Module 5: Processes and Perl Programming

5.PROCESS:

A **Process** is simply an instance of running program.

- A process is said to be **born** when the program execution starts and remains alive as long as the program is active
- After execution is complete, the process is said to **die.**
- Some attributes of every process are maintained by the kernel in memory in a separate structure called the **process table.**

Two important attributes of a process are

- **Process id(PID):** Each process is uniquely identified by a unique integer called the PID that is allotted by the kernel when the process is born. This PID is used to control a process .
- **Parent PID(PPID):**The PID of the parent is also available a s process attribute.

5.1 PROCESS STATUS:

ps command to display the some process attributes.

By default the ps command displays the processes owned by the user running the command.

```
$ps
PID   TTY       TIME CMD
291   console   0:00  bash\
```

ps options:

5.1.1 Full listing(-f)

To get a detailed listing which also shows the parent of every process, use -f option

```
$ps –f
UID     PID    PPID     C     STIME       TTY     TI ME              CMD
sumit    367     291    0     12:35:16     console   0:00    -bash
sumit    291      1     0     10:24:35     console     0:00      /usr/bin/bash
```

5.1.2 Displaying Processes of a user(-u)

To know the activities of any user.

```
$ps -u sumit
PID    TTY    TIME  CMD
378    ?      0:05  Xsun
403    ?      0:00  bash
347    ?      0:01   vi
460    pts/5  0:00  dtterm
```

5.1.3 Display all user processes(-a)

The -a option lists processes of all users but doesnot display the system processes.

```
$ps -a
PID    TTY    TIME  CMD
662    pts/01 00:00:00     ksh
705    pts/04 00:00:04     sh
680    pts/05 00:00:00     sort
1056   pts/08 00:00:00     ps
```

5.1.4 Displays system Processes(-e or A)

```
$ps -e
PID    TTY    TIME  CMD
```

```
1       ?      0:01    init
3       ?      0:00    inetd
194     ?      0:00    cron
231     ?      0:00    sendmail
```

5.2 MECHANISM OF PROCESS CREATION:

There are three distinct phases in the creation of process and uses three important system calls or functions

fork, exec and wait

5.2.1 Fork:

- A process in unix is created with fork system call, which creates a copy of the process that invokes it.
- The process image is practically identical to that of the calling process except for the few parameters like the PID.
- When a process is forked in this way, a child gets a new PID.
- The forking mechanism is responsible for the multiplication of the processes in the system.

5.2.2 Exec:

- Forking creates a process but its not enough to run a process.
- To do that, the forked child needs to overwrite its own image with the code and data of the new program.
- This mechanism is called the exec and child process is said to exec a new program.

5.2.3 Wait:

- The parent then executes the wait system call to wait for the child process to complete.
- It picks up the exit status of the child and then continues with its other functions.

5.3 PARENT AND CHILD PROCESS:

- Every process has a parent. This parent itself is another process and a process born from it is said to be its child.
- When you run the command

  cat emp.lst
- from the keyboard a process representing the cat command is started by the shelll process.
- The shell (sh,ksh,bash) is said to be parent of cat , while cat is the child of the shell.
- The ancestry of every process is ultimately traced to the first process (PID0) that is set up when the system is booted.

5.4 How the shell is created?

$$\begin{tikzcd}\textbf{init} \arrow[r, "\textbf{fork}"'] & \textbf{getty} \arrow[r, "\textbf{fork-exec}"'] & \textbf{login} \arrow[r, "\textbf{fork-exec}"'] & \textbf{shell}\end{tikzcd}$$

- When the system moves to multi user mode, **init** forks and execs a **getty** for every active communication port.
- Each one of the **getty** prints the login prompt on the respective terminal and then goes off to sleep.
- When a user attempts to login, getty wakes up and **fork execs** the log in program to verify the login name and password entered.
- On successful login, login forks execs the process representing the login shell.
- Repeated overlaying results in **init** becoming the immediate ancestor of the shell
- **init** goes off to the sleep, waiting for the death of its children processes.

- When the user logs off , user shell is killed and the death is intimated to the **init.**
- **Init** then wakes up and spawns the another **getty** for the next login.

