Perl in 20 pages

A guide to Perl 5 for C/C++, awk, and shell programmers

Russell Quong

November 2, 1998

Document version 98c

Keywords: Perl documentation, Perl tutorial, Perl beginners, Guide to Perl. (For internet search engines.)

1 Introduction

Perl is an interpreted scripting language with high-level support for text processing, file/directory management, and networking. Perl originated on Unix but as of 1997 has been ported to numerous platforms including the Win32 API (on which Win95/NT are based). It is the defacto language for CGI scripts. If I had to learn just one scripting language, it would be Perl.

This document is not meant to be a thorough reference manual; instead, see the concisely-written manual pages ("man pages") or buy the Perl book (Programming Perl 2nd Edition, by Wall, Christianson and Schwartz, ISBN 1-56592-149-6 [1]. This document attempts to help an experienced programmer unfamiliar with Perl up to speed as quickly as possible on the most commonly used features of Perl.

I am willing to sacrifice 100% correctness if there is a much simpler view that is correct 99% of the time. There are several reasons for taking this approach (I need to finish this paragraph).

My Perl programming philosophy emphasizes reuse and clarity over brevity. We happily acknowledge that much of the Perl code presented could easily be written in half the number of lines of code and with greater efficiency.

1. I name variables and avoid using the implicit `$_` or `@_` variables whenever possible.
2. I use subroutines to hold all code.
3. I use local variables and avoid globals whenever possible.

The latest version of this document [2] can be found at http://www.best.com/~quong/perlin20/. A gzipped 2-up (US letter) Postscript version of this document [3] is available, too. I expect to update this document every 4–6 months.

As of 11/98 this document is about 95% complete.

1.1 Perl Versions

This document covers Perl version 5. If you have an older version, upgrade immediately. As of 2/98, version 5.004 (and as of 9/98, 5.005) is available[4]. Run `perl -v` to see the version.

Before version 5, Perl was a cryptic language in large part to its use of variables. In Version 4 most built-in variables were named via single punctuation symbols, such as `$]`, `$_` and, even worse, most statements operated on an implicit variable, named `_` (yes, the variable named underscore) to increase

[1] Like the K&R book on C, this definitive reference on a popular language is dense and insightful, but not for all tastes

[2] `http://www.best.com/~quong/perlin20/perlin20.html`

[3] `http://www.best.com/~quong/perlin20/perlin20.2.ps.gz`

[4] I used 5.003 when initially writing this document 4/98

brevity. In Perl 5, released sometime in late 1995 (?), most of built-in variables now have descriptive english names and all statements can be rewritten to show explicitly the variables being used.

2 Obtaining Perl binaries, documentation

Check CPAN (the Comprehensive Perl Archive) for any Perl related material, documentation, source or modules. If anything, there is too much information at CPAN. CPAN is mirrored at many (over 40) different sites [5]. Pick one near you.

As of 3/98, the latest Win32 port of Perl is available at any of the CPAN sites at `authors/Gurusamy_Sara`
For example, here are a few locations

- USA West coast mirror [6].
- USA East coast mirror [7].

On a Unix system or Win32 system with Perl properly installed,

3 Basics

Perl is a polymorphic, interpreted language with built in support for textual processing, regular expressions, file/directory manipulation, command execution, networking, associative arrays, lists, and dbm access. We next present three increasingly complicated examples using perl

4 Command line usage: substituting text

In some cases, a script is not needed. For example, I often want to replace all occurrences of a regex (regular expression) `FROMX` to a new value `TOX` in one more files `FILESX`.

Here's the command:

```
## replace FROM with TOX in all files FILESX, renaming originals with .bak
% perl -p -i.bak -e "s/FROM/TOX/;" FILESX

## replace FROM with TOX in all files FILESX, overwriting originals
% perl -p -i -e "s/FROM/TOX/;"  FILESX
```

5 A simpler one-shot script

Sometimes you need a simple throw-away script to do a task once or twice, in which case the full-blown script in the next section is just too much. The following script one-shot.pl reads all files specified as command line arguments and prints out each line preceded by the file name and the line number. You may need to make the script file executable (via the Unix command `chmod 755 oneShot.pl`) first.

```
#! /usr/bin/perl -w
use English;

sub main () {
    my($filename, $line, $lineno) = (" / \\ :", undef, 0);
```

[5] `ftp://ftp.cdrom.com/pub/perl/CPAN/SITES.html`

[6] `ftp://ftp.digital.com/pub/plan/perl/CPAN/authors/Gurusamy_Sarathy/`

[7] `ftp://ftp.epix.net/pub/languages/perl/authors/Gurusamy_Sarathy/`

```
    while ( defined($line=<>) ) {
        if ($ARGV ne $filename) {       # detect when we switch files
            $lineno = 0;                # reset the line number
            $filename = $ARGV;          # $ARGV = current file name
        }
        $lineno ++;             # increment the line number
        chomp($line);           # stop off newline from the line
          ## print FILENAME, LINENUMBER: line=(LINE)
        print "file=$filename, $lineno: line=($line)\n";
    }
}

main();
0;
```

To run the script type

```
% oneShot.pl input-file(s)
```

or

```
% perl -w oneShot.pl input-file(s)
```

5.1 A prototype Perl script

We present a non-trivial prototype Perl script that illustrates many common Perl script operations, including

- command line flag handling
- variables, defining/calling functions, parameter syntax
- read multiple files
- write the results to a file
- text searching and matching using regular expressions,
- sorting an array of strings alphabetically

If this script is too much for your needs, use the preceding prototype script for simpler one-shot tasks in the next section. Remember, though, its much easier to remove parts from a big script than to add to a smalll script.

