http://www.coolinterview.com/type.asp?order=3&iType=108&iDBLoc=20
http://world.std.com/~swmcd/steven/perl/module_anatomy.html <= Read this before interview

Que: i. Scalar, Array, Hash and their references.

ii. How to know whether it's an array ref or hash ref or glob ref or subroutine ref as all are scalars after referencing ?

iii. How do you get the elements in an array ref and hash ref ?

iv. How to dereference ?

v. What are ***anonymous array, hash and subroutine*** ? Where do we use these ?

Ans:

i.

$scalar = “Samir”;

@arr = ();

%hash = ();

Creating References:

```
$array = \@ARGV;      # Create reference to array
$hash = \%ENV;        # Create reference to hash
$glob = \*STDOUT;     # Create reference to typeglob
$foosub = \&foo;      # Create reference to subroutine
```

ii.

ref ($array) # prints ARRAY
ref ($hash) # prints HASH
ref ($glob) # prints TYPEGLOB
ref ($foosub) # prints SUBROUTINE

e.g.

How to Check whether a variable is a hash or not ?

If the referenced object has been blessed into a package, then that package name is returned instead. You can think of ref() as a `typeof()` operator.

```
    if (ref($r) eq "HASH") {
        print "r is a reference to a hash.\n";
    }
    if (!ref($r)) {
        print "r is not a hash reference.\n";
    }
```

iii.

$array->[0] # gives you the first element
$hash->{key} # gives you the value of the key

iv.

@$array # array is back
%$hash # hash is back
*$glob # typeglob is back
&$foosub # subroutine is back

iv. Anonymous Array, Hash and Subroutine

We use these in case of array of array, array or hashes or hash or hash etc

$array = []

$hash = {}

```
$coderef = sub { print "Boink!\n" }; # Now &$coderef prints
"Boink!"
```

e.g.

$array = ['Bill', 'Ben, 'Mary'];

```
$hash = { 'Man' => 'Bill',
          'Woman' => 'Mary,
          'Dog' => 'Ben'
};
```

Example of ***nested datastructure*** (array of arrays, hash of hashes, hash of arrays, array of arrays and hashes etc)

```
@arrayarray = ( 1, 2, [1, 2, 3]);  # Array containing 3 elements one of which is an anonymous array

my @students = ( { name         => 'Clara',         # Array of arrays and hashes

                   registration => 10405,
                   grades       => [ 2, 3, 2 ] },
                 { name         => 'Amy',
                   registration => 47200,
                   grades       => [ 1, 3, 1 ] },
                 { name         => 'Deborah',
                   registration => 12022,
                   grades       => [ 4, 4, 4 ] } );
```

Que: What do you mean by '$^O'?
Ans: It prints the current OS whether Linux/Windows.
perl -e 'print $^O'
linux

Que: What is @INC and %INC ?
Ans:
@INC

This array variable holds a list of directories where Perl can look for scripts to execute. The list is used mainly by the require statement.

%INC

This hash variable has entries for each filename included by do or require statements. The key of the hash entries are the filenames and the values are the paths where the files were found.

e.g. @INC prints
/etc/perl
/usr/local/lib/perl/5.10.1
/usr/local/share/perl/5.10.1
/usr/lib/perl5
/usr/share/perl5
/usr/lib/perl/5.10
/usr/share/perl/5.10/
usr/local/lib/site_perl

Que: How to

i. do substitution and character-by-character substirtuion in perl?
ii. Do swap using regex ?
iii. Character-by-character translation i.e. a,b,c will be replaced by e,d,f

iv. you use grep in perl ?
v. use map function in perl ?

Ans:

i. $sntnce =~ s/search_str/replace_str/g # Remove g if you want to replace once
ii. $sntnce =~ s/^(.)(.*)(.)$/\3\2\1/ # 1st and last character would be swaped
or
s/^(.)(.*)(.)$/\$3\$2\$1/
iii. $sntnce =~ tr/abc/edf/
iv.

Que: Scalar operations ?
Ans:
Que: What is the difference between chop & chomp functions in perl?

chop $SCALAR => is used remove last character of a scalar,
chomp $SCALAR => function removes only line endings(i.e.\n). (in short chomp removes the $INPUT_RECORD_SEPARATOR, whenever you call it, it checks the value of '$/' and removes it; by default it is \n)
lc $SCALAR => converts to lower case characters
uc $SCALAR => converts to upper case characters

Que: Array operations ?
Ans:
$#arr +1 # length of the array length $string # String length
or $@arr or ***.@arr***
$#arr # last index of the array
$arr[-1] # last element of the array
sort @arr # sorts by character/alphabatically
sort {$a <=> $b}@arr # Numeric sort
sort {length $a <=> length $b}@arr # Length wise sort
sort {lc $a <=> lc $b} @arr # case-insensitive sort
push (@arr, @brr) or push (@arr, $str) # Adds to the end of an array
pop @arr # Removes the last element
shift @arr # Removes the first element
unshift @arr, $str # Adds to the beginning of an array
delete $array[index] # Removes an element by index number
spllice @arr, OFFSET, LENGTH, LIST
split (“\s+”, $str)
join (“,”, @arr)
reverse @arr # ***scalar*** reverse $str => Array and string reverse
qw(1234) # same as (“1”,”2”,”3”,”4”)

Que: How to delete a hash element ?
Ans: delete($hash{key});

Que: What are the 3 ways to empty an array ?
Ans:

1. **@arr= -1**
2. @arr= ()
3. @arr = undef (Assigning undef is not a good idea)

Que: How many types of prefix dereferenceing are there ?
Ans: **Their are six prefix dereferencer available in perl they are:**

(i) $-Scalar variables
(ii) %-Hash variables
(iii) @-arrays
(iv) &-subroutines
(v) (- code
(vi) * - typeglob

Que: Difference between array and list in perl ?
Ans: We can perform push, pop, shift and unshift on an array but not upon a list
e.g. @arr = (1, 2, 3) => array
$list = (1,2) => list

Que: What is Pack and Unpack in perl ?
Ans:
pack function converts values to a byte sequence containing representations according to a given specification, the so-called "template" argument. unpack is the reverse process, deriving some values from the contents of a string of bytes. (Be cautioned, however, that not all that has been packed together can be neatly unpacked - a very common experience as seasoned travellers are likely to confirm.)
$STRING = pack (TEMPLATE, LIST)
unpack (TEMPLATE, $STRING)
For unpacking something(a list) which you have packed you need to remember the template and the string containing packed value

e.g.
my $nrs = pack 'N*', 45320..45325;

print "\$nrs is $nrs";
# It will give you something which might be not readable
my @array = unpack 'N*', $nrs;

print "\@aaray is @array";

it displays: 45320 45321 45322 45323 45324 45325

OR

```
my $str = pack 'CCCC', 97, 98, 99, 100, 101, 102;
# 97 is the numeric value of the ASCII 'a' character
print "$str\n";     # abcd
```

Que: How do use perl function to find out the date and time ?
Ans:
```
my ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime(time);

print "local time is ",localtime(time);

print "\$mday, Day of the month is $mday"; # 1..
$mon += 1;

print "\$mon, month is $mon";   # Month is 0..11

$year += 1900;
```

```
print "year is $year";          # Year is 0...; So add it to 1900

print "\$wday, Day of the week is $wday";
 # 0..6 with 0 indicating Sunday and 3                                    #
indicating Wednesday.
print "\$yday, Day of the year is $yday";
 # 0..364 (or 0..365 in leap years.)
print "\$is dst is $isdst";
      # true(1) if the specified time occurs during
      # Daylight Saving Time, false(0) otherwise.
```

Que: How to get ASCII, Hexadecimal and Octal value in perl ?
Ans:
ord (EXPR)
hex (EXPR)
oct (EXPR)

Que: Why to not use perl for a project ?
**Ans: Only in the case of high performance graphics scenarios, like games, etc.
For all the other cases Perl can be used somehow.**

Que: Advantages of C over Perl for development.
Ans:

1. **There are more development tools for C than for PERL**
2. **There are less PERL programers around than C programmers**
3. **PERL execute slower than C programs do**
4. **Without additional tools it is impossible to create an executable of a Perl program**

Que: What exactly is grooving and shortening of the array?
Ans:
You can change the number of elements in an array simply by changing the value of the last index of/in the array $#array. In fact, if you simply refer to a non existent element in an array perl extends the array as needed, creating new elements. It also includes new elements in its array.

Que: How to set environment variables in perl ?
Ans:
"%ENV" is a special hash in Perl that contains the value of all your environment variables. Because %ENV is a hash, you can set environment variables just as you'd set the value of any Perl hash variable. Here's how you can set your PATH variable to make sure the following four directories are in your path::

$ENV{'PATH'} = '/bin:/usr/bin:/usr/local/bin:/home/yourname/bin';

Que: How do you give functions private variables that retain their values between calls?
Ans: Create a scope surrounding that sub that contains lexicals.
Only lexical variables are truly private, and they will persist even when their block exits if something still cares about them. Thus:
{ my $i = 0; sub next_i { $i++ } sub last_i { --$i } }
creates two functions that share a private variable. The $i variable will not be deallocated when its block goes away because next_i and last_i need to be able to access it.

Que: What are different loops ?
Ans:

for,
foreach,
while,
do while,
until => opposite of ***while***
goto LABEL

Que: What are different conditions ?
Ans:
if
unless => if not

Que: What is next and last in perl ?
Ans:
If you want to skip the rest of the processing of the body, but don't want to exit the loop, you can use
next to immediately go back to the start of the loop, passing the next value to the iterator.

The keyword ***last***, in the body of a loop, will make perl immediately exit, or 'break out of' that loop. The remaining statements are not processed, and you up right at the end.

e.g.

```
my @array = (8, 3, 0, 2, 12, 0);

for (@array) {

if ($_ == 0) {

print "Skipping zero element.\n";

next;

}

print "48 over $_ is ", 48/$_, "\n";

}
```

AND

```
my @array = ( "red", "blue", "STOP THIS NOW", "green");

for (@array) {

last if $_ eq "STOP THIS NOW";

print "Today's colour is $_\n";

}
```

Que: What's the significance of @ISA, @EXPORT @EXPORT_OK %EXPORT_TAGS list & hashes in a perl package? With example?
Ans:
@ISA -> each package has its own @ISA array. this array keep track of classes it is inheriting.

ex:

package child;

@ISA=(parentclass);

@EXPORT this array stores the subroutins to be exported from a module.

@EXPORT_OK this array stores the subroutins to be exported only on request.

Que: Types of eval statements ?
Ans:

1. **eval expression**
2. **eval block**

Que: What is typeglob ?
Ans:
Typeglobs are a way of accessing the symbol table. You can create a typeglob by prepending the * sigil to a bareword.

Three common uses for typeglobs are

i. **creation of aliases,**
ii. **passing global filehandles (such as STDIN) to functions, and**
iii. **the dynamic creation of functions.**

e.g.

Aliases:

```
my    $var   = "foo";


local *alias = \$var;



print "$var and $alias are the same\n";
                                    # foo and foo are the same
$var = "bar";


print "$var and $alias are still the same\n";                    # bar and bar are still the same
```

OR

```
$scalar = 'foo';
```

```
@array  = 1 .. 5;

*foo = \$scalar;

*foo = \@array;



print "Scalar foo is $foo\n";
                        # Scalar foo is foo
print "Array foo is @foo\n";
                        # Array foo is 1 2 3 4 5
```
OR
```
$myvar = 'foo';

@myvar = 1 .. 5;

%myvar = qw(One 1 Two 2 Three 3);

sub myvar { "I'm a subroutine!"};



*me = *myvar;



print "Scala me is $me\n";
                        # Scalar me is foo
print "Array me is @me\n";
                        # Array me is 1 2 3 4 5
print "Hash me is ",%me,"\n";
                # Hash me is Three3Two2One1
print "subroutine me is ",&me,"\n";
                # subroutine me is I'm a subroutine!

$myvar = 'Bar scalar';
```

@myvar = 6 .. 10;

print "Scalar is <$me>, array is <@me>\n";

Scalar is <Bar scalar>, array is <6 7 8 9 10>

typeglob

hide

Typeglobs are a way of accessing the symbol table. You can create a typeglob by prepending the * sigil to a bareword. A typeglob will return the symbol table entry proper for the context in which you use it. Three common uses for typeglobs are creation of aliases, passing global filehandles (such as STDIN) to functions, and the dynamic creation of functions.

Aliases

```
my    $var   = "foo";
local *alias = \$var;

print "$var and $alias are the same\n";
$var = "bar";
print "$var and $alias are still the same\n";
```

Passing global filehandles

```
sub log_message {
    my ($fh, $message) = @_;
    print $fh $message;
}

log_message(\*STDERR, "write to standard error\n");
```

Dynamic creation of functions

```
*newsub = sub { print "hello world\n" };
newsub();
```

Que: What is symbol table in perl ?
Ans:
Each package has a special hash-like data structure called the symbol table, which comprises all of the typeglobs for that package. It's not a real Perl hash, but it acts like it in some ways, and its name is the ***package name with two colons at the end***.

The main symbol table's name is thus `%main::`, or `%::` for short. Likewise the symbol table for the nested package mentioned earlier is named `%OUTER::INNER::`.

This isn't a normal Perl hash, but I can look in it with the `keys` operator. Want to see all of the symbol names defined in the `main` package? I simply print all the keys for this special hash:

```
#!/usr/bin/perl

foreach my $entry ( keys %main:: )
    {
    print "$entry\n";
    }
```

Que: What is qw(...) ?
Ans: qw is a construct which quotes words delimited by spaces.

Que: **Which has the highest precedence, List or Terms? Explain?**
Ans: Terms have the highest precedence in perl. Terms include variables, quotes, expressions in parenthesis etc. List operators have the same level of precedence as terms. Specifically, these operators have very strong left word precedence.

Que: What is the easiest way to download the contents of a URL with Perl?
Ans: Using module LWP::Simple
#!/usr/bin/perl
use LWP::Simple;
$url = get 'http://www.websitename.com/**';**

Que: What are the different types of perl operators?
Ans: There are four different types of perl operators they are
(i) Unary operator like the not operator
(ii) Binary operator like the addition operator
(iii) Tertiary operator like the conditional operator
(iv) List operator like the print operator

Que: How do you work with array slices?
Ans: An array slice is a section of an array that acts like a list, and you indicate what elements to put into the slice by using multiple array indexes in square brackets. By specifying the range operator you can also specify a slice.

Que: What is meant by a 'pack' in perl? What is 'unpack' ?
Ans:
**Pack Converts a list into a binary representation

Takes an array or list of values and packs it into a binary structure, returning the string containing the structure

Hope that kills the problem !!

**
e.g.
$rec = pack("l i Z32 s2", time, $emp_id, $item, $quan, $urgent);

print $rec;

($order_time, $monk, $itemname, $quantity, $ignore) = unpack("l i Z32 s2", $rec);

Que: Why is it hard to call this function: sub y ?
Ans: Because y is a kind of quoting operator.
The y operator is the sed-savvy synonym for tr. That means y(3) would be like tr(), which would be looking for a second string, as in tr/a-z/A-Z/, tr(a-z)(A-Z), or tr[a-z][A-Z].

