

**Process:**

- A **Process** is an instance of a running program. A process is said to be **born** when the program starts execution and remains **alive** as long as the program is active. After execution process is said to **die**.
- The process also has a name, usually the name of the program being executed.
- Process has attributes. These attributes of every process are maintained by the kernel in memory in a separate structure called the **process table**.
- The two important attributes of the process are:
  - The process-id (**PID**) Each process is uniquely identified by a unique integer called the process-id (PID).
  - The parent PID (**PPID**) the PID of the parent is also available as a process attribute.

**Mechanism of Process creation:**

There are three distinct phases in the creation of a process and uses three important **system calls** – fork, exec and wait.

- **Fork:** A process in UNIX is created with the 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, when a process is forked in this way, the child process get a **new PID**.
- **Exec:** the exec system call will overwrite the forked child's image with its own image. No new process is created here, the PID and PPID of the exec'd process remain unchanged.
- **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.

For example: when the user run a command (cat) from the shell,

- The shell first forks another shell process. (fork system call)
- The newly fork'd shell then overlays itself with the executable image of cat, which then starts to run. (exec)
- The parent waits for the cat to terminate and then picks up the exit status of the child (wait)

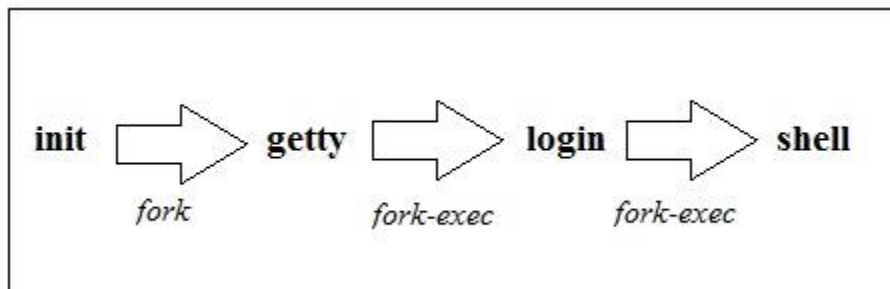
When a process is forked, the child has a different PID and PPID from its parent. However, it inherits most of the environment of its parent. The important attributes that are inherited are:

- The **real UID** and **real GID** of the process.
- The **effective UID** and **effective GID** of the process.

- The **current directory** from where the process was run.
- The **Environment variables**.
- The **descriptors** of all files opened by the parent process.

### Mechanism of Shell creation:

- The **init** process forks and execs a **getty** for every active communication port.
- Each one of the **gettys** prints the **login prompt** on the respective terminal and goes off to *sleep*.
- When a user attempts to login in; **getty** wakes up and fork-exec the login program to verify the login name and password entered.
- On successful login, **login** forks-exec the process representing the login shell.
- Repeated *overlaying* ultimately results in **init** becoming the immediate ancestor of the shell.



- **Init** goes off to sleep, waiting for the death of its children. The other processes **getty** and **login** have extinguished themselves by overlaying.
- When the user logs out, her shell is killed and the death is intimated to **init**.
- **Init** then wakes up and spawns another **getty** for that line to monitor the next login.

### Parents and children:

- Every process has a parent. This parent itself is another process and a process born from it is said to be its child.
- For example: when you run the command  
**cat emp.lst**
- A process representing cat command is started by the shell process.
- The shell is said to be the **parent** of the cat, While cat is said to be the **child** of the shell.
- The ancestry of every process is ultimately traced to the first process (**PID 0**) that is set up when the system is booted.

### PS command: Process status

- The ps command is used to display some process attributes.
- ps displays the processes owned by the user running the command.

- Example

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

- The header fields are **PID** which represents the process id, **TTY** represents the terminal name, **TIME** represents the cumulative processor time that has been consumed since the process has been started, and the **CMD** represents the process name.

## ps options:

The various options of ps commands are shown in the table below:

POSIX option	Significance
<b>-f</b>	Full list showing the PPID of each process.
<b>-e or -A</b>	All processes including the user and system processes
<b>-u</b> <i>usr</i>	Processes of user <i>usr</i> only
<b>-a</b>	Processes of all users excluding processes not associated with terminal
<b>-l</b>	Long listing showing memory-related information
<b>-t</b> <i>term</i>	Processes running on terminal <i>term</i>

**Full listing (-f):** is used to get the detailed listing of every process.

```
$ ps -f
UID  PID  PPID  C  STIME      TTY  TIME CMD
Sumith 367   291    0   12:35:16    console 00:00  cat emp.lst
Sumith 291    1    0   12:25:58    console 00:00  bash
Sumith 368   291    0   12:35:16    console 00:00  wc emp.lst
```

**UID** shows the owner of every process, **PPID** indicates the Parent PID, **C** indicates and index for recent processor utilization, **STIME** indicates the starting time of the process.

**Displaying the process of a user (-u):** the system administrator needs to use the **-u** (user) option to the know activities of any user

```
$ ps -u sumit
PID  TTY  TIME CMD
367    ?    00:05 Xsun
401    ?    00:00 Xsession
300    pts/3 00:00  bash
346    pts/3 00:60  vi
```

Displays all the processes belonging to the user Sumit.

**Displaying the processes of all the users (-u):** the -a (all) option lists processes of all users but doesn't display the system processes:

```
$ ps -a
PID   TTY    TIME CMD
347   pts/1   00:00 ksh
401   pts/2   00:00 sh
300   pts/3   00:00 bash
350   pts/1   00:60 vi
```

### **System processes (-e or -A)**

- Apart from the processes a user generates, a number of system processes keep running all the time. Most of them are not associated with any controlling terminal.
- They are spawned during system startup and some of them start when the system goes into multiuser mode. These processes are known as **daemons** because they are called without a specific request from a user.
- To list them use,

```
$ ps -e
PID   TTY          TIME          CMD
0      ?             0:34          sched
1      ?             41:55         init
23     Console      0:03          sh
272    ?             2:47          cron
7015   term/12      20:04         vi
```

### **at: One-Time Execution**

- To schedule one or more commands for a specified time, use the at command. With this Command, you can specify a time, a date, or both.