By breaking each of the majors steps into a separate function, you can modify this prototype script for your needs with minimial changes. Although this script is (too) long, it should be fairly easy to read. This example script `proto-getH1.pl` extracts and then sorts (alphabetizes) all the high-level headings from one or more HTML files, by looking for lines that contain

```
<Hn> ... </Hn>
```

This script proto-getH1.pl is run via:

```
% perl -w proto-getH1.pl [-o outputfile] input-file(s)
```

or

```
% proto-getH1.pl [-o outputfile] input-file(s)
```

All HTML headers are sent to the *output file*, which is `stdout` by default, or the file specified after the `-o` command line flag.


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

# Example perl file - extract H1,H2 or H3 headers from HTML files
# Run via:
#   perl this-perl-script.pl [-o outputfile] input-file(s)
# E.g.
#   perl proto-getH1.pl -o headers *.html
#   perl proto-getH1.pl -o output.txt homepage.htm
#
# Russell Quong         2/19/98

require 5.003;                  # need this version of Perl or newer
use English;                    # use English names, not cryptic ones
use FileHandle;                 # use FileHandles instead of open(),close()
use Carp;                       # get standard error / warning messages
use strict;                     # force disciplined use of variables

## define some variables.
my($author) = "Russell W. Quong";
my($version) = "Version 1.0";
my($reldate) = "Jan 1998";

my($lineno) = 0;                # variable, current line number
my($OUT) = \*STDOUT;            # default output file stream, stdout
my(@headerArr) = ();            # array of HTML headers

  # print out a non-crucial for-your-information messages.
  # By making fyi() a function, we enable/disable debugging messages easily.
sub fyi ($) {
    my($str) = @_;
    print "$str\n";
}

sub main () {
    fyi("perl script = $PROGRAM_NAME, $version, $author, $reldate.");
    handle_flags();
      # handle remaining command line args, namely the input files
    if (@ARGV == 0) {           # @ARGV used in scalar context = number of args
        handle_file('-');
    } else {
        my($i);
        foreach $i (@ARGV) {
            handle_file($i);
        }
    }
    postProcess();              # additional processing after reading input
}

  # handle all the arguments, in the @ARGV array.
  # we assume flags begin with a '-' (dash or minus sign).
  #
```

```
sub handle_flags () {
    my($a, $oname) = (undef, undef);
    foreach $a (@ARGV) {
        if ($a =~ /^-o/) {
            shift @ARGV;                # discard ARGV[0] = the -o flag
            $oname = $ARGV[0];          # get arg after -o
            shift @ARGV;                # discard ARGV[0] = output file name
            $OUT = new FileHandle "> $oname";
            if (! defined($OUT) ) {
                croak "Unable to open output file: $oname.  Bye-bye.";
                exit(1);
            }
        } else {
            last;                       # break out of this loop
        }
    }
}

  # handle_file (FILENAME);
  #   open a file handle or input stream for the file named FILENAME.
  # if FILENAME == '-' use stdin instead.
sub handle_file ($) {
    my($infile) = @_;
    fyi(" handle_file($infile)");
    if ($infile eq "-") {
        read_file(\*STDIN, "[stdin]");  # \*STDIN=input stream for STDIN.
    } else {
        my($IN) = new FileHandle "$infile";
        if (! defined($IN)) {
            fyi("Can't open spec file $infile: $!\n");
            return;
        }
        read_file($IN, "$infile");      # $IN = file handle for $infile
        $IN->close();           # done, close the file.
    }
}

  # read_file (INPUT_STREAM, filename);
  #
sub read_file ($$) {
    my($IN, $filename) = @_;
    my($line, $from) = ("", "");
    $lineno = 0;                        # reset line number for this file
    while ( defined($line = <$IN>) ) {
        $lineno++;
        chomp($line);                   # strip off trailing '\n' (newline)
        do_line($line, $lineno, $filename);
    }
}

  # do_line(line of text data, line number, filename);
```



```
  #   process a line of text.
sub do_line ($$$) {
    my($line, $lineno, $filename) = @_;
    my($heading, $htype) = undef;
    # search for a <Hx> .... </Hx>  line, save the .... in $header.
    # where Hx = H1, H2 or H3.
    if ( $line =~ m:(<H[123]>)(.*)</H[123]>:i ) {
        $htype = $1;            # either H1, H2, or H3
        $heading = $2;          # text matched in the parethesis in the regex
        fyi("FYI: $filename, $lineno: Found ($heading)");
        print $OUT "$filename, $lineno: $heading\n";

          # we'll also save the all the headers in an array, headerArr
        push(@headerArr, "$heading ($filename, $lineno)");
    }
}

  # print out headers sorted alphabetically
  #
sub postProcess() {
    my(@sorted) = sort { $a cmp $b } @headerArr;        # example using sort
    print $OUT "\n--- SORTED HEADERS ---\n";
    my($h);
    foreach $h (@sorted) {
        print $OUT "$h\n";
    }
    my $now = localtime();
    print $OUT "\nGenerated $now.\n"

}
 # start executing at main()
 #
main();
0;              # return 0 (no error from this script)
```

6 Control constructs

Perl has the similar syntax as C/C++/Java for control constructs such as `if`, `while`, `for` statements. The following table compares the control constructs between C and Perl. In Perl, the values `0`, `"0"`, and `""` (the empty string) are false; any other value is true when evaluating a condition in an if/for/while statement.

	C	Perl (braces required)
same	`if () { ... }`	`if () { ... }`
diff	`} else if () { ... }`	`} elsif () { ... }`
same	`while () { ... }`	`while () { ... }`
diff	`do  while ();`	`do  while (); (See below)`
same	`for (aaa;bbb;ccc) { ... }`	`for (aaa;bbb;ccc) { ... }`
diff	N/A	`foreach $var (@array) { ... }`
diff	`break`	`last`
diff	`continue`	`next`
similar	0 is FALSE	0, "0", and "" is FALSE
similar	$\neq$ 0 is TRUE	anything not false is TRUE

Note in Perl, the curly braces around a block are required, even if the block contains a single statement. Also you must use `elsif` in Perl, rather than `else if` as shown below.