Que: How do you connect to database in perl ?
Ans: Using DBI(database independent interface)
For Sybase:

use DBI;
my $dbh = DBI->connect('dbi:Sybase:server=$SERVER', 'username', 'password')

my $sth = $dbh->prepare("select name, symbol from table");

$sth->execute();

while (@row = $sth->fetchrow_array())

{
  print "name = $row[0], symbol = $row[1]";
}
$dbh->disconnect;

Que: **"perl regular expressions are greedy" what does this mean?**
Ans:
Perl regular expressions normally match the longest string possible. that is what is called as "greedy match" For instance:my($text) = "mississippi";$text =~ m/(i.*s)/;print $1 . " ";Run the preceding code, and here's what you get:ississIt matches the first i, the last s, and everything in between them. But what if you want to match the first i to the s most closely following it? Use this code:my($text) = "mississippi";$text =~ m/(i.*?s)/;print $1 . " ";Now look what the code produces:is

Que: How do you load a large file into a hash ? Or why do we use Data::Dumper **?**
Ans:
It dumps out objects. Allows us to convert the contents of a variable into a printable string of source code - in other words to let us display a variable's content.
e.g.

```
#!/usr/bin/perl
use Data::Dumper qw(Dumper);

########## In case of a large file(uncomment the following 3 lines) ########
#my $filename = shift;
#open my $fh,"<",$filename or die $!;
#my %hash = map { chomp; split /\t/ } <$fh>;

my %hash = ( a => 1, b => 2, c => 3 );
print Dumper(\\%hash);  # note the \ backslash; Dumper() takes references as
arguments; Note: Here "print %hash" would simply print "c3a1b2" and without
reference "print Dumper(%hash)" would print $VAR1, $VAR2, $VAR3 etc
```

Output:

```
$VAR1 = {
          'c' => 3,
          'a' => 1,
          'b' => 2
        };
```

lets do one more modification

```
#!/usr/bin/perl
use Data::Dumper qw(Dumper);
my %hash = ( a => 1, b => 2, c => [0,1,2,3] );
print Dumper(\\%hash);  # note the \ backslash; Dumper() takes references as
```

```
arguments


$./example
Output:
$VAR1 = {
          'c' => [
                    0,
                    1,
                    2,
                    3
                  ],
          'a' => 1,
          'b' => 2
        };
```

Que: How to handle large file in perl as it will take a long time for reading the file ?

Ans: Use Tie::File (http://perldoc.perl.org/Tie/File.html)
Tie::File - Access the lines of a disk file via a Perl array
The file is *not* loaded into memory, so this will work even for gigantic files.
`Tie::File` represents a regular text file as a Perl array. Each element in the array corresponds to a record in the file. The first line of the file is element 0 of the array; the second line is element 1, and so on.
e.g.

```
        # This file documents Tie::File version 0.97
use Tie::File;
tie @array, 'Tie::File', filename or die ...;
$array[13] = 'blah';     # line 13 of the file is now 'blah'
print $array[42];        # display line 42 of the file
$n_recs = @array;        # how many records are in the file?
$#array -= 2;            # chop two records off the end for (@array) {
s/PERL/Perl/g;         # Replace PERL with Perl everywhere in the file }

# These are just like regular push, pop, unshift, shift, and splice
# Except that they modify the file in the way you would expect

push @array, new recs...;
my $r1 = pop @array;
unshift @array, new recs...;
my $r2 = shift @array;
@old_recs = splice @array, 3, 7, new recs...;
untie @array;            # all finished
```

memory

This is an upper limit on the amount of memory that `Tie::File` will consume at any time while managing the file. This is used for two things: managing the *read cache* and managing the *deferred write buffer*.

Records read in from the file are cached, to avoid having to re-read them repeatedly. If you read the same record twice, the first time it will be stored in memory, and the second time it will be fetched from the *read cache*. The amount of data in the read cache will not exceed the value you specified for `memory` . If `Tie::File` wants to cache a new record, but the read cache is full, it will make room by expiring the least-recently visited records from the read cache.

The default memory limit is 2Mib. You can adjust the maximum read cache size by supplying the `memory` option. The argument is the desired cache size, in bytes.

```
# I have a lot of memory, so use a large cache to speed up access
tie @array, 'Tie::File', $file, memory => 20_000_000;
```
Setting the memory limit to 0 will inhibit caching; records will be fetched from disk every time you examine them.

The `memory` value is not an absolute or exact limit on the memory used. `Tie::File` objects contains some structures besides the read cache and the deferred write buffer, whose sizes are not charged against `memory` .

The cache itself consumes about 310 bytes per cached record, so if your file has many short records, you may want to decrease the cache memory limit, or else the cache overhead may exceed the size of the cached data.

Que: What arguments do you frequently use for the Perl interpreter and what do they mean?
Ans:

- -W for show all warnings mode (or -w to show less warnings)
- -e executes the prog given as an argument
- -d start perl debugger on the file name specified as an argument
- -c compile the file
- -T is enables the Taint mode it performs some checks how
  your program is using the data passed to it
  Turn on strict mode to make Perl check for common mistakes.
- -n and -p for line processing, and probably with -i for inline processing

Que: How to check if a hash key exists or not, an array index exists or not, a subroutine exists or not?
Ans:

```
print "Exists"    if exists $hash{$key};
print "Defined"   if defined $hash{$key};
print "True"      if $hash{$key};

print "Exists"    if exists $array[$index];
print "Defined"   if defined $array[$index];
print "True"      if $array[$index];

print "Exists"  if exists &subroutine;
print "Defined" if defined &subroutine;
```

print “Exists” if (exists $ref->{"Some key"})
A hash element can be TRUE only if it's defined, and defined if it exists, but the reverse doesn't necessarily hold true.

Que: How to check perl version ?
Ans: perl -v

Current Perl version. 5.12.2

Que: How will you check whether a particular module is installed or not ???

Ans: perl -e “use Module_name”. How do you use perl one-liner ?
Ans: perl -e ... can be used as a one-liner.
e.g.
perl -e “use XML::Simple”

Some of the Default perl modules:
samir@samir-laptop:~$ perl -e "use CGI::Apache"
samir@samir-laptop:~$ perl -e "use CGI::Push"
samir@samir-laptop:~$ perl -e "use CGI::Switch"
samir@samir-laptop:~$ perl -e "use Class::ISA"
samir@samir-laptop:~$ perl -e "use CPAN"
samir@samir-laptop:~$ perl -e "use Carp"
samir@samir-laptop:~$ perl -e "use Compress::Zlib"

If the module is present then it won't display anything, but if it's absent then it will give error like *Can't locate Classh/ISA.pm in @INC (@INC contains: /etc/perl /usr/local/lib/perl/5.10.0 /usr/local/share/perl/5.10.0 /usr/lib/perl5 /usr/share/perl5 /usr/lib/perl/5.10 /usr/share/perl/5.10 /usr/local/lib/site_perl .*

Que: When would you use Perl for a project?
1. When you are developing an application for a real time
system in which processing speed is of utmost importance

When to use Perl
1. For large text processing
2. When application does lot of data manipulation
3. When you need fast development
4. For database loading operations
5. When shell scrits grow to become libraries

Que: How to delete a file in perl ?

Ans: unlink($file)

```
#!/usr/bin/perl

@files = ("newtext.txt","moretext.txt","yetmoretext.txt");

foreach $file (@files) {

   unlink($file);

}
```

Que: Write a simple regular expression to match an IP address, e-mail address, city-state-zipcode
combination.
/([0-255])(\.)$1\1$1\1$1/;

Que: What?s your favorite module and why?

My Favourite module is CGI.pm Bcoz it can handle almost all
the tasks like
1. parsing the form input
2. printiing the headers

3. can handle cookies and sessions and much more

Test::More, Test::Most and Test::Exception

Que: Explain the difference between use and require?

Use :
1. The method is used only for the ***modules***(only to include
.pm type file)
2. The included objects are ***varified at the time of compilation***.
3. ***No Need to give file extension***.

Require:
1. The method is used for both ***libraries and modules***.
2. The included objects are ***varified at the run time.***
3. ***Need to give file Extension.***

The require Compiler Directive

The require directive is used to load Perl libraries. If you needed to load a library called Room.pl, you would do so like this:

require Room.pl;

No exporting of symbols is done by the require directive. So all symbols in the libraries must be explicitly placed into the main namespace. For example, you might see a library that looks like this:

```
package abbrev;

sub main'abbrev {
   # code for the function
}
```

Two things in this code snippet point out that it is Perl 4 code. The first is that the package name is in all lowercase. And the second is that a single-quote is used instead of double-colons to indicate a qualifying package name. Even though the abbrev() function is defined inside the abbrev package, it is not part of the %abbrev:: namespace because of the main' in front of the function name.

The require directive can also indicate that your script needs a ***certain version*** of Perl to run. For example, if you are using references you should place the following statement at the top of your script:

require 5.000;

And if you are using a feature that is only available with Perl 5.002 - like prototypes - use the following:

```
require 5.002;
```

Perl 4 will generate a fatal error if these lines are seen.

Note

Prototypes are not covered in this book. If you are using Perl 5.002 or later, prototypes should be discussed in the documentation that comes with the Perl distribution.

The use Compiler Directive

When it came time to add modules to Perl, thought was given to how this could be done and still support the old libraries. It was decided that a new directive was needed. Thus, use was born.

The use directive will automatically export function and variable names to the main namespace by calling the module's import() function. Most modules don't have their own import() function; instead they inherit it from the Exporter module. You have to keep in mind that the import() function is not applicable to object-oriented modules. Object-oriented modules should not export any of their functions or variables.

You can use the following lines as a template for creating your own modules:

```
package Module;
   require(Exporter);
   @ISA = qw(Exporter);
   @EXPORT = qw(funcOne $varOne @variable %variable);
   @EXPORT_OK = qw(funcTwo $varTwo);
```

The names in the @EXPORT array will always be moved into the main namespace. Those names in the @EXPORT_OK will only be moved if you request them. This small module can be loaded into your script using this statement:

```
use Module;
```

Since use is a compiler directive, the module is loaded as soon as the compiler sees the directive. This means that the variables and functions from the module are available to the rest of your script.

If you need to access some of the names in the @EXPORT_OK array, use a statement like this:

```
use Module qw(:DEFAULT funcTwo);     # $varTwo is not exported.
```

Que: Explain the difference between my and local?

The variables declared with "my" can live only within the block it was defined and there is no visibility to inherited functions which were called within that block, but one defined with "local"

they can live within the block and can have visibility in the functions which were called within that block.

e.g.:

```
&check_func();
sub check_func {
   my $var1 = 3;
   local $var2 = 9;
   &add_numbers();
}
sub add_numbers {
   $var = "";
   print "var1 value in the inherited function is " . $var1;
   print "var2 value in the inherited function is " . $var2;
   $var3 = $var1 + $var2;
   print "var3 value in the inherited function is " . $var3;
}
```

O/P:

var1 value in the inherited function is

var2 value in the inherited function is 9

var3 value in the inherited function is 9

Explanation:

Here var1 will not be visible to the inherited function add_numbers() as it has been declared **my**

var2 will be visible to the inherited function add_numbers() as it has been declared **local**

even if we won't declare var2 as local and keep it blank(e.g. $*var2* in place of *local $var2*) then also it will give the above output.

Que: What is ***our*** in perl ? Differentiate it with ***my***.

Ans:

Defines the variables specified in LIST as being global within the enclosing block, file, or eval statement. It is effectively the opposite of my***.it declares a variable to be global within the entire scope***, *rather than creating a new private variable of the same name*. All other options are identical to my;

An our declaration declares a global variable that will be visible across its entire lexical scope, even across package boundaries. The package in which the variable is entered is determined at the point of the declaration, not at the point of use.

e.g.

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

our $string = "We are the world";
```

```
my $string2 = "I am Indian";
print "$string\n";
print "$string2\n";
myfunction();
print "$string\n";
print "$string2\n";

sub myfunction
{
our $string = "We are the function";
my $string2 = "I am a boy";
print "$string\n";
print "$string2\n";
}
```

O/P:

```
We are the world
I am Indian
We are the function
I am a boy
We are the function
I am Indian
```

or

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

our $string = "We are the world";
print "$string\n";
myfunction();
print "$string\n";

sub myfunction
{
our $string = "We are the function";
print "$string\n";
}
```

It will produce following results:

We are the world

We are the function

We are the function

Que: What is ***warn*** in perl ?

Ans: Prints the value of LIST to STDERR. Basically the same as the die function except that no call is made to the exit and no exception is raised within an eval statement. This can be useful to raise an error without causing the script to terminate prematurely.

If the variable $@ contains a value (from a previous eval call) and LIST is empty, then the value of $@ is printed with .\t.caught. appended to the end. If both $@ and LIST are empty, then .Warning: Something.s wrong. is printed.

e.g.

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

warn("Unable to calculate value, using defaults instead.\n");
```

It will produce following results:

```
Unable to calculate value, using defaults instead
```

Que: What are Carp, Cluck, Croak, Confess etc ?

Ans:

carp - warn of errors (from perspective of caller)

cluck - warn of errors with stack backtrace (not exported by default)

croak - die of errors (from perspective of caller)

confess - die of errors with stack backtrace

Discussion: Croak is like shouting out for something without dieing although it works almost same as die. We need to use “use Carp” for using Croak.

Que: What are the different pragmas you have used ?

Ans:

use strict

use warnings

use diagnostics

Que: How do you ON/OFF “use strict” and “use warnings” ?

Ans:

use strict =>

use strict 'vars';

use strict 'refs';

use strict 'subs';

You may deactivate a pragma using:

no strict; # deactivates vars, refs and subs;

no strict 'vars';# deactivates use strict 'vars';

no strict 'refs'; # deactivates use strict 'refs';

no strict 'subs'; # deactivates use strict 'subs';

Que: What is *use strict* and *use warning* ? What does the command 'use strict' do and why should you use it?

Ans:

1. ***1.*** The use strict 'refs' pragma prevents you ***from using symbolic references.***
2. Prevents undeclared variables (you need to use "my $var" otherwise throws error :global symbol for explicit package name is missing)
3. Tells to use ***&mysub*** while calling a subroutine (Although we can call a subroutine in 3ways like: *mysub, mysub(), &mysub* but for the first case strict refs treat it as a barreword)
4. you have a typo in a variable name, without strict/warnings you will search some time to find it

use strict enforces strictness! :-) You should use it because it will help you avoid pestering errors that nobody likes (forgotten my declarations being the most usual ones). I nowadays use it most for discipline. It's not that one my programs wouldn't work without it, but I prefer to be on the safe side. Also note that I don't use strict for one-liners.
e.g.

1.

Here are two trivial examples of references (sorry that they do not make the case for using references obvious):

my $foo = "bar";

my $bar = "foo";

my $ref = \$foo;

hard reference

```
print $$ref;        # prints 'bar'
```

symbolic reference

```
print $$bar;        # also prints 'bar'
```

So what gives? In the first case we create a hard reference to the variable $foo. We then print the value in $foo by dereferencing it via the $$ construct. In the second case what happens is that because $bar is not a reference when we try to deference it Perl presumes it to be a symbolic reference to a variable whose name is stored in $bar. As $bar contains the string 'foo' perl looks for the variable $foo and then prints out the value stored in this variable. We have in effect used a variable to store a variable name and then accessed the contents of that variable via the symbolic reference.

4.

```
my $dir = "/tmp";
my $work_dir = "/";
chdir($work_dir);
# some hundred lines in between
system("rm -rf $dri/*");
# oops ;-)
```

e.g.

