Perl Programming, Part 1

Andrew Arensburger

February 13, 2001

Contents

Part I

1 Introduction

Perl has been called "the Swiss army chainsaw of scripting languages," and rightfully so. Anything you want to do, you can do with Perl. If it's not already in there, you can add it.

Perl has a reputation for being needlessly cryptic, with a byzantine syntax. To a great extent, this is true. Fortunately, you don't need to know all—or even a lot—of Perl in order to write Perl programs.[1] So if you doze off in the middle of this talk, that's okay. Just lean your head forward so that you don't snore.

Secondly, a lot of features are shortcuts were added in order to make certain types of things easier to do. From my own experience, I can say that fully 90% of all Perl scripts that I write are one-shots: most of them I just enter on the command line, the rest I put in `/tmp`, run once, and then forget about. When you're writing a short throwaway, you really get to appreciate all the shortcuts that Perl has to offer.

Perl goes to great lengths to make things do what you expect them to. My job is simply to tell you what it is that you expect.

A note about versions: Perl version 5 introduced a lot of new features, and generally made Perl even more powerful than it was before. Since version 5 has been out for years, I will be talking primarily about version 5. Perl 4 scripts will mostly run okay under Perl 5, but not vice-versa. I might occasionally remember to mention that such-and-such feature doesn't exist under Perl 4, but if you try something and it doesn't work, make sure that you're using Perl 5.

2 Variables

Perl, like any other programming language, has variables. There are three basic types of variables: *scalars*, *arrays* and *hashes*. In addition, Perl supports *references*, which allow one to do some rather interesting things.

Variables need not be declared in advance; if you refer to a new variable, Perl will create one on the spot; the way variables are used contains enough information for it to figure out what kind of variable you're talking about. Bear in mind, this also means that this can also cause hard-to-detect problems: if you make a typo, Perl will happily use the misspelled variable name and not tell you about it.[2]

Perl is case-sensitive: `$foo`, `$Foo` and `$FOO` are all different.

Variable names may consist of any sequence of letters, numbers and undeer-scores, but must begin with a letter or an underscore. Actually, there are a few

[1] It is a topic of some discussion as to whether Perl programs should be called "scripts" or "programs." I don't really care. I'll use the two terms interchangeably.

[2] Actually, the `-w` command-line option will warn you about identifiers that have only been used once, and may therefore be typos. But woe betide you, should you make the same typo twice.

more exotic characters that you can use, but most of them are already taken.

2.1 Scalars

A scalar is the simplest type of variable. It holds a single value, which can be a string or a number. Internally, Perl stores scalars as strings, which means that it is not terribly well-suited for number-crunching, since it has to keep converting strings to numbers and back all the time.

To assign a value to a variable, use

```
$var = value
```

and to refer to the value, use

```
$var
```

You can assign several variables in parallel, e.g.,

```
($a, $b, $c) = (1, 2, 3);
($a, $b, $c) = ($c, $b, $a);
```

The second line illustrates that values on the right are determined before the assignments happen on the left, so you can use this to swap two variables without a temporary.

You won't use parallel assignment in this way very often, though: usually, you'll use it to assign an array into several variables, or vice-versa.

2.2 Arrays

Arrays are ordered lists of values. The name "array" is traditional, although they're more akin to linked lists or deques. Use an array any time you need a list of things.

```
@array = @otherarray;
@mixed = ( "arensb", 2072, 10 );
@empty = ();
```

Array indices go from 0 to $n-1$, just as in C. Unlike C, however, you do not need to declare the size of an array in advance: if you assign to an array index beyond the end of the array, the array will be resized to accomodate this (any values in between will be assigned the undefined value).

Square brackets indicate subscripting:

```
$foo      = $array[9];
$array[3] = $bar;
@other    = @array[2,0,1];
@baz      = @array[4..8];
($a, $b, @rest) = @array;
```

Note that in the last three examples, I used `@array` instead of `$array`. The `@` indicates that the expression should return an array, rather than a scalar. The square brackets tell Perl that `array` is an array. This is because `$foo` and `@foo` are two different variables: the first is a scalar, the second is an array. This is perfectly legal.

`@array[2,0,1]` returns an array consisting of elements 2, 0 and 1 of `@array`, in that order, and `@array[4..8]` returns an array consisting of elements 4 through 8 inclusive of `@array`.

You can subscript anything that has an array value, *e.g.*, literal arrays:

```
$foo = ("a", "b", "c")[1];
```

though this is more useful with functions returning arrays.

There are two ways of determining the size of an array:

```
$size = $#array + 1;
$size = @array;
$size = scalar(@array);
```

The `$#`*name* construct returns the index of the last element in the array. Since indices begin at 0, this returns the size of the array, minus 1. The last two lines are equivalent, but the third line makes the scalar context explicit (we'll cover contexts in a little bit).

You can also assign to `$#array`. This has the effect of growing or shrinking the array, as necessary. Note that if you shrink an array, any values past the new end are lost forever: you cannot recover them by growing the array again.

I mentioned that arrays in Perl are more akin to deques than to C arrays. The four deque-related functions are:

`push` *array*`,` *value...* Appends *value* to the end of *array*.

`pop` *array* Pops the last value off of the end of *array* (shortening it by one), and returns it.

`shift` *array* Removes the first element from the beginning of *array* (shortening it by one) and returns it.

`unshift` *array*`,` *value...* Prepends *value* to the beginning of *array*.

If you omit *array*, `pop` and `shift` will use `@ARGV`, the list of command-line arguments (or `@_`, inside of a function).

2.3 Hashes

The third basic type in Perl is the *hash table*, also known as an *associative array* or *hash*. Like arrays, a hash is a collection of values. However, whereas array indices are integers, hash indices can be any string. You can think of arrays as being lists, and hashes as being look-up tables.

```
%user2group = (
        "arensb" => 10,
        "root"   => 0,
        "bin"    => 3,
);
$user2group{"arnie"} = 199;
%empty = ()
```

The token => is equivalent to a comma, and is intended as syntactic sugar when you're defining hashes.

Extracting values from hashes works much the same way as for arrays, except that you use curly braces instead of square brackets for the subscripts:

```
$group  = $user2group{"arensb"};
@groups = @user2group{"root", "bin"};
```

Again, note that `$` indicates that the expression should return a scalar value, `@` indicates that it should return an array value, and the curly braces tell Perl that `user2group` is a hash.

If you need to get all of the values in a hash, you can use the `keys` and `values` functions. `keys %hash` returns an array of all of the keys (indices) in `%hash`, in no particular order, and `values %hash` returns a list of all of the values.

Often, though, you don't care what the keys are, you just want to do something to all of the elements in a hash. The `each` function returns an array of two elements: a key and a value. Each time you call `each %hash`, it returns the next key-value pair in `%hash`.

To find out whether a hash contains a given value, you can use the `defined` function:

```
if (defined($user2group{"arnie"}))
...
```

To remove a hash entry, use `delete`:

```
delete($user2group{"arnie"});
```

By the way, there's nothing that says that the value part of a hash entry has to represent anything. One common trick is to use a hash as an unordered set:

```
%isastaffer = (
        arensb  => 1,
        root    => 1,
        bin     => 1,
);

...
if ($isastaffer{$user}) {
        ...
}
```

2.4 Barewords

A word by itself that has no other interpretation is known as a "bareword," and is treated as if it were a double-quoted string. Avoid barewords, since they only lead to trouble.

2.5 Numbers

This is a brief section, since numbers in Perl work pretty much the way you'd expect them to. The following are all legal numbers:

```
  123
 0456           # Octal
0x8a0           # Hex
0.998
 .998
9.98e-1
1_234_567_890
```

The underscores in the last example are just for legibility: `1_234` is equivalent to `1234`.

2.6 Context

Every expression is evaluated in a particular *context*. The two major ones are the *scalar* and *array* contexts. Intuitively enough, if an expression is evaluated in a scalar context, it yields a scalar, and if it is evaluated in an array context, it yields an array.

We've already seen an example of scalar vs. array context:

```
@array = ("a", "b", "c");
 $foo  = @array;    # Set $foo to size of @array
($bar) = @array;    # Set $bar to first element of @array
```

You can use `scalar(`*expression*`)` to force *expression* to be evaluated in a scalar context. There is no equivalent function to force an array context, though you can use parentheses to good effect.

Some functions return different values depending on the context in which they're evaluated. The `localtime` function, for instance, returns either the time and date in human-readable form, or an array giving the year, month, day, hours, minutes and seconds of the current time:

```
$time = localtime;      # Returns "Sat Jun 13 14:19:35 1998"
@time = localtime;      # Returns (35 19 14 13 5 98 6 163 1)
```

If you want to write functions that behave this way, you'll want to use the `wantarray` function, which returns true if the function is being evaluated in an array context.