```
if ( conditionAAA ) {
   ...
} elsif ( conditionBBB ) {
   ...
} else {
   ...
}
```

Finally, although the `do { body } while (...)` is legal Perl, it is not an actual loop construct in Perl. Instead, it is the `do` statement with a `while` modifier. In particular, `last` and `next` will not work inside the body.

7 Variables

There are four types of data in Perl, scalars, arrays, hashes and references. Scalars and arrays are ubiquitous (used everywhere). Hashes are common in large programs and not unusual in smaller programs. References are an advanced topic and can be completely ignored initially. In the following listing, the initial symbol is the context specifier for that type.

1. (`$`) A *scalar* is a single string or numeric value. More advanced scalar types include references, and typeglobs.
2. (`@`) A *list* or *array* is a one-dimensional vector of zero or more scalars. Arrays/lists are indexed as arrays via [ ]; the starting index is 0, like C/C++. The Perl reference documentation intermixes the terms *list* and *array* freely; so shall we.
3. (`%`) A *hash* is a list of *(key, value)* pairs, in which you can search for a particular *key* efficiently. In practice, a hash is implemented via in a hash table, hence the name.
4. (\ ) A *reference* refers to another value, much like a pointer in C/C++ refers to some other value.

7.1 Scalar types

A scalar holds a single value; an array or list holds zero or more values. The scalar *types* in Perl are string, number, and reference[8]. Like `awk`, a scalar data value in Perl contains either a string or a (floating point) number. For reference we create scalars of all four types.

[8]There is also a symbol table entry scalar type, poorly named a *typeglob* in Perl, but you are not likely to use it initially


```
$numx = 3.14159;                # numeric
$strx = "The constant pi";      # string
$refx = \$numx;                 # reference
$tglobx = *numx;                # typeglob (different from file name globbing)
```

A numeric value is a real or floating point value and can use any of the standard C specifications, e.g. (1.2, 12+e

A string value is enclosed in matching single or double quotes. Within double quotes, variable references (but not expressions involving operators) are evaluated, like `t/csh`; within single quotes nothing is evaluated. Double quotes are especially convenient when printing out values.

```
$i = 123;
print('i = $i\n');                         # print: i = $i\n
print("i = $i\n");                         # print: i = 234
print("i = $i+4\n");                       # print: i = 123+4
print("i = " . ($i+4) . "\n");             # print: i = 127
print("i = " . $i+4 . "\n");               # print: 4
print((("i = " . $i) + 4) . "\n");         # print: 4 (same as previous)
```

7.2 String or number

Perl automatically converts from string to number or vice versa as needed, based on the operation being done. Below, + is arithmetic plus and `.` is string concatenation.

```
$pi = "3.14";
$two_pi = 2 * $pi;              # $two_pi = 6.28
$pi_pi = $pi . $pi;             # $pi_pi = "3.143.14"
```

The following table shows that a non-numeric string value is viewed as 0 (zero), and a numeric value viewed as a string is the ASCII representation of the number.

Type of $x	(Value of) $x	$x+1	$x . "::"	if ($x) {
string	"abc"	1	abc::	true
number	3	4	3::	true
string	"45.0"	46	45.0::	true
number	0	1	0::	false
string	""	0	::	false
undefined	""	0	::	false

Because strings are converted to numbers on demand and vice versa, there is no practical difference between a number and its string equivalent. Thus, in the following statements `i` and `j` are assigned the same value.

```
$i = 3;         # same as $i = "3"
$j = "3";       # same as $j = 3
$k = $i + $j;   # $k = 6
$s = $i . $j;   # $s = "33"
$f = "3.0"      # not the same as "3" as $f . 1 would give "3.01"
```

7.3 Null string/zero versus no value

A scalar variable that has a valid string or numeric value, such as 4.3 or "hello" or even "" (the empty string), is *defined*. In contrast, if a variable without a valid value is *undefined*. The builtin value `undef` represents this undefined value, much like `NULL` in C/C++, `null` in Java or `nil` in Lisp/Ada are undefined values. An array is defined if has previously data,

held, the empty array () is undefined; all other array values are considered defined. Use the `defined()` function to test if a variable is defined.