For example,

```
$ at 14:23 Friday
at> date
at> echo "Good Morning"
[Ctrl-d]
commands will be executed using /usr/bin/bash
job 1041198880.a at Fri Oct 12 14:23:00 2016
```

- All at jobs go into a queue.
- at shows the job number, the date and time of scheduled execution. T

- his job number is derived from the number of seconds elapsed since the Epoch. A user should remember this job number to control the job.
- The standard output and standard error of the program are mailed to the user.
- at offers the key word *now*, *noon*, *midnight*, *today* and *tomorrow*. Moreover it accepts + symbol to act as an operator. The words that can be used with this operator includes *hours*, *days*, *weeks*, *months* and *years*.
- The following forms show the use of some of the key words and operators:
  - at      15**
  - at      5pm**
  - at      3:08pm**
  - at      noon**
  - at      now + 1 year**
  - at      3:08pm + 1 day**
  - at      15:08 December 18,2001**
  - at      9am tomorrow**

### **cron: Running jobs periodically**

- cron program is a daemon which is responsible for running repetitive tasks on a regular schedule.
- It is a perfect tool for running system administration tasks such as backup and system logfilemaintenance. It can also be useful for ordinary users to schedule regular tasks including calendarreminders and report generation.
- To schedule commands or processeson a regular basis, you use the cron (short for *chronograph*) program. You specify the times anddates you want to run a command in crontab files.
- Times can be specified in terms of minutes,hours, days of the month, months of the year, or days of the week.
- cron is listed in a shell script as one of the commands to run during a system boot-up sequence.
- Individual users don't have permission to run cron directly.
- If there's nothing to do, cron "goes to sleep" and becomes inactive; it "wakes up" every minute,however, to see if there are commands to run.
- *cron* looks for instructions to be performed in a control file in/var/spool/cron/crontabsAfter executing them, it goes back to sleep, only to wake up the next minute.
- To a create a crontab file, First use an editor to create a crontab file say cron.txt Next use crontab command to place the file in the directory containing crontab files.
- crontab will create a file with filename same as user name and places it in /var/spool/cron/crontabs directory.

**A typical entry in crontab file:**

A typical entry in the crontab file of a user will have the following format.

**Minute hour day-of-month month-of-year day-of-week command**

Where, Time-Field Options are as follows:

Field	Range
-----	
<i>minute</i>	00 through 59 Number of minutes after the hour
<i>hour</i>	00 through 23 (midnight is 00)
<i>day-of-month</i>	01 through 31
<i>month-of-year</i>	01 through 12
<i>day-of-week</i>	01 through 07 (Monday is 01, Sunday is 07)

- The first five fields are time option fields. You must specify all five of these fields. Use an **asterisk (\*)** in a field if you want to ignore that field.
- Example 1:  
**00-10 17 \* 3.6.9.12 5 find / -newer .last\_time -print >backuplist**

In the above entry, the find command will be executed every minute in the first 10 minutes after 5 p.m. every Friday of the months March, June, September and December of every year.

**Running Jobs in Background**

- The basic idea of a background job is simple. It's a program that can run without prompts or other manual interaction and can run in parallel with other active processes.
- There are two ways of starting a job in the background – with the shell's **&** operator and the **nohup** command.

**&: No Logging out**

- Use the **&** symbol at the end of the command line to direct the shell to execute the command in the background.

**\$ sort -o emp.dat emp.dat &**

[1] 1413 *The job's PID*

Note:

1. Observe that the shell acknowledges the background command with two numbers. First number [1] is the *job ID* of this command. The other number 1413 is the PID.

2. When you specify a command line in a pipeline to run in the background, all the commands are run in the background, not just the last command.
3. The shell remains the parent of the background process.

### **nohup: Log out Safely**

- A background job executed using **&** operator ceases to run when a user logs out. This is because, when you logout, the shell is killed and hence its children are also killed.
- The UNIX system provides **nohup** statement which when prefixed to a command, permits execution of the process even after the user has logged out. You must use the **&** with it as well.
- In the following command, the sorted file and any error messages are placed in the file **nohup.out**.

```
$ nohup sort sales.dat &
```

```
1252
```

```
Sending output to nohup.out.
```

- The shell has returned the PID (1252) of the process.
- When the user logs out, the child turns into an orphan. The kernel handles such situations by reassigning the PPID of the orphan to the system's init process (PID 1) - the parent of all shells.
- When the user logs out, init takes over the parentage of any process run with nohup. In this way,
- you can kill a parent (the shell) without killing its child.