2.7 Special Variables

Perl has a whole slew of special variables, most of which have funny names like `$_`, which contributes to Perl's reputation for looking like line noise.

If this bothers you, you can put

```
use English;
```

at the top of your program. This will allow you to use either the "traditional" names, or English (or *awk*) equivalents.

For the most part, special variables will be listed next to the section to which they are pertinent. A few don't fall into convenient categories, however, so they're listed here (along with their alternate names):

`$ARG`
`$_` The root of all being. `$_` is used everywhere, usually as a default argument. If you read a line and don't specify where to put it, it'll go into `$_`. If you want to see if a string matches a certain pattern, and don't specify which string, Perl will use `$_`. Even some mathematical functions use `$_` as their default argument.

`$CHILD_ERROR`
`$?` The status returned by the last pipe close, backtick command, or `system` call.

`$OS_ERROR`
`$ERRNO`
`$!` The error returned by the last system call. If you use it in a string context (see section **??**), its value is an error message (what `perror()` would have printed). In a numeric context, its value is the current value of `errno`.

`$PROCESS_ID`
`$PID`
`$$` The process ID of the Perl process running this script.

`$REAL_USER_ID`
`$UID`
`$<` The real uid of this process.

`$EFFECTIVE_USER_ID`
`$EUID`
`$>` The effective uid of this process.

`$REAL_GROUP_ID`
`$GID`
`$(` The real gid of this process.

`$EFFECTIVE_GROUP_ID`
`$EGID`
`$)` The effective gid of this process.

@ARGV The list of command-line options passed to the script (not to the Perl interpreter). Unlike C, `$ARGV[0]` is not the name of the script, but rather the first command-line argument.

$PROGRAM_NAME
$0 The file containing the Perl script being executed, like `argv[0]` in C.

%ENV Contains the current values of environment variables. This is similar to `getenv` in C, except that you can manipulate the environment in a more intuitive fashion.

If you change a value in `%ENV`, the new value will be passed down to subprocesses.

%SIG This hash is used to manipulate signal handlers. If you say

```
$SIG{"QUIT"} = handler;
```

or

```
$SIG{"QUIT"} = \&handler;
```

(the latter is preferable), Perl will call the function `handler` when it receives a SIGQUIT.

If you want to restore the default handler for a signal, use

```
$SIG{"HUP"} = 'DEFAULT';
```

Or, if you want to ignore the signal altogether, use

```
$SIG{"HUP"} = 'IGNORE';
```

@INC A list of directories in which to look for included files. You can specify additional directories by passing the `-I`*directory* option to the Perl interpreter (not to your program, since it probably wouldn't know what to do with it).

%INC Contains an entry for each file that has been included using `do` or `require` (we'll talk about those later).

$PERL_VERSION
$] The version of Perl you're using.

$OSNAME
$^O The name of the operating system under which this version of Perl was built.

Actually, if you need to look at this stuff, you may want to use the `Config` module, which contains lots of other juicy details about the local setup.

$EXECUTABLE_NAME
$^X The name of the Perl executable itself.

`$BASETIME`
`$^T` The time at which the script began running, in seconds since the epoch.

`$WARNING`
`$^W` Whether or not this script is running with the `-w` (warning) command-line option.

It can be daunting trying to remember which variable is which, so the perlvar(1) manual page lists a mnemonic for almost every variable.

Some variable names consist of a caret followed by a letter. Rest assured, that really is a caret, and not a control character!

2.8 Quoting

Perl has several methods of quoting strings, analogous to those in the Bourne shell.

`'...'`
`q{...}` Single quotes simply quote what's between them. What you type is what you get. The only exception to this is that you can use a backslash to escape a single quote (`\'`) or another backslash (`\\`).

`"..."`
`qq{...}` Double quotes differ from single quotes mainly in that if you have a variable (either a scalar or an array) in double-quotes, its value will be expanded and interpolated into the string at that point. The same is true of back-quoted expressions (see below).

`` `...` ``
`qx{...}` An expression inside backquotes is passed to the shell as a command; its value is whatever the command prints to *stdout*. As with double quotes, scalar and array values are interpolated.

`qw{...}`[3] This is sort of a single-quote for lists: this allows you to create a list of elements without having to single-quote each one. The words are separated by whitespace instead of commas.

```
@array = qw( sys$disk  It's!  `quot"ing' );
```

`<<`*word* Also known as a "here" document, a `<<`*word* string starts at the next line, and continues until *word* appears on a line by itself.

When an array value is interpolated into a string, Perl inserts the value of the special variable `$"` between each element.

"Here" strings are very handy for multi-line strings, but are somewhat error-prone. Remember that `<<`*word* behaves as if you had just inserted a string at that point on the line, even though the body of the string hasn't started yet. In particular, don't forget the semicolon at the end of the command!

Wrong:

[3] The Camel Book says that this is equivalent to `(...)`, but it doesn't seem to be.

```
print <<EOT
This is a string.
EOT;
```

Right:

```
print <<EOT, " and also ", <<EndOfSecondString;
This is a string.
EOT
this is another.
EndOfSecondString
```

As this example illustrates, you can have multiple "here" documents on one line. They are read in the order in which they appear on the line (for obvious reasons, I think).

By default, <<*word* behaves like a double-quoted string. However, you can enclose *word* in the quotes of your choice (just the *word* after the <<, not the terminating one), and the string will behave as a string of that type.

The q* style of quoting allows you to use any character you like: q/abc/ is equivalent to q#abc#. Alternatively, you can use one of the three symmetrical delimeters: q{abc}, q(abc), q<abc>. Pick whatever seems most readable.

2.9 Operators

Like any respectable programming language, Perl has a full complement of operators. They are:

Math

+, -, *, / — The usual arithmetic operators.

% — Modulus.

** — Exponentiation.

&, |, ^, ~ — Bit-wise and, or and exclusive-or, and not.

<<, >> — Left and right bit-shift.

+=, -=, etc. — *operator*-and-assign. Works just as in C, with all of the operators above.

++, -- — Perl supports both pre- and post-increment and -decrement. They work the same way as in C, with the following flourish: if you have a variable whose value is a string of letters followed by a string of numbers, *var*++ will increment the value anyway: e012 will become e013, aaa will become aab, and so forth. This is useful for programs that need to generate identifiers automatically.

Note that -- is not magical.

<, <=, ==, >=, >, != — Numeric comparison. A string that isn't a proper number has a numeric value of 0.

`<=>` A generalized comparison operator: `$a <=> $b` returns -1 if `$a` is less than `$b`, 1 if `$a` is greater than `$b`, or 0 if they are equal. This is typically used in sorting functions.

Strings

`.` (dot) Concatenates two strings.

`.=` Concatenate and assign.

`x` String multiplier: *string* `x` *number* yields *number* copies of *string*, concatenated together. Thus, `"ab" x 3` yields `"ababab"`.

`x=` Multiply and assign.

`lt`, `le`, `eq`, `ge`, `gt`, `ne` String comparison operators, similar to the ones in the test(1) utility. They compare strings according to (case-sensitive) dictionary order. Thus, `"bar" le "baz"`. If a string happens to be a number, it will be treated as a string: `"100" lt "2"`.

(NB: the "not equal" operator is `ne`, not `neq`.)

`cmp` The generalized string comparison operator. `$a cmp $b` returns -1 if `$a` is less than `$b`, 1 if `$a` is greater than `$b`, and 0 if they are equal.

Files Perl has an inordinate number of file-testing operators, written as `-`*X expr*, where *expr* is either a filehandle, or an expression giving the name of a file.

These operators may appear strange at first, but they harken back to the test(1) utility that the Bourne shell uses.

`-r`, `-w`, `-x`, `-o` File is readable, writable, executable or owned by the effective uid.

`-R`, `-W`, `-X`, `-O` File is readable, writable, executable or owned by the real uid.

`-e` File exists.

`-z` File has zero size.

`-s` Size of the file (so this functions as a "file has non-zero size" test).

`-M`, `-A`, `-C` Returns the time, in days, since the file's last modification time (mtime), access time (atime) or inode modification time (ctime).

`-f` File is a plain file.

`-d` File is a directory.

`-l` File is a symbolic link.

`-p` File is a named pipe.

`-S` File is a socket.

`-b` File is a block special file.

`-c` File is a character special file.

- `-t` File is a filehandle opened to a terminal (this is Perl's version of `isatty()`).
- `-u` File has the setuid bit set.
- `-g` File has the setgid bit set.
- `-k` File has the sticky bit set.
- `-T` File is a text file.
- `-B` File is a binary file.