```
my($emptystr) = "";
my(@nonemptylist) = ( undef );
if ( defined($emptystr) && defined(@nonemptylist) ) {
   print "will see this\n";
}
my($invalid);
my(@empylist) = ();
if ( defined($invalid) || defined(@emptylist)) {
   print "will NOT see this\n";
}
@emptylist = (1, 2);
@emptylist = ();
if ( defined(@emptylist)) {
   print "emptylist is empty but is defined now\n";
}
```

If you read or access an undefined variable `var` as a string or number, you get the undefined value, which is then converted to `""` or `0`. Thus an undefined variable is considered false.

7.4 Operators

Most Perl operators, such as + or < or `.` work either on numbers or on strings but not both.

Description	string op	numeric op
equality	`eq`	`==`
inequality	`ne`	`!=`
ternary compare	`cmp`	<=>
concatenation	`.` (a dot)	N/A
arithmetic	N/A	`+, -, *, /`
relational	`lt, le, gt, gt`	<, <=, >, >=
ANSI C ops		

ASCII strings are ordered character by character based on the underlying ASCII value. For purely alphabetic strings, this results in normal alphabetization, as A < B < ... < Z < a < b < ... < z. In general, strings are ordered using the local collating property. The ternary compare operations `xx cmp yy` or `xx <=> yy`, returns -1, 0, or 1 if `xx` is less than, equal or greater than `yy` for strings and numbers respectively.

7.5 Lists/arrays

A list/array is a one-dimensional vector that holds zero or more values. To Perl, lists and arrays are identical, and we shall use the terms interchangably, using the poor justification the existing documentation does so, too. In Perl, a list/array value is denoted by scalars enclosed in parethesis. Arrays can be indexed; like C/C++/Java, the first element has index 0.

```
@fib = (0, 1, 1, 2, 3, 5);
@mixed = ("quiet", +4, 3.14, "hot dog");
@empty = ();
@emptyAlso = ( (), (), () );
$five = pop @fib;              # get $five
$three = $fib[4];
```

The length or size of an array is can be obtained in two different ways.

```
$len = @array          ## need SCALAR CONTEXT.  Number of items in the array.
$last_index = $#array    ## index of last element in the array.
```

Finally, here are three ways to iterate through an array, `@arr`. In this example, we simply print out each element. For accessing each element, I prefer `foreach`; if the index is needed too, I use the second method.

```
my($item);
foreach $item (@arr) {          ## cleanest, but no index
  print $item;
}
my($i);
for ($i=0; $i<@arr; $i++) {     ## just like C
  print $arr[$i];
}
my($j);
for ($j=0; $i<=$#arr; $j++) {   ## I don't use this much
  print $arr[$j];
}
```

The next block shows some common array operations. Push and pop add/remove elements at the right-end of the array. We show how to construct the list ("one1", "two2", "three3", "four4") in the following steps.

```
@list = ("one1");
push(@list, "two2");
$list[2] =  "three3";
$nelements = @list;              # get three, as there are three elements
$list[$nelements] = "four" . "4";
```

Perl automatically and dynamically enlarges an array so you do not have predeclare the size of an array. However, if you know you will need a very large array, `largeArr`, you can pre-allocate space by assigning to `$#largeArr`. Pre-allocating is slightly more efficient, but potentially wastes a lot of space, and should only be done for arrays bigger than 16K elements.

```
$#largeArr = 987654;          ## preallocate 987K worth of space.
```

7.6 Hashes

A *hash* variable stores a map of `(key, value)` pairs. Typically, the key and value are different but related values, such as a person's name and phone number. A *hash* is implemented in Perl so that you can quickly look up the value given the key, when there are many (key, value) pairs. From a computer science data structures standpoint, a Perl hash implements a dictionary.

For example, given the name of a state, such as `california`, I want the Postal abbreviation, `CA`. We define, initialize, and modify, and use a hash, `%abbrevTable` as follows.

```
my(%abbrevTable) = (            # this is the initialization syntax.
    "california" => "CA",       # key = california, value = CA
    "oregon" => "OR",
);
sub printAbbrev($) {
    my($state) = @_;
    if (exists $abbrevTable{$state}) {
        print "Abbreviation for $state = $abbrevTable{$state} \n";
    } else {
        print "No known abbreviation for $state\n";
    }
}
sub hashdemo () {
    printAbbrev("arizona");             # no such key
    $abbrevTable{"arizona"} = "AZ";     # add a new (key, value) pair
    printAbbrev("arizona");             # this will succeed
}
```

Calling the function `hashdemo()` gives

```
No known abbreviation for arizona
Abbreviation for arizona = AZ
```

Note that we use the `exists $hash{$key}` syntax to test if a key exists in the hash table. Also a hash is assymetric in that we can lookup up entries based on the key, not the value.

If treated as an normal array/list, a hash will appear as

```
(keyA, valueA, keyB, valueB, keyC, valueC, ... ).
```

The order of the keys will appear random[9].

7.7 Variables declaration

Declare local variables using the `my(var-decl)`, which creates list context, or `my scalar-var`, which creates scalar syntax. A local variable only exists in and can only be used in the function (or block) where it was declared.

```
sub some_function () {
  my($i, $mesg) = (0, "hi");    # local variables for some_function
  foreach $i (@ARGV) {
    my($arg) = $ARGV[$i];       # $arg only exists in the for loop
  }
  print $arg;                   # Arghh.  ERROR, $arg does not exist here.
}
```

[9]The key order is based on the underlying hash function being used, we are simply listing the hash table buckets.

In older Perl code, you may see the `local` keyword instead of `my`. If in doubt, use `my` instead of `local`[10]. A `local` variable is dynamically-scoped[11]; a `my` variable is statically-scope, which is faster and almost certainly what you want. For example, C/C++/Java use static scoping.

7.8 Barewords

A *bareword* is a unquoted literal not used as a variable or function name. Barewords are used mainly for labels and for filehandles [12]. The following code snippet shows three bare words, `A_FILE_HANDLE`, `bare` and `bareword`. filehandles are uppercase to avoid naming conflicts, and to follow the normal Perl naming convention. (If you use the FileHandle package, you don't need to make your own file handles.)