### **nice: Job Execution with Low Priority**

- Processes in UNIX system are usually executed with equal priority, sometimes this is not desirable.
- UNIX offers the **nice** command, which is used with the **&** operator to reduce the priority of jobs.
- To run a job with low priority, the command name should be prefixed with nice.

```
nice wc -l a.txt &
```

- The nice values are system dependent and typically range from 1 to 19. Commands execute with a nice value that is generally in the middle of the range – usually 10.
- Higher nice values imply lower priority. Nice reduces the priority of any processes, thereby raising its nice values.
- nice values can be explicitly specified with the **-n** option:

```
nice -n 5 wc -l a.txt &
```

## **Killing Processes with Signals:**

- The UNIX system communicates the occurrences of an event to a process by sending **signal** to the process.
- Each signal is identified by a **number** and a **symbolic name** having the prefix **SIG** and is designed to perform a specific function.
- Some of the signals available in UNIX are:

SIGNAL NAME	NUMBER
SIGTERM	15
SIGINT	2
SIGKILL	9

- The **kill** command sends a signal, with the intention of killing one or more processes.
- The command uses one or more PIDs as its arguments, and by default uses **SIGTERM (15)** signal. Thus

**kill 105** *it's like using kill -s TERM 105*  
terminates the job having the PID 105.

- kill command can accept more than one PIDs as arguments

**kill 121 122 123 124 135 136**

### ***Using Kill with other signals:***

- By default kill command kills with the SIGTERM but it is also possible to kill a process with different signals

**kill -s KILL 121**

or

*kills the process with PID 121 with KILL Signal*

**kill -9 121**

## **Job Control**

A job is a name given to a group of processes that is typically created by piping a series of commands using pipeline character. You can use job control facilities to manipulate jobs. You can use job control facilities to,

1. Relegate a job to the background (bg)
2. Bring it back to the foreground (fg)
3. List the active jobs (jobs)
4. Suspend a foreground job ([Ctrl-z])
5. Kill a job (kill)

The following examples demonstrate the different job control facilities.

- Assume a process is taking a long time. You can **suspend** it by pressing [Ctrl-z].

**[1] + Suspended wc -l hugefile.txt**

- A suspended job is not terminated. You can now relegate it to background by,

**\$ bg**

- You can start more jobs in the background any time:

**\$ sort employee.dat > sortedlist.dat &**



[2] 530

\$ grep 'director' emp.dat &

[3] 540

- You can see a listing of these jobs using jobs command,  
\$ jobs  
[3] + Running grep 'director' emp.dat &  
[2] - Running sort employee.dat > sortedlist.dat &  
[1] Suspended wc -l hugefile.txt
- You can bring a job to foreground using fg %jobno OR fg %jobname as,  
\$ fg %2  
OR  
\$ fg %sort

### **find : locating files**

- It recursively examines a directory tree to look for files matching some criteria, and then takes some action on the selected files.
- The syntax of find is as follows:  
**find path\_listselection\_criteriaaction**
- The find operates:
  - Recursively examines all files specified in **path\_list**
  - It then matches each file for one or more **selection-criteria**
  - It takes some **action** on those selected files.
- The **path\_list** comprises one or more subdirectories separated by whitespace. There can also be a host of **selection\_criteria** that you can use to match a file, and multiple **action** can also be performed on these files.
- The various **selection\_criteria** are shown in the table below:

<b>Selection Criteria</b>	<b>Select Files</b>
-inum n	Having inode number n
-type x	If of type x; x can be f, d, l
-type f	If an ordinary file
-perm nnn	If octal permissions match nnn completely
-links n	If having n links
-user username	If owned by username
-group gname	If owned by group gname
-size +x	If size is greater than x blocks
-mtime -x	If modified is less than x days
-newer fname	If modified after fname
-mmin -x	If modified is less than x minutes
-atime +x	If accessed in more than x days

-amin +x	If accessed in more than x minutes
-name fname	fname
-iname fname	As above, but match is case-insensitive
-follow	After following a symbolic link
-prune	But don't descend directory if matched
-mount	But don't look in other file systems.

- The various **actions** are shown in the table below:

Action	Significance
-print	Print selected file on standard output
-ls	Executes <code>ls -lids</code> command on selected files
-exec <i>cmd</i>	Executes UNIX command <i>cmd</i> followed by <code>{ } \;</code>
-ok <i>cmd</i>	Executes UNIX command <i>cmd</i> interactively followed by <code>{ } and \;</code>

### Find operators (!, -o and -a):

- The **negation operator !** is used before an option to negate its meaning:  
**find . !-name "\*.c" -print**  
selects all but c program files.
- The OR operator **-o** is used to represent the OR condition:  
**find /home \( -name "\*.sh" -o -name "\*.pl" \) -print**  
The ( and ) are special characters that are interpreted by the shell to group commands.
- The **-a** operator is used to represent the AND condition, and is implied by default whenever two selection criteria are placed together.

### Examples:

- **\$ find / -name a.out -print**  
searches the file with name a.out in the root directory structure.
- **\$ find . -name "\*.c" -print**  
searches the file with name ending with .c (C program files) in the current working directory structure.
- **\$ find \$HOME -perm 777 -type d -print**  
Searches the directory with the permission 777 in the HOME directory.
- **\$ find . -type f -mtime +2 -mtime -5 -ls**  
Searches a file with last modified time greater than 2 minutes and less than 5 minutes in the current working directory, once found `ls -lids` command is executed on the files found.
- **\$ find \$HOME -links 2 -exec rm {} \;**  
Searches and removes *all* the files with link count equal to 2 in the HOME directory.
- **\$ find \$HOME -perm 666 -ok rm {} \;**  
Searches and removes the files with permission 666 *interactively* in the HOME directory.

## Perl Programming

- Perl: Perl stands for Practical Extraction and Reporting Language.
- The language was developed by Larry Wall. Perl is a popular programming language because of its powerful pattern matching capabilities, rich library of functions for arrays, lists and file handling.
- Perl is also a popular choice for developing CGI (Common Gateway Interface) scripts on the www (World Wide Web). Perl is a simple yet useful programming language that provides the convenience of shell scripts and the power and flexibility of high-level programming languages.
- Perl programs are interpreted and executed directly, just as shell scripts are; however, they also contain control structures and operators similar to those found in the C programming language. This gives you the ability to write useful programs in a very short time.

A perl program runs in a special interpretive model; the entire script is compiled internally in memory before being executed. Script errors, if any, are generated before execution.