You can also pass any of these operators a special filehandle, called `_` (underscore). This causes the operator to reuse the results from the last `stat()` call:

```
if (-u $filename || -g $filename)
```

calls `stat()` twice, whereas

```
if (-u $filename || -g _)
```

only calls it once.

Others

`=~`, `!~` Certain functions, notably the string-match and string-replace functions, need a string as their argument. By default, they use `$_`. The `=~` makes them work on some other string, e.g., `$foo =~ m/abc/`.

`!~` works the same way as `=~`, but negates the result of the operation.

`&&`, `||` Logical and, or. These work like C's `&&` and `||`, and perform short-circuit evaluation.

One difference, though: in Perl, these operators return not 0 or 1, as in C, but rather the last value seen. Thus, if your program needs to run the user's favorite editor, a good way to do that is to use

```
$EDITOR = $ENV{VISUAL} ||
          $ENV{EDITOR} ||
          "/usr/ucb/vi";
```

`&`, `|`, `^`, `~` Bitwise and, or, xor; one's complement.

`!` Not.

`?:` An in-line `if` statement.
expression `?` *if-expr* `:` *else-expr*
works the same way as in C.

`..` (dot dot) The range operator is rather magical. It's one of my favorites.

In a list context, *num1* `..` *num2* returns the list of numbers from *num1* to *num2*. This is handy for taking array slices (`@array[3..10]`) or for repeating a loop a fixed number of times (`for (1..50)...`). Be aware, though, that this does generate a temporary array, so you can waste a lot of memory by using this with a large range.

In a scalar context, *expr1* `..` *expr2* acts as a "flip-flop." It starts out false. Then, once the left-hand expression becomes true, `..` returns true and starts evaluating its right-hand expression, until that becomes true. *After* that, `..` flip-flops back to being false.

This is useful for things like finding text between delimiters in a file:

```
perl5 -ne 'print if /BEGIN/../END/'
```

will read standard input and print all lines between lines delimited by `BEGIN` and `END`.

There is also a `...` (dot dot dot) operator, related to `..`, but I won't cover it here.

`and`, `or`, `xor`, `not` These work just like `&&`, `||`, `^` and `!`, but have a much lower precedence.

The motivation for this is that, since most functions return some value that tells whether it succeeded, it is rather common to write *expr1* `&&` *expr2* as a way of saying, "do *expr1*, and if that succeeds, do *expr2*."

Unfortunately, depending on what operators *expr1* and *expr2* might contain, Perl might not parse things the way you want it to. `and`, `or` and `not` have the lowest precedence, so your code will always be parsed as `(`*expr1*`) and (`*expr2*`)`.

There is a hierarchy of precedence to these operators, but don't memorize it (aside from the rule about `and`, `or`, `xor` and `not`). Use parentheses to make explicit how you want expressions to be evaluated, and your code will be more readable for it.

3 Flow Control

Perl has mostly the same flow-control operators as C does, with just a few flourishes of its own.

3.1 `if` and `unless`

```
if (condition) {
   commands
}
[elsif (condition) {
   commands
}...]
[else {
   commands
} ]
```

Perl's `if` statement is reminiscent of both C and the Bourne shell, except that *sh*'s `elif` has mutated into `elsif`.

One potential pitfall for C programmers is the fact that in Perl, braces are mandatory, even if you only have one statement. This avoids ambiguity caused by nested `if` statements.

The `unless` construct is similar to `if`.

```
unless ($i < 100)...
```

is equivalent to

```
if (not $i < 100)...
```

I won't tell you how `unless` affects `elif` and `else` blocks, because such constructs are confusing and should be avoided. `unless` is best suited for one-line postfix conditions.

3.2 Truth And Other Booleans

At this point, it might be useful to digress for a moment to expore the nature of truth. Actually, it's easier to define what is false: empty arrays and hashes, and nonexistent variables, are false. That's the easy bit.

For scalars, 0 is false, as is the empty string. Actually, there are two varieties of empty string: the first is `""`, the second is the undefined value.

The undefined value is what you get if, say, you try to get a hash element that doesn't exist. It is similar to the `NULL` pointer in C. You can find out whether a variable has the undefined value by calling `defined(`*variable*`)` (or `!defined(`*variable*`)`, as the case may be).

By the way, you can explicitly set a variable to the undefined value by using `undef`:

```
$var = undef;
```

That's it for falsehood. Anything that isn't false is true.

3.3 `while` and `until`

```
while (condition) {
        commands
}
[continue {
        commands
} ]
```

The `while` loop should also look fairly familiar to C programmers. Again, as with `if`, the braces are mandatory.

Note, however, the optional `continue` block. The statements in the `continue` block will be executed every time the loop repeats, whether by falling off the end of the `while` block, or because of an explicit loop-control construct (which we'll see in a bit). It allows you to make sure that a particular piece of code gets executed every time you iterate through a loop. It's not used often in practice, but it's there if you need it.

`until` works just like `while`, except that the test is negated.

3.4 Postfix Conditionals

`if`, `unless`, `while` and `until` also come in postfix versions. That is,

```
if ($foo eq "abc")
{
        print "Foo is abc\n";
}
```

is equivalent to

```
print "Foo is abc\n" if $foo eq "abc";
```

Likewise for the other postfix conditionals.

Note that in the postfix version, parentheses are not required around the condition, since there is no ambiguity as to where the condition begins and ends.

One caveat: if you have a `do` block followed by a postfix `while` or `until`, the `do` block will execute at least once. This is so that

```
open FILE, "$filename";
do {
        $line = <FILE>;
        print $line;
} until $line =~ /END/;
close FILE;
```

will work as expected.

3.5 `for`

```
for (init; condition; continue) {
    commands
}
```

This looks a lot like C's `for` statement, doesn't it? No surprises here. This is equivalent[4] to

```
init;
while (condition) {
      commands
```

[4]In fact, Larry added the `continue` block so that `for` could be defined precisely this way.

```
} continue {
      continue
}
```

3.6 The other `for`, and `foreach`

```
for [var] (list) {
    commands
}
```

Perl's `for` and `foreach` loops (the two are synonymous) behave much like *sh*'s `for` and *csh*'s `foreach` loops. They iterate over *list*, setting *var* to each element in turn. If *var* is omitted, `$_` is used.

Again, the parens and curly braces are mandatory.

3.7 Bare blocks

Technically, a bare block—that is, a pair of curly braces with statements between them—is a loop that executes exactly once. I bring this up because you do a few interesting things by treating a block as a loop, as we'll see later on.

Oh, and by the way: you can omit the semicolon after the last statement in a block.

3.8 do

`do {` *BLOCK* `}` executes the commands in *BLOCK* and returns the value returned by the last statement. Useful when you can't use a bare block, or when you'd like to write something as one statement, but goshdarnit, you need two or more:

```
open INFILE, "/my/file" or do {
                print STDERR "I don't know what to do.\n"
                exit 1;
        };
```

3.9 Loop-control commands

There are times when you don't want to finish executing the body of the loop you're in. For these cases, Perl provides not one, not two, but three special commands:

`last`: this is similar to C's `break` statement: it exits the loop immediately (without passing through the `continue` block, if any), causing control to continue with the next statement after the closing brace.

`next`: this is similar to C's `continue` statement: control passes through the `continue` block, if any, then back up to the condition at the top of the loop.

`redo` has no equivalent in C. It causes control to go back to the top of the `while` block, to the first statement after the opening brace. The `continue` block is not executed, and the condition is not evaluated.

3.10 Block labels

But wait, there's more! You can also put labels in front of blocks and loop commands:

```
OUTER: for ($i = 0; $i < 100; $i++)
{
    INNER: for ($j = 0; $j < 10; $j++)
    {
        if (&an_error_has_occurred)
        {
            last OUTER;
        }
    }
}
```

By default, `last`, `next` and `redo` act on the innermost loop that they're in. If you give them a label, they apply to the innermost enclosing block that has that label.

3.11 Gotos

Perl has three `goto` invocations, two of which are highly magical, but you'll have to look them up yourselves, because they're evil and, with all the loop-control stuff we've just seen, unnecessary.

4 I/O

Since Perl was originally written as a report generator, it's not surprising that it can perform various I/O operations. Since I/O is such an important function of Perl's, it has more magic than most other parts.

4.1 Files

Most of the time, when you need to read a file, you'll do the following:

```
open INFILE, "/my/file";
while ($line = <INFILE>)
{
        # Do something
}
close INFILE;
```

The `open` function opens a file, naturally enough. The first word, `INFILE` is the *filehandle* (by convention, filehandles are in all-caps, to set them apart). The second argument specifies the filename. By default, the file is opened for reading.