```
open(A_FILE_HANDLE, "./perlscript.pl");
bare: while ($line = <A_FILE_HANDLE>) {
  bareword: while ($line[$i] ne "") {
    if ($line[$i] =~ /\s*#/) {
      next bare;
    }
  }
}
```

A bareword not used as a filehandle or label, and which is not a known function, is viewed as string constant.

```
$str = hi;            # AVOID.   Use of bareword hi, same as "hi".
$str = "hi";          # same, but much easier to read.
```

We advise against use barewords as strings, since it impedes clarity, as function calls are typically barewords. Instead, put your strings in double quotes, which is standard across most languages.

8 Context: scalar, list, hash or reference

A context specifier, which is one of the characters $, @, % must be used before all variable references. The context indicates the kind value that will be used or assigned. The context is not part of a variable name. Consider the following assignment statements.

```
$eight = 8;                     # numeric scalar
@nulllist = ();                 # null or empty list.
$four = $eight / 2;             #
@cubes = (1, 8, 27, 64);        # assign an entire array/list.
$eight = $cubes[1];             # huh?  cubes is an array, why not @cubes[1].
```

[10]There are advanced situations, beyond the scope of this document, where `local` must be used.

[11]With dynamic scoping, we use the variable in the closest function-call stack frame, which means that the same line of code might use different non-local variables as it depends on the function call nesting.

[12]and for package names, but this is an advanced topic

The $ specifier in the statement …= *$varX* … means that we expect to read a scalar value from a variable named `varX`. Thus, Perl uses the scalar variable named `varX`. Similarly, the `@` specifier in …= *@varX* means that we expect to read an array/list value from a variable `varX`; Perl uses the array/list variable `varX`.

While it might seem that the `$` and the `@` are part of the variable names in `$varX` and `@varX`, **this view is wong**. In reality, there are two different variables, each named `varX`; one is a scalar, the other an array. In an expression like `varX[...]`, because array subscripting is used, Perl selects the array variable. The last statement in the preceding example `$eight = $cubes[1];` illustrates the preceding rule.

An expression like `@aaa = @bbb[$ccc]` means that we expect the element bbb[$ccc] to produce an list/array value, which is probably wrong thinking. Since Perl arrays elements must be scalars, `@bbb[$ccc]` results in a one-element array containing $bbb[$ccc], namely `( $bbb[$ccc] )`.[13]

In an expression like `...= $varX[kk]`, we first interpret the array brackets, which means *varX* must be an array. We get the *kk*th element. Finally the leading `$` specifier indicates we expect this element to be a scalar value.

What happens if the LHS and RHS contexts do not match in an assignment statement? Perl uses the following rules which are often convenient but sometimes unexpected.

Value assigned to LHS in LL = RR			
LHS	Original RHS Value		
Value	Scalar $RR	List @RR	Hash %RR
	`"hi"`	`(1, 4, 9)`	`("one", 1, "two", 2)`
scalar, $LL	`"hi"`	9 [last element]	used/alloc
list, @LL	`("hi")`	`(1, 4, 9)`	`("one", 1, "two", 2)`
hash, %LL	[empty hash]	`(1, 4)`	`("one", 1, "two", 2)`

Variables of different types (scalar, list, hash) can have the same name, because each type has its own namespace. Thus, the following code refers to three different variables, so that no data values are overwritten.

```
$xyz = "my foot";                                # scalar mode variable
@xyz = ("tulip", "rose", "mum is the word");     # list mode variable
%xyz{$xyz} = $xyz[1];                            # %xyz{"my foot"} = "rose";
```

Even the Perl book is misleading as it states that "all variable names start with a `$`, or `%`," (page 37) which would imply that `$cubes[1]` is using the `$cubes` variable, *which is incorrect*. (It is accurate to say that all variable uses begin with a $, @ or a %

The condition of an if-statement or while-loop is evaluated in scalar context. Thus it is acceptable and indeed common Perl programming practice to say

```
if ( @array > 4 ) {            ## @array ==> number of items in it.
   ...
}
```

Many functions and operators behave differently depending on the context. Beware that using `my($var)` produces a list context, because the parenthesis denote a list. Thus, to get a single string of the current time here several correct ways. The following table shows some commonly encountered cases.

```
my($now1) = scalar(localtime());      # CORRECT, force scalar context
my $now2 = localtime();               # CORRECT, no parens, scalar context
```

[13] If `$bbb[$ccc]` is undefined, we get the array `( undef )`


```
my($now3);
$now3 = localtime();                  # CORRECT,
my($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime();  # OK
my($nowWRONG) = localtime();          # WRONG, list context, get $sec
```

8.1 Forcing scalar or list context

Use the `scalar(...)` function to force scalar context. Use `(...)` to force array/list context.

```
$scalarVar = scalar(@arrayVar);       # force scalar context.
my($line) = scalar( < file > );       # just read one line
```

9 Functions

9.1 Calling functions

Perl functions take a single list/array as a parameter, which naturally handles the case of passing several scalars. Parameters are separated by commas, because they are separate elements of the parameter list/array.

```
$two = sqrt 4.00;                  # square root of 4
open FILEHANDLE, "input.txt";      # open the file input.txt for reading
$i = index "abcdefg", "cde";       # index of substring cde in abcdefg
print "i = $i \bsl n";             # print value of i
if (defined $somevar) { ... }      # test if $somevar has been used
```

You may optionally put parenthesis around the arguments, resulting in the standard call-syntax of most langauges as shown below. I personally prefer using parenthesis. However, I prefer no parenthesis if the function call is the entire conditional of an `if` or `while` statement.

