

---

## *Brief Summary of Perl*

Perl acts like an interpreter; you type the source into a file and then tell the Perl program (usually called perl or perl.exe) to execute it. In actuality, Perl will compile the program before it runs it, but the user won't really notice this; the only difference from a true interpreter is that syntax errors anywhere in the program will cause an error before execution starts.

Perl treats all white space, including new lines, as the same. Blocks of code are enclosed between { and }, and statements end with a semi-colon (;).

Comments are marked with a # symbol; anything following that on a line is ignored.

Variables names start with \$. They do not need to be declared; they can simply be used. Variables that have not been assigned a value will evaluate to the special reserved value undef.

Perl stores all numbers as floating point; numbers and strings together are known as "scalars". Perl distinguishes between strings and numbers in literals, so you can assign one or the other to a scalar variable:

---

```
$x = "Hello";
```

or

---

```
$y = 5;
```

However, Perl will automatically convert between strings and numbers as needed, depending on which operator is being used. For example, the operator + is defined to be numeric addition of two scalars, so the statement

---

```
$x = 5 + "7";
```

will set \$x to 12; the string "7" is automatically converted to the number 7. Similarly, the operator . concatenates two strings together, so the statement:

---

```
$x = 5 . "7";
```

sets \$x to "57".

In practice, this means that you can almost always treat a number and its string representation as the same. When converting a string to a number, only the characters up to the first non-numeric one are evaluated, so "123ABC" evaluates to 123. A variable that is undef will do the "right thing" when used in an expression, evaluating to the number 0 or an empty string as appropriate.

Assignment is done with the = sign, as seen above, and you can use an assignment statement anywhere you would use a variable: the assignment is done first, then the new value of the variable is used:

---

```
$x = ($y = 4);    # $x will be 4 also
```

Perl uses +, -, \*, and / for the basic mathematical operations. % is modulo (numbers are truncated to integers first) and \*\* is the exponentiation operator, for example 2\*\*16. ++ and -- work as they do in C/C++ and Java. Perl supports binary assignment operators such as +=, -=, %=, and even .= and \*\*=:

---

```
$x += 5;  
$mystring .= "(s)";
```

---

Strings can be quoted with either single or double quotes. Within single quotes, the only special characters in a string are the backslash and the single quote; to put an actual backslash in a single-quoted string you use \\, and to put a single quote you use \', as in these examples:

---

```
$directory = 'windows\\system32';  
$answer = 'I don\'t agree';
```

---

Within double quotes, you can specify backslash escapes such as \n and \t, as well as more advanced one such as \L and \E which delineate an area in which all letters should be lower-cased (we won't use those fancy ones in programs in this book). Perl also does variable interpolation within double-quoted strings, meaning variables are replaced with their values:

---

```
$prompt = "Enter your name\n";  
$name = "Sally";  
$reply = "Hello, $name!";
```

---

The length() function returns the length of a string; substr() is used to return a part of a string, and index() to find the index of a match within a string. String positions are 0-based, so the first character is at position 0. You can use negative numbers to count backwards from the end, with index -1 the last character in a string. Note that with these built-in functions, as with most functions in Perl, the parenthesis around the arguments are optional, as long as the fact that it is a function call is unambiguous (nonetheless I will continue to use () as a way to indicate functions):

---

```
$j = length $somestring;  
$secondfivechars = substr ($x, 5, 5);  
$lastchar = substr $inputline, -1;  
$firstspace = index($text, " ");
```

---

If the third parameter to `substr()` is not provided, then the rest of the string is returned. `index()` can also take a third parameter which is the offset within the string to start searching.

You can modify a string by assigning to a substring, even if the new string is not the same length as the substring:

---

```
substr($currency, index($currency, "$"), 1) = "<dollar>";
```

The `x` operator repeats a string a specified number of times, so you could add three exclamation marks to the end of a string with:

```
$phrase .= ('!' x 3);
```

Perl uses the term `list` to describe an ordered collection of scalars. An array is a variable that contains a list, so the terms "array" and "list" are often thought of as being the same.