The special filehandles `STDIN`, `STDOUT` and `STDERR` come pre-opened, so you don't even need to open them to use them.

Above, we used `open INFILE, "/my/file"`. We could also have said open INFILE, "</my/file", to explicitly say that it's being opened for reading. Similarly, open OUTFILE, ">/my/file" opens /my/file for writing, and open OUTFILE, ">>/my/file" for appending.[5]. If you put a `+` in front of the < or >, you'll get both read and write access to the file.[7]

`open CMD, "|`*command*`"` will invoke the shell and run *command*; if you write to the `CMD` filehandle, your text will be fed to the standard input of *command*. Likewise, `open CMD, "`*command*`|"` will run *command*, and reading from `CMD` will read from its standard output. You can't put a pipe at both ends of a command this way. You have to jump through a few hoops to do that.

The `close` statement closes the file, naturally enough.

The expression <*FILEHANDLE*> reads the next line from *FILEHANDLE* and returns it. When it reaches the end of the file, <*FILEHANDLE*> will return the undefined value, which is false, as we've seen above, and the `while` loop will terminate.

In an array context, however, the angle operator will read every line in *FILEHANDLE*, put them into an array, and return it. This is a quick and easy way to read in an entire file (and potentially waste tons of memory).

The angle operator has a bit of magic built in: if it's the only thing in a `while` loop condition, it'll read the next line and assign it to the `$_` variable. The following:

```
while ($_ = <FILE>)
{
        print $_;
}
```

is equivalent to

```
while (<FILE>)
{
        print $_;
}
```

And as this example shows, you write using the `print` function. The syntax for `print` is

`print` *[FILEHANDLE] [expression]*

(Note that there is no comma between *FILEHANDLE* and *expression*)

[5]If you're dealing with user-supplied filenames, don't use `open FILE, "$filename"` since if `$filename` contains < or >, it will be interpreted as part of the open mode specification. Similarly, don't use open FILE, ">$filename", since if `$filename` begins with >, you will append to the file instead of zeroing it first. Use open FILE, "> $filename" instead.[6]

[6]You can't use `open` to open a file whose name begins with a space. Use `sysopen` instead.

[7]+> zeroes the file first; +< doesn't.

If you omit *FILEHANDLE*, it defaults to STDOUT. If you omit *expression*, it defaults to $_.

Unlike some other languages, Perl does not take care of the ends of lines for you: when you print a line, you need to include the \n at the end. Similarly, when you read a line with <*FILEHANDLE*>, it still has the newline at the end.

Since trailing newlines usually get in the way of what you want to do, Perl provides the chop and chomp functions. chop $*var* chops off the last character of $*var* (and returns it). chomp $*var* looks for a newline (or whatever the record separator is set to) at the end of $*var*, and removes it if it is there (and returns the number of characters it removed).

4.2 <>

You might think that since

```
print "Hello, world!\n";
```

is equivalent to

```
print STDOUT "Hello, world!\n";
```

that <> would be equivalent to <STDIN>. Well, not quite. <>, sometimes called the diamond operator, has magic of its own.

Most Unix filters, like *grep*, *sed*, *awk* etc., will read the files named on the command line, or standard input if there aren't any, or if you specify - (dash) as a filename. <> does all this for you.

When you use <>, it looks at @ARGV, the array of command-line options. It will remove the first argument from @ARGV, and open the file that it names. Once it has finished reading that file, it will close it, grab the next filename from @ARGV, and so forth.

If there weren't any filenames in @ARGV to begin with, <> will first set @ARGV to ("-"), then proceed as above. That way, it will open the filename - (dash), which is special, and gives you STDIN when it's opened for reading, and STDOUT when it's opened for writing.

So to answer the question posed at the beginning of this section, if you want to be sure of reading from STDIN, you need to explicitly say <STDIN>.

In fact, while (<>) loops are so common that Perl provides not one, but two command-line options that provide one automatically. They are primarily intended to be used with the -e option, which says that the next command-line argment is the script.

perl5 -ne *script* is equivalent to

```
while (<>)
{
        script
}
```

and perl5 -pe *script* is (almost) equivalent to

```
while (<>)
{
        script
} continue {
        print
}
```

You may think that Perl I/O is needlessly burdened with special cases and exceptions, but it does simplify many short scripts. The simplest way to write *cat* in Perl is

```
perl5 -pe '' filename...
```

and *grep* becomes

```
perl5 -ne 'print if /pattern/' filename...
```

If you were to spell it out, without any magic,[8] this last example would become

```
@ARGV = ('-') unless @ARGV;
while ($ARGV = shift @ARGV) {
        open FILE, $ARGV or die("Can't open $ARGV: $!\n");
        while ($_ = <FILE>)
        {
                print STDOUT, $_ if /pattern/;
        }
        close FILE;
}
```

The `-i`*extension* command-line option turns on in-place editing for `<>`. That is, if you specify `-i.bak`, then when `<>` opens a file `foo`, it will rename it as `foo.bak`, and also open `foo` for writing and make this the default filehandle for `print` statements. Thus,

```
perl5 -i.bak -pe 'tr/a-z/A-Z/' foo bar baz
```

will translate the contents of files `foo`, `bar` and `baz` to upper case, and leave backup copies in `foo.bak`, `bar.bak` and `baz.bak`.

Note that `<>` really does use `@ARGV`, so it's perfectly legal to say

```
@ARGV = qw( foo bar baz );
while (<>)
{
        # Do something
}
```

[8] Well, hardly any.

4.3 <...> the Globber

One last thing about `<...>`: if you put something other than a filehandle between the angle brackets, they become something else: the filename globbing operator. The expression between brackets will be interpreted as a pattern. The globbing operator will return every filename that matches the pattern. In a list context, it'll return them all at once; in a scalar context, it'll return one at a time.

```
@etcfiles = </etc/*>;

while (</tmp/*>)
{
        print "$_\n"
}
```

4.4 End Of File

The `eof` function tests end-of-file status. Normally, it is invoked as `eof` *FILEHANDLE*, which returns true if *FILEHANDLE* is currently at the end of file (*i.e.*, if the next read would return the undefined value).

If you omit the *FILEHANDLE* argument, `eof` tests the last filehandle that was read from.

You can also say `eof()`, which tests the pseudo-file that `<>` uses, that composed of all of the files listed on the command line. In other words, `eof()` will tell you whether you've reached the end of the last file listed on the command line.

If you're using `<>` and want to detect the end of each file, you can either use `eof` without any arguments (assuming you haven't read from any other files), or use the special filehandle `ARGV`:[9] `eof(ARGV)`.

4.5 Formats

Formats are Perl's way of letting you print pretty formatted reports. The way it does this is rather nice: you just draw a picture of how you want the data to come out, fill in the blanks, and `write`.

That's `write`, not `print`. `print` just does plain, ordinary printing. `write` outputs the next record for the format you're using.

Perhaps the easiest way to show what formats are all about is with an example:

```
format NEWHOST =
Name:   @>>>>>>>>>  IP: @<<<<<<<<<<<<<< Ether: @||||||||||||||||
        $hostname,      $ip,            $ether
Domain:    @>>>>>>          CPU Tag: @<<<<<<<<<<<<<<<<<<<<
        $domain,        $cpu_tag,
                        Monitor Tag: @<<<<<<<<<<<<<<<<<<<<
```

[9] Yes, `ARGV` is magical. Are you surprised?

```
                        $mon_tag
Problems with the installation:
    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
        $problems
~   ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
        $problems
~   ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
        $problems
~   ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<...
        $problems
# Should there be more than four lines for problems?

Comments:
@*
$comments
.
```

As you see, a format declaration begins with `format` *formatname* `=`, and continues up to a dot on a line by itself.

The text just prints the way it's laid out. The things that look like `@<<<<<` and such are *picture fields*. They specify where the data should go on the line. `@>>>>` means that the data should be flushed right, `@<<<<<` means it should be flushed left, and `@|||||` means it should be centered within the field.[10]

You can also have fields of the form `@######` or `@###.##`, for specifying numeric values. If the field includes a dot, the decimal will line up with it.

Underneath each picture field is a variable name. This is, intuitively enough, the variable whose value will be plugged into the field. The variables are separated by commas. And they don't have to line up with the fields, they just have to be in the same order. Lining them up just makes it clearer what belongs with what.

A `#` at the beginning of a line marks that line as being a comment.

You'll note that some of the fields begin with `^`, and some lines have a `~`. Normally, if a value is too wide to fit in a field, the end is chopped off, and you never see it. If, however, the field begins with `^`, Perl will fit as much of the value as it can into the field, chop that off,[11] and save the rest until the next time that variable is used.