```
&check_func();
sub check_func {
    my $var1 = 3;
    local $var2 = 9;
    &add_numbers();
}
sub add_numbers {
    $var3 = $var1 + $var2;
    print "var3 value in the inherited function is " . $var3;
}
```

If we will **use warning** in the above program then on compilation it will give the following:

Name "main::var1" used only once: possible typo at mylocal line n($var3 = $var1 + $var2).

mylocal syntax OK

If we will **use strict** in the above program then on compilation it will give the following:

Global symbol "$var2" requires explicit package name at mylocal line 9.

Global symbol "$var3" requires explicit package name at mylocal line 16.

Global symbol "$var1" requires explicit package name at mylocal line 16.

Global symbol "$var2" requires explicit package name at mylocal line 16.

Global symbol "$var3" requires explicit package name at mylocal line 17.

mylocal had compilation errors.

Que: What is “use diagnostics” ?
Ans:
Three lines - what could be easier. We get some input from STDIN and exit if the user types exit. Otherwise we print something. When we run the code we find that regardless of what we type it prints. Why no exit?? Lets add warnings and see what happens:

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

my $input_received = <STDIN>;
exit if $input_recieved =~ m/exit/i;
print "Looks like you want to continue!\n";
```

```
Name "main::input_received" used only once: possible typo at test.pl line 3.
Name "main::input_recieved" used only once: possible typo at test.pl line 4.
test.pl syntax OK
```

So we have a syntax error, fix that and we are off and running. If we were using strict the code would not have run in the first place but that's another story. Typos like this can be hard for humans to spot but perl does it in a jiffy.
Use of a what in a where like how? - use diagnostics

When you first start to use warnings some of the messages appear quite cryptic. Don't worry, the "use diagnostics;" pragma has been designed to help. When this is active the warnings generated by "-w" or "use warnings;" are expanded greatly. You will probably only need to use diagnostics for a few weeks as you soon become familiar with all the messages!

To finish off, here is another example:

```
my @stuff = qw(1 2 3 4 5 6 7 8 9 10);
print "@stuff" unless $stuff[10] == 5;
```

If you run this code it runs OK, or does it? Sure it prints the array but there is a subtle problem. $stuff[10] does not exist! Perl is creating it for us on the fly. Use warnings catches this subtle trap but if we add the use diagnostics; pragma we will get a blow by blow description of our sins.

```
use warnings;
use diagnostics;
my @stuff = qw(1 2 3 4 5 6 7 8 9 10);
```

print "@stuff" unless $stuff[10] == 5;

With warnings on the error is caught and with diagnostics on it is explained in detail.

Use of uninitialized value in numeric eq (==) at test.pl line 4 (#1)

(W uninitialized) An undefined value was used as if it were already
defined. It was interpreted as a "" or a 0, but maybe it was a mistake.
To suppress this warning assign a defined value to your variables.

Que: What is -> Symbol in perl
reference the method of a module

Que: How do you check the return code of a command in Perl or how do you check whether a command is successful or not ?
Ans:
print $?
If it prints 0 then successful,
> 0 (non zero) => unsuccessful
system calls *traditionally* returns 0 when successful and 1 when it fails.
system (cmd) && die "Error - could run cmdn";

Que: What are perl array exec() and system() function ?
Ans:
exec(PROGRAM);
$result = system(PROGRAM);

Both execute the command directly, considering the first element of the array as the command name and the remaining array elements as arguments to the command to be executed.

For that reason, it's highly recommended for efficiency and safety reasons (specially if you're running a cgi script) that you use an array to pass arguments to system()

e.g. `system("command", "arg1", "arg2", "arg3");`

Both Perl's **exec() function and system() function execute a system shell command.** The big difference is that system() creates a fork process and waits to see if the command succeeds or fails - ***returning a value(0*** for success and nonzero for failure***)***.

exec() ***does not return anything***, it simply executes the command. Neither of these commands should be used to capture the output of a system call. If your goal is to capture output, you should use the backtick operator:

$result = `PROGRAM`;

Note: All the statements after exec() ***doesn't get executed*** irrespective of success/failure. (while after system() it gets executed)
e.g.
system("ls", "-l");

if ($? == -1)

{

print "command failed: $!\n";

}

```
else

{

  printf "command exited with value %d", $? >> 8;

}

exec("ls -l bar");

print "Executed the exec command";
                    # This statement won't be printed
```

Que: How to substitute a particular string in a file containing millions of records?
perl -p -i.bak -e 's/search_str/replace_str/g' filename... => Creates a back up and replaces in original

perl -p -i -e 's/search_str/replace_str/g' filename... => Only replaces in original

Que:
Ans:
1.An object is simply a reference that happens to know which class it belongs to.

2.A class is simply a package that happens to provide methods to deal with object references.

3.A method is simply a subroutine that expects an object reference (or a package name, for class methods) as the first argument.

Que: What is bless in perl ?
Ans: bless is used to create an object(from a reference) of a class

- **bless REF, CLASSNAME** **e.g. bless $self, $classname**
- **bless REF**

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

e.g.

```
$foo    = { };
$fooRef = $foo;

print("data of \$foo is "    . ref($foo)    . "\n");
print("data of \$fooRef is " . ref($fooRef) . "\n");

bless $foo, "Bar";

print("data of \$foo is "    . ref($foo)    . "\n");
print("data of \$fooRef is " . ref($fooRef) . "\n");
```

This program displays the following:

```
data of $foo is HASH
data of $fooRef is HASH
data of $foo is Bar
data of $fooRef is Bar
```

I have a variable named $objref which is defined in main package. I want to make it as an Object of Class XYZ. How could I do it?
bless $objref,'XYZ';

Que: What is shebang line ? Can we run perl scripts without shebang line?
Ans: #!/usr/bin/perl

as the first line of a Perl script. This line is a bit of magic employed by UNIX-like operating systems to automatically execute interpreted languages with the correct **command interpreter i.e. It tells the interpreter where is perl located.** This line is called a shebang line due to the first two characters: **# is sometimes called *sharp***, and **! is sometimes called *bang***. This line normally won't work for Perl-for-Win32 users,[5] although it doesn't hurt anything since Perl sees lines beginning with # as comments. windows ignores the shebang line, ***but if you plan on writing CGI scripts the http server may need to read the shebang line.*** If you are not writing CGI scripts just use the Unix shebang line and windows will ignore it. If you are writing CGI scripts then you will need to change the shebang line or inform the end-user it will need to be changed.

YES, we can execute perl scripts without a shebang line.

What is perl?
perl is a programming language based off of C, shell, Lisp and a few other things. It is mostly used for OS program, network operations and some website development (mostly cgi).

Write a script for 'count the no.of digits using regular expressions in perl..
#!/usr/bin/perl -w

```
use strict;
my($test,$number) = ("","");
$test = "12344tyyyyy456";
$number = ($test =~ tr/[0-9]/[0-9]/);
print "Number of digits in variable is :- $number ";
exit;
```

Que: How would you replace a char in string and how do you store the number of

replacements?

```
$str='Hello';
$cnt= ($str=~s/l/i/g);
print $cnt;
```

Que: What operators are used with regular expression?

Ans: tr(translate), g, i,s

Que: What is Hash?
Hash is an associative array where data is stored in "key"->"value" pairs.
Eg : fruits is a hash having their names and price
%fruits = ("Apple", "60", "Banana", "20", "Peers", "40")

Que: How do you open a file for writing?
open FILEHANDLE, ">$FILENAME"

Que: How to install a package in perl???

Ans:

perl -MCPAN -e shell

cpan> install HTML::Template

cpan> install Module::Name

or perl -MCPAN -e 'install HTML::Template'

But it's always advisable to download a module yourself, especially if you're having problems installing with CPAN. It is better to follow the below steps.You have a file ending in .tar.gz (or, less often, .zip). You know there's a tasty module inside. There are four steps you must now take:

- **DECOMPRESS** the file : *gzip -d yourmodule.tar.gz*
- **UNPACK** the file into a directory: *tar -xof yourmodule.tar*
- **BUILD** the module (sometimes unnecessary): Go into the newly-created directory and type: *cd yourmodule*

  perl Makefile.PL

  make

  make test

  INSTALL the module: *make install*

Que: Name an instance where you used a CPAN(Comprehensive Perl Archive Network) module?

we will get a no.of modules from CPAN ,
there so many module for database interface,
database drivers.
to do the mathematical operations ,text processing .
each module is to handle a small tasks .

There are no.of instances for this
For eg; we use DBI module for database connectivity,
and we also use CGI Module for parsing the form input and
printing the headers.

I like Test::More, Test::Most and Test::Exception as we can use ok, lives_ok, dies_ok, like, isa etc

Que: what is meant by a 'pack' in perl?

Pack Converts a list into a binary representation

Takes an array or list of values and packs it into a binary structure,
returning the string containing the structure
Takes a LIST of values and converts it into a string using the rules given by the TEMPLATE

Que: How to implement a stack in a perl?

Ans. Stack is LIFO (Last in First out), In perl that could be inplemented using the push() and shift() functions. push() adds the element at the last of array and shift() removes from the beginning of an array.

Que: What is eval in perl?
"eval" and "if ($@)" is the equivalent of a try/catch in the C
programming world. My question is how to basically throw an error in

one perl script that can be caught by another
Consider the following example:

```
eval {
die("REASON"); # the script dies for some reason
};
if ($@) # $@ contains the exception that was thrown
{print $@;
}
```

Que : What is Grep used for in Perl?

You can able check the array contains a particular value is exist or not? For example see the below code.

```
@array = (1,2,3,4,5,6,7,8,100,200,500); # array
$result = grep(/600/, @array); # checking the value.
print $result; # the result will be 0. (if 1 then it contains the value
```

Que: what does this mean :

'$_' ?
It is a default variable which hold automatically the list of arguments passed to subroutines within parentheses.
What is @_ ?
It is an array that contains all the data that is passed to a function.

Que. what are the characteristics of project that is well suited to

PERL?
actually it is used in automation testing....as a tester i am using per script for automation testing like running the tools etc.

Perl has a great text processing feature like grep, sed etc

Que: What is caller function in perl?

Returns information about the current subroutines caller. In a list context, with no arguments specified, caller returns the

- **package name,**
- **file name and**
- **line within the file for the caller of the current subroutine**

If EXPR is specified, caller returns extended information for the caller EXPR steps up. That is, when called with an argument of 1, it returns the information for the caller (parent i.e package name) of the current subroutine, with 2 the caller of the caller (grandparent i.e. filename) of the current subroutine, and so on

Return Value

- undef on failure
- Basic information when called with no arguments
- Extended information when called with an argument

Example:

Information returned when calling without argument:

($package, $filename, $line) = caller;

Information returned when calling with argument:

($package, $filename, $line, $subroutine, $hasargs, $wantarray, $evaltext, $is_require) = caller($i);

The $evaltext and $is_require values are only returned when the subroutine being examined is actually the result of an eval( ) statement.

Que: How to count no of occurrence of a unique patterns in perl?

$count = 1; while ($string =~ /<pattern>/g) { $count++ }

or

($count)=$string =~ s/<pattern>/<pattern>/g;

Que: how to search a unique pattern in a file by using perl hash map function ???

open(FH, "file");

@array = <FH>;

%seen = ();

@uniq = grep{!seen{$_}++} @array:

Que: How to use each with hash ?

Ans:

```
use strict;
my %table = qw/schmoe joe smith john simpson bart/;
my($key, $value);  # @cc{Declare two variables at once}
```

```
while ( ($key, $value) = each(%table) ) {
    # @cc{Do some processing on @scalar{$key} and @scalar{$value}}
}

Que: How do you print a hash, array or scalar with some other string ?

Ans: print "\$key_table is ", $key_table;

print "%coins is ", %coins;

print "\@kkk is ", @kkk;

Common mistakes:

Don't put any dot(.) while dealing with array or hash especially whether as it
will work for scalars. So always put a comma(,) while dealing with arrays or
hashes.

e.g.

print "\@kkk is ".@kkk; => This will give the length of the array
```

Que: How will you copy and move/rename a file in perl ?

Ans: use File::Copy <= This module can be used for copy and move both

e.g.

use File::Copy;

$filetobecopied = "myhtml.html.";

$newfile = "myhtml.html.";

copy($filetobecopied, $newfile) or die "File cannot be copied.";

move($oldlocation, $newlocation);

Que: What is the difference between die and exit ?

Ans: die prints out STDERR message in the terminal before exiting the program while exit jusst terminate the program without giving any message.

Que: How does Perl know about end of file while reading a file?

@array = <FILE>;

$last_line = $array[-1];

it is one of the ways but is there some better way as the file is pretty huge & there is memory crunch so reading the whole file seems to be really wasting a lot of memory. So the **best way** to do this to use ***eof()*** function as follows:

open (FH, "+< $file") or die "can't update $file: $!";

while (<FH>) {

$addr = tell(FH) unless eof(FH);

}

truncate(FH, $addr) or die "can't truncate $file: $!";

This sort of code is used to “Removing the Last Line of a File ”.This is much more efficient than reading the file into memory all at once, since it only holds one line at a time. Although you still have to grope your way through the whole file, you can use this program on files larger than available memory.

If we will print eof(FH) then it will return 1. If the file is a blank file then it will return nothing.

#Reference: http://oreilly.com/catalog/cookbook/chapter/ch08.html

Que: How do you reading a File Backwards by Line or Paragraph

```
@lines = reverse <FILE>;
foreach $line (@lines) {
   # do something with $line
}
```

Que: Why do we write ***1*** at the end of the package everytime ? What is ***warn*** ?

Ans:e.g.

Perl class names often use the StudlyCaps or CamelCase style. Use the name `TutorialConfig`. Because Perl looks for a module by its filename, the filename should be *TutorialConfig.pm*.

Put the following into a file called *TutorialConfig.pm*:

```
package TutorialConfig;

warn "TutorialConfig is successfully loaded!\n";

1;
```

(I've sprinkled debugging statements throughout the code. You can take them out in practice. The `warn` keyword is useful to bring things to the user's attention without ending the program the way `die` would.)

The `package` keyword tells Perl the name of the class you're defining. This is generally the same as the module name. (It doesn't *have* to be, but it's a good idea!) ***The `1;` will return a true value to Perl, which indicates that the module has loaded completely and successfully. If you forget this (and you will), Perl will give you an error message saying that your package did not return a true value.***

You now have a simple module called `TutorialConfig`, which you can use in your code with the `use` keyword. Run this very simple, one-line program:

```
use TutorialConfig;
```

... and you should see:

```
TutorialConfig is successfully loaded!
```

```
Que. How do we convert a floating point number to an integer ?

Ans. Best way to use is to use POSIX module which will give us the lower and
upper boundary. e.g.

    use POSIX;

    $ceil   = ceil(3.5);                # 4
```

```
    $floor  = floor(3.5);                # 3
```

Otherway we can do it is by ***printf*** or ***sprintf. e.g.***

```
my $rounded = sprintf "%.0f", $float;

printf("%.3f", 3.1415926535); # prints 3.142
```

We can use the Math module to round it up.e.g.

```
use Math::Round;

my $rounded = round( $float );
```

Pattern matching:

There are some special characters in Regular Expressions(or RE). These are {}[]()^$.|*+?\

They are also called metacharacters. They have special meaning in REs. Now lets see the meaning of every metacharacters...

Metacharacters	Description
*	0 or more matches of the atom
+	1 or more matches of the atom
?	0 or 1 matches of the atom
{m}	exactly m matches of the atom
{m,}	m or more matches of the atom
{m,n}	m through n (inclusive) matches of the atom

An atom is one of:

Atom	Description
(re)	(where re is any regular expression) matches a match for re, with the matched string stored as a variable
[chars]	a bracket expression, matching any one of the chars
.	matches any single character
\c	where c is alphanumeric (possibly followed by other characters), an escape.
^	matches at the beginning of a line
$	matches at the end of a line

Escapes

Escape	Description
\d	Matches any digit
\s	Matches space
\w	Matches any alphabet, any number and underscore(_)

\D Matches all non digit

\S Matches all non space

\n Matches newline

\r Matches carriage return

\b “word” boundary

\B not a “word” boundary

\xhh character with hex. code hh

Character sets: specialities inside [...]