A list is specified using comma-separated scalars within parenthesis, which can then be assigned to an array variable:

---

```
@mylist = (1, 2, 3);
```

An entire list is referenced by preceding the name with `@`. If a list is included in another list, the elements in the list are included, not the list itself, so you don't get the "list member that is itself a list":

---

```
@listA = ('A', 'B', 'C');  
@listB = (@listA, 'D', 'E');
```

---

`@listB` will now be a list with five elements: ('A', 'B', 'C', 'D', 'E').

The range operator can be used as a shortcut for a list that is a sequence of numbers:

---

```
@numbers1to5 = (1..5);
```

As a shortcut for lists of strings, you can use `qw` (the letter `q` followed by the letter `w`):

---

```
@stringlist = qw/ january february march /;
```

Access to elements in an array uses 0-based indexing, and supports negative numbers to indicate counting back from the end. When indexing into an array, the list name is preceded with `$`, not `@`, except for certain circumstances which we won't get into:

---

```
$firstel = $mylist[0];  
$lastel = $mylist[-1];
```

---

The index of the last element in an array can be retrieved with  `$#arrayname`; this is one less than the size, since arrays are 0-based. You can assign to this value to chop the end off an array:

---

```
$array[$#arrayname+1] = $value    # extend array by one
$#arrayname = 3;                # drop all elements after the fourth one
```

---

Consistent with being one less than the size of the array,  `$#arrayname` will be `-1` for an array that has an empty list.

You can assign to an element of an array past the current size; the array is extended and any intervening elements will return the value `undef` if accessed (as will any elements past the new end of the array):

---

```
$array[0] = "A";
$array[1] = "B";
$array[25] = "Z"; # array[2] through array[24] are undef
$k = $array[26]; # this is undef also
```

---

You can also assign to a list containing variable names, which lines up the values as you would expect; including `undef` in the list means no assignment is done:

---

```
@numarray = (1..5);
($a, undef, undef, $b, undef) = @numarray;
```

---

sets `$a` to 1 and `$b` to 4.

You can "slice" a list, as it is known, by specifying a list of indices into a list or array to produce a smaller list or array. For example:

---

```
($a, $b) = @arr[2, 3]; # $a = $arr[2], $b = $arr[3]
print @arr[0..4];     # print first five elements
```

---

Remember that negative indices count back from the end of the list, so `@arr[0..-1]` is the whole array.

The `shift()` function takes an element off the beginning of a list (index 0) and `pop()` takes an element off the end (index `-1`). `unshift()` and `push()` place an element or list of elements back on the list at the beginning or end:

---

```
$next = shift @mylist;
push ($newelement, @mylist);
unshift (1, 2, 3), @numberlist;
```

---

The `splice()` function takes a set of elements in an array and replaces them with new elements, or removes them if new elements are not provided:

---

```
splice ($listA, -1, 1, "last");    # replace last element
splice ($listB, 1, 2); # remove second and third elements
```

---

The `split()` function splits a string into arrays using a separator defined by a regular expression, and `join()` converts an array back to a string using a separator specified by a string (the "regular expression vs. string" difference is why the first parameters to the two functions look different):

---

```
@fields = split /;/, $inputline;
$outputline = join "|", @fields;
```

---

When a program is invoked, the list `@ARGV` contains the command-line parameters that were passed to it:

---

```
$firstarg = shift @ARGV;
```

Unlike in the `argv[]` array in C, which stores the name of the program in `argv[0]` and the first argument in `argv[1]`, the first element in `@ARGV` is the first argument to the program; the name of the program is stored in the variable `$0` (that's the number 0, not the letter O)..

A Perl hash is similar to a list, except it is indexed using strings known as keys. The entries are also unordered. To access an element of a hash, the key is surrounded by curly braces:

---

```
$iphash{"router"} = "192.0.0.1";
```

There can be only one value for a given key; it is replaced if a new value is assigned.

The entire contents of a hash are referred to by preceding the name with `%`. The functions `keys()` and `values()` return lists of the keys and values of a hash:

---

```
$machinenames = keys %iphash;
$ipaddrs = values %iphash;
```

---

Although the hash is unordered, the elements in the lists returned by `keys()` and `values()` will line up as long as the hash is not modified in between.

The `exists()` function will check for the existence of an element with a given key, and `delete()` removes an element given its key (it does nothing, silently, if no element exists for that key).

