACEMENT`

`substr EXPR,OFFSET,LENGTH`

`substr EXPR,OFFSET`

Extracts a substring out of EXPR and returns it. First character is at offset 0, or whatever you've set \$[ to (but don't do that). If OFFSET is negative (or more precisely, less than \$[]), starts that far from the end of the string. If LENGTH is omitted, returns everything to the end of the string. If LENGTH is negative, leaves that many characters off the end of the string.

You can use the substr() function as an lvalue, in which case EXPR must itself be an lvalue. If you assign something shorter than LENGTH, the string will shrink, and if you assign something longer than LENGTH, the string will grow to accommodate it.

If OFFSET and LENGTH specify a substring that is partly outside the string, only the part within the string is returned. If the substring is beyond either end of the string, substr() returns the undefined value and produces a warning. When used as an lvalue, specifying a substring that is entirely outside the string is a fatal error.

An alternative to using substr() as an lvalue is to specify the replacement string as the 4th argument. This allows you to replace parts of the EXPR and return what was there before in one operation, just as you can with splice().

If the lvalue returned by substr is used after the EXPR is changed in any way, the behaviour may not be as expected and is subject to change. This caveat includes code such as `print(substr($foo,$a,$b)=$bar)` or `(substr($foo,$a,$b)=$bar)=$fud` (where \$foo is changed via the substring assignment, and then the substr is used again), or where a substr() is aliased via a foreach loop or passed as a parameter or a reference to it is taken and then the alias, parameter, or deref'd reference either is used after the original EXPR has been changed or is assigned to and then used a second time.

`tr//cds`

The transliteration operator. Same as `y///`. Transliterates all occurrences of the characters found in the search list with the corresponding character in the replacement list. It returns the number of characters replaced or deleted. If no string is specified via the `~` or `!~` operator, the `$_` string is transliterated.

- `c` Complement the SEARCHLIST.
- `d` Delete found but unreplaced characters.
- `s` Squash duplicate replaced characters.

`uc EXPR`

`uc`

Returns an uppercased version of EXPR. This is the internal function implementing the `\U` escape in double-quoted strings.

If EXPR is omitted, uses `$_`.

`ucfirst EXPR`

`ucfirst`

Returns the value of EXPR with the first character in uppercase (titlecase in Unicode). This is the internal function implementing the `\u` escape in double-quoted strings.

If EXPR is omitted, uses `$_`.

`y///`

The transliteration operator. Same as `tr///`.

## Shell Commands

`chdir EXPR`

Changes the working directory to EXPR, if possible. If EXPR is omitted, changes to the directory specified by `$ENV{HOME}`, if set; if not, changes to the directory specified by `$ENV{LOGDIR}`. (Under VMS, the variable `$ENV{SYS$LOGIN}` is also checked, and used if it is set.) If neither is set, `chdir` does nothing. It returns true upon success, false otherwise. See the example under `die`.

`chmod LIST`

Changes the permissions of a list of files. The first element of the list must be the numerical mode, which should probably be an octal number

`chown LIST`

Changes the owner (and group) of a list of files. The first two elements of the list must be the *numeric* uid and gid, in that order. A value of -1 in either position is interpreted by most systems to leave that value unchanged. Returns the number of files successfully changed.

`index STR,SUBSTR,POSITION`

`index STR,SUBSTR`

The index function searches for one string within another, but without the wildcard-like behavior of a full regular-expression pattern match. It returns the position of the first occurrence of SUBSTR in STR at or after POSITION. If POSITION is omitted, starts searching from the beginning of the string. The return value is based at 0 (or whatever you've set the `$[` variable to--but don't do that). If the substring is not found, returns one less than the base, ordinarily -1.

`link OLDFILE,NEWFILE`

Creates a new filename linked to the old filename. Returns true for success, false otherwise.