5.5 THREE TYPES OF COMMANDS

The shell recognizes three types of commands.

External commands: The shell creates a process for each of these commands that it executes while remaining their parent. Ex: cat,ls

Internal commands: The shell has a number of built in commands. Ex: cd , echo

Shell scripts: The shell executes these scripts by spawning another shell which then executes the command listed in the script.The child shell becomes teh parent of the commands that features in the script.

Why directory change cant be made in separate process?

- As the child process inherits the current working directory from its parent as one of its environmental parameters. This inheritance has important consequences for the cd command.
- Its necessary for the cd command not to spawn the child to achieve a change of directory.
- If it did so, then after the child process completes its run control would revert back to parent and original directory would be restored. It would then be impossible to change the directories.

5.6 EXECUTING A COMMAND AT SPECIFIED POINT OF TIME:
at and batch command

5.6.1 at command:ONE TIME EXECUTION

- at takes its argument the time the job is to be executed and displays the at> prompt.
- Input has to be supplied from the standard input.

```
$ at > 14:08
at>empawk2.sh
[ctrl-d]
```

- commands will be executed using /usr/bin/bash
- job 1041188880 at Wed Nov 09 14:08:00 2016
- The job goes to the queue and at 2:08 p.m today ,the script file empawk2.sh will be executed .
- at shows the job number, date and time of scheduled execution.

- At also offers the keyword now, noon,today and tomorrow.
- Moreover it accepts the + symbol to act as an operator.The words that can be used with this operator
- include hours,days,weeks,month,years.
- The following form shows the use of some keywords and operators.

```
at 15
at 5pm
at 3:08 pm
at noon
at now + 1 year        //at current time after one year
at 3:08pm + 1 year     //at 3:08 pm after one year
at 9 am tomorrow
```

5.6.2 batch: Execute in batch queue:

The batch command also schedules job for later execution.
The command doesnot take any arguments but uses an internal algorithm to determine the execution time.

```
$batch < empawk2.sh
commands will be executed using /usr/bin/bash
job 1041188880 at Wed  Nov 09 14:08:00 2016
```

5.7 EXECUTE COMMAND PERIODICALLY:

5.7.1 cron and crontab files

- The ps -e command always shows the cron daemon running.
- cron executes the programs at regular intervals.
- It is mostly dormant but for every minute it wakes up and looks in crontab file /var/spool/cron/crontabs for instructions to be performed at that time.
- A user may also be permitted to place a crontab file named after his login name in the crontabs directory.
- For ex:Kumar can place his crontab commnads in the file /var/spool/cron/crontabs/kumar
- A specimen entry in the file/var/spool/cron/crontabs/kumar can look like this:

```
00-10        17     *       3,6,9,12       5       find  /  -newer  .lst_time  -print  >  backuplist
```

- Each line consists of six fields separate dby whitespaces.
- The first field:(values 00 to 59) specifies the number of minutes after the hour when the command has to be executed.
  The range here 00-10 schedules execution every minute in the first 10 minutes of the hour.
- The second field(values 1-24): indicates the hour in 24 hour format for scheduling.
  Here 17 represents 5 p.m
- The third field(values 1 to 31): controls the day of the month
  This field here uses * implies that the command is to be executed every minute for the first 10 minutes starting 5 p.m everyday.
- The fourth field(values 1 to 12): specifies the month
  Here 3,6,9,12 represents march,june,september,december
- The fifth field (values 0 to 6): indicates the days of the week.
  Here 5 represents friday.
- The sixth field indicates the command to be executed.

5.7.2 Crontab:

Creating a crontab file:

- use the crontab command to place the file in directory containing crontab files for cron to read the file again.
- crontab cron.txt
- If kumar runs this command, a fil enamed after kumar will be created in /var/spool/cron/crontabs containing the contents of cron.txt.
- Cron is mainly used by admins to perform housekeeping operations like removing the outdated files, collecting data or system performance.
- Its also used to periodically dial up to an internet mail server to send and retrieve mail.