---

```
if (exists $hashA{"keyname"}) {
```

The `each()` function can be used to iterate through the key-value pairs in a hash:

```
while ( ($key, $value) = each %somehash) {
```

We'll cover `if` statements and `while` loops next; suffice it to say that this works as expected, and will cause the loop to exit once every element of the hash has been iterated through. This example also shows how you can assign the two-element list returned by `each()` to a two-element list containing two variables.

Conditionals can be tested with the `if` statement, which is followed by a block of code inside curly braces. Curly braces are required even if the block only has one line of code:

---

```
if ($i == 5) {  
    $i = 0;  
}
```

---

A scalar that is equal to `undef` will evaluate false, as will an empty string and the number 0. To preserve the rule that a number and its string representation can be treated as equivalent, the string `'0'` will also evaluate to false. Everything else evaluates to true.

If a string has a number in it, Perl does not know whether to compare it as a number or a string. Therefore, there are two complete sets of comparison operators. Numeric comparisons are done using `==`, `!=`, `<`, `>`, `<=`, and `>=`; string comparisons are done using `eq`, `ne`, `lt`, `gt`, `le`, and `ge`. Thus, with the following assignments:

---

```
$a = "5";  
$b = "10";
```

---

`($a < $b)` will be true, but `($a lt $b)` will be false.

Perl uses `||` and `&&` for logical or and logical and. It guarantees that in the expression

---

```
if ((expr1) || (expr2)) {
```

`expr2` will only be evaluated if `expr1` is false (and similarly, in the case of `(expr1 && expr2)`, `expr2` will only be evaluated if `expr1` is true).

Perl also supports the words `or` and `and`. The difference is that `or` and `and` have lower precedence than `||` and `&&`; in particular, the `=` assignment operator has higher precedence than `or` and `and` but lower precedence than `||` and `&&`, so a test such as

---

```
($j = myfunc() || $x)
```

won't do what you are probably expecting, but

---

```
($j = myfunc() or $x)
```

will.

Perl supports `else` and `elsif` (note the spelling) blocks after `if` statements:

---

```
if ($command = "sort") {
    do_sort();
} elsif ($command = "print") {
    do_print();
} else {
    invalid_command();
}
```

---

Perl also supports `unless`, which is like `if` except that the sense is reversed – the `unless` block will execute if the condition is false:

---

```
unless (defined($name)) {
    $name = "default";
}
```

---

(`defined()` is a built-in function that returns false if the argument is `undef`). An `unless` statement can have `elsif` and `else` clauses, but the meaning is not reversed for those:

---

```
unless ($age < 21) {
    print ("can drive and vote\n");
} elsif ($age > 16) {
    print ("can drive but not vote\n");
} else {
    print ("cannot drive or vote\n");
}
```

---

Perl has several ways to loop. The `while` loop works as it does in many other languages:

---

```
while ($k < 100) {  
    $k = $k + 1;  
}
```

---

There are also `until` loops, which execute as long as their test is false (`while` and `until` are related the same way as `if` and `unless`), and also `do/while` and `do/until` loops.

Perl has `for` loops that look the same as C and Java:

---

```
for ($j = 0; $j < 10; $j++) {  
    print $j;  
}
```

---

It also has `foreach` loops which loop through a list:

---

```
foreach $counter (0..9) {  
    print $counter;  
}  
  
foreach @mylist {  
    print $_;  
}
```

---

The second example shows the Perl default variable `$_`. If a `foreach` loop does not specify the name of its loop control variable, then the control variable is stored in a variable named `$_`. Perl uses the `$_` default in other places too. For example, the `print()` function by default takes `$_` as its parameter, thus the body of the second loop could simply have been `print;`.

Perl allows `if`, `unless`, `while`, `until`, and `foreach` to be written as "modifiers" to expressions, which can be easier to read in some cases:

---

```
$x += 1 unless $x > 100;  
print $_ foreach (1..10);
```

---

This is just a reordering of the traditional way. In particular, the conditional is still evaluated before the code is executed, even though it is to the right of it. With `foreach` written as a modifier, the control variable can't be named; it is always `$_`.