If all of the picture fields on a line are blank, and there's a `~` anywhere on the line, that line will not be printed (if it is printed, though, the `~` will be turned into a blank).

Thus, in the above example, the variable `$problems` can take up to four lines of text. The dots at the end of the fourth line are just there in case that's not enough, to let the reader know that there was more text.

[10]The `@` is part of the picture field, and does count toward its width. One consequence of this is that if you have a one-character field, you cannot specify whether it is to be flushed left or right, or centered. This is not considered a problem.

[11]This *does* change your variable, so make sure it's not something you're going to need later.

Perl tries to be reasonably smart about splitting lines this way: by default, it'll break only on whitespace or a dash, though you can change this by setting the `$:` variable.

Finally, we get to the special field `@*`. This just inserts the value of its variable as-is, without any splitting, just the way it appears, for as long as it takes.

To print using this format, use:

```
$~ = NEWHOST;

$hostname = "glitnir";
$domain   = "CfAR";
$ip       = "128.8.132.40";
$ether    = "00:40:05:4a:c0:0a";
$cpu_tag  = "12345";
$mon_tag  = "67890";
$problems = "None";
$comments = `cat /tmp/long.rant`;

write;
```

Each filehandle has a default format that has the same name as the filehandle (note, however, that the two are not otherwise related; one is a format, the other is a filehandle). In this case, however, we're using a format called `NEWHOST`. Rather than open a new file called `NEWHOST`, let's simply set `$~` to associate the format `NEWHOST` with the default filehandle, which is currently `STDOUT`.[12]

We then assign values to each of the variables that appear in the format, and call `write`. `write` also takes an optional filehandle as an argument, in case you don't want to `write` to the current default filehandle.

Each format also has a top-of-form format that gets printed every time `write` begins a new page. By default, the top-of-form format has the same name as the default format for that filehandle, but with `_TOP` at the end. You can use this to print page numbers (available in `$%`), or to print column headings, *e.g.*:

```
format HOSTINFO_TOP =
Host information                                Page @<<
                                                     $%

Keep this info up to date, or feel the wrath of Theresa!

Name            Room     Type     Model    Comment
.
```

[12] You can specify which filehandle is the default for `print` and `write` with `select` *filehandle*.[13]

[13] Note that this is different from the `select()` system call, which is also available in Perl, and is also called `select`.

```
@<<<<<<<<<<<<<  @<<<<   @<<<<<  @<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<
$hostname,      $room,  $type   $model  $comments
~                                       ^<<<<<<<<<<<<<<<<<<<<<<<
                                        $comments
.
```

You can change the top-of-form format by setting the `$^` variable.

4.6 I/O Special Variables

As you might expect, there are a number of special variables associated with I/O. They are:

`$_`
: `while (<FILEHANDLE>)` reads the next line into `$_` by default.

`$ARGV`
: The name of the current file when reading with `<>`.

`$INPUT_RECORD_SEPARATOR`
`$RS`
`$/`
: The input record separator. This is a newline (`"\n"`) by default.

  Note that this is magical: if you set it to the empty string (`""`), it will behave as if you had set it to two newlines (`"\n\n"`), with this exception: two or more blank lines in a row will be compressed into one blank line. This makes it easy to read files one paragraph at a time.

`$OUTPUT_AUTOFLUSH`
`$|`
: If set to a nonzero value, forces a flush every time you write to the currently-selected filehandle.

`$OUTPUT_FIELD_SEPARATOR`
`$OFS`
`$,`
: Output field separator. When you `print` several items, separated by commas, Perl inserts the value of `$,` between each item.

`$LIST_SEPARATOR`
`$"`
: Like `$,`, but applies to arrays interpolated into a double-quoted string.

`$INPUT_LINE_NUMBER`
`$NR`
`$.`
: The current input line number for the last filehandle that was read from.

`$FORMAT_LINES_PER_PAGE`
`$=`
: The number of lines per page on the currently-selected output channel.

`$FORMAT_LINES_LEFT`
`$-`
: The number of lines left on the current output page.

`$FORMAT_PAGE_NUMBER`
`$%`
: The current page number of the currently-selected output channel.

$FORMAT_NAME
$~ The name of the current format for the currently-selected output channel.

$FORMAT_TOP_NAME
$^ The name of the current top-of-page format for the currently-selected output channel.

$FORMAT_LINE_BREAK_CHARACTERS
$: A string containing the characters after which it is okay to break a long line in a format, and start filling in continuation (`^`) fields. This is `"\n-"` by default.

$FORMAT_FORMFEED
$^L The string that formats should output to produce a form feed. This is `"\f"` by default.

$ACCUMULATOR
$^A The current value of the `write` accumulator for format lines. See perlform(1) and perlfunc(1) for details.

$INPLACE_EDIT
$^I The current value of the inplace-edit extension. If Perl is running with the `-i` command-line option, but no backup extension specified, `$^I` will be the empty string. If the `-i` option was not specified, `$^I` has the undefined value.

5 Regular Expressions

Regular expressions, often abbreviated "regexps", "REs", or simply "patterns" are a major part of Perl, simply because Perl is so good at them.

There's a mathematical definition of "regular expression" that you may have run into if you've taken compiler design. However, Perl has added so much on top of that that it's nearly useless, so we'll ignore it. Suffice it to say that a regular expression is a pattern of characters. For instance, "abc" is a regular expression, and

```
m/abc/
```

will return a true value if the current line contains an 'a' followed by a 'b' followed by a 'c'.

Any ordinary character is a regular expression that matches itself. Several regular expressions in a row mean that the string must match the first one, then immediately the second one, then the next, and so forth.

There are also several special characters that have special meanings within regexps:

`^` Matches the beginning of a line: `foobar` matches `m/bar/`, but not `m/^bar/`.

`$` Matches the end of a line: `foobar` matches `m/foo/`, but not `m/foo$/`.

Using `^` or `$` is known as *anchoring* a pattern: you're saying that one end or the other of the pattern (or both) has to be at the beginning or end of the string.

One thing to watch out for: `$` normally precedes a variable name, and variables *are* interpolated into patterns. So if you say

```
m/foo$bar/
```

This does not mean `foo`, followed by a newline, followed by `bar`. Perl will first expand the variable `$bar`, an treat the whole thing as a pattern.

Since the end of a line normally only occurs at the end of a pattern, Perl is usually smart enough to figure out what you meant, but it's still something to bear in mind. And if you *did* mean to say "`foo`, followed by newline, followed by `bar`," there are ways of doing that, which we'll cover in a bit.

`.` (dot) Matches any character except newline.

pat1 | *pat2* Matches either pattern *pat1* or *pat2*.

`[`*range*`]` Matches any character in the range *range*. For instance, `m/[abc]/` will match either `a`, `b` or `c`. For longer ranges, you can use a dash: `m/[0-9a-f]/` will match any (lower case) hex digit. (If you want to include a dash in your range, make it the first character in the range: `m/[-a-z]/` will match any letter, or a dash.)

You can also negate a range by putting `^` at the beginning of the range: `m/[^a-z]/` will match any character except a lower case letter.

`(`*pattern*`)` Parentheses perform grouping. They also make individual parts of the matched string available through the `$`*digit* variables (see below).

`\` A backslash escapes the next character, so that it loses any special meaning that it might have. Thus, `\(` matches a literal open parenthesis.[14]

pattern`*` Matches zero or more occurrences of *pattern*.

pattern`+` Matches one or more occurrences of *pattern*.

pattern`?` Matches zero or one occurrences of *pattern*. (*i.e.*, *pattern* is optional)

pattern`{`*n*`}` Matches exactly *n* occurrences of *pattern*.

pattern`{`*n*`,}` Matches at least *n* occurrences of *pattern*.

pattern`{`*n*`,`*m*`}` Matches at least *n*, but no more than *m* occurrences of *pattern*.

Watch out when you use *pattern*`*`: remember that it does match zero occurrences of the pattern. Thus, if you want to see if a string has more than one word in it, you might be tempted to write

[14]This is in contrast to some other regular expression implementations: in *sed*, for example, you have to use `\(...\)` to create a backreference.

```
m/[a-z]\s*[a-z]/
```

However, the string `ab` would match, since it consists of a letter, followed by zero spaces, followed by another letter. In this case, you'd need to use `\s+`, to make sure that there was at least one space between the two letters.