5.8 nice command: Job Execution with low priority.:

- Processes in UNIX system are usually executed with equal priority.
- This is not always desirable, since high priority jobs has to be executed first.
- UNIX offers the nice command which is used with & operator to reduce the priority of jobs.
- To run a job with low priority the command name shouls be prefixed with nice.
  $nice wc -l uxmanual
- nice is a shell built in command and its value are system dependent and typically range from 1 to 19.
- A higher nice value implies low priority.
- nice reduces the priority of any process and thereby raising the nice value.
- We can also specify the nice value explicitly with -n option
  $nice -n 5 wc -l uxmanual &

5.9 RUNNING JOBS IN BACKGROUND:

5.9.1 &:No Logging Out

The & is the shell operator used to run a process in the background. The parent in this case doesnot wait for the childs death.Just terminate the command with a & and the command will run in background.

```
$sort -o emp.lst emp.lst &
550
```

The shell immediately returns a number the PID of the invoked command..
The prompt is returned and the shell is ready to accept another command even though the previous command has not been terminated yet.
The shell however remains th parent of the background process.

5.9.2 nohup: Log out safely:

The nohup(no Hangup) command when prefixed to command, permits execution of the process even after the user has logged out.
You must use the & with it as well.

```
$ nohup  sort emp.lst  &
586
Sending output to nohup.out
```

The shell return the PID this time too, and some shell displays this message as well.
When the nohup command is run in these shells , nohup sends the standard output to file nohup.out.

5.10 JOB CONTROL

A job is the name given to group of processes.
Following are the list of job control commands:

- Relegate a job to background(bg)
- Bring it back to foreground(fg)
- List the active jobs(jobs)
- Suspend a foreground job([ctrl-z])
- kill a job(kill)

The bg command

```
$bg
[1]     spell   uxtip02 > uxtip02.spell  &
$bg %2                          // sends second job to background
```

The fg command:

```
$fg
$fg %1                          //Bring the first job to foreground
$fg %sort                       // brings sort job to foreground
```

5.11 Signals:
Killing the processes with signals:

5.11.1 kill:Premature termination of a process:

- The kill command sends a signal usually with the intention of killing on eor more processes.
- kill is an internal command in most shells; the external /bin/kill is executed only when the shell lacks the kill capability.
- The kill command uses one or more PID as its arguments and by default uses SUGTERM(15)signal

```
kill 105                to terminate the job with pid 105
```

To terminate more than on job

```
kill 121  123  125  132         //to terminate the jobs with all these pid
```

Killing the last background job.

```
$sort -o emp.lst emp.lst &
345
$ kill $!                       // kills the last background job
```

Kill with other signals:

```
kill -9 121                     //kills the job with PID 121 with SIGKILL(9) signal
```

5.12 find command:LOCATING FILES

- find is one of the powerful tools of the UNIX system.it recursively examines a directory tree to look for files matching some criteria and then takes some action on the selected files.
- find has three components

  find path_list   selection_criteria   action
- This is how find operates:
- First it recursively examines all files in the directories specified in path_list
- It then matches each file for one or more selection_criteria.
- Finally it takes some action on those selected files.

5.12.1 Selection Criteria:

- **Locating files by inode number(-inum):**

Find allows us to locate files by their inode numbers. Use the –inum option to find all filenames that have the same inode number.

```
$ find  /  -inum  13875  -print
```

- **File types and permissions(-type and –perm)**

The –type option followed by letter f,d or l selects the files of ordinary , directory and symbolic link type.

```
$find . –type d -print
```

The –perm option specifies the permissions to match.

```
$find $HOME –perm 777 –type d -print
```

- **Finding unused files(-mtime and –atime)**
  The –mtime is files modification time and (-atime) access times to select them.
  find options can easily match files which are unaccessed or unmodified for months
  $find . –mtime -2 -print

5.12.2 The find operators(!, -0 and –a)

- The ! operator is used before an option to negate its meaning.
- The –o option represents the OR condition
- The –a option represents the AND condition.

5.12.3 options in Action component:

- **Displaying the listing:(ls)**
  The –print option belongs to action component.
- **Taking action on seleted files(-exec and –ok)**
- The –exec option lets you take the action by running Unix command on the selected files.
- The –exec takes the ommand to execute its own argument followed by {} and finally the rather cryptic symbols \;
- The –ok option is used for user confirmation as interactive option(-i).