```
$two = sqrt(4.00);                 # square root of 4
open (FILEHANDLE, "input.txt");    # open the file input.txt for reading
$i = index("abcdefg", "cde");      # index of substring cde in abcdefg
print ("i = $i \bsl n");           # print value of i.
if (defined($somevar)) { ... }     # test if $somevar has been used (ugly)
```

A few functions, such as `print`, `grep`, `map`, and `sort` have secondary syntaxes that require spaces after the first parameter. If you use parenthesis around the arguments, you must still use a space.

```
print STDERR "i = $i \bsl n";          # print value of i to STDERR
print(STDERR "i = $i \bsl n");         # print value of i to STDERR
print(STDERR, "i = $i \bsl n");        # (ACK) print 'STDERR' followed by i
```

Beware that the first set of outermost parenthesis fully delimit the parameters, so that subsequent values are not parameters. Whitespace does not affect things.

```
$ten = sqrt (1+3)*5;               # Ack. same as $ten = (sqrt(4)) * 5;
$ten = 5 * sqrt (1+3);             # Arithmetically the same as preceding.
$n = sqrt ((1+3)*5);               # Good.  $n = sqrt (20);
```

9.2 Defining functions

A function definition looks as follows. The parameters are passed in the `@_` list/array. This is one time where use of this cryptic variable cannot be avoided. I always immediately rename the parameters as shown in the prototype code.

```
sub do_line ($$$) {
    my($line, $lineno, $filename) = @_;
    ...
}
```

As of Perl 5.002, you can pre-declare the number and types of the function parameters (see *Prototypes* section in the perlsub manpage) using a function prototype, so that the parameters can be interpreted in a user specified manner. In the function declaration `sub do_line ($$$) {`, each of the $ signifies a single scalar parameter. A @ in the parameter list signifies a list; nothing can follow it as the list parameter gobbles up all remaining parameters. Warning: the function-prototype for a function *fn* must be seen before calling *fn* for Perl to do parameter checking.

A Perl function can return any type of value including a scalar, an array, or nothing (void). Unfortunately, the return type of a function cannot be specified in the function prototype. If a function returns one type, say an array, and you expect a scalar, Perl will silently do a conversion.

You can write functions that return different types based on expected return type (known as *the calling context)* by using the `wantarray` function. For example,

```
sub scalarOrList () {
    return wantarray ? ("red", "green", "blue") : 88;
}
  ...
  $i = scalarOrList();             # scalar context, get 88
  @color = scalarOrList();         # list context, get ("red", "green", "blue")
```

10 Regular Expressions

10.1 Symbols, syntax

In regular expressions, Perl understands the following convenient character set symbols which match a single character. Thus, to handle arbitrary blank space you must use `\s+`. You may use these symbols in a character set. For example, when looking for a hex integer you might look for `[a-fA-F\d]`. Also, the term *regex* is short for regular expressions.

Symbol	Equiv	Description
\w	`[a-zA-Z0-9_]`	A "word" character (alphanumeric plus "_")
\W	`[^a-zA-Z0-9_]`	Match a non-word character
\s	`[ \t\n\f\r]`	Match a whitespace character
\S	`[^\s]`	Match a non-whitespace character
\d	`[0-9]`	Match a digit character
\D	`[^0-9]`	Match a non-digit character

Perl has the standard regex quantifiers or closures, where r is any regular expressions.

r*	Zero or more occurences of r (greedy match).
r+	One or more occurences of r (greedy match).
r?	Zero or one occurence of r (greedy match).
r*?	Zero or more occurences of r (match minimal).
r+?	One or more occurences of r (match minimal).
r??	Zero or one occurence of r (match minimal).

Let q be a regex with a quantifier. If there are many ways for q to match some text, a greedy quantifier will match (or "eats up") as much text as possible; a minimal matcher does the opposite. If a regex contains more than one quantifier, the quantifiers are "fed" left to right.

10.2 Searching and substituting

The two main regex operations are searching/finding and substituting. In searching, we test if a string contains a regular expression[14]. In substituting, we replace part of the original string with a new string; the new string is often based on the original. Both of these operations use the regular expression operator $\boxed{=\sim}$, which consists of two characters. This operator is not related to either equals = or ~[15]

Searching: For example, to determine if the string `$line` *contains* a recent year such as 1998 or 1983, we use the search operator =~ `/.../`. Here the slashes '/' *delimit* or mark the beginning and the end of the regular expression.

```
if ($line =~ /19[89]\d/) {
  # we found a year in $line
}
```

In general, to determine if string `$var` contains the regular expression *re* use any of the following forms. If the regular expression contains a slash '/' itself, then you must use `m`X`\textslre`X form, where each X is the same single character not appearing in *re*.

In `mX...X`, the `m` stands for "match".