By default, Perl's regular expressions are "greedy," *i.e.*, they try to match as much as possible:

```
"one two three" =~ /(.*)\s(.*)/;
```

will set `$1` to `one two` and `$2` to `three`. If this is not what you want, you can append a `?` to the standard numeric modifiers to change the greediness. Thus, *pattern*`*?` will match zero or more instances of *pattern* (but as few as possible), *pattern*`+?` will match one or more instances (but as few as possible), and *pattern*`??` will match zero or one instances (but preferably zero). Thus,

```
"one two three" =~ /(.*?)\s(.*)/;
```

will set `$1` to `one` and `$2` to `two three`, and

```
"one two three" =~ /(.*?)\s(.*?)/;
```

will set `$1` to `one` and `$2` to the empty string.[15]

The following parenthesized expressions, of the form `(?...)` go a long way toward making Perl regular expressions not merely powerful, but obscenely powerful:

`(?#`*text*`)` A comment. *text* is ignored.

`(?:`*pattern*`)` Groups a regular expression like `( )`, but doesn't make a backreference (*i.e.*, doesn't set `$1`).

`(?=`*pattern*`)` A zero-width, positive lookahead assertion. That is,

```
/foo(?=bar)/
```

matches `foo`, but only if it is followed by `bar`. However, `$&` will only contain `foo`, not `bar`.

`(?!`*pattern*`)` A zero-width, negative lookahead assertion.

```
/foo(?!bar)/
```

will match `foo`, unless it is followed by `bar`.

Note that this can sometimes be nonintuitive. For instance, `"aaab"` matches the pattern `/a+(?!b)/`, since `"aa"` *is* a string of `a`s that isn't followed by a `b`: it's followed by another `a`!

[15] Actually, if you think about it some more, you'd expect the last example to set both `$1` and `$2` to the empty string. For an explanation of why not, see the perlre(1) man page.

There are certain ranges that occur over and over, so Perl has predefined shorthand for them:

`\w` Matches any alphanumeric ("word") character, or an underscore (_).

`\W` Matches anything but a "word" character (*i.e.*, anything that `\w` doesn't match.

`\s` Matches a whitespace character.

`\S` Matches a non-whitespace character.

`\d` Matches a digit (0–9).

`\D` Matches a non-digit character.

`\b` Matches a word break. This doesn't match any actual characters. Rather, it matches the place between a word character (`\w`) and a non-word character (`\W`).

`\B` Matches a non-word break.

`\t` Matches a tab.

`\n` Matches a newline.

`\r` Matches a carriage return.

`\f` Matches a form-feed.

`\a` Matches a bell character.

`\e` Matches the escape character.

n Matches the *n*th parenthesized expression (where *n* is a digit) within a pattern. Thus, `"foo-foo"` will match the pattern `(.*)-\1`, but `"foo-bar"` won't.

`\0`*nn* Matches the character whose ASCII code, in octal, is *nn*. Note: this is not the same as the same as the "match a previous substring" escape; the leading `0` distinguishes the two cases.

`\x`*nn* Matches the character whose ASCII code, in hex, is *nn*. Note: this, too, is not the same as the same as the "match a previous substring" escape; the leading `x` distinguishes the two cases.

These next few escapes actually apply to strings in general, but we might as well mention them here:

`\l` Convert the next character to lower case.

`\u` Convert the next character to upper case.

`\L` Convert to lower case until `\E`.

`\U` Convert to upper case until `\E`.

`\Q` Quote regexp metacharacters until `\E`.

5.1 Extended Regular Expressions

With all of this stuff going on, you may have gotten the impression that Perl regular expressions are a write-only language. Unfortunately, there's a lot of truth in this. If you have an error in a particularly hairy regexp, it may be easier to just rewrite it from scratch than to try to fix it.

However, you can use the `/x` option to `m//` and `s///` to enable extended regular expressions. In an extended regular expression, all whitespace is ignored (unless it's escaped or in a character range), so you can split it up into lines, and indent it for legibility. Also, you can use `#` to introduce comments.

Let's look at a fairly hairy regular expression:

```
m/^                                 # Anchor beginning

 # Start with the day
 (mon|tue|wed|thu|fri|sat|sun)      # Day of the week
 \.?                                # Optional dot
 ,\s+                               # Comma, whitespace

 # Now try to match the date
 ((jan|mar|may|jul|aug|oct|dec)     # The 31-day months
        \s+                         # Whitespace
        (0?[1-9]        |           # 1-9 (01 also allowed)
         [12][0-9]      |           # 10-29
         3[01]                      # 30 and 31
        )
  | (apr|jun|sep|nov)               # The 30-day months
        \s+
        (0?[1-9]        |           # 1-9 (01 also allowed)
         [12][0-9]      |           # 10-29
         30                         # 30
        )
  | feb                             # February: we don't
                                    # allow leap years
        \s+
        (0?[1-9]        |           # 1-9 (01 also allowed)
         1[0-9]         |           # 10-19
         2[0-8]                     # 20-28
        )
 )
```

```
 # Finally, get the year
 (19)?\d{2}

$                                   # Anchor end
/xi
```

This pattern matches a date of the form, `Mon., Jun 4, 1998`. The complexity comes from the fact that it only maches valid dates (*i.e.*, it doesn't match `Feb. 44`).

Granted, this is still a mess, but it's better than

```
m/^(mon|tue|wed|thu|fri|sat|sun)\.?,\s+((jan|mar|may|jul|aug|
oct|dec)\s+(0?[1-9]|[12][0-9]|3[01])|(apr|jun|sep|nov)\s+(0?
[1-9]|[12][0-9]|30)|feb\s+(0?[1-9]|1[0-9]|2[0-8]))(19)?\d{2}
$/i
```

5.2 Special Variables

Perl has a number of special variables associated with regular expressions.

`$_` The default variable to match.

If you just say `m/abc/`, Perl will see if `$_` contains `abc`. If you want to see if some other string matches the pattern, you need to use `$var =~ m/abc/` (actually, you don't need to use a variable. You can use any string).

$*number* Matches the *number*th parenthesized expression.

As I mentioned earlier, parentheses perform grouping in a regular expression. They also indicate to Perl that you're interested in that part of the string, so they make it available through the variables `$1`, `$2`, etc.

The part of the string that matches the first set of parentheses will be placed in `$1`, the part that matches the second set of parentheses will be put in `$2`, and so forth. So if you have

```
$_ = "name = arensb   uid = 2072";
m/name = (\w+)\s+uid = (\d+)/;
```

`$1` will be set to `arensb` and `$2` will be set to `2072`.

Remember, parenthesized expressions can nest. To get the number of a parenthesized expression, just count the open-parens from the left:

```
$_ = "user n arensb";
m/user (n (\w+)|# (\d+))/;

$_ = "user # 2072";
m/user (n (\w+)|# (\d+))/;
```

In the first case, `$1` will be set to `n arensb`, `$2` will be set to `arensb`, and `$3` will have the undefined value.

In the second case, `$1` will be set to `# 2072`, `$2` will have the undefined value, and `$3` will be set to `2072`.

$MATCH
$& Gets set to the text that matched the pattern.

$PREMATCH
$` Gets set to the text preceding the text that matched the last pattern.

$POSTMATCH
$' Gets set to the text following the text that matched the last pattern.

$LAST_PAREN_MATCH
$+ Gets set to the text that matched the last parenthesized expression that matched something. For instance:

```
m/Username: (\w+)|UID: (\d+)/;
```

5.3 Operations

Okay, now that you know what a regular expression is, what can you do with one? First of all, you can see if a string matches it:

5.3.1 m//

`m/`*pattern*`/`*[options]*

`m//` returns a true value if a string matches *pattern*, and a false value otherwise.

Note that you can use any character[16] as the pattern delimiter, instead of slashes. This is especially useful when you're matching a filename and don't want to write `\/` all the time.

Again, as with the `q*`-style quoting operators, you can use the symmetrical delimiters: `m{...}`, `m(...)`, `m<...>`.

If, however, you choose to use slashes, then you don't need the `m` at the beginning. In addition, if you don't say otherwise, Perl will match `$_` against the pattern. That's why you'll often see

```
if (/^user.*/)
{
        ...
}
```

Otherwise, you can specify

[16] Any non-alphanumeric, non-whitespace character, that is. Otherwise, `magenta` would be a valid Perl program, which would be confusing.

```
$var =~ /pattern/
```

or

```
$var !~ /pattern/
```

to say "`$var` matches *pattern*" or "`$var` doesn't match *pattern*," respectively.

`m//` takes a number of options:

`i` Do case-insensitive pattern-matching.