`mkdir FILENAME,MASK`

`mkdir FILENAME`

Creates the directory specified by FILENAME, with permissions specified by MASK (as modified by `umask`). If it succeeds it returns true, otherwise it returns false and sets `$!` (errno). If omitted, MASK defaults to 0777.

`rename OLDNAME,NEWNAME`

Changes the name of a file; an existing file NEWNAME will be clobbered. Returns true for success, false otherwise.

`rmdir FILENAME`

`rmdir`

Deletes the directory specified by FILENAME if that directory is empty. If it succeeds it returns true, otherwise it returns false and sets `$!` (errno). If FILENAME is omitted, uses `$_`.

`sleep EXPR`

`sleep`

Causes the script to sleep for EXPR seconds, or forever if no EXPR. Returns the number of seconds actually slept.

`symlink OLDFILE,NEWFILE`

Creates a new filename symbolically linked to the old filename. Returns 1 for success, 0 otherwise.

`unlink LIST`

`unlink`

Deletes a list of files. Returns the number of files successfully deleted.

If LIST is omitted, uses `$_`.

## IO

**-X FILEHANDLE**

**-X EXPR**

**-X**

A file test, where X is one of the letters listed below. This unary operator takes one argument, either a filename or a filehandle, and tests the associated file to see if something is true about it. If the argument is omitted, tests \$\_, except for -t, which tests STDIN. Unless otherwise documented, it returns 1 for true and "" for false, or the undefined value if the file doesn't exist. Despite the funny names, precedence is the same as any other named unary operator, and the argument may be parenthesized like any other unary operator. The operator may be any of:

- 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.
  
- R File is readable by real uid/gid.
- W File is writable by real uid/gid.
- X File is executable by real uid/gid.
- O File is owned by real uid.
  
- e File exists.
- z File has zero size (is empty).
- 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.
- 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 Filehandle 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 text file (heuristic guess).
- B File is a "binary" file (opposite of -T).
  
- M Script start time minus file modification time, in days.
- A Same for access time.
- C Same for inode change time (Unix, may differ for other platforms)

**Example:**

```
while (<>) {
    chomp;
    next unless -f $_;      # ignore specials
    #...
}
```

If any of the file tests (or either the stat or lstat operators) are given the special filehandle consisting of a solitary underline, then the stat structure of the previous file test (or stat operator) is used, saving a system call. (This doesn't work with -t, and you need to remember that lstat() and -l will leave values in the stat structure for the symbolic link, not the real file.) (Also, if the stat buffer was filled by a lstat call, -T and -B will reset it with the results of stat \_). Example:

**close FILEHANDLE**

**close**

Closes the file or pipe associated with the file handle, returning true only if IO buffers are successfully flushed and closes the system file descriptor. Closes the currently selected filehandle if the argument is omitted.

FILEHANDLE may be an expression whose value can be used as an indirect filehandle, usually the real filehandle name.

**closedir DIRHANDLE**

Closes a directory opened by opendir and returns the success of that system call.

**eof FILEHANDLE**

**eof ()**

**eof**

Returns 1 if the next read on FILEHANDLE will return end of file, or if FILEHANDLE is not open. FILEHANDLE may be an expression whose value gives the real filehandle.

An eof without an argument uses the last file read. Using eof() with empty parentheses is very different. It refers to the pseudo file formed from the files listed on the command line and accessed via the <> operator. Since <> isn't explicitly opened, as a normal filehandle is, an eof() before <> has been used will cause @ARGV to be examined to determine if input is available. Similarly, an eof() after <> has returned end-of-file will assume you are processing another @ARGV list, and if you haven't set @ARGV, will read input from STDIN.

**getc FILEHANDLE**

**getc**

Returns the next character from the input file attached to FILEHANDLE, or the undefined value at end of file, or if there was an error (in the latter case \$! is set). If FILEHANDLE is omitted, reads from STDIN.