Major expressions used by find command

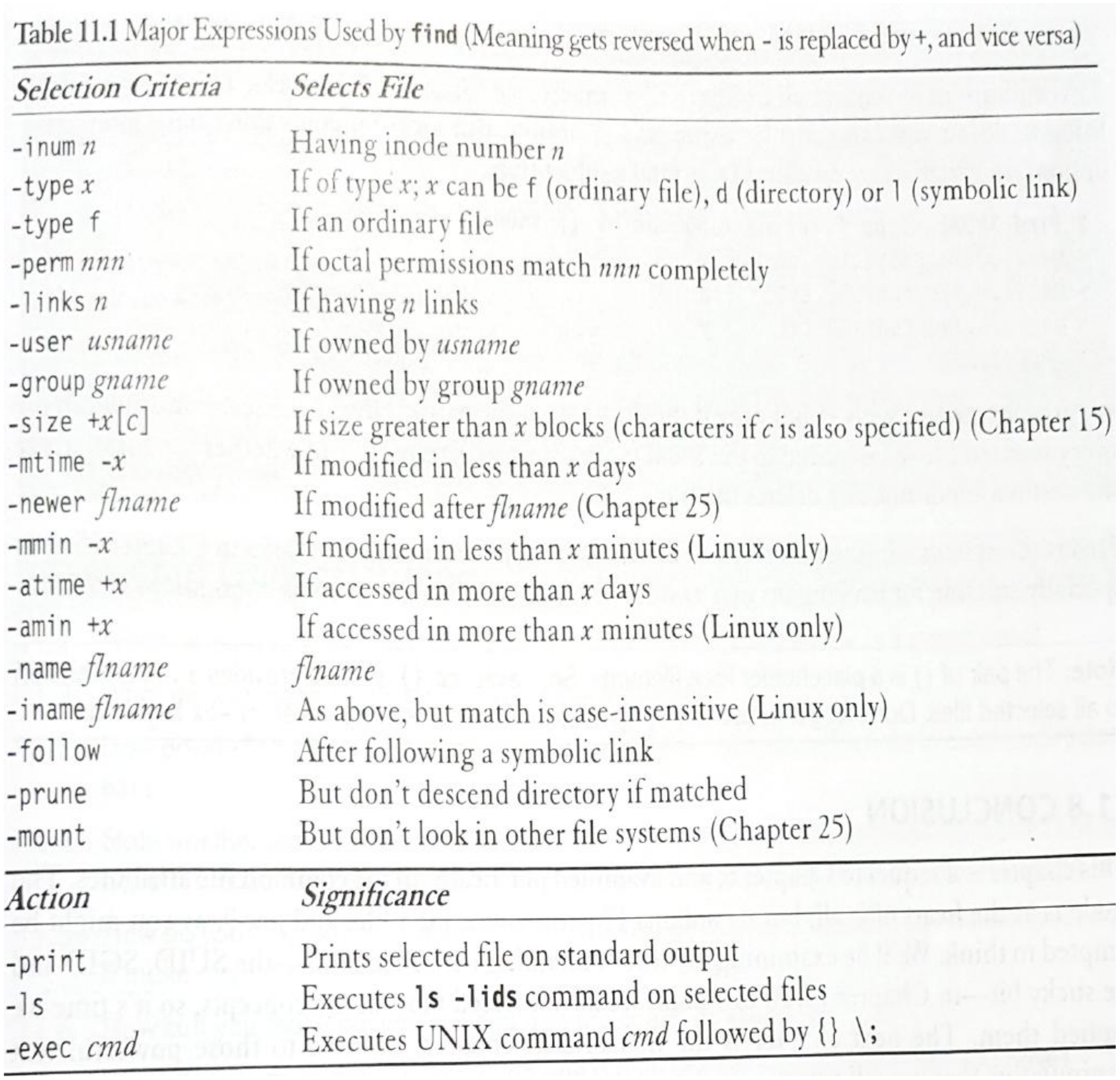

Table 11.1 Major Expressions Used by `find` (Meaning gets reversed when - is replaced by +, and vice versa)