Different meanings apply inside a character set (“character class”) denoted by [...] so that, instead of the normal rules given here, the following apply:

[characters] matches any of the characters in the sequence

[x-y] matches any of the characters from x to y (inclusively) in the ASCII code

[\-] matches the hyphen character “-”

[\n] matches the newline; other single character denotations with \ apply normally, too

[^something] matches any character except those that [something] denotes; that is, immediately after the leading “[”, the circumflex “^” means “not” applied to all of the rest

Examples

expression matches...

abc abc (that exact character sequence, but anywhere in the string)

^abc abc at the beginning of the string

abc$ abc at the end of the string

a|b either of a and b

^abc|abc$ the string abc at the beginning or at the end of the string

ab{2,4}c an a followed by two, three or four b’s followed by a c

ab{2,}c an a followed by at least two b’s followed by a c

ab*c an a followed by any number (zero or more) of b’s followed by a c

ab+c an a followed by one or more b’s followed by a c

ab?c an a followed by an optional b followed by a c; that is, either abc or ac

a.c an a followed by any single character (not newline) followed by a c

a\.c a.c exactly

[abc] any one of a, b and c

[Aa]bc either of Abc and abc

[abc]+ any (nonempty) string of a’s, b’s and c’s (such as a, abba, acbabcacaa)

[^abc]+ any (nonempty) string which does not contain any of a, b and c (such as defg)

\d\d any two decimal digits, such as 42; same as \d{2}

\w+ a “word”: a nonempty sequence of alphanumeric characters and low lines (underscores), such as foo and 12bar8 and foo_1

100\s*mk the strings 100 and mk optionally separated by any amount of white space (spaces, tabs, newlines)

abc\b abc when followed by a word boundary (e.g. in abc! but not in abcd)

perl\B perl when not followed by a word boundary (e.g. in perlert but not in perl stuff)

Examples of simple use in Perl statements

These examples use very simple regexps only. The intent is just to show contexts where regexps might be used, as well as the effect of some "flags" to matching and replacements. Note in particular that matching is by default case-sensitive (Abc does not match abc unless specified otherwise).

s/foo/bar/;

replaces the first occurrence of the exact character sequence foo in the "current string" (in special variable $_) by the character sequence bar; for example, foolish bigfoot would become barlish bigfoot

s/foo/bar/g;

replaces any occurrence of the exact character sequence foo in the "current string" by the character sequence bar; for example, foolish bigfoot would become barlish bigbart

s/foo/bar/gi;

replaces any occurrence of foo case-insensitively in the "current string" by the character sequence bar (e.g. Foo and FOO get replaced by bar too)

if(m/foo/)...

tests whether the current string contains the string foo

This is how it works...

RE	Description	Example
a?	match 'a' 1 or 0 times	a
a*	match 'a' 0 or more times, i.e., any number of times	aaaaaa
a+	match 'a' 1 or more times, i.e., at least once	aaaaaa
a{2,5}	match at least 2 times, but not more than 5 times.	aaaa
a{2,}	match at least 2 or more times	aaa
a{5}	match exactly n times	aaaaa

Que: How to start Perl in interactive mode ?

Ans: First install ***perlconsole (perl -MCPAN -e 'install perlconsole')***

$ perl -e -d | PerlConsole

or

$ perlconsole

Que: Assume both a local($var) and my($var) exist, what's the difference between ${var} and ${“var”} ?

Ans:

${var} is the lexical variable $var

${“var”} is the dynamic variable $var

Note: As the second is a symbol table lookup, it's disallowed under `use strict “refs”`. The words global, local, package, symbol table and dynamic all refer to the kind of variables that local() affects, whereas the other sort, those governed by my(), are variously knows as private, lexical, or scoped variable.

Que: What happens when you return a reference to a private variable ?

Ans: Perl keeps track of your variables, whether dynamic or otherwise, and doesn't free things before you're done using them.

Imp MySql questions:

1.Different Types of Joins-
Inner Join
Outer Join - Right Outer Join
Left Outer Join
Cross Join

2. How will you copy the structure of a table without copying the data?
create table xyz
as (select * from abc where 1=0)
Diffrence between a “where” clause and a “having” clause

Answers
1.”where” is used to filter records returned by “Select”
2.”where” appears before group by clause
3.In “where” we cannot use aggrigate functions like where count(*)>2 etc
4.”having” appears after group by clause
5.”having” is used to filter records returned by “Group by”
6.In”Having” we can use aggrigate functions like where count(*)>2 etc
there are two more

How to create a database link?
A database link is an object in the local database that allows you to access objects on a remote database or to mount a secondary database in read-only mode.

CREATE [PUBLIC] DATABASE LINK dblink
[CONNECT TO user IDENTIFIED BY password]
[USING ‘connect_string’]

What is the difference among “dropping a table”, “truncating a table” and “deleting all records” from a table.
Drop Table - will remove the existance of the table from the database along with its data and structure and all the constraints. The table will be no longer available.

Truncate Table - will remove all the rows (only the rows) from a table. it will not delete the table. Its a DDL statement that means the deleted rows cannot be reverted back by ROLLBACK statement;

Delete Table - is a DML statement which will delete rows from a table according to the matching criteria mentions in the ‘where’ clause. And these rows can be reverted back by ‘ROLLBACK’ statement if ‘COMMIT’ is not fired.

Table 12.2 - Perl's Special Variables

Important Ones: $_, $0, @ARGV, @INC, %ENV, %SIG, <>, <DATA> and __DATA__, $!, $@

Variable Name	Description
Variables That Affect Arrays	
$"	**The separator used between list elements when an array variable is interpolated into a double-quoted string. Normally, its value is a space character.** **$" = "-";** **@ar = ("one", "two", "three");** **print "@ar";** **will print one-two-three.**
$[	Holds the base array index. Normally, set to 0. Most Perl authors recommend against changing it without a very good reason.
$;	Holds the subscript separator for multi-dimensional array emulation. If you refer to an associative array element as: $foo{$a,$b,$c} it really means: $foo{join($;, $a, $b, $c)} but don't put @foo{$a,$b,$c} which means ($foo{$a},$foo{$b},$foo{$c})
Variables Used with Files (See Chapter 9, "Using Files")	
$.	This variable holds the current record or line number of the file handle last read. It is read-only and will be reset to 0 when the file handle is closed.
$/	**This variable holds the input record separator. The record separator is usually the newline character. However, if $/ is set to an empty string, two or more newlines in the input file will be treated as one.**
$|	This variable, if nonzero, will flush the output buffer after every `write()` or `print()` function. Normally, it is set to 0.

`$^F`	This variable holds the value of the maximum system file description. Normally, it's set to 2. The use of this variable is beyond the scope of this book.
`$ARGV`	This variable holds the name of the current file being read when using the diamond operator `(<>)`.
`_`	This file handle (the underscore) can be used when testing files. If used, the information about the last file tested will be used to evaluate the latest test.
`DATA`	This file handle refers to any data following `__END__`.
`STDERR`	This file handle is used to send output to the standard error file. Normally, this is connected to the display, but it can be redirected if needed.
`STDIN`	This file handle is used to read input from the standard input file. Normally, this is connected to the keyboard, but it can be changed.
`STDOUT`	This file handle is used to send output to the standard output file. Normally, this is the display, but it can be changed.

Variables Used with Patterns (See Chapter 10, "Regular Expressions")

`$&`	**This variable holds the string that was matched by the last successful pattern match.**
`$``	This variable holds the string that preceded whatever was matched by the last successful pattern match. **$_ = 'abcdefghi';** **/def/;** **print "$`:$&:$'\n"; # prints abc:def:ghi**
`$'`	This variable holds the string that followed whatever was matched by the last successful pattern match.
`$+`	This variable holds the string matched by the last bracket in the last successful pattern match. For example, the statement `/Fieldname: (.*)|Fldname: (.*)/ && ($fName = $+);` will find the name of a field even if you don't know which of the two possible spellings will be used. **/Version: (.*)\|Revision: (.*)/ && ($rev = $+);**
`$*`	This variable changes the interpretation of the `^` and `$` pattern anchors. Setting `$*` to 1 is the same as using the `/m` option with the regular expression matching and substitution operators. Normally, `$*` is equal to 0.
`$<number>`	This group of variables (`$1`, `$2`, `$3`, and so on) holds the regular expression pattern memory. Each set of parentheses in a pattern stores the string that matches the components surrounded by the parentheses into one of the `$<number>` variables.

Variables Used with Printing

`$,` This variable is the output separator for the `print()` function. Normally, this variable is an empty string. However, setting `$,` to a newline might be useful if you need to print each element in the parameter list on a separate line.

`$\` **The variable is added as an invisible last element to the parameter list passed to the `print()` function. Normally, it's an empty string, but if you want to add a newline or some other suffix to everything that is printed, you can assign the suffix to `$\`.**

`$#` **This variable is the default format for printed numbers. Normally, it's set to `%.20g`, but you can use the format specifiers covered in by the section "Example: Printing Revisited" in Chapter 9 to specify your own default format.**

Variables Used with Processes (See Chapter 13, "Handing Exceptions and Signals")

`$$` **This UNIX-based variable holds the process number of the process running the Perl interpreter.**

`$?` **This variable holds the status of the last pipe close, back-quote string, or `system()` function. More information about the `$?` variable can be found in Chapter 13, "Handling Exceptions and Signals."**

e.g. print $?
If it prints **0 then successful,**
> 0 (non zero) => unsuccessful

`$0` **This variable holds the name of the file containing the Perl script being executed.**

`$]` This variable holds a string that identifies which version of Perl you are using. When used in a numeric context, it will be equal to the version number plus the patch level divided by 1000.

`$!` **This variable, when used in a numeric context, holds the current value of `errno`. If used in a string context, it will hold the error string associated with `errno`. For more information about `errno`, see Chapter 13, "Handling Exceptions and Signals."**
e.g. open($fh, "| /usr/bin/osascript >/dev/null") or die "cant open osascript $!";

`$@` **This variable holds the syntax error message, if any, from the last `eval()` function call. For more information about `errno`, see Chapter 13, "Handling Exceptions and Signals."**

`$<` This UNIX-based variable holds the read uid of the current process.

`$>` This UNIX-based variable holds the effective uid of the current process.

`$)`	This UNIX-based variable holds the read gid of the current process. If the process belongs to multiple groups, then `$)` will hold a string consisting of the group names separated by spaces.
`$^T`	This variable holds the time, in seconds, at which the script begins running.
`$^X`	This variable holds the full path name of the Perl interpreter being used to run the current script.
`%ENV`	This hash variable contains entries for your current environment variables. Changing or adding an entry will affect only the current process or a child process, never the parent process. $ENV{'PATH'} = '/bin:/usr/bin'; # Change to your real path Delete @ENV{'IFS','CDPATH','ENV','BASH_ENV'};
`%SIG`	Perl has a rich signal handling capabilities; using the %SIG variable, you can make any subroutine run when a signal is sent to the running process. This is especially useful if you have a long-running process, and would like to reload configuration files by sending a signal (using SIGHUP) instead of having to start and stop the process. You can also change the behaviour of die and warn by assigning $SIG{__DIE__} and $SIG{__WARN__}, respectively.

Variables Used with Reports (see Chapter 11, "Creating Reports")

`$%`	This variable holds the current page number for the default file handle. If you use `select()` to change the default file handle, `$%` will change to reflect the page number of the newly selected file handle.
`$=`	This variable holds the current page length for the default file handle. Changing the default file handle will change `$=` to reflect the page length of the new file handle.
`$-`	This variable holds the number of lines left to print for the default file handle. Changing the default file handle will change `$-` to reflect the number of lines left to print for the new file handle.
`$~`	This variable holds the name of the default line format for the default file handle. Normally, it is equal to the file handle's name.
`$^`	This variable holds the name of the default heading format for the default file handle. Normally, it is equal to the file handle's name with `_TOP` appended to it.
`$:`	This variable holds a string that consists of the characters that can be used to end a word when word-wrapping is performed by the `^` report formatting character. Normally, the string consists of the space, newline, and dash characters.
`$^L`	This variable holds the string used to eject a page for report printing. Chapter 11, "Creating Reports," shows how to use this variable to create simple footers.

Miscellaneous Variables

`$_`	This variable is used as the default parameter for a lot of functions. **while (<>) {... # equivalent only in while!** **while ($_ =<>) {...** **/^Subject:/** **$_ =~ /^Subject:/** **y/a-z/A-Z/** **$_ =~ y/a-z/A-Z/** **chop** **chop($_)**
`$^D`	This variable holds the current value of the debugging flags. For more information, see Chapter 16, "Debugging Perl."
`$^I`	This variable holds the file extension used to create a backup file for the in-place editing specified by the `-i` command line option. For example, it could be equal to ".bak."
`$^P`	This variable is an internal flag that the debugger clears so that it will not debug itself.
`$^W`	This variable holds the current value of the `-w` command line option.
`@ARGV`	This array variable holds a list of the command line arguments. You can use `$#ARGV` to determine the number of arguments minus one.
`@F`	This array variable holds the list returned from autosplit mode. Autosplit mode is associated with the `-a` command line option.
`@INC`	This array variable holds a list of directories where Perl can look for scripts to execute. The list is used mainly by the require statement. You can find more information about `require` statements in Chapter 15, "Perl Modules."
`%INC`	This hash variable has entries for each filename included by `do` or `require` statements. The key of the hash entries are the filenames and the values are the paths where the files were found.

<DATA> and __DATA__:

If a program contains the mafic token __DATA__ on a line by itself, anything following it will be a variable to the program through the magic <DATA> filehandle.

This is useful if you want to include data with your program, but want to keep it separated from the main program logic.

<>: Diamond operator or STDIN expecting user input

Perl Libraries and Modules:

==========

Perl Libraries

Perl has become a language of choice for World Wide Web development, text processing, Internet services, mail filtering, systems administration, and most every other task requiring a portable and easily developed solution. Your virtual server has the Perl interpreter already installed at the following location.

~/usr/local/bin/perl

If you require the use of the Perl Standard Libraries or other Perl modules, you will need to install these into a local directory on your virtual server.

Installing the Perl Standard Libraries

Connect to your virtual server via Telnet or SSH and run the following commands that match your virtual server operating system.

FreeBSD

```
% vinstall perl5
```

Removing the Perl Standard Libraries

If you would like to remove the Perl Standard Libraries you may do so by running the following commands that match your virtual server operating system.

FreeBSD

```
% vrmperl
```

Installing Perl Modules

Perl Modules can greatly extend the functionality of your Perl programming language interpreter. By using prepared modules written by others, instead of using your own code, you can save yourself both time and effort. Many popular Perl5 modules can be easily installed on your virtual server.

Pre-Packaged Perl Modules

The following is a list of pre-packaged Perl Modules that can be installed on your virtual server.

perl-lwp

perl-lwp contains the LWP (Library for WWW access in Perl) modules as well as their dependencies. Note that Bundle::libnet (perl-libnet below) is required for optimal performance.

perl-mysql

perl-mysql contains the DBD::mysql database driver for Perl's DBI module (which must be installed separately, using vcpan).

perl-pg

perl-pg contains Pg and DBD::Pg, two Perl5 modules built to access PostgreSQL databases. DBD::Pg is for use with Perl's DBI module (which you must install separately) while Pg is for standalone access to Postgres.

perl-msql (FreeBSD only)

perl-msql includes all the necessary Perl libraries to talk to an mSQL database via Perl's DBI module (which must be installed separately, using vcpan).

perl-libnet

This is a collection of Perl5 modules which provides a simple and consistent programming interface (API) to the client side of various protocols used in the internet community.

perl-gd

This is an autoloadable interface module for libgd, a popular library for creating and manipulating PNG files. With this library you can create PNG images on the fly or modify existing files.

perl-xml_parser

XML::Parser is an XML parser written in Perl. (It makes use of Expat, so Expat must be installed on your server if you're going to use XML::Parser. Use vinstall expat.)

Installation

Connect to your server via Telnet or SSH and do the following, depending on your virtual server O/S.

FreeBSD

% vinstall MODULE-NAME

Installing Other Perl Modules

If you require a module that is not included above or in the Perl Standard Libraries, you may be able use the vcpan utility to install it.