**open FILEHANDLE,EXPR**

**open FILEHANDLE,MODE,EXPR**

**open FILEHANDLE,MODE,EXPR,LIST**

**open FILEHANDLE,MODE,REFERENCE**

**open FILEHANDLE**

Opens the file whose filename is given by EXPR, and associates it with FILEHANDLE.

If FILEHANDLE is an undefined scalar variable (or array or hash element) the variable is assigned a reference to a new anonymous filehandle, otherwise if FILEHANDLE is an expression, its value is used as the name of the real filehandle wanted. (This is considered a symbolic reference, so use strict 'refs' should not be in effect.)

If EXPR is omitted, the scalar variable of the same name as the FILEHANDLE contains the filename. (Note that lexical variables--those declared with my--will not work for this purpose; so if you're using my, specify EXPR in your call to open.)

If three or more arguments are specified then the mode of opening and the file name are separate. If MODE is '<' or nothing, the file is opened for input. If MODE is '>', the file is truncated and opened for output, being created if necessary. If MODE is '>>', the file is opened for appending, again being created if necessary.

If the filename begins with '|', the filename is interpreted as a command to which output is to be piped, and if the filename ends with a '|', the filename is interpreted as a command which pipes output to us.

**opendir DIRHANDLE,EXPR**

Opens a directory named EXPR for processing by readdir, telldir, seekdir, rewinddir, and closedir. Returns true if successful. DIRHANDLE may be an expression whose value can be used as an indirect dirhandle, usually the real dirhandle name. If DIRHANDLE is an undefined scalar variable (or array or hash element), the variable is assigned a reference to a new anonymous dirhandle. DIRHANDLEs have their own namespace separate from FILEHANDLEs.

**print FILEHANDLE LIST**

**print LIST**

**print**

Prints a string or a list of strings. Returns true if successful. FILEHANDLE may be a scalar variable name, in which case the variable contains the name of or a reference to the filehandle, thus introducing one level of indirection. If FILEHANDLE is omitted, prints by default to standard output or to the last selected output channel. If LIST is also omitted, prints \$\_ to the currently selected output channel. The current value of \$, (if any) is printed between each LIST item. The current value of \$ (if any) is printed after the entire LIST has been printed.

**printf FILEHANDLE FORMAT, LIST**

**printf FORMAT, LIST**

Equivalent to print FILEHANDLE sprintf(FORMAT, LIST), except that \$\ (the output record separator) is not appended. The first argument of the list will be interpreted as the printf format.

**read FILEHANDLE, SCALAR, LENGTH, OFFSET**

**read FILEHANDLE, SCALAR, LENGTH**

Attempts to read LENGTH characters of data into variable SCALAR from the specified FILEHANDLE. Returns the number of characters actually read, 0 at end of file, or undef if there was an error (in the latter case \$! is also set). SCALAR will be grown or shrunk so that the last character actually read is the last character of the scalar after the read.

An OFFSET may be specified to place the read data at some place in the string other than the beginning. A negative OFFSET specifies placement at that many characters counting backwards from the end of the string. A positive OFFSET greater than the length of SCALAR results in the string being padded to the required size with "\0" bytes before the result of the read is appended.

**readline EXPR**

Reads from the filehandle whose typeglob is contained in EXPR. In scalar context, each call reads and returns the next line, until end-of-file is reached, whereupon the subsequent call returns undef. In list context, reads until end-of-file is reached and returns a list of lines. Note that the notion of "line" used here is however you may have defined it with \$/ or \$INPUT\_RECORD\_SEPARATOR).

**readlink EXPR**

**select FILEHANDLE**

**select**

Returns the currently selected filehandle. Sets the current default filehandle for output, if FILEHANDLE is supplied. This has two effects: first, a write or a print without a filehandle will default to this FILEHANDLE. Second, references to variables related to output will refer to this output channel.

**sprintf FORMAT, LIST**

Returns a string formatted by the usual printf conventions of the C library function sprintf.