```
if ($var =~ /re/) { ... }
if ($var =~ m:re:) { ... }      # can replace ':' with any other character
while ($var =~ m/re/) { ... }  # can replace '/' with any other character
```

To access the substring in `$var` matched by part of the regular expression *re*, put the part of *re* in parenthesis. The matched text is accessible via the variables $1, $2, ..., $$k$, where $$k$ matches the k-th parenthesized part of the regular expression. For example to break up an e-mail address `user@machine` in `$line` we could do

```
if ($line =~ /(\S+)@(\S+)/) {          # \S = any non-space character
    my($user, $machine) = ($1, $2);
    ...
}
```

The variables $1, $2, ... $$k$ are updated after each *successful* regex operation, which wipes out the previous values. I store these values into other well-named variable immediately after the regex operation, if I want them.

[14]"Regex searching" is often incorrectly called "regex matching".

[15](1) The choice of symbols was quite confusing to me initially. (2) The =~ is officially called the "binding operator", as there are other non-regex operations that use it.

Use \ k (not $\$k$) in the regular expression itself to refer to a previously matched substring. For example, to search for identical begining and ending HTML tags <xyz> ... </xyz> on a single line $line use

```
if ($line =~ m|<(.*)>(.*)</\1>|) {       # search for: <xyz>stuff</xyz>
   my($stuff) = $2;
   ...
}
```

Substitution: To replace or substitute text in $var from the regular expression *old* to *new* use the following form.

```
$var =~ s/old/new/;                     # replace old with new
if ($var =~ s|old|new|) { ... }         # replace ':' with any other character
```

The use part of the actual text matched by the *orig* regex, the *new* regex can use the $\$k$ variables.
Taking our previous example involving years, to replace the year 19xy with xy, use

```
$line =~ s/19(\d\d)/$1/;
```

Modifiers: When searching or substituing, there are several optional modifiers you can use to alter the regular expression. For example, in if ($var =~ /<title>/i), the i at the end specifies a case-insensitive search. We use m// and s/// to represent searching and substituing.

Option	Where	What
i	m//,s///	case insensitive (upper=lower case) pattern
m	m//,s///	$var as multiple lines
g	s///	replace all orig with new. I.e. apply repeatedly.
g	m///	(Adv) search for all occurences. On next evaluation, continue where previous search
s	m//,s///	(Adv) treat $var as a single line, even if imbedded '\n' chars
x	m//,s///	(Adv) allow extended regex syntax. Ignore spaces in the regex (for readability)

The regex operations return different results depending on the context. For clarity, I recommend using the scalar context

context	return value
scalar	true, if there was a match (or substitution)
list/array	list of sub-matches ($1, $2, ...) found in the match

11 Built-in Perl functions

Perl has many built-in functions.

There are numerous ways to access documentation about Perl functions.

- On a Unix system with Perl installed, run %man perfunc.
- On a Win 95 PC with standard Perl installed in perldir on, look at perldir/lib/Pod/perlfunc.ht

Here are some of the more common functions I've used. If the function has additional options for a function, the description starts with a (+).

@arr=split(/[t:]+/, $line);	(+) Split $line into words. Words are seprated by spaces or colons (but not tabs). Store words in @arr, spaces and colons are discarded.
@arr = stat(filename);	Returns a 13 element list ($dev, $ino, $mode (permissions on this file), $nlink, $uid, $gid, $rdev, $size (in bytes), $atime, $mtime (last modification time), $ctime, $blksize, $blocks) containing information about a file.
$str = join("::", @arr);	Concatenate all elements of @arr into a single scalar string; separate all the elements by a double colon. Useful when printing out an array.

11.1 File tests

Perl has several functions which test properties about files. These functions have the name -X, for some character X. (Yes, the function name starts with a dash.) These names mimic the Unix csh and the Unix sh test operations. These functions take a filename or a file handle, as in -X filename.

For example, if you want to run a command /bin/ccc on the data file ../input/ddd, you might want to check if ccc is executable and ddd is readable first.

```
if ( (-x "/bin/ccc") && (-r "../input/ddd") ) {
   my(@cccout) = `/bin/ccc ../input/ddd`;    # run the command.
} else {
   ... complain ...
}
```

I give the descriptions directly from the perlfunc manual page, listed from most common to least common, based on my own usage.

-f	File is a plain file.
-e	File exists.
-d	File is a directory.
-l	File is a symbolic link.
-r	File is readable by effective uid/gid.
-x	File is executable by effective uid/gid.
-w	File is writable by effective uid/gid.
-z	File has zero size.
-s	File has non-zero size (returns size).
-o	File is owned by effective uid.
-R	File is readable by real uid/gid.
-W	File is writable by real uid/gid.
-X	File is executable by real uid/gid.
-O	File is owned by real uid.

-p	File is a named pipe (FIFO).
-S	File is a socket.
-b	File is a block special file.
-c	File is a character special file.
-t	Filehandle is opened to a tty.
-u	File has setuid bit set.
-g	File has setgid bit set.
-k	File has sticky bit set.
-T	File is a text file.
-B	File is a binary file (opposite of -T).
-M	Age of file in days when script started.
-A	Same for access time.
-C	Same for inode change time.

12 Command line arguments

When you run a Perl script, perl puts the command line arguments in the global array @ARGV. For example, if you run the command

```
% perl somescript.pl -o abc -t one.html two.html
```

will result in

`$ARGV[0]`	`-o`
`$ARGV[1]`	`abc`
`$ARGV[2]`	`-t`
`$ARGV[3]`	`one.html`
`$ARGV[4]`	`two.html`

The prototype code at the begining of this document shows one way to process `@ARGV`.

13 File I/O

See the prototype example for reading/writing from/to a file.

Given a file handle `FH` from either `open()` or a `new FileHandle`, the operation `<FH>` reads the next line in scalar context or the entire file in list context.