Selection Criteria	*Selects File*
`-inum` *n*	Having inode number *n*
`-type` *x*	If of type *x*; *x* can be `f` (ordinary file), `d` (directory) or `l` (symbolic link)
`-type f`	If an ordinary file
`-perm` *nnn*	If octal permissions match *nnn* completely
`-links` *n*	If having *n* links
`-user` *usname*	If owned by *usname*
`-group` *gname*	If owned by group *gname*
`-size +`*x*`[c]`	If size greater than *x* blocks (characters if *c* is also specified) (Chapter 15)
`-mtime -`*x*	If modified in less than *x* days
`-newer` *flname*	If modified after *flname* (Chapter 25)
`-mmin -`*x*	If modified in less than *x* minutes (Linux only)
`-atime +`*x*	If accessed in more than *x* days
`-amin +`*x*	If accessed in more than *x* minutes (Linux only)
`-name` *flname*	*flname*
`-iname` *flname*	As above, but match is case-insensitive (Linux only)
`-follow`	After following a symbolic link
`-prune`	But don't descend directory if matched
`-mount`	But don't look in other file systems (Chapter 25)

Action	*Significance*
`-print`	Prints selected file on standard output
`-ls`	Executes **`ls -lids`** command on selected files
`-exec` *cmd*	Executes UNIX command *cmd* followed by {} \;

6. PERL
Practical Extraction and Report Language

6.1 A perl program runs in a special interpretive mode; the entire script is compiled internally memory before being executed.

```
$perl  -e 'print ("GNU not Unix\n");
output:
GNU not Unix
```

<u>A sample perl program</u>

to run a perl program: filename.pl

```
#!/usr/bin/perl
#script sample.pl
print(" Enter your name:");
$name = < STDIN>;
print ("enter a temperature in centigrade:");
$centigrade=<STDIN>;
$fahrenheit=$centigrade*9/5 + 32;
print "The temperature $name in Fahrenheit is $fahrenheit \n";
```

output:

```
$sample.pl
Enter your name: stallman
Enter a temperature in centigrade:  40.5
The temperature stallman in Fahrenheit is 104.9
```

6.2: The chop function: Removing the last character

- The chop function is used to remove the last character.\
- The syntax of the chop function
  chop variable
  chop(LIST)
- This function returns the character removed from EXPR and in list context, the character is removed from the last element of LIST.

```
#!/usr/bin/perl
# Script chop.pl
$string1="This is test";
$retval = chop($string1);
print "Choped string is :$string1 \n";
print "character removed:$retval\n";
```

output:

```
Choped string is : This is tes
Character removed: t
```

6.3 :chomp() function:

- The chomp() function will remove (usually) any newline character from the end of a string. The reason we say usually is that it actually removes any character that matches the current value of $/ (the input record separator), and $/ defaults to a newline.

- The syntax of chomp function is:

chomp variable
chomp (LIST)

```
#!/usr/bin/perl
$string1 = "This is test";                    /*string without \n(newline character)
$retval  = chomp( $string1 );
print " Choped String is : $string1\n";
print " Number of characters removed : $retval\n";

$string1 = "This is test\n";                          /*string with \n
$retval  = chomp( $string1 );
print " Choped String is : $string1\n";
print " Number of characters removed : $retval\n"

output:
Choped String is : This is test
Number of characters removed : 0
Choped String is : This is test
Number of characters removed : 1
```

6.4 VARIABLES AND OPERATORS:

- Perl variables have no type and do not require initialization.
- There are some of the variable attributes one should remember:
- When a string is used for numeric comparison, perl immediately converts it into a number.
- If a variable is undefined it is assumed to be a null string and a null string is equivalent to zero.

  Ex:
  $ perl -e '$x++; print("$x");'
  output
  $x is uninitialized.so it automatically takes the value(0) null.
  Next incrementing $x will result in 1
  output:1
- perl uses same set of numeric comparison operators with ==, !=, >=,<=,>,<.
- perl uses similar operators as shell for string comparison- -ne,-lt,gt,eq,-le,-ge.
- Perl supports both unquoted and quoted strings(single and double).
- When perl compares strings , it has to match ASCII value of each character.