Using vcpan to install Perl modules

For more information about installing Perl modules:

Installing Your Own Perl Modules

perldoc - Perl Documentation Viewer

Do the following to install the perldoc utility, which you can use to view Perl documentation, on your virtual server.

FreeBSD

% vinstall perldoc

This command links in a variety of required terminal macro definitions as well as several groff/troff/nroff files required for proper man page formatting.

Once installed, you may run the following command to access documentation for your favorite Perl5 module. Substitute the module name for MODULE::FAVORITE below.

FreeBSD

% virtual perldoc MODULE::FAVORITE

Que: Describe about functions split, join, slice, splice.

Ans: (Ref: http://www.misc-perl-info.com/perl-splice.html)

Split: splits up a string and places it into an array

$_ = "Capes:Geoff::Shot putter:::Big Avenue";

```
@personal = split(/:+/);

@chars = split(//, $word);
@words = split(/ /, $sentence);
@sentences = split(/\./, $paragraph);
```

Splice: Perl splice function is for arrays and is used to remove, replace or add elements. It has four syntax forms:

```
splice ARRAY, OFFSET, LENGTH, LIST
splice ARRAY, OFFSET, LENGTH
splice ARRAY, OFFSET
splice ARRAY
```

where:

* ARRAY is the array to manipulate
* OFFSET is the starting array element to be removed
* LENGTH is the number of elements to be removed
    - LIST is an ordered list of elements to replace the removed elements with

```
splice ARRAY, OFFSET, LENGTH, LIST
```

It is the general form of this function and it uses all its arguments. Here's an example:

```
# initialize an array
my @array = qw(1 2 3 11 12 10);
my @returnedArray = splice @array, 3, 2, 4..9;

print "\@array = (@array)\n";
print "\@returnedArray = (@returnedArray)\n";
```

This code produces as output:

```
@array = (1 2 3 4 5 6 7 8 9 10)
```

@returnedArray = (11 12)

The second syntax form

splice ARRAY, OFFSET, LENGTH

In this syntax form, the LIST argument is omitted and you can use this format any time you need to remove a chunk of elements from an array. See the example below for this:

```
# initialize and fill an array
my @array = qw(1 2 3 4 5 6 7 8 9 10);
my @returnedArray = splice @array, 5, 7;

print "\@array = (@array)\n";
print "\@returnedArray = (@returnedArray)\n";
```

The output:

```
@array = (1 2 3 4 5)
@returnedArray = (6 7 8 9 10)
```

This code will remove the elements of the array starting with the index 5 (the sixth element of the array) and it will try to remove 7 elements. Because there are only 5 elements available, it will remove the elements until the end of the array is reached.

The third syntax form

splice ARRAY, OFFSET

In this syntax form, both the LIST and the LENGTH argument are omitted. In this case the Perl splice function will remove all the elements from OFFSET onward. If OFFSET is negative, it starts that far from the end of the array. See the following snippet code:

```
# initialize and fill an array
my @array = qw(1 2 3 4 5 6 7 8 9 10);

# using a positive OFFSET
splice @array, 8;
# the @array is now (1, 2, 3, 4, 5, 6, 7, 8)

# using a negative OFFSET
splice @array, -3;
# the @array is now (1, 2, 3, 4, 5)
```

In the last example of this code I used a negative OFFSET (-3) – as result the Perl splice function removed the last 3 elements of the array.

The fourth syntax form

splice ARRAY

In this format, the Perl splice function has only one argument – the array variable. In this case it removes all the elements of the array.

```
my @array = qw(1 2 3 4 5 6 7 8 9 10);
splice @array;
```

This code will practically empty the array. If you need to empty an array, you may consider other options, too:

```
@array = ();  # or
undef @array; # or
$#array = -1
```

Slice:

There is no specific slice() function for slicing up elements of an array. Instead PERL allows us to create a new array with elements of another array using array indexing.

slicendice.pl:

```
#!/usr/bin/perl

print "content-type: text/html \n\n"; #HTTP HEADER

@coins = qw(Quarter Dime Nickel Penny);
@slicecoins = @coins[0,2];
print "@slicecoins\n";
print "<br />";
```

When handling lists of sequential numbers, the range operator can quickly become your favorite tool for slicing up arrays.

myrangefriend.pl:

```
#!/usr/bin/perl

print "content-type: text/html \n\n"; #HTTP HEADER

# SEQUENTIAL ARRAY
@nums = (1..200);
@slicenums = @nums[10..20,50..60,190..200];
print "@slicenums";
```

myrangefriend.pl:

11 12 13 14 15 16 17 18 19 20 21 51 52 53 54 55 56 57

58 59 60 61 191 192 193 194 195 196 197 198 199 200

join: function is used to concatenate the elements of an array or a list into a string, using a separator given by a scalar variable value.

e.g.

```
#initialize an array
my @perlFunc = ("substr","grep","defined","undef");
my $perlFunc = join " ", @perlFunc;
print "Perl Functions: $perlFunc\n";
```

If you run it, you'll get as output:

Perl Functions: substr grep defined undef

Que: What is @ISA variable in perl or what is ***class inheritance*** in perl ?

Ans: http://docstore.mik.ua/orelly/perl2/prog/ch12_05.htm

As with the rest of Perl's object system, inheritance of one class by another requires no special syntax to be added to the language. When you invoke a method for which Perl finds no subroutine in the invocant's package, that package's `@ISA` array[5] is examined. This is how Perl implements inheritance: each element of a given package's `@ISA` array holds the name of another package, which is searched when methods are missing. For example, the following makes the `Horse` class a subclass of the `Critter` class. (We declare `@ISA` with `our` because it has to be a package variable, not a lexical declared with `my`.)

```
package Horse; our @ISA = "Critter";
```

You should now be able to use a `Horse` class or object everywhere that a `Critter` was previously used. If your new class passes this *empty subclass test*, you know that `Critter` is a proper base class, fit for inheritance.

[5] Pronounced "is a", as in "A horse is a critter."

Suppose you have a `Horse` object in `$steed` and invoke a `move` method on it:

```
$steed->move(10);
```

Because `$steed` is a `Horse`, Perl's first choice for that method is the `Horse::move` subroutine. If there isn't one, instead of raising a run-time exception, Perl consults the first element of `@Horse::ISA`, which directs it to look in the `Critter` package for `Critter::move`. If this subroutine isn't found either, and `Critter` has *its* own `@Critter::ISA` array, then that too will be consulted for the name of an ancestral package that might supply a `move` method, and so on back up the inheritance hierarchy until we come to a package without an `@ISA`.

If `@ISA` contains more than one package name, the packages are all searched in left-to-right order. The search is depth-first, so if you have a `Mule` class set up for inheritance this way:

```
package Mule;

our @ISA = qw(Horse Donkey);        # our @ISA = ("Horse", "Donkey")
```

Perl looks for any methods missing from `Mule` first in `Horse` (and any of its ancestors, like `Critter`) before going on to search through `Donkey` and its ancestors.

When Perl searches for a method, it makes sure that you haven't created a circular inheritance hierarchy. This could happen if two classes inherit from one another, even indirectly through other classes. Trying to be your own great-grandfather is too paradoxical even for Perl, so the attempt raises an exception. However, Perl does not consider it an error to inherit from more than one class sharing a common ancestry, which is rather like cousins marrying. Your inheritance hierarchy just

stops looking like a tree and starts to look like a directed acyclic graph. This doesn't bother Perl--so long as the graph really is acyclic.

When you set `@ISA`, the assignment normally happens at run time, so unless you take precautions, code in `BEGIN`, `CHECK`, or `INIT` blocks won't be able to use the inheritance hierarchy. One precaution (or convenience) is the `use base` pragma, which lets you `require` classes and add them to `@ISA` at compile time. Here's how you might use it:

```
package Mule; use base ("Horse", "Donkey"); # declare
superclasses
```

This is a shorthand for:

```
package Mule; BEGIN { our @ISA = ("Horse", "Donkey");
require Horse; require Donkey; }
```

except that `use base` also takes into account any `use fields` declarations.

When you invoke a method *methname* on an invocant of type *classname*, Perl tries six different ways to find a subroutine to use:

1. First, Perl looks in the invocant's own package for a subroutine named *classname*`::`*methname*. If that fails, inheritance kicks in, and we go to step 2.
2. Next, Perl checks for methods inherited from base classes by looking in all *parent* packages listed in `@`*classname*`::ISA` for a *parent*`::`*methname* subroutine. The search is left-to-right, recursive, and depth-first. The recursion assures that grandparent classes, great-grandparent classes, great-great-grandparent classes, and so on, are all searched.
3. If that fails, Perl looks for a subroutine named `UNIVERSAL::`*methname*.
4. At this point, Perl gives up on *methname* and starts looking for an `AUTOLOAD`. First, it looks for a subroutine named *classname*`::AUTOLOAD`.
5. Failing that, Perl searches all *parent* packages listed in `@`*classname*`::ISA`, for any *parent*`::AUTOLOAD` subroutine. The search is again left-to-right, recursive, and depth-first.
6. Finally, Perl looks for a subroutine named `UNIVERSAL::AUTOLOAD`.

Perl stops after the first successful attempt and invokes that subroutine. If no subroutine is found, an exception is raised, one that you'll see frequently:

```
Can't locate object method "methname" via package "classname"
```

Que:

i. How do you execute a simple global variable to print in shell ?

ii. What is @INC in perl ?

Ans:

i. $ perl -e "print @INC"

ii. The @INC array is a list of directories Perl searches when attempting to load modules. To display the current contents of the @INC array:

perl -e "print join(\"\n\", @INC);"

e.g.

samir@samir-laptop:~$ perl -e "print join(\"\n\", @INC);"

/etc/perl

/usr/local/lib/perl/5.10.0

/usr/local/share/perl/5.10.0

/usr/lib/perl5

/usr/share/perl5

/usr/lib/perl/5.10

/usr/share/perl/5.10

/usr/local/lib/site_perl

.samir@samir-laptop:~$

The following two methods may be used to append to Perl's @INC array:

1. Add the directory to the PERL5LIB environment variable.
2. Add use lib 'directory'; in your Perl script.

Que: How many ways you can express a string ?

Many. For example 'this is a string' can be expressed in:
"this is a string"
qq/this is a string like double-quoted string/
qq^this is a string like double-quoted string^
q/this is a string/
q&this is a string&
q(this is a string)

Que: What is the difference between compile time and run time ?

Ans:"Compile time" is when you build your code - . In order to be understood the language by machine it compiler converts your source code into Machine language

"Runtime" is when your code is executed

In short, C is a high level language which means it is meant for humans to understand not computers. For a computer to understand a C program it has to be compiled, that is changed into machine language. Once a program is in machine language then the computer can understand it and perform the actions it describes (that is to run the program). A C program need only be compiled once but (once compiled) it can be run any number of times.

runtime error: an error that occurs then your program is run such as division by 0, buffer overruns and corrupt stack. There are an infinite (or nearly infinite) number of these errors so its not possible to list them all.

compile time error: errors that your compiler produces when it tries to build the program.

Que: How to install Moose and what is it ?

Ans:

```
sudo apt-get install libmoose-perl
```

```
Description:
```

```
===========
```

Moose, it’s the new Camel. Moose is an extension of the existing Perl 5 object system, inspired by the object model available in Perl 6. Because Moose is built on top of Class::MOP (libclass-mop-perl), it is easier to build normal Perl objects, while also providing the power of metaclass programming. It is designed to provide as much convenience during definition and construction of classes as possible, but can still stay out of your way. Moose also conveniently manages all attributes (including inherited ones) that are defined, but also provides facilities for properly initializing instance slots, setting defaults where appropriate and performing any necessary type constraint checking or coercion. More details about the structure of Moose as well as its features can be found in the ever-expanding Moose::Cookbook document.

```
@names = qw(john alice christine william robert);
foreach (@names) {
    $_ = uc($_);
}
map ($_ = lc($_), @names);
@names = map {ucfirst} @names;
```

Que: How to write simple SSH script in perl ?

```
#!/usr/bin/perl

use Net::SSH::Perl;

my $host = "perlhowto.com";
my $user = "user";
my $password = "password";

#-- set up a new connection
my $ssh = Net::SSH::Perl->new($host);
#-- authenticate
$ssh->login($user, $pass);
#-- execute the command
my($stdout, $stderr, $exit) = $ssh->cmd("ls -l /home/$user");
```

-- IMPORTANT NOTE!!

Execution of Net::SSH::Perl commands can be quite slow if you don't have the Math::BigInt::GMP module; so be sure that you have that module installed (or install it if you don't) in order to avoid the slowness problem.

Que: How to code Perl to implement the tail function in unix ?

Ans: You have to maintain a structure to store the line number and size of the file at that time e.g. 1-10bytes, 2-18bytes. You have a counter to increase the number of lines to find out the number of lines in the file. Once you are through the file, you will know the size of the file at any nth line, use 'sysseek' to move the file pointer back to that position(last 10) and then start reading till the end.

Que: How to create and remove a directory in perl ?

Ans:

mkdir $dir_name

rmtree($dir) => removes the whole directory tree; also you can use the module File::Remove to delete the directory

Anonymous:

Parsing Styles

`XML::Parser` supports several different styles of parsing to suit various development strategies. The style doesn't change how the parser reads XML. Rather, it changes how it presents the results of parsing. If you need a persistent structure containing the document, you can have it. Or, if you'd prefer to have the parser call a set of routines you write, you can do it that way. You can set the style when you initialize the object by setting the value of `style`. Here's a quick summary of the available styles:

Debug
: This style prints the document to `STDOUT`, formatted as an outline (deeper elements are indented more). *parse( )* doesn't return anything special to your program.

Tree
: This style creates a hierarchical, tree-shaped data structure that your program can use for processing. All elements and their data are crystallized in this form, which consists of nested hashes and arrays.

Object
: Like `tree`, this method returns a reference to a hierarchical data structure representing the document. However, instead of using simple data aggregates like hashes and lists, it consists of objects that are specialized to contain XML markup objects.

Subs
: This style lets you set up *callback functions* to handle individual elements. Create a package of routines named after the elements they should handle and tell the parser about this package by using the `pkg` option. Every time the parser finds a start tag for an element called `<fooby>`, it will look for the function *fooby( )* in your package. When it finds the end tag for the element, it will try to call the function *_fooby( )* in your package. The parser will pass critical information like references to content and attributes to the function, so you can do whatever processing you need to do with it.

Stream
: Like `Subs`, you can define callbacks for handling particular XML components, but callbacks are more general than element names. You can write functions called *handlers* to be called for "events" like the start of an element (any element, not just a particular kind), a set of character data, or a processing instruction. You must register the handler package with either the `Handlers` option or the *setHandlers( )* method.

custom
: You can subclass the `XML::Parser` class with your own object. Doing so is useful for creating a parser-like API for a more specific application. For example, the `XML::Parser::PerlSAX` module uses this strategy to implement the SAX event

processing standard.

Que: What does '$result = f()..g()' really return ?

Ans:
False so long as f() returns false, after which it returns true until g() returns true and starts the cycle again.
As double dot is a range operator, for the first time f() returns true, g() is entirely ignored, adn f() will be ignored while g() later when g() is evaluated. F() and g() will both be evaluated on the same record. If you don't want that to happen, then exclusive range operator, triple dots can be used instead.

Que: Why does Perl not have overloaded functions like C++ ???
Ans:

Que: What is ternary operator in Perl ? What is wantarray and how do you use it with ternary operator ?
Ans:
?: => Ternary operator

```
CONDITION-PART ? TRUE-PART : FALSE-PART
```

which is shorthand for the following statement:

```
if (CONDITION-PART) {
    TRUE-PART
} else {
    FALSE-PART
}
```

wantarray returns true if the context of the currently executing function is looking for a list value. Returns false in a scalar context

```
e.g.
sub foo {
  return(wantarray() ? (“A”, “B”, “C”) : '1');
}

$result = foo();    # scalar context
@result = foo();    # array context

print("foo(): $result\n");      # 1
print("foo(): @result\n");      # A, B, C
```

Que: Why does Perl not have overloaded functions ?