The following is a sample perl script.

```
#!/usr/bin/perl
# Script: sample.pl – Shows the use of variables
#
print("Enter your name: ");
$name=<STDIN>;
print("Enter a temperature in Centigrade: ");
$centigrade=<STDIN>;
$fahr=$centigrade*9/5 + 32;
print "The temperature in Fahrenheit is $fahr\n";
print "Thank you $name for using this program.";
```

There are two ways of running a perl script.

- One is to assign execute (x) permission on the script file and run it by specifying script filename (chmod +x filename).
- Other is to use perl interpreter at the command line followed by the script name. In the second case, we don't have to use the interpreter line viz., #!/usr/bin/perl.

```
$ chmod +x sample.pl ; sample.pl
Enter your name: stallman
Enter a temperature in Centigrade: 40.5
The temperature stallman
in Fahrenheit is 104.9
```

## **The chop function**

The chop function is used to remove the last character of a line or string. In the above program, the variable \$name will contain the input entered as well as the newline character that was entered by the user.

In order to remove the \n from the input variable, we use chop(\$name).

**Example:** chop(\$var); will remove the last character contained in the string specified by the variable var.

Note that you should use chop function whenever you read a line from the keyboard or a file unless you deliberately want to retain the newline character.

```
#!/usr/bin/perl
# Script: name.pl - Demonstrates use of chop
#
print("Enter your name: ") ;
$name = <STDIN> ;
chop($name) ;                      # Removes newline character from $name
if ( $name ne "" ) {
    print("$name, have a nice day\n") ;
} else {
    print("You have not entered your name\n") ;
}
```

## **Variables and Operators**

Perl variables have no type and need no initialization. However, we need to precede the variable name with a \$ for both variable initialization as well as evaluation.

**Example:** \$var=10;  
print \$var;

Some important points related to variables in perl are:

1. When a string is used for numeric computation or comparison, perl converts it into a number.
2. If a variable is undefined, it is assumed to be a null string and a null string is numerically zero. Incrementing an uninitialized variable returns 1.
3. If the first character of a string is not numeric, the entire string becomes numerically equivalent to zero.
4. When Perl sees a string in the middle of an expression, it converts the string to an integer. To do this, it starts at the left of the string and continues until it sees a letter that is not a digit.

**Example:** "12O34" is converted to the integer 12, not 12034.

**Comparison Operators:**

Perl supports operators similar to C for performing numeric comparison. It also provides operators for performing string comparison. \

**Numeric comparison**

==  
!=  
>  
<  
>=  
<=

**String comparison**

eq  
ne  
gt  
lt  
ge  
le

**Concatenating Operators:**

Perl provides three operators that operate on strings:

- The . operator, which joins two strings together;  
**\$ perl -e '\$x=maruti' ; \$y=".com" ; print(\$x . \$y . "\n") ;**

**maruti.com**

- The x operator, which repeats a string  
**Ex:** The following statement prints 40 asterisks on the screen:

**\$ perl -e 'print "\*" x 40'**

\*\*\*\*\*

**String Handling Functions:**

Perl has all the string handling functions that you can think of.

Some of the frequently used functions are:

- **length** determines the length of its argument.
- **index(s1, s2)** determines the position of a string **s2** within string **s1**.
- **substr(str,m,n)** extracts a substring from a string **str**, **m** represents the starting point of extraction and **n** indicates the number of characters to be extracted.

```
$x = "abcdijklm" ;  
print length($x) ;  
print index($x,j) ;  
substr($x,4,0) = "efgh" ;  
print "$x" ;
```

*This is 9  
This is 5  
Stuffs \$x with efgh  
\$x is now abcdefghijklm*

It can extract characters from the right of the string, and insert or replace the string,

- **uc(str)** converts all the letters of **str** into uppercase.
- **ucfirst(str)** converts first letter of all leading words into uppercase.

**Ex:**

```
$ name = "larry wall" ;  
$ result = uc($name) ;  
$ result = ucfirst($name) ;
```

\$result is LARRY WALL  
\$result is Larry wall

- **reverse(str)** reverses the characters contained in string **str** and returns the reversed string

**Ex:**

```
$ x = "abcd" ;  
print reverse($x);
```

Prints dcba

The functions **lc** and **lcfirst** perform opposite functions of their “uc” counterparts.

### **Specifying Filenames in Command Line**

- Perl provides specific functions to open a file and perform I/O operations on it.
- Perl also supports special symbols that perform the same functionality.
- The diamond operator, **<>** is used for reading lines from a file.

When you specify STDIN within the **<>**, a line is read from the standard input.

**Example:**

1. **perl -e 'print while (< >)' sample.txt**
2. **perl -e 'print < >' sample.txt**

In the first case, the file opening is implied and **< >** is used in scalar context (reading one line).

In the second case, the loop is also implied but **< >** is interpreted in list context (reading all lines).

We can also read from multiple files:

**perl -e 'print while (< >)' foo1 foo2 foo3**

### **Reading Files in a Script:**

The following script will print all Gupta's and Agarwal/Aggarwal's contained in a file (specified using an ERE) that is specified as a command line parameter along with the script name.

```
#!/usr/bin/perl  
printf("%30s", "LIST OF EMPLOYEES\n");  
while(<>) {  
    print if /\bgupta|Ag+[ar][ar]wal/ ;  
}
```

**\$\_ (Dollar-Underscore): The Default Variable**

perl assigns the line read from input to a special variable, \$\_, often called the default variable. chop, <> and pattern matching operate on \$\_ by default. It represents the last line read or the last pattern matched.

By default, any function that accepts a scalar variable can have its argument omitted. In this case, Perl uses \$\_, which is the default scalar variable. chop, <> and pattern matching operate on \$\_ by default, the reason why we did not specify it explicitly in the print statement in the previous script.