6.5 The concatenation operator . and x

- The .(dot)operator is used to concatenate variables.
  $ perl -e '$x="yahoo"; $y=".com"; print($x . $y);'
  output:
  yahoo.com

- **$perl -e '$x="David"; $y="Marshall"; print ($x . " " . $y);'**
  output:
  David Marshall
  Here $x (David) is concatenated with space and then space is concatenated with $y(Marshall).

- Perl **uses x (repeat operator)** to repeat a string
  $ perl -e 'print "*" x 40;'
  It prints the 40 asterisk on the screen

6.6 STRING HANDLING FUNCTIONS

a. length:returns the length of the string
b.index:returns the index value of character within string.Index value starts from 0
c.substr(stg,offset,replacement)=value;
insert or replace a string. Substring value is placed with or without replacing characters.0 denotes non replacement
d.reverse:returns the reverse of the given string.
e.uc:returns the given string in uppercase letters
f.ucfirst:returns the string with first letter as upper case
g.lc:returns the given string in lowercase letters
h. lcfirst:returns the string with first letter as lower case.
Examples:
Length:

```
$x = "abcdijklm"
print length($x);
        output:                         /*length of the string=9.  Count starts from 1
        9
```

Index:

```
print index($x,j);
        output:
        5                               /* index value of  j in string abcdijklm is 5.
                                /*count starts from 0
```

substr()

```
substr($x,4,0)="efgh";          /* substring  efgh is placed at offset 4 without replacement
print($x);
        output:
        abcdefghijklm
```

reverse()

```
reverse($x);
print($x);
        output:
        mlkjihgedcba
```

lc()

```
$name =" larry wall";
print lc($name);
        output:
        larry wall
```

uc()

```
print uc($name);
```

output:
LARRY WALL

lcfirst()
print lcfirst($name);
output:
lARRY WALL
ucfirst():
print ucfirst($name);
output:
Larry wall

$x ="List of employees"
substr ($x,7,0)="sales";
print($x);
output:
List of sales employees

6.7 $_ THE DEFAULT VARIABLE:

- perl assigns the line read from input to a special variable $_; often called default variable.
- Suppose you have to prefix a line number to every line.
- This is where you need $_ to explicitly specify the line.

6.8 ($.)CURRENT LINE NUMBER AND (..) RANGE OPERATOR.

- Perl stores the current line number in another special variable called $.
  ex: $perl -ne 'print if ($. < 4)' foo

- two dots (..) is used as range operator.
  (1..5) represents 1,2,3,4,5

6.9 LISTS and ARRAYS:
List:is an ordered collection of scalar data
Arrays:is a variable that contains list.

- Example of a list
  ("Jan",123,"How are you",-34.56)
- A list need not contain data of same type.For this list to be usable, it needs to be assigned to set of variables.
  ($mon, $num, $stg,$neg)=("Jan",123,"How are you",-34.56)

- Arrays in perl are not of fixed size, they grow and shrink dynamically as elements are added and deleted.
- Arrays are defined using @ symbol.
  @array_name.

- Each element of array is accessed by $variable[n] where n is the index value,which starts from zero.
  Ex :lets assign the list to a three element array.

```
@month=("Jan","Feb","Mar");
print $month[1];
```
output:

```
Feb
```
$month[0] evaluates to string Jan, $month[2] evaluates to Mar.

6.10 ARGV[]: Command line arguments:

perl uses command line arguments which are stored in system array @ARGV[].the first argument is ARGV[0].

Perl program to illustrate the usage of ARV[] command line arguments

or

Perl Program to find the square root of a number.

```
#!/usr/bin/perl
# square_root.pl
print ("The program you are running is $0 \n");
foreach $number (@ARGV)
{
print("The square root of number is " sqrt($number));
}

$square_root.pl  123  25 436                    //to run the program
```
output

```
The program you are running is square_root.pl
The square root of 123 is 11.09
The square root of 25 is 5
Th esquare root of 456 is 21.35
```

6.11 MODIFYING ARRAY CONTENTS: PERL ARRAY FUNCTIONS:

perl has a number of functions for manipulating the contents of array.