**write FILEHANDLE**

**write EXPR**

**write**

Writes a formatted record (possibly multi-line) to the specified FILEHANDLE, using the format associated with that file. By default the format for a file is the one having the same name as the filehandle, but the format for the current output channel (see the select function) may be set explicitly by assigning the name of the format to the \$\_ variable.

If FILEHANDLE is unspecified, output goes to the current default output channel, which starts out as STDOUT but may be changed by the select operator. If the FILEHANDLE is an EXPR, then the expression is evaluated and the resulting string is used to look up the name of the FILEHANDLE at run time.

## System calls

**exec PROGRAM LIST**

The exec function executes a system command and never returns-- use system instead of exec if you want it to return. It fails and returns false only if the command does not exist and it is executed directly instead of via your system's command shell (see below).

Since it's a common mistake to use exec instead of system, Perl warns you if there is a following statement which isn't die, warn, or exit (if -w is set - but you always do that). If you really want to follow an exec with some other statement, you can use one of these styles to avoid the warning:

**exit EXPR**

Evaluates EXPR and exits immediately with that value.

**rewinddir DIRHANDLE**

Sets the current position to the beginning of the directory for the readdir routine on DIRHANDLE.

**seek FILEHANDLE, POSITION, WHENCE**

Sets FILEHANDLE's position, just like the fseek call of stdio. FILEHANDLE may be an expression whose value gives the name of the filehandle. The values for WHENCE are 0 to set the new position in bytes to POSITION, 1 to set it to the current position plus POSITION, and 2 to set it to EOF plus POSITION (typically negative). Returns 1 upon success, 0 otherwise.

**seekdir DIRHANDLE, POS**

Sets the current position for the readdir routine on DIRHANDLE. POS must be a value returned by telldir. Has the same caveats about possible directory compaction as the corresponding system library routine.

**system LIST**

**system PROGRAM LIST**

Does exactly the same thing as exec LIST, except that a fork is done first, and the parent process waits for the child process to complete. Note that argument processing varies depending on the number of arguments. If there is more than one argument in LIST, or if LIST is an array with more than one value, starts the program given by the first element of the list with arguments given by the rest of the list. If there is only one scalar argument, the argument is checked for shell metacharacters, and if there are any, the entire argument is passed to the system's command shell for parsing (this is /bin/sh -c on Unix platforms, but varies on other platforms). If there are no shell metacharacters in the argument, it is split into words and passed directly to execvp, which is more efficient.

**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, assumes the file last read.

**telldir DIRHANDLE**

Returns the current position of the readdir routines on DIRHANDLE. Value may be given to seekdir to access a particular location in a directory.

**time**

Returns the number of non-leap seconds since whatever time the system considers to be the epoch, suitable for feeding to gmtime and localtime.

**times**

Returns a four-element list giving the user and system times, in seconds, for this process and the children of this process.

(\$user, \$system, \$cuser, \$csystem) = times;

In scalar context, times returns \$user.

**truncate FILEHANDLE,LENGTH**

**truncate EXPR,LENGTH**

Truncates the file opened on FILEHANDLE, or named by EXPR, to the specified length. Produces a fatal error if truncate isn't implemented on your system. Returns true if successful, the undefined value otherwise.

**readdir DIRHANDLE**

Returns the next directory entry for a directory opened by opendir. If used in list context, returns all the rest of the entries in the directory. If there are no more entries, returns an undefined value in scalar context or a null list in list context.

**readlink**

Returns the value of a symbolic link. If there is some system error, returns the undefined value and sets \$! (errno). If EXPR is omitted, uses \$\_.

## **Math**

**abs**

**atan2 Y,X**

**cos**

**exp**

**int**

**log**

**rand EXPR**

**rand**

Returns a random fractional number greater than or equal to 0 and less than the value of EXPR. (EXPR should be positive.) If EXPR is omitted, the value 1 is used.

**sin**

**sqrt**

**srand**