Ans:

Because you can inspect the argument count, return context, and object tyupes all by yourself. In Perl, the no of arguments is trivially available to a function via the scalar sense of @_, the return context via wantarray(), and the types of the arguments via ref() if they've references and simple pattern matching like /^\d+$/ otherwise. In languages like C++ where you can't do this, you simply must resort to overloading of functions.

Que: What does read() or sysread() return at end of file ?

Ans: 0

Que: What are Packages ? What is Begin and End in Perl ?

- A package is a collection of code which lives in its own namespace
- A namespace is a named collection of unique variable names (also called a symbol table).
- Namespaces prevent variable name collisions between packages

You may define any number of code blocks named BEGIN and END which act as constructors and destructors respectively.

```
BEGIN { ... }
END { ... }
BEGIN { ... }
END { ... }
```

- Every **BEGIN** block is executed after the perl script is loaded and compiled but before any other statement is executed
- Every END block is executed just before the perl interpreter exits.
- The BEGIN and END blocks are particularly useful when creating Perl modules.

Que: How to create the Perl Module Tree ?

Or How do you start Perl programming from start ?

When you are ready to ship your PERL module then there is standard way of creating a Perl Module Tree. This is done using **h2xs** utility. This utility comes alongwith PERL. Here is the syntax to use h2xs

```
$h2xs -n  Module::Name

# For example, if your module is available in Person.pm file
$h2xs -n Person::Samir

This will produce following result
Writing Person-Samir/lib/Person/Samir.pm
Writing Person-Samir/Makefile.PL
Writing Person-Samir/README
Writing Person-Samir/t/Person.t
Writing Person-Samir/Changes
Writing Person-Samir/MANIFEST
```

Here is the descritpion of these options (h2xs -AX -n Person::Samir => you can use these options too)

- **-A** omits the Autoloader code (best used by modules that define a large number of infrequently used subroutines)
- **-X** omits XS elements (eXternal Subroutine, where eXternal means external to Perl, i.e. C)
- **-n** specifies the name of the module

So above command creates the following structure inside Person directory. Actual result is shown above.

- Changes

- Makefile.PL
- MANIFEST (contains the list of all files in the package)
- README
- t/ (test files)
- lib/ (Actual source code goes here

So finally you **tar** this directory structure into a file Person.tar and you can ship it. You would have to update README file with the proper instructions. You can provide some test examples files in t directory.

Installing Perl Module

Installing a Perl Module is very easy. Use the following sequence to install any Perl Module.

```
perl Makefile.PL
make
make install
```

Que: What is fork ? What is its return value ? Give an example.

Ans: fork is used to create parallel processes. You must ensure that you wait on your children to prevent "zombie" processes from forming.

Return Value

- undef on failure to fork
- Child process ID to parent on success
    - and 0 to child on success

Example

Following are the usage...

```
#!/usr/bin/perl

$pid = fork();
if( $pid == 0 ){
   print "This is child process\n";
   print "Chile process is existing\n";
   exit 0;
}
else
{
    print "This is parent process and child ID is
$pid\n";
    print "Parent process is existing\n";
    exit 0;
}

Output:
#This will produce following result
```

```
This is parent process and child ID is 16417
Parent process is existing
This is child process
Chile process is existing
```

Que: How to create softlink and hardlink in perl like in linux ?

Softlink(ln -s src_file dest_file):

symlink("src_file", "dest_file")

Hardlink(ln src_file dest_file):

link("src_file", "dest_file")

OOP:

Defining a Class(http://www.tutorialspoint.com/perl/perl_oo_perl.htm)

Its very simple to define a class. In Perl, a class is corresponds to a Package.To create a class in Perl, we first build a package. A package is a self-contained unit of user-defined variables and subroutines, which can be re-used over and over again. They provide a separate namespace within a Perl program that keeps subroutines and variables from conflicting with those in other packages.

To declare a class named Person in Perl we do:

```
package Person;
```

The scope of the package definition extends to the end of the file, or until another package keyword is encountered.

Creating and Using Objects

To create an instance of a class (an object) we need an object constructor. This constructor is a method defined within the package. Most programmers choose to name this object constructor method new, but in Perl one can use any name.

One can use any kind of Perl variable as an object in Perl. Most Perl programmers choose either references to arrays or hashes.

Let's create our constructor for our Person class using a Perl hash reference;

When creating an object, you need to supply a constructor. This is a subroutine within a package that returns an object reference. The object reference is created by blessing a reference to the package's class. For example:

```
package Person;
sub new
{
```

```
    my $class = shift;
    my $self = {
        _firstName => shift,
        _lastName  => shift,
        _ssn       => shift,
    };
    # Print all the values just for clarification.
    print "First Name is $self->{_firstName}\n";
    print "Last Name is $self->{_lastName}\n";
    print "SSN is $self->{_ssn}\n";
    bless $self, $class;
    return $self;
}
```

Every method of a class passes first argument as class name. So in the above example class name would be "Person". You can try this out by printing value of $class. Next rest of the arguments will be rest of the arguments passed to the method.

Now Let us see how to create an Object

```
$object = new Person( "Mohammad", "Saleem", 23234345);
```

You can use simple hash in your consturctor if you don't want to assign any value to any class variable. For example

```
package Person;
sub new
{
    my $class = shift;
    my $self = {};
    bless $self, $class;
    return $self;
}
```

Defining Methods

Other object-oriented languages have the concept of security of data to prevent a programmer from changing an object data directly and so provide accessor methods to modify object data. Perl does not have private variables but we can still use the concept of helper functions methods and ask programmers to not mess with our object innards.

Lets define a helper method to get person first name:

```
sub getFirstName {
    return $self->{_firstName};
}
```

Another helper function to set person first name:

```
sub setFirstName {
    my ( $self, $firstName ) = @_;
    $self->{_firstName} = $firstName if defined($firstName);
    return $self->{_firstName};
}
```

Lets have a look into complete example: Keep Person package and helper functions into Person.pm file

```
#!/usr/bin/perl

package Person;

sub new
{
    my $class = shift;
    my $self = {
        _firstName => shift,
        _lastName  => shift,
        _ssn       => shift,
    };
    # Print all the values just for clarification.
    print "First Name is $self->{_firstName}\n";
    print "Last Name is $self->{_lastName}\n";
    print "SSN is $self->{_ssn}\n";
    bless $self, $class;
    return $self;
}
sub setFirstName {
    my ( $self, $firstName ) = @_;
    $self->{_firstName} = $firstName if defined($firstName);
    return $self->{_firstName};
}

sub getFirstName {
    my( $self ) = @_;
    return $self->{_firstName};
}
1;
```

Now create Person object in mail.pl fileas follows

```
#!/usr/bin/perl

use Person;

$object = new Person( "Mohammad", "Saleem", 23234345);
# Get first name which is set using constructor.
$firstName = $object->getFirstName();

print "Before Setting First Name is : $firstName\n";

# Now Set first name using helper function.
$object->setFirstName( "Mohd." );

# Now get first name set by helper function.
$firstName = $object->getFirstName();
print "Before Setting First Name is : $firstName\n";

This will produce following result
First Name is Mohammad
Last Name is Saleem
SSN is 23234345
Before Setting First Name is : Mohammad
Before Setting First Name is : Mohd.
```

Inheritance

Object-oriented programming sometimes involves inheritance. Inheritance simply means allowing one class called the Child to inherit methods and attributes from another, called the Parent, so you don't have to write the same code again and again. For example, we can have a class Employee which inherits from Person. This is referred to as an "isa" relationship because an employee is a person. Perl has a special variable, @ISA, to help with this. @ISA governs (method) inheritance.

Following are noteable points while using inheritance

- Perl searches the class of the specified object for the specified object.
- Perl searches the classes defined in the object class's @ISA array.
- If no method is found in steps 1 or 2, then Perl uses an AUTOLOAD subroutine, if one is found in the @ISA tree.
- If a matching method still cannot be found, then Perl searches for the method within the UNIVERSAL class (package) that comes as part of the standard Perl library.
- If the method still hasn't been found, then Perl gives up and raises a runtime exception.

So to create a new Employee class that will inherit methods and attributes from our Person class, we simply code: Keep this code into Employee.pm

```
#!/usr/bin/perl

package Employee;
use Person;
use strict;
our @ISA = qw(Person);    # inherits from Person
```

Now Employee Class has all the methods and attributes inherited from Person class and you can use it as follows: Use main.pl file to test it

```
#!/usr/bin/perl

use Employee;

$object = new Employee( "Mohammad", "Saleem", 23234345);
# Get first name which is set using constructor.
$firstName = $object->getFirstName();

print "Before Setting First Name is : $firstName\n";

# Now Set first name using helper function.
$object->setFirstName( "Mohd." );

# Now get first name set by helper function.
$firstName = $object->getFirstName();
print "After Setting First Name is : $firstName\n";

This will produce following result
First Name is Mohammad
Last Name is Saleem
SSN is 23234345
Before Setting First Name is : Mohammad
Before Setting First Name is : Mohd.
```

Method Overriding

The child class Employee inherits all the methods from parent class Person. But if you would like to override those methods in your child class then you can do it by givig your implementation. You can add your additional functions in child class. It can done as follows: modify Employee.pm file

```
#!/usr/bin/perl

package Employee;
use Person;
use strict;
our @ISA = qw(Person);    # inherits from Person

# Override constructor
sub new {
    my ($class) = @_;

    # Call the constructor of the parent class, Person.
    my $self = $class->SUPER::new( $_[1], $_[2], $_[3] );
    # Add few more attributes
    $self->{_id}   = undef;
    $self->{_title} = undef;
    bless $self, $class;
    return $self;
}

# Override helper function
sub getFirstName {
    my( $self ) = @_;
    # This is child class function.
    print "This is child class helper function\n";
    return $self->{_firstName};
}

# Add more methods
sub setLastName{
    my ( $self, $lastName ) = @_;
    $self->{_lastName} = $lastName if defined($lastName);
    return $self->{_lastName};
}

sub getLastName {
    my( $self ) = @_;
    return $self->{_lastName};
}

1;
```

Now put following code into main.pl and execute it.

```
#!/usr/bin/perl

use Employee;

$object = new Employee( "Mohammad", "Saleem", 23234345);
# Get first name which is set using constructor.
$firstName = $object->getFirstName();

print "Before Setting First Name is : $firstName\n";
```

```
# Now Set first name using helper function.
$object->setFirstName( "Mohd." );

# Now get first name set by helper function.
$firstName = $object->getFirstName();
print "After Setting First Name is : $firstName\n";


This will produce following result
First Name is Mohammad
Last Name is Saleem
SSN is 23234345
This is child class helper function
Before Setting First Name is : Mohammad
This is child class helper function
After Setting First Name is : Mohd.
```

Default Autoloading

Perl offers a feature which you would not find any many other programming languages: a default subroutine.

If you define a function called AUTOLOAD() then any calls to undefined subroutines will call AUTOLOAD() function. The name of the missing subroutine is accessible within this subroutine as $AUTOLOAD. This function is very useful for error handling purpose. Here is an example to implement AUTOLOAD, you can implement this function in your way.

```
sub AUTOLOAD
{
   my $self = shift;
   my $type = ref ($self) || croak "$self is not an object";
   my $field = $AUTOLOAD;
   $field =~ s/.*://;
   unless (exists $self->{$field})
   {
      croak "$field does not exist in object/class $type";
   }
   if (@_)
   {
      return $self->($name) = shift;
   }
   else
   {
      return $self->($name);
   }
}
```

Destructors and Garbage Collection

If you have programmed using objects before, then you will be aware of the need to create a .destructor. to free the memory allocated to the object when you have finished using it. Perl does this automatically for you as soon as the object goes out of scope.

In case you want to implement your destructore which should take care of closing files or doing some extra processing then you need to define a special method called **DESTROY**. This method

will be called on the object just before Perl frees the memory allocated to it. In all other respects, the DESTROY method is just like any other, and you can do anything you like with the object in order to close it properly.

A destructor method is simply a member function (subroutine) named DESTROY which will be automatically called

- When the object reference's variable goes out of scope.
- When the object reference's variable is undef-ed
- When the script terminates
- When the perl interpreter terminates

For Example:

```
package MyClass;
...
sub DESTROY
{
    print "   MyClass::DESTROY called\n";
}
```

Another OOP Example

Here is another nice example which will help you to understand Object Oriented Concepts of Perl. Put this source code into any file and execute it.

```
#!/usr/bin/perl

# Following is the implementation of simple Class.
package MyClass;

sub new
{
   print "   MyClass::new called\n";
   my $type = shift;            # The package/type name
   my $self = {};               # Reference to empty hash
   return bless $self, $type;
}

sub DESTROY
{
   print "   MyClass::DESTROY called\n";
}

sub MyMethod
{
   print "   MyClass::MyMethod called!\n";
}


# Following is the implemnetation of Inheritance.
package MySubClass;

@ISA = qw( MyClass );

sub new
{
   print "   MySubClass::new called\n";
   my $type = shift;            # The package/type name
```

```
   my $self = MyClass->new;     # Reference to empty hash
   return bless $self, $type;
}

sub DESTROY
{
   print "   MySubClass::DESTROY called\n";
}

sub MyMethod
{
   my $self = shift;
   $self->SUPER::MyMethod();
   print "   MySubClass::MyMethod called!\n";
}

# Here is the main program using above classes.
package main;

print "Invoke MyClass method\n";

$myObject = MyClass->new();
$myObject->MyMethod();

print "Invoke MySubClass method\n";

$myObject2 = MySubClass->new();
$myObject2->MyMethod();

print "Create a scoped object\n";
{
        my $myObject2 = MyClass->new();
}
# Destructor is called automatically here

print "Create and undef an object\n";
$myObject3 = MyClass->new();
undef $myObject3;

print "Fall off the end of the script...\n";
# Remaining destructors are called automatically here
```

You can put Perl stuff anywhere you want and add any directory to @INC, but you can't add "." while using the taint mode.

You can add the directory to PERL5LIB or

you can add use lib 'directory'; to your perl script or

you can push your directory onto @INC in a BEGIN block like

BEGIN { push @INC,"/home/you/yourperlstuff/";}

We will look at how the contents of this array are constructed and can be manipulated to affect where Perl interpreter will find the module files.

Default @INC

Perl interpreter is compiled with a specific default value of @INC that it was compiled with. To find out this value, run env -i perl -V command (`env -i ignores PERL5LIB environmental variable - see #2) and in the output you will see something like this:

```
$ env -i perl -V
...
@INC:
 /usr/local/lib/perl5/5.8.6/i686-linux
 /usr/local/lib/perl5/5.8.6
 /usr/local/lib/perl5/site_perl/5.8.6/i686-linux
 /usr/local/lib/perl5/site_perl
```

To change the default path when configuring Perl binary compilation, use otherlibdirs variable:

```
Configure -Dotherlibdirs=/usr/lib/perl5/site_perl/5.8.1
```

Environmental variable PERL5LIB (or PERLLIB)

Perl pre-pends @INC with a list of directories (colon-separated) contained in PERL5LIB (if it is not defined, PERLLIB is used) environmental variable of your shell. To see the contents of @INC after PERL5LIB and PERLLIB environment variables have taken effect, run perl -V.

```
$ perl -V
...
%ENV:
  PERL5LIB="/home/myuser/test"
@INC:
 /home/myuser/test
 /usr/local/lib/perl5/5.8.6/i686-linux
 /usr/local/lib/perl5/5.8.6
 /usr/local/lib/perl5/site_perl/5.8.6/i686-linux
 /usr/local/lib/perl5/site_perl
```

-I command line parameter

Perl pre-pends @INC with a list of directories (colon-separated) passed in to it as a value of -I command line parameter. This can be done in one of two ways, as usual with Perl parameters:

Pass it on command line: perl -I /my/moduledir your_script.pl

Pass it via the first (shebang) line of your Perl script:

```
#!/usr/local/bin/perl -w -I /my/moduledir
```

Pass it as part of PERL5OPT (or PERLOPT) environmantal variable (see chapter 19.02 in Programming Perl)

Pass it via the use lib pragma

Perl pre-pends @INC with a list of directories passed in to it via use lib pragma.