1.shift 2.unshift 3.push 4.pop 5.splice

- shift : For moving elements to the left/to delete element from the beginning
- unshift: to add elements to the beginning of array
- push:to add elements to the end of array
- pop:to remove and return the array last element.
- splice(arg1,arg2,arg3,arg4):

Ex :

```
@list=(3..5, 9);            /*(3..5, 9)  is 3 4 5 9 where .. is range operator
shift() :
shift (@list);
  output: 4 5 9             /* 3 is deleted from beginning
pop (@list);
output: 4 5                 /* removes 9  at the end
unshift (@list, 1..3);
```

output: 1 2 3 4 5 /*adds 1, 2, 3 to beginning
push (@list, 9);
output: 1 2 3 4 5 9 /*pushes 9 at end

The splice function can do everything that these four functions can do. Additionally it uses upto four arguments to add or remove elements at any location of the array.

- The first argument takes up the list.
- The second argument is the offset from where the insertion or removal should begin.
- The third argument represents the number of elements to be removed.If the third argument is 0 then the elements has to be added.
- The fourth argument takes new list to be added or replaced

Ex:
@list=1 2 3 4 5
splice(@list,5, 0,6..8) //offset -5 , elements to be replaced is 0
//6..8(range) add 6,7,8 to list at 6th position
Adds at 6th location – 6,7,8
output:
1 2 3 4 5 6 7 8

@list =1 2 3 4 5 6 7 8
splice(@list, 0, 2)
Removes from beginning 2 elements
output:
3 4 5 6 7 8

6.12 Split() and join()

split(): This function splits a string expression into fields based on the delimiter specified by PATTERN. If no pattern is specified whitespace is the default. An optional limit restricts the number of elements returned.
Syntax:
split (/PATTERN(delimiter) /, EXPR, LIMIT)

```
split /PATTERN(delimiter) /, EXPR
Return Value
```

- Return Value in Scalar Context: Not recommended, but it returns the number of fields found and stored the fields in the @_ array.
- Return Value in Array Context: A list of fields found in EXPR or $_ if no expression is specified.

Following is the example code showing its basic usage −

```
#!/usr/bin/perl
@fields = split( / : /, "1:2:3:4:5");        // delimiter value is :
print "Field values are: @fields\n";
```

When above code is executed, it produces the following result −

```
Field values are: 1 2 3 4 5                    //delimiter (:) is used to split up values
```

join():This function combines the elements of LIST into a single string using the value of EXPR to separate each element. It is effectively the opposite of split.

- Note that EXPR is only interpolated between pairs of elements in LIST; it will not be placed either before the first or after the last element in the string.
- Following is the simple syntax for this function −
  join (delimiter,List)

Example 1:

```
#!/usr/bin/perl
$string = join( "-", "one", "two", "three" );   /* delimiter (–) is used to join values one, two, three
print"Joined String is $string\n";
```

output:

Joined String is **one-two-three**

Example 2:

```
#!/usr/bin/perl
$string = join( "", "one", "two", "three" );   /*no value is assigned to delimiter.
print"Joined String is $string\n";
```

output:

Joined String is **onetwothree**

6.13 FILE HANDLES

open():

- Perl also provides low level file handling functions that let you hard code the source and destinationof the data stream in the script itself.
- A file is opened for reading like this:
  open(INFILE, “/home/henry/mbox“);
- INFILE here is a filehandle(an identifier) of the mailbox
- Once a file has been opened , funtions that read and write the file will use the filehandle to access the file.
- A filehandle is similar to file descriptor.
- A file is opened for writing with the shell like operators > and >> having their usual meaning.
  open(OUTFILE, “>rep_out.lst“);
  open(OUTFILE, “>>rep_out.lst“);

close():

close FILEHANDLE: closes the file or pipe associated with the file handle ,closes th esystem file descriptor and flushes the IO buffers.

die:

- die List raises an exception .perl has two special variables $? and $! that helps in finding out what happened after an error has occured. The $? Variable hold sthe status of the last pipe close or system function.The $! variable can be used either a numeri or string context.
- In numeric context it holds the value of errno and in string context it holds the error string associated with errno.
- Once you detect an error and you cannot correct the problem without outside intervention, you need to communicate problem to user.
- This is usualluy done by die() funtion