`g` Do a global pattern match. In a list context, `m//g` returns a list of all of the patterns that it found in the string. In a scalar context, it finds the first match and returns true; then, if you match the same pattern, Perl will remember where it left off and start from there. Thus,

```
$_ = "a1b2c3";
@numbers = /\d/g;
```

will set `@numbers` to `(1, 2, 3)`, and

```
while ("a1b2c3" =~ /\d/g)
{
        print "$&\n";
}
```

will print

```
1
2
3
```

If you're using `m//g`, you can use `\G` in the pattern, to match the place where the last match left off. This acts as a weak `^`. For instance,

```
$_ = "abc123def456";
m/\d/g;
print "$&\n";
while (/\G\d/g)
{
        print "$&\n";
}
```

will print

```
1
2
3
```

`c` Normally, when `m//g` doesn't match, it resets the search position to the beginning of the string. `m//gc` prevents this.

m This is a multi-line pattern. That is, `^` will match the beginning of every line, and `$` will match the end of every line in the string.

   Inside of `m//m`, you can use `\A` and `\Z`. `\A` matches only the very beginning of the string, and `\Z` matches only the very end.

o Only compile the pattern once.

   If you have variables in a pattern, Perl will interpolate them, then compile the pattern. It'll do this every time it sees the pattern. This can be expensive, so if you specify the `/o` option, Perl will compile the pattern only once. Of course, if the variables in your pattern change value, this won't work.

s Treat the string as a single line: `.` will match newlines.

x Enable extended regular expressions. These get a section of their own.

(Are you overwhelmed yet?)

5.3.2 s///

The next really useful thing you can do with regular expressions is string replacement. This is done with

   s/*pattern*/*replacement*/*[options]*

This replaces the text matched by *pattern* with the *replacement* text, and returns the number of substitutions made. Note that *replacement* is a string, not a pattern.

Again, just as with `m//`, `s///` will work on `$_` by default, or you can use `=~` to have it work on some other variable. Likewise, you can use a delimiter other than slashes, if you like.

Since *pattern* is a regular expression, all of the special variables are available on the right, so

```
$string = "ABCdef";
$string =~ s/^(...)/$1($1)/;
```

will set `$string` to `ABC(ABC)def`.

`s///` takes the same options as `m//`, with the addition of

e Treat *replacement* as an expression to be evaluated.

   When you have the `/e` option, `s///` will match the pattern on the left, then evaluate the string on the right as a Perl expression, and replaces *pattern* with whatever the expression returns. For example:

```
s/\d+/(getpwuid($&))[0]/eg;
```

   replaces any integer in `$_` with the name of the user with that uid.

   A word of caution: Perl doesn't compile the replacement expression until it needs to, so if it contains a syntax error, you won't see any error messages about it until it is encountered at runtime.

5.3.3 `tr///` and `y///`

`tr/`*searchlist*`/`*replacementlist*`/`*[options]*

`tr///` replaces the characters on the left with the corresponding characters on the right, much as the *tr* program does (`y` is a synonym for `tr`). Thus,

```
tr/A-Z/a-z/
```

converts all upper case letters to lower case, and leaves everything else alone.

Note that *searchlist* and *replacementlist* are strings, not full-fledged regular expressions (except that you can have ranges), but this seemed like the right place to talk about this.

As you're probably getting used to by now, `tr` operates on `$_` by default, or you can use `=~` to make it work on another variable.

`tr///` only takes a few options:

`c` Complement the search list: anything that doesn't match will be replaced by the last character of *replacementlist*

`d` Delete any characters that were found in the search list, but don't have a corresponding replacement. Normally, they are replaced by the last character in the replacement list.

`s` Squash identical characters in the output:

```
s/ \t/_/s
```

converts `a b      c` to `a_b_c`.

5.3.4 `split` and `join`

`split /`*pattern*`/, `*[string*`, `*[limit]]*
`join `*string*`, `*array*

`split` looks for instances of *pattern* in the string, and returns an array consisting of everything else. Thus,

```
split /:/, "staff:*:10:arensb, arnie"
```

will return `("staff", "*", "10", "arensb, arnie")`.

If you omit the string, `split` will use—you guessed it—`$_`.

If you also omit the pattern, `split` will split on whitespace, and will also strip leading whitespace so that you don't get an empty first element.

If you specify a number as *limit*, then `split` will split the *string* into no more than that many parts.

`join` is the converse of `split`: it returns a string, consisting of all of the elements in *array*, with *string* in between.

One thing to watch out for: it *is* legal to split on an empty pattern (*e.g.*, `split // "abc"`), but this usually isn't what you want: this will return an array of every character in the string. Another common mistake is, as with `m//`, using `*` instead of `+`: `split /:*/` will split `ab::c` into `("a", "b", "c")`.

6 Functions

Of course, a programming language wouldn't be much good if it didn't allow you to define your own functions, now would it?

To define your own function, use

```
sub name
{
        ...
}
```

and call it using

```
&name(arg...);
```

The body of the function can contain the same sorts of things that you can do in the main program: you can manipulate variables and use all of the flow-control constructs, as you'd expect. But you can do just about anything you like, including switching packages (we'll talk about packages later), and even define new functions.

You can define functions wherever you like: the Perl compiler will find them during the compilation phase, and make them available to your code by the time the body of the program is executed. You don't have to worry about defining functions before calling them.[17]

6.1 Local Variables

The next question is, what does the following code do?

```
$var = 1;
&myfunc;
print $var;

sub myfunc {
        # ...
        $var = 2;
}
```

The answer depends, of course, on what the `#...` is. Normally, this will set the global variable `$var` to 2. Unlike the Bourne shell, variables inside Perl functions are not automaticaly local. So if you don't say anything in the `#...` above, the `myfunc` function will set the global variable `$var` to 2, and `$var` will retain this value when `myfunc` exits.

Of course, it can be extremely handy to have local variables inside of a function. To do so, simply put

```
my $var;
```

[17] In fact, since you can define functions at runtime, you can even put in calls to functions that don't yet exist. Of course, you shouldn't do this without good reason.

or

```
my $var = 2;
```

at the beginning of your function body, and everything will work as you expect it to.[18] my variables do not get propagated outside of the enclosing block, so if myfunc had a my $var at the top, then $var would still be 1 back in the main program.

6.2 Function Arguments

Function arguments are passed to a function through the @_ array. Thus, you'll often see this sort of thing:

```
sub myfunc {
        my $arg  = shift;
        my @rest = @_;
my ($num1, $num2) = (3, 98);
```

As was briefly mentioned in the section on arrays, the shift function removes the first element from an array, and returns it. If you don't specify which array to do this to, it'll use @ARGV in the main program, or @_ in functions.

So here, the line my $arg = shift declares $arg as being local to the function, and also initializes it to the first argument.

The next line declares the array @rest to be local to myfunc, and assigns it all of the other arguments that were passed to the function.

Finally, note how to place the parentheses if you want to declare several my variables on one line. In my opinion, however, you should only declare one my variable per line, since it makes your code more readable.

6.3 Other tricks

Actually, there's another way to call functions: if the function is declared *before* you call it, the compiler already knows about it, so you don't need the & to tell it, "this is a function call." Thus, you can say

```
sub japh {
my $language = shift;
        print "Just another $language hacker\n";
}

japh "Perl";
```

And as you can see, you can also omit the parentheses around the arguments. This allows you to make your functions look like the built-in ones.

[18] You may also see scripts that use local instead of my. Unless you know what you're doing, you should use my, and I'll talk about why later on. In the meantime, if you're impatient, I'll tell you that my uses lexical scoping, whereas local uses dynamic scoping.

You can also have a stand-alone declaration in one place, and a definition someplace else.[19] As long as the declaration comes before the call, you can safely omit the &:

```
sub japh;

japh "Perl";

sub japh {
my $language = shift;
        print "Just another $language hacker\n";
}
```

If you call a function as

```
&myfunc;
```

i.e., if you leave off the arguments *and* parentheses, `myfunc` will be called with the current value of `@_`. This means that, if you were so inclined, you could write a function that manipulated its argument list, then passed it off to another function to do the real work. If the argument list is long, this can avoid copying arrays needlessly.

If you want to explicitly call a function with no arguments, use

```
&myfunc();
```

6.4 Return values

To return a value, use

`return` *value*`;`

And that's about it. *value* can be a scalar, array, or hash variable, or a literal value.

6.5 Function Prototypes

As you can tell from the above, all functions in Perl take a variable number of arguments, so you can call a function any which way, and it'll work. There are times, however, when you *do* want to be told that a certain function requires two arguments, and you're only passing it one. For this, Perl has function prototypes.

A function prototype looks like this:

```
sub myfunc ($$@) {
        my $a = shift;
        my $b = shift;
```

[19]A *declaration* merely says that the function exists, or will exist at some point. The definition specifies the body of the function, *i.e.*, which commands it executes.

```
        my @c = @_;

        ...
}
```

It's like a little picture of the way the function should be called. A `$` in the prototype means that the argument is a scalar; a `@` means that the argument is an array, and `%` means that it's a hash. Thus, `myfunc` above takes two scalars and an array.