Inside a loop, the `last` statement exits the loop (similar to `break` in some other languages), the `next` statement moves to the next iteration of the loop (similar to `continue` in some other languages), and the `redo` statement restarts the current iteration without checking the exit condition or changing the control variable.

An important concept in perl is scalar context vs. list context. This refers to where an expression is used. For example, when assigning to a scalar, the right-hand side of the assignment statement is in scalar context. When assigning to a list, the right-hand side of the assignment statement is in list context. The conditional expression of a `while()` statement is in scalar context; the expression controlling a `foreach()` loop is in list context.

This matters because certain expressions, such as the name of an array, produce different values in list context vs. scalar context. In scalar context the name of an array returns the number of values; in list context it returns the whole array. Thus you can say

---

```
$arraysize = @myarray;    # scalar context
```

and also

---

```
@arraycopy = @myarray;    # list context
```

Scalar vs. list context also matters when dealing with file handles. The most commonly-used file handle is `STDIN`, which is the standard input to the Perl program. File handles are accessed by enclosing the handle between `<` and `>` and assigning the result to a variable. In scalar context a file handle returns the next line of a file, or `undef` when end-of-file is reached; in scalar context it returns every line of the file. Thus, you can loop through a file either with:

---

```
while (<STDIN>) {    # scalar context
    process($_);
}
```

---

or:

---

```
foreach (<STDIN>) {    # list context
    process($_);
}
```

---

But in the first case only one line of the file is read into memory at a time, while in the second the entire file is read into a list, which is then stepped through.

When Perl reads a line out of a file, it includes the newline character (`'\n'`) at the end. Since it is common to want to remove this, Perl provides a built-in function `chomp()` whose only function is to remove the last character from a string if it is `\n`.

Files can be opened by name using the `open()` function, which also takes a handle as a parameter. The handle can then be used the same as `STDIN`, and then closed.

---

```
open FH, "logfile";
```

```
while (<FH>) {  
    chomp($_);  
    process($_);  
}  
close FH;
```

---

The name of a file handle can also be stored in a variable.

Of special note is the diamond operator, so called because of its appearance: <>. The diamond operator is used for programs that specify a list of files as command-line parameters. It is a magic file handle that reads in turn from each file specified on the command line, or from standard input if no files were specified. The diamond operator uses the @ARGV array to see what files to use, so you can tweak @ARGV as you like before invoking the diamond operator:

---

```
$matchstring = shift @ARGV;  
while (<>) {  
    lookformatches($matchstring, $_);  
}
```

---

The diamond operator follows the Unix convention that a filename that is a single hyphen refers to the standard input stream.

Perl has a series of tests which can be performed on a file, which are all specified by a hyphen and a letter followed by a file name or file handle. Some of these tests return true or false, but others return numeric values. Among these are -f which is true if the filename refers to a plain file, -d which returns true if the filename refers to a directory, and -s which return the size of a file in bytes:

---

```
(if -f $file) {  
    print ("$file is a plan file\n");  
    $size = -s $file;  
    print ("Size is $size bytes\n");  
}
```

---

For even more information, the `stat()` function returns a 13-element list of information about the file.

Directories can be read in Perl much like files, but using different functions: `opendir()`, `readdir()`, and `closedir()`. Each line read from a directory is the name of a file.

Perl includes built-in regular expression matching. The simplest form compares a regular expression to the value of `$_`:

---

```
while (<>) {
```

```
    if (/hello/) {
        ++$hellolinecount;
    }
}
```

---

In addition to matching literal strings, the regular expressions can include the following:

- . matches any character except newline
- \ escapes the next character (so \. matches only a period)
- () can be used to group parts of a regular expression
- \* means to match the previous item zero or more times
- + means to match the previous item one or more times
- ? makes an item optional; it can appear zero or one time
- {n} means to match the previous item n times
- {n,m} means to match the previous item between n and m times
- | between two items means to match either one
- [abcd] matches any of the characters listed
- [a-z] matches any character in a range
- [^abcd] matches any character *except* the ones listed
- \d matches any digit, same as [0-9]
- \w matches a word character, same as [a-zA-Z0-0\_]
- \s matches whitespace, same as [\f\t\n\r ]
- \D, \W, \S match any character *except* their lowercase equivalent
- ^ matches the beginning of the string
- \$ matches the end of the string