6.14 ASSOCIATIVE ARRAYS

- Perl also supports hash or associative array.It alternates the array subscripts(called **keys)** and values in series of strings.
- When declaring array these strings are delimited by commas .
- This array uses the % symbol to prefix the array name.This assignment creates an array of four elements where the subsript precedes the value in the array definition.
- The array subscript which an also be a string is enclosed within a pair of curly braces rather than [ ].
- For ex $region("N") produces NORTH.

- We use associative arrays %region in the program, region.pl to expand region codes.
- The program also shows how to use two associative arrays functions :keys and values.
- Keys stores the list of subsripts in a separate array.(here @key_list)
- Value holds the value of each element in another array(here @value_list)

Ex:**region.pl**

```
#!/usr/bin/perl
#
%region =( "N", "NORTH", "S","SOUTH","E","EAST","W","WEST");
foreach $ letter(@ARGV)
{
print("The letter $letter stands for $region($letter)");
}
@key_list=keys(%region);
print("The subscripts are @key_list\n");
@value_list= values % region;
print("The values are @value_list \n");
```

output

```
$region.pl  S   W
The letter S stands for South
The letter W stands for West
The subscripts are S E N W
The values are South East North West
```

6.15 DECISION MAKING LOOP CONTROL STRUCTURE:

foreach: Looping through a list

- perl provides an extremely useful foreach construct to loop through a list.
```
foreach $var (@arr)
{
statements
}
```
- Each element of the array @arr is picked up and assigned to the variable $var.
- The iteration is continued as many times as there are items in the list.

6.16 REGULAR EXPRESSIONS:

SIMPLE AND MULTIPLE SEARCH PATTERNS:
THE MATCH AND SUBSTITUTE OPERATORS

- Perl offers a grand superset of all possible regular expressions found in the UNIX system.

The s and tr functions:

- These functions handles all substitution in perl.
- The s funtion(substitute command) is just the same way as in sed.
- tr translates characters in just the way the UNIX tr command does but slightly with different syntax:
- s/:/~/g ;
- tr /a-z/A-Z/ ;   in Unix we use tr '[a-z]' '[A-Z]'
- s and tr also accepts flags for global substitution.
- Identifying whitespaces,digits and words
- Perl also offers some escaped characters to represent the whitespace, digits and word Boundaries.

- Here are some commonly used ones

\s → A whitespace character
\d → A digit
\w→ A word character
Additional regular expressions Sequences used by Perl

Symbols	**Significance**
\w	Matches a word character
\W	Doesnot match a word character
\d	Matches a digit
\D	Doesnot match a digit
\s	Matches a whitespace character
\S	Doesnot math a whitespace character
\b	Matches on word boundary
\B	Doesnot Matches on word boundary

6.17 DEFINING AND USING SUB ROUTINES

- Perl supports functions but calls the subroutines.
- A subroutine is called by **the & symbol followed by the subroutine name.**
- If the subroutine is defined without any formal parameter , perl uses the **array @_** as the default.
- Variables inside the subroutine must be declared by **my** to make them invisible in the calling program.
  The general form of a subroutine definition in perl programming language is as follows:
  sub subroutine
  {
  body of the subroutine
  }

The typical way of calling the subroutine in perl is as follows:
&subroutine_name(list of arguments);

Example of simple function and then call it.

```
 #!/usr/bin/perl
#
sub  Hello
{
print “Hello World“;
}
#function call
Hello()
```

Output:

```
Hello World
```

Passing List to Sub routines:

Because the @_ variable is an array, it can be used to supply lists to a subroutine.However because of the way in which perl accepts and parses list and arrays, it can be difficult to extract the individual elements from @_.

```
#!/usr/bin/perl
#
sub PrintList
{
my @list =@_;
print (“Given list is @list\n“);
}
$a=10;
@b= (1,2,3,4);

#Function call with list parameter.
PrintList($a,@b);
```

Output:

```
Given list is 10  1  2  3  4
```