```
while ( $line = <FILE_DATA> ) {          # read a line at a time.
    if ( $line =~ /keyboard/ ) {
        print $line;
    }
}

my(@whole_file) = <FILE_DATA>;           # be careful, file could be BIG.
my($numlines) = scalar(@whole_file);     #
```

If you only want to read from stdin, use an use

```
while ($line = <STDIN>) { # read a line at a time
  ...
}
```

But how can we read from a file sometime and from STDIN at other times in the same Perl script? The routines `handle_file()` and `read_file()` in the prototype code show how read from *any input stream* such as a file, `stdin` (which itself could be a file, the keyboard or a network connection), a network connection, the keyboard, and so on.[16] The function `handle_file()` is a "driver" for `read_file()` that passes as a parameter either `STDIN` or a `FileHandle` input stream to `read_file()`.

In `read_file(istream, fname)` the first parameter, `istream`, is the input stream, from whic we read input data. The second parameter `fname` is the file name, which is used for say, reporting errors. To pass `STDIN` as a parameter to `read_file()`, we use `\*STDIN`[17] Sadly explaining `\*STDIN` is beyond the scope of this document.

[16] An input stream is any source of input data and is a generalization of an input file. In C an input stream is a file descriptor or a FILE* pointer (from `stdio.h`), such as `stdin`. In C++ an input stream is an `istream`, such as `cin`.

[17] This is a very advanced topic as we are passing a reference to the typeglob for STDIN.

14 Running external commands

(This may or may not work on Win32) You can run an external command, such as `ls -l` by placing it in back quotes (also known as back ticks or grave accents, `` `ls -l` ``. The returned value is the output the command sends to stdout. In scalar context, you get one big string, with a `\n` character separating lines; in array context, each output line is a separate array item.

Thus, see the contents of a tar file, `xyz.tar` in Perl, you could do

```
my(@tarlist) = `tar tfv xyz.tar`;
```

Commands are run in current working directory, which is initially the directory where you started the Perl script. You can change the current working directory to DDD by calling the built-in Perl function `chdir DDD`.

15 References

A reference in Perl is equivalent to a pointer in C. Any Perl scalar value/variable can be a reference. The address-of operator in Perl is the \ (backslash) and the dereference operator is *confusingly* the $ (dollar sign).

Thus the following lines are equivalent in Perl and C; in both cases we change the value of `str` from `"hi"` to `"bye"` via `ptr` and we add 5 to the value of `num` via a pointer. In Perl, we can use the same reference variable `ptr` becuse references are not typed; in C we must use different pointers `sptr` and `iptr`.

Perl	C/C++
`$str = "hi";`	`char* str = "hi";`
`$ptr = \$str;`	`char** sptr = &str;`
`$$ptr = "bye";`	`*sptr = "bye";`
`$num = 4;`	`int num = 4;`
`$ptr = \$num;`	`int* iptr = #`
`$$ptr += 5;`	`(*iptr) += 5;`

15.1 Passing references to functions

Here's an example of a function `clearEntry` which clears the specified index `idx` of an array of strings `arr` and increments index. Because both variables are modified, they are both passed as references.

```
sub clearEntry ($$) {
    my($idx, $arr) = @_;
    $arr->[$$idx] = "";
    $$idx ++;
}
sub callClear () {
    my(@stuff) = ("aa", "bb", "cc", "dd");
    my($indexer) = 1;
    print "BEFORE indexer = $indexer " . join(":", @stuff) . "\n";
    clearEntry(\$indexer, \@stuff);
    print "AFTER  indexer = $indexer " . join(":", @stuff) . "\n";
}
```

To dereference the array reference arr, use the syntax `$arr->[...]`.

Calling `callClear()` gives

```
BEFORE indexer = 1 aa:bb:cc:dd
AFTER  indexer = 2 aa::cc:dd
```

16 Quoting

There are a variety of other quoting mechanisms as summarized in the table below, which borrows directly from the *Quote and Quotelike Operators* section in the perlop manpage. Interpolates means that variables are evaluated, which in turn means that all variable references starting with $, @, or % are fully evaluated.

```
@squares = (0, 1, 4, 9, 16, 25);
$i = 2;
print("i = $i, 3+i = (3+$i)\n");           # print: i = 2, 3+i=(3+2)
print("squares[i+3] = $squares[$i+3]\n"); # print: squares[i+3] = 23
```

In the first `print()` statement, the arithmetic expression `(3+i)` is not evaluated, because it is not a variable; however, the reference to `$squares[$i+3]` is fully evaluated.

Customary	Generic	Meaning	Interpolates
'xxx'	q:xxx:	Literal	no
"xxx"	qq:xxx:	Literal	yes
\`xxx\`	qx:xxx:	Command	yes
none	qw:xxx:	Word list	no
/xxx/	m:xxx:	Pattern match	yes
none	s:xxx:yyy:	Substitution	yes
none	tr:xxx:yyy:	Translation	no

The generic quoting mechanism allows you to delimit a string with arbitrary characters, which is especially convenient when the string contains single and/or double quotes.

```
$where = "a hot dog stand";
$proverb =  'Don't buy sushi from a hot dog stand.';
$proverb = q/Don't buy sushi from a hot dog stand./;
$proverb = q(Don't buy sushi from a hot dog stand.);
$proverb =   "Don't buy sushi from $where.";
$proverb = qq/Don't buy sushi from $where./;
$proverb = qq(Don't buy sushi from $where.);
```

17 Packages, Modules, Records and Objects in Perl

I have no plans to cover these topics in this introductory document. Perhaps in a not-in-the-near future "Reusable Perl code in 10 pages" document.

18 Feedback

Any constructive feedback on this document is welcome.

Unfortunately, I will ignore all questions of the form ``How do I do ...??''

19 Motivation and afterthoughts

I am writing this document because I wish some one had done so when I was learning Perl.