In a program:

```
use lib ("/dir1", "/dir2")
```

On the command line:

perl -Mlib=/dir1,/dir2

You can also remove the directories from @INC via no lib

You can directly manipulate @INC as a regular Perl array.

NOTE: Since @INC us used during compilation phase, this must be done inside of a BEGIN {} block, which precedes the use MyModule statement;

Add directories to the beginning via unshift @INC, $dir

Add directories to the end via push @INC, $dir

Do anything else you can do with a Perl array.

NOTE The directories are unshifted onto @INC in the order listed in this answer, e.g. default @INC is last in the list, preceded by PERL5LIB, preceded by -I, preceded by "use lib" and direct @INC manipulation, the latter two mixed in whichever order they are in Perl code.

```
There are many ways to execute external commands from Perl. The most commons
are:

    system function
    exec function
    backticks (``) operator
    open function

All of these methods have different behaviour, so you should choose which
one to use depending of your particular need. In brief, these are the
recommendations:

method     use if ...
system()   you want to execute a command and don't want to capture its
output
exec  you don't want to return to the calling perl script
backticks  you want to capture the output of the command
open  you want to pipe the command (as input or output) to your script



How to uninstall a CPAN module ?
CPAN module does not have the option of un-installing a perl module as of
now. This can be especially frustrating for newbies (as it was for me). I
struggled quite a bit on google and found this solution. For this, you need
the CPANPLUS module, which needs to be first installed through CPAN.
```

```
C:\Users\Pooja Verma>perl -MCPAN -e shell

cpan> install CPANPLUS

cpan> exit
```

Unlike CPAN, you cannot invoke CPANPLUS by typing cpanplus on command prompt, even if its installed.

```
C:\Users\Pooja Verma>cpanplus
'cpanplus' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Pooja Verma>perl -MCPANPLUS -e shell
CPANPLUS::Shell::Default -- CPAN exploration and module installation (v0.84)
*** Please report bugs to <bug-cpanplus@rt.cpan.org>.
*** Using CPANPLUS::Backend v0.84.  ReadLine support disabled.

*** Type 'p' now to show start up log

Did you know...
    The documentation in CPANPLUS::Module and CPANPLUS::Backend is very
useful
CPAN Terminal> help

[General]
    h | ?                   # display help
    q                       # exit
    v                       # version information
[Search]
    a AUTHOR ...            # search by author(s)
    m MODULE ...            # search by module(s)
    f MODULE ...            # list all releases of a module
    o [ MODULE ... ]        # list installed module(s) that aren't up to date
    w                       # display the result of your last search again
[Operations]
    i MODULE | NUMBER ...   # install module(s), by name or by search number
    i URI | ...             # install module(s), by URI (ie
http://foo.com/X.tgz)
    t MODULE | NUMBER ...   # test module(s), by name or by search number
    u MODULE | NUMBER ...   # uninstall module(s), by name or by search
number
    d MODULE | NUMBER ...   # download module(s)
    l MODULE | NUMBER ...   # display detailed information about module(s)
    r MODULE | NUMBER ...   # display README files of module(s)
    c MODULE | NUMBER ...   # check for module report(s) from cpan-testers
    z MODULE | NUMBER ...   # extract module(s) and open command prompt in it
[Local Administration]
    b                       # write a bundle file for your configuration
    s program [OPT VALUE]   # set program locations for this session
    s conf    [OPT VALUE]   # set config options for this session
    s mirrors               # show currently selected mirrors
```

Now you can uninstall the module that you want to. If needed, you can use the --verbose and --force option with it.

```
CPAN Terminal> u MIME::Head --force --verbose
```

Something else is weird here...

```
~> cat test.pl
$a = "234*343";
$i = "FOO";

$a =~ s/\*/$i/;
print $a;

~> perl test.pl
234FOO343
```

Found something:

```
~> cat test.pl
$a = "234*343";
$i = "*4";

$a =~ m/$i/;
print $a;

~> perl test.pl
Quantifier follows nothing in regex; marked by <-- HERE in m/* <-- HERE 4/
at test.pl line 4.
```

Solution, escape the special characters from the variable using \Q and \E, for example (TIMTOWTDI)

```
~> cat test.pl
$a = "234*343";
$i = "*4";

$a =~ m/\Q$i\E/;
print $a;

~> perl test.pl
234*343
```

Que: How to unbless ?

Hash

```
$obj = bless {}, 'Obj';
print ref $obj, "\n";
$obj = { %$obj };
print ref $obj, "\n";
```

Array

```
$obj = bless [], 'Obj';
print ref $obj , "\n";
$obj = [ @$obj ];
print ref $obj, "\n";
```

Scalar

```
$obj = bless \$a, "Obj";
print ref $obj, "\n";
$obj = \${ $$obj };
```

```
print ref $obj, "\n";
```

Que. How to print a symbol table ? i.e How to print and verify all the subroutines and variables exported from a package/library?
Ans:
Use YAML;
Print %YAML::

To print it in a better way:
Print join(“\n”, %YAML::);

Symbol Tables

Each namespace -- and therefore, each module, class, or package -- has its own symbol table. A *symbol table*, in Perl, is a hash that holds all of the names defined in a namespace. All of the variable and function names can be found there. ***The hash for each namespace is named after the namespace with two colons.*** For example, the symbol table for the `Foo` namespace is called `%Foo::`.

The next example shows a program that displays all of the entries in the `Foo::` namespace. It basically operates as follows:

- Define the *`dispSymbols()`* function.
- Get the hash reference that should be the first parameter.
- Declare local temporary variables.
- Initialize the *`%symbols`* variable. This is done to make the code easier to read. Initialize the *`@symbols`* variables. This variable is also used to make the code easier to read.
- Iterate over the symbols array displaying the key-value pairs of the symbol table.
- Call the *`dispSymbols()`* function to display the symbols for the Foo package.
- Start the Foo package.
- Initialize the *`$bar`* variable. This will place an entry into the symbol table. Define the *`baz()`* function.
- This will also create an entry into the symbol table.

The Perl code to do this is as follows, `symbol.pl`:

```
sub dispSymbols {

    my($hashRef) = shift;
```

```
    my(%symbols);

    my(@symbols);


    %symbols = %{$hashRef};

    @symbols = sort(keys(%symbols));


    foreach (@symbols) {

        printf("%-10.10s| %s\n", $_, $symbols{$_});

    }

}


dispSymbols(\%Foo::);


package Foo;

    $bar = 2;


    sub baz {

        $bar++;

    }
```

This program displays:

```
bar       | *Foo::bar
baz       | *Foo::baz
```

Que: What is the difference between return, die, exit 0 and exit 1 ?
Ans:
You normally use exit(0) if everything went ok.
and for abnormal program termination or incomplete ones use 1 (recommended)

these exit status are indication to the environment from which the program has run about the status of exit of the program. Or you might use exit(1), exit(2), etc, with each exit code meaning some specific error.

1) die is used to throw an exception (catchable using eval).
exit is used to exit the process.

2) die will set the error code based on $! or $? if the exception is uncaught.
exit will set the error code based on its argument.

3) die outputs a message
exit does not.

The difference between return and exit() is that return only ends the current function, while exit() ends the whole program. ***In main(), return and exit() are identical.***

- The syntax for the Exit command is:

  "exit EXPR."

  The expression is evaluated before existing the interpreter (note the difference with Perl die and "lists" compared to exit "expressions"). An example of an exit command follows:

  $ans = $a ;

  exit 1 if $ans ~= /test/;

  This assigns a value to the "$ans" variable and exits the program with a status of 1 if the value contains the word "test."

In C actually you should use

#include <stdlib.h>

exit (EXIT_SUCCESS);
exit(EXIT_FAILURE);

Flushing Output

Que: How to disable buffering?
Ans: $| =1;

e.g.:

```
$old_fh = select(OUTPUT_HANDLE);
$| = 1;
select($old_fh);
```

Or, if you don't mind the expense of loading an IO module, disable buffering by invoking the `autoflush` method:

```
use IO::Handle;
OUTPUT_HANDLE->autoflush(1);
```

This works with indirect filehandles as well:

```
use IO::Handle;
$fh->autoflush(1);
```

Problem

When printing to a filehandle, output doesn't appear immediately. This is a problem in CGI scripts running on some programmer-hostile web servers where, if the web server sees warnings from Perl before it sees the (buffered) output of your script, it sends the browser an uninformative `500 Server Error` . These ***buffering problems arise with concurrent access to files by multiple programs and when talking with devices or sockets.***

e.g. If you are playing upon a growing array with a while loop and writing the element to a file(say logfile) then the logs doesn't appear in the logfile.

```
open(OUTFILE, ">>$outfile") or die "File error!";
while (defined($u = shift @urls_to_operate)) {
    print OUTFILE $u;
}
```

So for efficiency, Perl does not read or write the disk or the network when you ask it to. Instead, it reads and writes large chunks of data to a `buffer' in memory, and does I/O to the buffer; this is much faster than making a request to the operating system for every read or write. Usually this is what you want, but sometimes the buffering causes problems. Typical problems include communicating with conversational network services and writing up-to-date log files. *In such circumstances,* ***you would like to disable the buffering. You can do that in Perl by setting $|=1.*** This special variable makes the currently selected filehandle hot, so that the buffer is flushed after every write.

What is Buffering?

I/O is performed by your operating system. When Perl wants to read data from the disk, or to write it to the network, or to read or write data anywhere, Perl has to make a request to the operating system and ask that the data be read or written. This is an example of `making a system call'. (Don't confuse this with Perl's `system` function, which is totally unrelated.)

Making a system call is a relatively slow operation. On top of that, if the data is coming from the disk, you might have to wait for the disk to spin to the right position (that's called the `latency time') and you might have to wait for the disk heads to move over to the right track (that's called the `seek time'), and as computer operations go, that wait is unbearably long---several milliseconds is typical. By contrast, a typical computer operation, such as assigning to a variable or adding two numbers, takes a fraction of a microsecond.

Suppose you're reading a ten-thousand line file line by line:

```
while (<FILE>) {
  print if /treasure/;
}
```

If Perl made a system call for every read operation, that would be 10,001 system calls in all (one extra to detect end-of-file), and if the file was on the disk, it would have to wait for the disk at least 10,000 times. That would be very slow.

For efficiency, Perl uses a trick called *buffering*. The first time you ask to read from the file, Perl has to make a system call. Since system calls are expensive, it plans ahead. If you tried to read a little bit of text, you will probably want to read the rest later. The blocks on your disk are probably about 8K bytes, and your computer hardware is probably designed to transfer an entire block of data from the disk at once. So instead of asking the system for a little bit of text, Perl actually asks the system for an entire blockful, which hardly takes longer to get than a little bit would have. Then it stores this block of data in a ***region of memory*** that is called a *buffer*, and gives you back the one line you asked for. The next time you ask for a line, Perl already has the line you want in memory in

the 8K buffer. It doesn't have to make another system call; it just gives you the next line out of the buffer. Eventually you read up to the end of the buffer, and then Perl makes another system call to get another bufferful of data.

If lines typically have about 60 characters each, then the 10,000-line file has about 610,000 characters in it. Reading the file line-by-line with buffering only requires 75 system calls and 75 waits for the disk, instead of 10,001. On my system, a simple program with buffered reading ran about 40% faster than the corresponding program that made a system call for every line.

For writing, Perl uses the same trick. When you write data to a file with `print`, the data doesn't normally go into the file right away. Instead, it goes into a buffer. When the buffer is full, Perl writes all the data in the buffer at once. This is called *flushing the buffer*. Here the performance gain is even bigger than for reading, about 60%. But the buffering can sometimes surprise you.

Solution

Disable buffering by setting the per-filehandle variable `$|` to a true value, customarily `1` :

```
$old_fh = select(OUTPUT_HANDLE);
$| = 1;
select($old_fh);

OR

select(STDOUT); $| = 1;               # make unbuffered

OR
```

Or, if you don't mind the expense, disable it by calling the `autoflush` method from the IO modules:

```
use IO::Handle;
OUTPUT_HANDLE->autoflush(1);

e.g.
open(OUTFILE, ">>$outfile") or die "File error!";

# make unbuffered
my $fh = select(OUTFILE);
$| = 1;
select($fh);
#

while (defined($u = shift @urls_to_operate)) {
    print OUTFILE $u;
}



e.g.
        #!/usr/bin/perl
        open(SAVEOUT, ">&STDOUT");
        open(SAVEERR, ">&STDERR");

        open(STDOUT, ">foo.out") || die "Can't redirect stdout";
        open(STDERR, ">&STDOUT") || die "Can't dup stdout";

        select(STDERR); $| = 1;               # make unbuffered
        select(STDOUT); $| = 1;               # make unbuffered

        print STDOUT "stdout 1\n";     # this works for
        print STDERR "stderr 1\n";     # subprocesses too
```

```
close(STDOUT);
close(STDERR);

open(STDOUT, ">&SAVEOUT");
open(STDERR, ">&SAVEERR");

print STDOUT "stdout 2\n";
print STDERR "stderr 2\n";
```

eg3

- #!/usr/bin/perl
- use strict;
- use warnings;
- use diagnostics; # good for detailed explanations about any problems in code
- $| =1;
- my $val = 4;
- for (0..$val) {
- print "$_ ";
- sleep(1);
- }

FROM http://www.cs.cmu.edu/afs/cs/usr/rgs/mosaic/pl-exp-io.html

If you open a pipe on the command "-", i.e. either "|-" or "-|", then there is an implicit fork done, and the return value of open is the pid of the child within the parent process, and 0 within the child process. (Use defined($pid) to determine if the open was successful.) The filehandle behaves normally for the parent, but i/o to that filehandle is piped from/to the *STDOUT*/ STDIN of the child process. In the child process the filehandle isn't opened--i/o happens from/to the new *STDOUT* or *STDIN*. Typically this is used like the normal piped open when you want to exercise more control over just how the pipe command gets executed, such as when you are running setuid, and don't want to have to scan shell commands for metacharacters. The following pairs are more or less equivalent:

```
open(FOO, "|tr '[a-z]' '[A-Z]'");
open(FOO, "|-") || exec 'tr', '[a-z]', '[A-Z]';

open(FOO, "cat -n '$file'|");
open(FOO, "-|") || exec 'cat', '-n', $file;
```

Explicitly closing any piped filehandle causes the parent process to wait for the child to finish, and returns the status value in $?. Note: on any operation which may do a fork, unflushed buffers remain unflushed in both processes, which means you may need to set $| to avoid duplicate output.

The filename that is passed to open will have leading and trailing whitespace deleted. In order to open a file with arbitrary weird characters in it, it's necessary to protect any leading and trailing whitespace thusly:

```
$file =~ s#^(\s)#./$1#;
open(FOO, "< $file\0");
```

fork():

```
The child process executes the END {} block as it is exiting, but usually I
don't want that to happen. Is there a way to prevent a child process from
calling the END block at exit? Barring that, is there a way for a program to
"know" that it is a child process, so I could say something like
END { unless (i_am_a_child_process()) { &some_cleanup_code } }
```

?

```
The exit() function does not always exit immediately. It calls any defined END
routines first, but these END routines may not themselves abort the exit.
Likewise any object destructors that need to be called are called before the
real exit. If this is a problem, you can call POSIX: _exit($status) to avoid END
and destructor processing. See perlmod for details.

e.g.
use POSIX "_exit"; POSIX::_exit(0);
```

"This Perl not built to support threads"

$ /usr/bin/perl -e "require threads;"
This Perl not built to support threads
Compilation failed in require at -e line 1.