The \$\_ is an important variable, which makes the perl script compact.

For example, instead of writing

```
$var = <STDIN>;
```

```
chop($var);
```

you can write,

```
chop(<STDIN>);
```

In this case, a line is read from standard input and assigned to default variable \$\_, of which the last character (in this case a \n) will be removed by the chop() function.

Note that you can reassign the value of \$\_, so that you can use the functions of perl without specifying either \$\_ or any variable name as argument.

**\$. (Current Line number) And .. (The range operator)**

\$. (Dollar-Dot) is the current line number. It is used to represent a line address and to select lines from anywhere.

**Example:**

```
perl -ne 'print if ($. < 4)' in.dat
```

# is similar to head -n 3 in.dat

```
perl -ne 'print if ($. > 7 && $. < 11)' in.dat
```

.. (Dot Dot) is the range operator. Example:

```
perl -ne 'print if (1..3)' in.dat
```

# Prints lines 1 to 3 from in.dat

```
perl -ne 'print if (8..10)' in.dat
```

# Prints lines 8 to 10 from in.dat

You can also use compound conditions for selecting multiple segments from a file.

**Example:** if ((1..2) || (13..15)) { print ;}

# Prints lines 1 to 2 and 13 to 15

## **Lists and Arrays:**

- Perl allows us to manipulate groups of values, known as lists or arrays.
- These lists can be assigned to special variables known as array variables, which can be processed in a variety of ways.
- **A list is a collection of scalar values enclosed in parentheses.**

The following is a simple example of a list:

(1, 5.3, "hello", 2)

This list contains four elements, each of which is a scalar value: the numbers 1 and 5.3, the string "hello", and the number 2.

For a list to be usable, it needs to be assigned to a set of variables

**( \$num1, \$str, \$num2, \$num3 ) = ( 1, 5.3, "hello", 2 );**

To indicate a list with no elements, just specify the parentheses: ()

You can use different ways to form a list. Some of them are listed next.

- Lists can also contain scalar variables:  
(17, \$var, "a string")
- A list element can also be an expression:  
(17, \$var1 + \$var2, 26 << 2)
- Scalar variables can also be replaced in strings:  
(17, "the answer is \$var1")
- The following is a list created using the list range operator:  
(1..10) □ same as (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
- The list range operator can be used to define part of a list:  
(2, 5..7, 11)

The above list consists of five elements: the numbers 2, 5, 6, 7 and 11



## Arrays

- Arrays are placeholders of list.
- An array is created by assigning a list to it
- Arrays in perl need not contain similar type of data.
- Also arrays in perl can dynamically grow or shrink at run time.

```
@array = (1, 2, 3);           # Here, the list (1, 2, 3) is assigned to the array variable
                                @array.
```

```
@month = ("Jan", "Feb", "March");           $month[0] is Jan
```

Perl also supports the **qw** function that can make short work of this assignment:

```
@month = qw/Jan Feb March/;
```

Perl uses **@** and **\$** to distinguish array variables from scalar variables, the same name can be used in an array variable and in a scalar variable:

```
$var = 1;
@var = (11, 27.1, "a string");
```

Here, the name **var** is used in both the scalar variable **\$var** and the array variable **@var**. These are two completely separate variables. You retrieve value of the scalar variable by specifying **\$var**, and of that of array at index 1 as **\$var[1]** respectively.

Following are some of the examples of arrays with their description.

```
x = 27;           # list containing one element
@y = @x;          # assign one array variable to another
@x = (2, 3, 4);
@y = (1, @x, 5);   # the list (2, 3, 4) is substituted for @x, and the resulting list
                  # (1, 2, 3, 4,5) is assigned to @y.
```

```
$len = @y;        # When used as an rvalue of an assignment, @y evaluates to the
                  # length of the array.
```

```
$last_index = $#y; #    $# prefix to an array signifies the last index of the array.
```

```
#!/usr/bin/perl
# ar_in_ar.pl - Shows use of arrays
#
@days_between = ("Wed", "Thu") ;
@days = (Mon, Tue, @days_between, Fri) ;
@days[5,6] = qw/Sat Sun/ ;
$length = @days ;                               # @days in scalar context
#
print("The third day of the week is $days[2]\n") ;
print("The days of the week are @days\n") ;
print("The number of elements in the array is $length\n") ;
print("The last subscript of the array is $#days\n") ;
$#days = 5;                                     #Resize the array
print("\$days[6] is now $days[6]\n") ;
```

```
$ ar in ar.pl
The third day of the week is Wed
The days of the week are Mon Tue Wed Thu Fri Sat Sun
The number of elements in the array is 7
The last subscript of the array is 6
$days[6] is now
```

Note that after resizing the array (with \$#days = 5), \$days[6] is now a null string.

### **Reading a File into an Array:**

An array can also be populated by the <> operator. Each line then becomes an element of the array:

```
@line = <> ;           #Reads entire file from command line
print @line ;          #Prints entire file
```

The entire file is read with a single statement (@line = <>), and each element of the array , @line contains a line of the file (including the newline).

### **ARGV[]: Command Line Arguments**

The special array variable @ARGV is automatically defined to contain the strings entered on the command line when a Perl program is invoked. For example, if the program (test.pl):

```
#!/usr/bin/perl
print("The first argument is $ARGV[0]\n");
```

Then, entering the command

```
$ test.pl 1 2 3
```

produces the following output: The first argument is 1

Note that \$ARGV[0], the first element of the @ARGV array variable, does not contain the name of the program.

```
#!/usr/bin/perl
# Script: leap_year.pl - Determines whether a year is a leap year or not
#
die("You have not entered the year\n") if (@ARGV == 0 ) ;
$year = $ARGV[0] ;                               # The first argument
$last2digits = substr($year, -2, 2) ;           # Extract from the right
if ($last2digits eq "00") {
    $yesorno = ($year % 400 == 0 ? "certainly" : "not" ) ;
}
else {
    $yesorno = ($year % 4 == 0 ? "certainly" : "not" ) ;
}
print("$year is " . $yesorno . " a leap year\n") ;
```

## **List Operators:**

### **1.) The Push and Pop Functions:**

The Push operator works with two arguments:

- The first argument is an array variable name and the second argument is the element to be pushed.