A semicolon (`;`) separates mandatory arguments from optional ones:

```
sub settime ($$;$)
```

Here, the function `settime` takes either two or three arguments, so it can be called as

```
settime 19, 30;
```

or

```
settime 19, 30, 59;
```

but not

```
settime 19, 30, 59, 30;
```

A star (`*`) indicates a glob, and is usually used for filehandles. We haven't covered this yet, but don't worry: it won't make much more sense after we do.

A backslash (\) in front of a character indicates that that argument *must* begin with that character. That is, if you have

```
sub sort_list (\@)
```

it can be called as

```
sort_list @my_array;
```

but not

```
sort_list 19, 101, 38, 54, "hike!";
```

This will make more sense when we get to references. Trust me.

A function prototype *does* count as a declaration if you don't want to put the `&` in front of your function calls.

Having said all this, I must confess that prototypes aren't quite as useful as one might hope. As I mentioned, the prototype must come before the function call for it to have any effect. This doesn't mean that you should put all of your function definitions at the top: you can also have a stand-alone prototype at the top:

```
sub myfunc ($$@);
```

However, if you do this, you also need to include a prototype when you define the function later on.

Of course, if your function is inside of a module, the module will typically be included at the top of the main program, so you only have to maintain one prototype, which simplifies everything.

In addition, object methods (which we haven't covered yet, but which are special types of functions) aren't affected by prototypes; also, if you use `&`*func*, prototypes have no effect. The intent is to allow you to write functions that look like the built-in functions; if you stray too far from that, you don't get their benefits.

7 Built-In Functions

Perl includs a whole slew of built-in functions, just to start you off. I won't go into a lot of detail about them; this list is just to give you an idea of the sorts of things you can do, and the wheels you don't have to reinvent.

7.1 Scalar Functions

`chr` *num* Returns the character whose value is *num* in the character set you're using.

`ord` *expr* Returns the ASCII value of the first character in *expr*.

`chmod` *nnn*`,` *files...* Changes the mode of *files* to *nnn*.

`chown` *uuu*`,` *ggg*`,` *files...* Changes the owner of *files* to user *uuu* and group *ggg*.

`die` *message* Prints the value of *message*, and exits with the current value of `$!`. If *message* does not end with a newline, appends `at` *FILE* `line` *LINE*.

Inside of an `eval`, however, `die` makes the `eval` exit with the undefined value, and sets `$@` to the value of *message*. This allows you to implement exception-catching *à la* C++ or Java.

You can also catch `die`s by setting up a handler for the pseudo-signal `$SIG{__DIE__}`. It will be passed the error message as its argument. If it calls `die` again, the second error message will be printed.

`warn` *message* Prints *message* to standard error, just like `die`, but doesn't exit.

Like `die`, you can install a handler for the pseudo-signal `$SIG{__WARN__}`. However, if you do so, it is your responsibility to take any appropriate action (including printing an error message), since Perl will assume that you know what you're doing when you install such a handler.

If you want to get the default behavior of `warn` inside the handler, just call `warn` again. The hook will not be invoked recursively.

`exit` *code* — Exits the program immediately and returns an exit code of *code* to the caller.

`exec` *prog[, arg...]* — Executes the program *prog* and replaces the current program with it. That is, this function never returns.

`exec` will either pass the program to the shell, or call `execvp()` directly, depending on whether it looks as if you're passing it a shell expression or an `argv[]` list.

`system` *list* — Executes a program in the same way as `exec`, except that it `forks` first, and waits for the program to exit. The return value is the program's exit code, times 256.

`grep` *expr* `,` *list* — `grep` evaluates *expr* (often just a pattern-match) for each element in *list*, and returns the list of those for which *expr* returned a true value. Inside *expr*, `$_` is set to the current element.

`map` *expr* `,` *list* — Evaluates *expr* for each element of *list*, and returns the list of values returned by *expr*.

`sort` *expr list* — Sorts the elements of *list*, and returns the sorted list. *expr* can be either the name of a function, or a block of code.[20]

expr, be it a function or a block, is a comparison function which will have available the variables `$a` and `$b`. It should return -1 if `$a` comes before `$b`, 1 if `$a` comes after `$b`, or 0 if they're equal. The `<=>` and `cmp` operators come in really handy here.

`splice` *array* `,` *offset* `,` *[length* `,` *list]]* — Generalized list-substitution function. Replaces *length* elements of *array*, starting at *offset*, with *list*. See perlfunc(1) for the details.

`lc` *expr* — Converts *expr* to lower case.

`lcfirst` *expr* — Converts the first character of *expr* to lower case.

`uc` *expr* — Converts *expr* to upper case.

`ucfirst` *expr* — Converts the first character of *expr* to upper case.

`int` *expr* — Looks for an integer at the beginning of *expr* and returns its value.

`hex` *expr* — Interprets *expr* as a hex number (with an optional leading `0x`), and returns the corresponding value.

`oct` *expr* — Interprets *expr* as an octal number, and returns its value. Oh, and if *expr* begins with `0x`, it'll interpret it as a hex number.

[20] Which is actually an anonymous inline function

`pack` *template* , *list*
: Takes a list of values and converts them to a binary string. You can use this to write binary files that will be read by some other program.

  template is too complex to describe here. See perlfunc(1) for the gory details.

`unpack` *template* , *expr*
: The reverse of `pack`. Takes *expr*, parses it according to *template*, and returns the corresponding list of values.

  You can use this to read binary files.

`length` *expr*
: Returns the string length of *expr*.

`printf` *[FILEHANDLE] format* , *list*
: C's `printf()`, in case you need to, say, print a floating-point number with a certain number of decimals. In general, however, you should prefer `print` over `printf`.

`sprintf` *[FILEHANDLE] format* , *list*
: Like C's `sprintf()`, except that it returns the resulting value.

`reverse` *list*
: Reverses an array or string.

`select` *[FILEHANDLE]*
: Sets or returns the default filehandle. This is handy for setting `$~` and such.

`substr` *expr* , *offset* , *[len]*
: Returns the *len*-length substring of *expr* that begins at *offset*. If *offset* is negative, it refers to the position that far from the end of *expr*.

`sysopen` *filehandle* , *filename* , *mode* , *[perm]*
: Opens a file, but gives you all the power of open(2).

`sysread` *filehandle* , *var* , *len* , *[offset]*
: Uses read(2) to read from a file. Typically, if you need to use this, you'll want to feed the results to `unpack`.

`syswrite` *filehandle* , *var* , *len* , *[offset]*
: Uses write(2) to write to a file. Typically, *var* will be the output of `pack`.

`study` *var*
: Makes Perl take extra time to study the pattern *var*, to optimize matching it. This may or may not save time.

`index` *str* , *substr [, pos]*
: Returns the position of the first occurrence of *substr* in *str* at or after *pos*, or -1. If *pos* is omitted, starts at the beginning of *str*.

`rindex` *str* , *substr [, pos]*
: Just like `index`, but returns the last occurrence of *substr* within *str*.

`pos` *var*
: Returns the position where the last `m//g` over *scalar* left off.

`vec` *expr* , *offset* , *bits*
: Treats *expr* as an array of bit-fields, and allows you to get or set values.

`getlogin` Looks up the current username in */etc/utmp* and returns it.

`glob` *expr* Same as <*file pattern*>.

`reset` *[expr]* Resets certain matches, or variables beginning with *expr*. Deprecated.

7.2 Unix Functions

The following functions do pretty much the same thing as the standard Unix functions of the same name.

abs	gethostbyname	getsockopt	rename	sleep
accept	gethostent	gmtime	rewinddir	socket
atan2	getnetbyaddr	ioctl	rmdir	socketpair
bind	getnetbyname	kill	seek	sprintf
chdir	getnetent	link	seekdir	sqrt
chroot	getpeername	listen	select	srand
connect	getpgrp	log	semctl	stat
cos	getppid	lstat	semget	symlink
crypt	getpriority	mkdir	semop	syscall
exp	getprotobyname	msgctl	send	tell
fcntl	getprotobynumber	msgget	setpgrp	telldir
fileno	getprotoent	msgrcv	setpriority	time
flock	getpwent	msgsnd	setsockopt	times
fork	getpwnam	opendir	shmctl	truncate
getc	getpwuid	pipe	shmget	umask
getgrent	getservbyname	rand	shmread	unlink
getgrgid	getservbyport	readdir	shmwrite	utime
getgrnam	getservent	readlink	shutdown	wait
gethostbyaddr	getsockname	recv	sin	waitpid