Thus, you can get quite sophisticated with your matching:

---

```
if ($phone =~ /\d{3}\-\d{4}/) {
    print ("$phone looks like a US phone number\n");
}
if ($number =~ /[0-9a-fA-F]+)/ {
    print ("$number is a valid hexadecimal number\n");
}
if ($inputline =~ /^#/) {
    print("comment line, ignored");
}
}
```

---

Beyond grouping, parenthesis (()) around a part of the match string tells Perl to remember what part of the string matched that part of the regular expression. The escape \1 can be used later in the match string to refer back to the first grouped match. So the match string /(.)\1/ will match any character repeated twice in a row. Furthermore, the part of the string that matched will be put in a special variable \$1. The same goes for \2 and \$2, \3 and \$3, etc:

---

```
if ($word =~ /([aeiou])\1/) {
    print("$word has a repeated vowel: $1\n");
}
}
```

---

When a match is complete, the part of the string that matched the regular expression is stored in `$&`, the part before is stored in `$``, and the part after is stored in `$'`:

---

```
if ($text =~ /[\\w\\.]+\\. (com|org|net)/) {  
    print ("`<a href=\""$&\"">$&</a>$'");  
}
```

---

Finally, the `/i` modifier after the match string makes the matching case-insensitive. Perl regular expressions allow even more escapes and modifiers, but we won't use them here.

Perl user-defined functions (called subroutines) are declared using `sub`. The parameters to the subroutine are passed in the `@_` array:

---

```
sub addtwo {  
    return $_[0] + $_[1];  
}
```

---

The `return` statement is actually optional; if it is missing, then the subroutine will return the value of the last expression calculated, or `undef` if no expressions were calculated.

Variables local to a function can be declared with the `my` operator, so the previous function could be written:

---

```
sub addtwo {  
    my ($a, $b);    # declare them local  
    ($a, $b) = @_;  # list assignment  
    return $a + $b;  
}
```

---

You don't have to put `my` on a separate line, it can be applied the first time the variables are used:

---

```
my ($a, $b) = @_;
```

---

There is also a `local` operator. This is an older Perl operator that works sort of like `my`, except rather than create a truly local variable for the subroutine, it will reuse a global variable (if one exists with the same name) but save away the current value of the global until the subroutine is done. If that wasn't clear, it really only matters if the subroutine calls another subroutine that accesses the global variable by name. And, for reasons which are best left to Perl wizards to explain, you can't use `my` on a file handle, you have to use `local`.

Printing in Perl is done with the `print()` function, which we have seen in previous examples. More sophisticated printing can be done with `printf()`, which takes formatting similar to the C `printf()`, and its relative `sprintf()`, which returns the formatted string rather than printing it:

---

```
printf "The date is %2d/%2d/%4d\n", $day, $month, $year;
$time = sprintf "%2d:%2d:%2d", $hour, $minute, $second;
```

---

Perl has a built-in `sort()` function. By default it sorts a list in ASCII order:

---

```
@sortedlist = sort @unsortedlist;
```

However, you can also specify a subroutine to use for comparisons. This can be a named subroutine, or it can be specified inline as a parameter to `sort`. The semantics of the function are that it always has two arguments, `$a` and `$b`, and it should return `-1`, `0`, or `1`, respectively, if `$a` should be sorted before, at the same place, or after `$b`. For convenience, Perl defines two built-in operators which have these same semantics: `<=>` (known as the "spaceship" operator) does numeric comparisons, and `cmp` does string comparisons:

---

```
@sortedbylengthlist =
    sort { length($a) <=> length($b) } @unsortedlist;
@sortedbysecondcol =
    sort { substr($a,1) cmp substr($b,1) } @unsortedlist;
```

---

That's all we're going to use of Perl. The language is more sophisticated than that; among other things it supports object-oriented programming, with user-defined classes, inheritance, overloading, and the works. Perl has also been extended in every direction by the writing of modules, which are add-in libraries of functionality. There are also more built-in functions, pragmas, bitstring support, list-processing operators, references, networking, etc, etc...all of which we will leave undocumented here.