**Ex:**

```
@x = (1,2,3,4) ;  
push(@x,5) ;  
print @x ;
```

**Output:**

**12345**

- The Pop operator works with only one argument – an array variable name.

**Ex:**

```
@x = (1,2,3,4) ;  
pop(@x);  
print @x ;
```

**Output:**

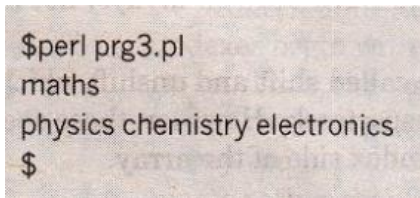
**1234**

### **2.) The Splice Operator:**

- This function allows adding or removing items even from the middle of an array, allowing the array to grow or shrink as required.
- This operator works with a maximum of four arguments.
- The general format is :  
**splice (@array, \$offset, [\$length], [\$list]);**
- The first argument must be an array on which the splice works and this argument must be present.
- The second argument is the offset from where the insertion or removal begins.

**Ex:**

```
$cat prg3.pl  
#!/usr/bin/perl  
@subjects = ("physics", "chemistry", "maths");  
$x = splice(@subjects, 2, 1, "electronics");  
print $x @subjects;  
$
```

**Output:**

```
$perl prg3.pl
maths
physics chemistry electronics
$
```

**Modifying Array Contents**

For deleting elements at the beginning or end of an array, perl uses the **shift** and **pop** functions. In that sense, array can be thought of both as a stack or a queue.

Example:

```
@list = (3..5, 9);
shift(@list); # The 3 goes away, becomes 4 5 9
pop(@list);   # Removes last element, becomes 4 5
```

The **unshift** and **push** functions add elements to an array.

```
unshift(@list, 1..3);      # Adds 1, 2 and 3 -- 1 2 3 4 5
push(@list, 9);            # Pushes 9 at end -- 1 2 3 4 5 9
```

- The **splice** function can do everything that shift, pop, unshift and push can do.
- It uses upto **four** arguments to add or remove elements at any location in the array.
- 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 it is 0, elements have to be added. The new replaced list is specified by the fourth argument (if present).

```
splice(@list, 5, 0, 6..8); # Adds at 6th location, list becomes 1 2 3 4 5 6 7 8 9
splice(@list, 0, 2);       # Removes from beginning, list becomes 3 4 5 6 7 8 9
```

**foreach: Looping Through a List**

foreach construct is used to loop through a list.

Its general form is,

```
foreach $var in (@arr) {
statements
}
```

**Example:** To iterate through the command line arguments (that are specified as numbers) and find their square roots,

```
foreach $number (@ARGV) {
```

```
print("The square root of $number is " . sqrt($number) . "\n");
}
```

You can even use the following code segment for performing the same task. Here note the use of `$_` as a default variable.

```
foreach (@ARGV) {
print("The square root of $_ is " . sqrt() . "\n");
}
```

### Another Example

```
#!/usr/bin/perl
@list = ("This", "is", "a", "list", "of", "words");
print("Here are the words in the list: \n");
foreach $temp (@list) {
print("$temp ");
}
print("\n");
```

Here, the loop defined by the `foreach` statement executes once for each element in the list `@list`. The resulting output is **Here are the words in the list: This is a list of words**

The current element of the list being used as the counter is stored in a special scalar variable, which in this case is `$temp`. This variable is special because it is only defined for the statements inside the `foreach` loop.

perl has a `for` loop as well whose syntax similar to C. Example:

```
for($i=0 ; $i < 3 ; $i++) { ...
```

**Array Handling Functions:**

- Split
- Join

**split : Splitting into a List or Array:**

- The split function is used to break a string into its constituent elements according to a separator.
- The separator could be anything like a whitespace, tab, colon or anything.
- The general format of this function is :

**split(/separator/, \$string);**

**Ex:**

```
$color_list = ("red, yellow, blue");
($var1, $var2, $var3) = split(/ , /,$color_list);
print "$var1" ;
print "$var2" ;
print "$var3" ;
```

- When the split function is used and the returned values are not stored in an array explicitly, they will be stored in the special array @\_ by implication.

**Ex:**

```
split(/:/);           #Returned fields are stored in the array@_
```

- When no string is mentioned explicitly , the split function works on the default variable \$\_

**Ex:**

```
@colors = split(/:/);      #$_ is the default string
```

- When no field separator is mentioned explicitly, the white spaces are taken as the field separator by default.

**The Join Function:**

- The Join function pastes the elements of a list into a string.
- The general format is:  

**join EXPR, LIST;**
- The first argument EXPR may be any string.
- This function puts the EXPR string between individual elements of the LIST and returns the resulting string.

**Ex:**

```
$result = join " ", ("This", "is", " an", "example");  
print $result;
```

**Output:**

**This is an example**

In the above example, the value of EXPR is a white space represented by “ ”.