"my" variable $rc masks earlier declaration in same scope at Sanity.pm line 406.

This warning can be suppressed by including

```
no warnings qw/misc/;
```

Passing Filehandles

To pass filehandles to subroutines, use the `*FH` or `\*FH` notations. These are ``typeglobs" - see Typeglobs and Filehandles and especially Pass by Reference for more information.

Here's an excerpt:

If you're passing around filehandles, you could usually just use the bare typeglob, like *STDOUT, but typeglobs references would be better because they'll still work properly under `use strict 'refs'`. For example:

```
splutter(\*STDOUT);
sub splutter {
    my $fh = shift;
    print $fh "her um well a hmmm\n";
}
$rec = get_rec(\*STDIN);
sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

If you're planning on generating new filehandles, you could do this:

```
sub openit {
    my $name = shift;
    local *FH;
    return open (FH, $path) ? *FH : undef;
}
```

```
$fh = openit('< /etc/motd');
print <$fh>;
```

Passing Regexps

To pass regexps around, you'll need to either use one of the highly experimental regular expression modules from CPAN (Nick Ing-Simmons's Regexp or Ilya Zakharevich's Devel::Regexp), pass around strings and use an exception-trapping eval, or else be be very, very clever. Here's an example of how to pass in a string to be regexp compared:

```
sub compare($$) {
    my ($val1, $regexp) = @_;
    my $retval = eval { $val =~ /$regexp/ };
    die if $@;
    return $retval;
}
$match = compare("old McDonald", q/d.*D/);
```

Make sure you never say something like this:

```
return eval "\$val =~ /$regexp/";   # WRONG
```

or someone can sneak shell escapes into the regexp due to the double interpolation of the eval and the double-quoted string. For example:

```
$pattern_of_evil = 'danger ${ system("rm -rf * &") } danger';
eval "\$string =~ /$pattern_of_evil/";
```

Those preferring to be very, very clever might see the O'Reilly book, *Mastering Regular Expressions*, by Jeffrey Friedl. Page 273's `Build_MatchMany_Function()` is particularly interesting. A complete citation of this book is given in the perlfaq2 manpage.

Passing Methods

To pass an object method into a subroutine, you can do this:

```
call_a_lot(10, $some_obj, "methname")
sub call_a_lot {
    my ($count, $widget, $trick) = @_;
    for (my $i = 0; $i < $count; $i++) {
        $widget->$trick();
    }
}
```

Or you can use a closure to bundle up the object and its method call and arguments:

```
my $whatnot =  sub { $some_obj->obfuscate(@args) };
func($whatnot);
sub func {
    my $code = shift;
    &$code();
}
```

You could also investigate the `can()` method in the UNIVERSAL class (part of the standard perl distribution).

Namespace:

Consider a situation, when we have two persons with the same name, Zara, in the same class. Whenever we need to differentiate them definitely we would have to use some additional information along with their name, like either the area if they live in different area or their mother or father name, etc.

Same situation can arise in programming languages.

More about perl buffering:

http://perl.plover.com/FAQs/Buffering.html

Surprise!

Suppose you have a program like this:

```
foreach $item (@items) {
  think_for_a_long_time($item);
  print LOG "Finished thinking about $item.\n";
}
```

Suppose there are 1,000 items, and the program will have to think about each one for two minutes. The program will take about 35 hours to complete, and you'd like to be able to peek at the log file to see how it is doing. You start up the program, wait ten minutes, and peek at the log file---but there's nothing there. Disaster! What happened? Is the program stuck? Is it taking five times as long to think about the items as you thought it would?

No, the program is not stuck, or even slow. The problem is that the `print`s to the log file are being buffered. The program has thought about the first five items, and it wrote the log messages for them, but the writes went into the buffer, and Perl isn't going to make a system call to send the buffer to the disk until the buffer is full. The buffer is probably 8K bytes, and the log messages are about 28 bytes each, so what's going to happen is that you won't see anything in the log file until about ten hours from now, when the first 292 messages will appear all at once. After that, you won't get any more messages for another ten hours.

That's a problem, because it's not what you wanted. Here you don't really care about the efficiency gain of buffered writes. On the plus side of buffering, you're saving about four seconds over the 35-hour lifetime of the program. On the minus side, it's making your logging feature useless. You think having the log is worth waiting an extra four seconds, so you'd like to turn off the buffering.

Disabling Inappropriate Buffering

In Perl, you can't turn the buffering off, but you can get the same benefits by making the filehandle *hot*. Whenever you print to a hot filehandle, Perl flushes the buffer immediately. In our log file example, it will flush the buffer every time you write another line to the log file, so the log file will always be up-to-date.

Here's how to make the filehandle hot:

```
{ my $ofh = select LOG;
  $| = 1;
  select $ofh;
}
```

The key item here is the `$|` variable. If you set it to a true value, it makes the current filehandle hot. What's the current filehandle? It's the one last selected with the `select` operator. So to make `LOG` hot, we `select' it, set `$|` to a true value, and then we re-`select' whatever filehandle was selected before we selected `LOG`.

Now that `LOG` is hot, Perl will flush the buffer every time we print to `LOG`, so messages will appear in the log file as soon as we print them.

Sometimes you might see code like this:

```
select((select(LOG), $|=1)[0]);
```

That's a compressed way of writing the code above that makes `LOG` hot.

If you happen to be using the `FileHandle` or `IO` modules, there's a nicer way to write this:

```
use FileHandle;          # Or `IO::Handle' or `IO::'-anything-else
```

```
...

LOG->autoflush(1);       # Make LOG hot.

...
```

Hot and Not Hot

If Perl is really buffering output, how is it you didn't notice it before? For example, run this program, `favorite`:

```
print "What is your favorite number?   ";
$num = <STDIN>;
$mine = $num + 1;
print "Well, my favorite is $mine, which is a much better number.\n";
% ./favorite
What is your favorite number?     119
Well, my favorite is 120, which is a much better number.
```

If you run this, you find that it works the way you expect: The prompt appears on the screen right away. Where's the buffering? Why didn't Perl save up the output until it had a full buffer? Because that's almost never what you want when you're writing to a terminal, the standard I/O library that Perl uses takes care of it for you. When a filehandle is attached to the terminal, as `STDOUT` is here, it is in *line buffered mode* by default. A filehandle in line buffered mode has two special properties: It's flushed automatically whenever you print a newline character to it, and it's flushed automatically whenever you read from the terminal. The second property is at work here: `STDOUT` is flushed automatically when we read from `STDIN`.

But now let's try it with `STDOUT` attached to a file instead of to the terminal:

```
% ./favorite > OUTPUT
```

Here the `STDOUT` filehandle has been attached to the file `OUTPUT`. The program has printed the prompt to the file, and is waiting for you to enter your favorite number. But if you open another window, and look into `OUTPUT`, you'll see that the prompt that `favorite` printed isn't in the file yet; it's still in the buffer. `STDOUT` is attached to a file, rather than to a terminal, so it isn't line-buffered; only filehandles attached to the terminal are line-buffered by default.

When the program finishes, it flushes all its buffers, so after you enter your favorite number, all the output, including the prompt, appears in the file at the same time.

There's one other exception to the rule that says that filehandles are cold unless they're attached to the terminal: The filehandle `STDERR`, which is normally used for error logging, is always in line buffered mode by default. If our original example had used `STDERR` instead of LOG we wouldn't have had the buffering problem.

Other Perils of Buffering

``My output is coming out in the wrong order!''

Here's a typical program that exhibits this common problem:

```
print "FILE LISTING OF DIRECTORY $dir:\n";
print "---------------------------------\n";
system("ls -l $dir");
print "---------------------------------\n";
```

Run this on a terminal, and it comes out OK. But if you run it with `STDOUT` redirected to a file, it doesn't work: the header appears after the file listing, instead of at the top.

So why didn't it work? Standard output is buffered, so the header lines are saved in the buffer and don't get to the file just yet. Then you run `ls`, which has its own buffer, and `ls`'s buffer is flushed when `ls` exits, so `ls`'s output goes into the file. Then you print the footer line and it goes into your programs buffer. Finally, your program finishes and flushes its buffer, and all three lines go into the output file, after the output from `ls`. To fix this, make `STDOUT` hot.

Now that we know why the data got into the file in the wrong order, that raises another question: If we have it print to the terminal, why does the output come out in the right order instead of the wrong order? Because when `STDOUT` is attached to the terminal, it is in line buffered mode, and is flushed automatically whenever we print a newline character to it. The two header lines are flushed immediately, because they end with newlines; then comes the output of `ls`, and finally the footer line.

``My web server says I didn't send the right headers, but I'm sure I did!"

Here's a typical program that exhibits this common problem:

```
print "Content-type: text/html\n\n";

print "<title>What Time Is It?</title>\n";
print "<h1>The Current Time in Philadelphia is</h1>\n\n";
print "<pre>\n";

system("date");

print "</pre>\n\n";
```

You might think that the output is going to come out in the order you put it in the program, with the `Content-type` header, then the title, and with the date in between the `<pre>` tags. But it isn't. The `print` statements execute, but the output goes into your program's buffer. Then you run `date`, which generates some output, this time into the `date` command's buffer. When `date` exits (almost immediately), this buffer is flushed, and the server (whch is listening to your standard output) gets the date before it gets the output from your `print`s; your `print` data is still in the buffer. Later, when your program exits, its own buffer is flushed and the server gets the output from the `print`s.

The server was expecting to see that `Content-type` line first, but instead it got the date first. It can't proceed without knowing the content-type of the output, so it gives up and sends a message to the browser that says something like `500 Internal Server Error`.

Solution 1: Make `STDOUT` hot. Solution 2: Collect the output from `date` yourself and insert it into your buffer in the appropriate place:

```
...

$the_date = `date`;
print $the_date;

...
```

Here's a similar sort of problem that stems from the program aborting before you wanted it to:

```
print "Content-type: text/html\n\n";

print "<title>Division Table</title>\n";
print "<h1>Division Table</h1>\n\n";

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

```
      print $i/$j, "\t";                # Ooops
    }
    print "\n";
}
```

The program is going to abort at the line that says `Ooops', because it divides by zero, and it's going to print the message

```
Illegal division by zero at division.cgi line 8.
```

What you actually see on your web browser depends a lot on the details of the web server, but here's one possible scenario: The server will collect all the output from your program, and send it back to the browser. You might think that the `Content-type` line will come first, followed by the title, and then the `division by zero` message, but you'd be wrong. The content-type and the title are printed to `STDOUT`, which is buffered, but the `divsion by zero` message is printed to `STDERR`, which isn't buffered. Result: The content-type and title are buffered. Then the error message comes out, and then, when the program exits, the `STDOUT` buffer is flushed and the content-type and title come out at last. The server was expecting to see the `Content-type` line right away, gets confused because it appears to be missing, and reports an error.

As usual, you can fix this by making `STDOUT` hot. Another way: Redirect error messages to a separate file, like this:

```
open STDERR, "> /tmp/division.err";
```

A third way:

```
use CGI::Carp 'fatalsToBrowser';
```

The `CGI::Carp` module will arrange that fatal error messages are delivered to the browser with a simple prefabricated HTTP header, so that the browser displays the error message properly.

Making both handles cold won't work, because when the program finishes, there's no telling which of the two will be flushed first.

``I'm trying to send data over the network, but nothing is sent!"

This one is the plague of novice network applications programmers; it bites almost everyone the first time they write a network application.

For concreteness, suppose you're writing a mail client, which will open a connection to the mail server and try to send a mail message to it. Your client will need to use the SMTP (Simple Mail Transfer Protocol) to talk to the server. In SMTP, the client opens a connection and then engages the server in a conversation that goes something like this:

```
Server says: 220 gradin.cis.upenn.edu ESMTP
Client says: HELO plover.com
Server: 250 gradin.cis.upenn.edu Hello mailuser@plover.com
Client: MAIL From: <mjd@plover.com>
Server: 250 <mjd@plover.com>... Sender ok
(And so on...)
```

The usual complaint: ``I opened the connection all right, and I got the greeting from the server, but it isn't responding to my client's commands!" And by now you know the reason why: The client's output to the network socket is being buffered, and Perl is waiting to send the data over the network until there's a whole bufferful. The server hasn't responded because it hasn't received anything to respond to.

Solution: Make the socket filehandle hot. Another solution: Use the `IO::Socket` module; recent versions (since version 1.18) make sockets hot by default.

``When my program terminates abnormally, the output is incomplete!"

When the program exits normally, by executing `die` or `exit` or by reaching the end of the program, it flushes all the buffers. But if the program is killed suddenly, it might exit without getting to flush the buffers, and then the output files will be incomplete. The Unix `kill` command destroys a process in this way and can leave behind incomplete files.

Even worse, a file that exits in this way can leave behind *corrupt* data. For example, imagine a program that writes out a database file. The database file is supposed to contain records of exactly 57 characters each.

Suppose the program has printed out 1,000 records, and then someone kills it and it doesn't have a chance to flush its buffer. It turns out that only 862 complete records have made it into the file, but that's not the worst part. The buffer is flushed every 8,192 bytes, and 57 does not divide 8,192 evenly, so the last record that was flushed to the file is incomplete; only its first 18 bytes appear in the file. The other 39 bytes were still in the buffer, and they're lost. The file is now corrupted, and any program that reads it assuming that each record is exactly 57 bytes long is going to get garbled data and produce the wrong results.

One possible solution to this is to simply make the filehandle hot. Another is to do the buffering yourself: Accumulate 57-byte records into a scalar variable until you have a lot of them, and then write them all at once. A third solution is to use the `setvbuf` method provided by the `Filehandle` and `IO::` modules to make the buffer size an exact multiple of 57 bytes; then it'll never contain any partial records when it's flushed. That looks like this:

```
use IO::Handle '_IOFBF'; # `FBF' means `Fully Buffered'

FH->setvbuf($buffer_var, _IOFBF, 8151);
```

(I picked 8151 here because it's the largest number less than 8K that is a multiple of 57.)

A fourth solution is to manually flush the buffer before it gets completely full. The next section explains how to do this.

Flushing on Command

Sometimes you'd like to have buffering, but you want to control when the buffer is flushed. For example, suppose you're writing a lot of data over the network to a logging service. For efficiency, you'd like buffering, but you don't want the log to get too far out of date. You want to let data accumulate in the buffer for up to ten seconds, and then flush it out, at least six times per minute.

Here's a typical strategy for doing that:

```
if (time > $last_flush_time + 10) {
  my $ofh = select LOG;
  $| = 1;                     # Make LOG socket hot
  print LOG "";               # print nothing
  $| = 0;                     # LOG socket is no longer hot
  select $ofh;
  $last_flush_time = time;
}

... Do something else ...
```

We select the `LOG` filehandle and make it temporarily hot. Then we print the empty string to the filehandle. Because the handle is hot, this flushes the buffer---printing to a hot filehandle always flushes the buffer. Then we return the filehandle to its un-hot state so that future writes to `LOG` will be buffered.

If you're using the `FileHandle` or `IO` modules, there's a simpler interface:

```
        $filehandle->flush();          # Flush the buffer
```

It does just the same as the code above.

Other Directions

If for some reason you want to avoid the buffering entirely, you can use Perl's `sysread` and `syswrite` operators. These don't use buffering at all. That makes them slow, but they are often appropriate for tasks like network communications where you don't want buffering anyway. All Perl's other I/O functions, including `write`, `print`, `read`, `<FILEHANDLE>`, and `getc`, are buffered. If you do both buffered and unbuffered I/O on the same filehandle, you're likely to confuse yourself, so beware.