**Another example:**

```
#!/usr/bin/perl  
$x = join ":",3,4,12,15;  
print "$x\n";  
@y = split / : /, $x;  
print "@y\n";  
$x = join "-",@y;  
print "$x\n";
```

**Output:**

**3:4:12:15**

**3 4 12 15**

**3-4-12-15**

**Handling Files:**

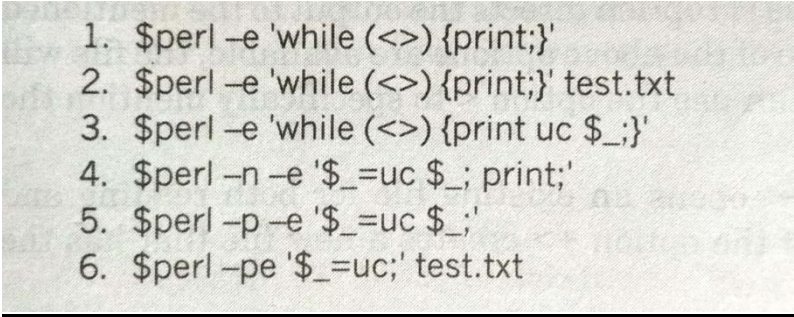
**FileHandle:**

- FileHandle is the name of a Perl program that provides input/output connection between a Perl process and the outside world.
- In other words, a filehandle is just the name or label of a connection; it is not exactly the name of a file.
- The names of filehandles are constructed using alphanumerics plus underscores.
- The names of filehandlers should not begin with a digit.
- The names of filehandlers are usually written in uppercase.
- Perl has three standard input/output streams, the standard input, the standard output and the standard error streams, identified using STDIN,STDOUT AND STDERR filehandles.
- Perl has four more reserved filehandles:
  - NULL
  - DATA
  - ARGV

- ARGVOUT

**The NULL Filehandle:**

- This is a special filehandle that allows scripts to get input from either STDIN or from each file listed on the command line.
- It is written as `<>` and is called the diamond operator or line reading operator or angle operator.
- The use of the `-n` option implies the existence of the **while** `<> {...}`. The `-p` is another option that serves a similar purpose. Both of these options eliminate the use of **while** `<>{...}`

**Ex:**

```
1. $perl -e 'while (<>) {print;}'
2. $perl -e 'while (<>) {print;} test.txt'
3. $perl -e 'while (<>) {print uc $_;}'
4. $perl -n -e '$_=uc $_; print;'
5. $perl -p -e '$_=uc $_;'
6. $perl -pe '$_=uc;' test.txt
```

**The open() Function:**

- Accessing a file means opening a file either for reading writing/appending or piping.
- In Perl, a file is opened using the `open()` function. The format of open function is:  
**`open(FILEHANDLE,"> | >> | FILENAME");`**
- The use of option `>` opens the mentioned file for writing. If the file does not exist, it will be created. If the file is already present it will be overwritten.
- The option `>>` is used to open the file for appending.
- The use of `|` option directs the output to the mentioned destination as a filter.
- If none of the options are available, then the file opened for reading.

**The close() Function:**

- An open file has to be closed after the necessary processing has been done.
- This is done by using the `close()` function.
- The statement **`close(FILEHANDLE);`** closes the file referred to by the FILEHANDLE.
- Opening a file that is already open, closes the file and reopens it.



**File Tests:**

- It is a good and recommended practice to find out whether a target file can be manipulated or not.
- For this, certain properties of these files must be tested before carrying out any type of operation on them.
- Perl has an elaborate system of file tests.

<i>Operator</i>	<i>Meaning</i>
-e	File or directory exists
-r	File or directory is readable
-w	File or directory is writable
-x	File or directory is executable
-f	Regular file
-d	Directory file
-T	Text file
-B	Binary file

**The die Function:**

- There are situations when the user may make an attempt to execute a program with a wrong filename or insufficient number of arguments.
- It is necessary to trap such type of situations and display a proper message. In Perl this is achieved using the **die function**.

```
...STDLIST = "./student.lst";
open(STDLIST) or die("Unable to open file $STDLIST Program terminated");
...
```

- Consider a situation wherein the user goes wrong in giving the path of the file on which the processing is to be done. Then the output of the program above looks like:

```
$perl fileIO.pl
Unable to open file ./studnet.lst
Program terminated /root/fileIO.pl line 5.
$
```

### **The chop() and chomp() Functions:**

Consider a Perl code segment:

```
$cat prg5.pl
#!/usr/bin/perl
print "Input a line of text\n";
$line = <STDIN>;
print "$line\n";
$
```

The output of the above code segment will be:

```
$perl prg1.pl
Input a line of text
Good Luck
Good Luck
# additional new line
$
```

We can observe that an extra new line is generated in the output.

**The chop() function is used to remove or eliminate the last character.**

Now consider the following code segment:

```
$cat prg6.pl
#!/usr/bin/perl
print "Input a line of text\n";
$line = <STDIN>;
chop($line);
print "$line\n";
$
```

The output of the above code segment will be:

```
$perl prg6.pl
Input a line of text
Good Luck
Good Luck
$
```

When a Perl code line like `chop($line );` is executed, the contents of the variable `$line` will be cleaned up, that is, the last character will be discarded and **the `chop()` function returns the character that is discarded or eliminated.**

Perl makes use of one more function, `chomp()`.

**The `chomp()` function removes only the newline character, `\n`, if that appears as the last character.**

If more than one new line character exists at the end then only one new line character is removed.

When no trailing newline character is present, the `chomp()` function returns as 0, as no character is removed; else `chomp()` function will return 1.

### **Associative Arrays (Hashes):**

- An associative array is a list of paired elements.
- It is defined as follows:

**Ex:**

```
%parrot = ("has","wings","can","fly","isa","bird");
```

- The associative array name begins with a `%`
- The elements like `"has"` and `"wings"`, `"can"` and `"fly"`, `"isa"` and `"bird"` constitute paired elements.
- These pairs are ordered pairs where the first element is referred to as the key and the second element is referred to as the value of the key element.
- Thus an associative array is made up of a certain number of key-value pairs.
- Values (data) in an associative array are accessed using keys.
- For example, the value `"fly"` is obtained using the key `"can"` along with the associative array name as a scalar by using a statement like: **`$parrot{"can"}`**
- The use of curly braces is required to access a required value in an associative array.
- The character `$` is used as all independent key values are always scalars.

An example below shows in which all the working days (assuming, 5 working days/week) are stored in an associative array with their short names as keys and long names as values.

```
%work_days = ("mon", "monday", "tue", "tuesday", "wed", "wednesday", "thu",
               "thursday", "fri", "friday");
```

When a statement of the type given above is executed, an associative array called %work\_days is created. Now assume that Saturday is also made a working day. This necessitates the addition of new pair of element—"sat", "saturday" to %work\_days. This is accomplished by using the statement

```
$work_days{"sat"} = "saturday";
```

A key-value pair can be removed by using the delete( ) function. For example the pair "fri" and "friday" can be removed by writing a delete statement as shown below.

```
delete $work_days{"fri"};
```

The use of the character \$ in the above two examples should be noted carefully.

### **keys and values Function:**

- Keys and values are two useful functions that are used with associative arrays,
- The function keys returns all the keys in the form of a list, which is normally stored in an array for any further usage,
- The function values returns all the values in the form of a list, which also is normally stored in an array and used.
- The list of keys or values that are obtained will not be in any order.
- If required, they can be obtained in a sorted order by using the sort function.

```
$cat work_days.pl
#!/usr/bin/perl
%work_days = ("mon", "monday", "tue", "tuesday", "wed", "wednesday", "thu",
               "thursday", "fri", "friday");
foreach $short_name (@ARGV)
{
    print("The short name $short_name stands for $work_days{$short_name} \n");
}
$work_days{"sat"} = "saturday";

print("\n");
foreach $short_name (@ARGV)
{
    print("The short name $short_name stands for $work_days{$short_name} \n");
}
print("\n");
@short_names_list = keys(%work_days);
print("The short names are @short_names_list \n");
@long_names_list = values(%work_days);
print("The long names are @long_names_list \n");
$
```



Upon execution of the above code segment, the following output will be obtained.

```
$perl work_days.pl mon tue sat
The short name mon stands for monday
The short name tue stands for tuesday
The short name sat stands for
The short name mon stands for monday
The short name tue stands for tuesday
The short name sat stands for saturday
The short names are sat fri thu wed tue mon
The long names are saturday friday thursday wednesday tuesday monday
$
```

## **Decision Making and Loop Control Structures:**

### **Decision Making:**

#### **1) The if Control Statement**

In Perl, the if statement is the core conditional statement.

This construct can take any of the following three general formats: that is simple if format, the if-else format or the if-elsif format.

```
1. if(expr)
{
    True_Block
}
```

```
2 if(expr)
{
    True_Block
}
else
{
    False_Block
}
3 if(expr)
{
    True_Block
}
elseif
{
    True_Block
}
-----
else
{
    False_Block
}
```

The presence of either the else or elseif portions is optional.

Whenever an **expr** is evaluated to be true, the block that appears immediately next to it is executed.

Whenever an **expr** is evaluated to be false, the block that appears immediately next to it is not executed, and the block afterwards, if present, will be executed next.

***The unless Control Statement:***

This control structure has the same general format as that of the if control statement(with the keyword if replaced by the keyword unless). However, unless works exactly the opposite way to that of the if.

**Loop Control:****1) The while Control Structure:**

The general format of this control structure is as follows:

```
while(test)
{
    block of statements
}
```

When the control comes across this structure, the test expression is evaluated first. The block of statements that follows this test expression is repeatedly evaluated as long as the test remains true. The control goes out of the scope of the while loop as soon as the test becomes false. Thus, this is an entry controlled loop structure.

**2) The for Control Structure:**

The general format of the for is shown below:

```
for(index; test; increment )
{
    -----;
    -----;
}
```

Where

- i. The index is initialized to a suitable start or initial value
- ii. The test tests whether the index value is within the limits or not, if it is within the limits the body of the statements is executed, if not the control goes out of the for loop and
- iii. Each time the body is executed, the index is incremented by an increment value and the entire process restarts.

**3) The foreach Control Structure:**

This control structure is also used to execute a set of statements repeatedly.

Ex:

```
foreach $num(1,3,5)
{
    print "The number is $num\n"
}
```

The number is 1

The number is 3

The number is 5

**Controlling Loops:**

Depending on certain conditions, it is necessary to stop looping , to stop executing or somehow to control the execution of the loop itself.

In Perl, such situations are managed using last, next and redo constructs.

**The last Construct:** This keyword or construct stops the looping immediately (like break in C). Execution continues from the statement that appears immediately after the current loop.

**The next Construct:** This keyword or construct stops the execution of the current iteration of the loop, goes back to the top, and starts the next iteration with the test. It is like continue in C.

**The redo Construct:** This keyword or construct stops the execution of the current iteration, goes back to the top, and starts the re-execution of the same iteration (without testing or incrementing anything).

**Regular Expressions:**

- Perl is mainly used to extract required records by providing certain patterns.
- In Perl, pattern matching is done by writing a pattern, that is , a regular expression, within a pair of forward slashes: **/PATTERN/**
- When nothing is mentioned explicitly the searching takes place on the contents of the **\$\_** special variable.

```
$_ = "Krishnamurthy Ramamurthy Venkateshmurthy";
```

```
if(/murthy/)
```

```
{
```

```
    print "Pattern Exists\n";
```

```
}
```

- The expression **/murthy/** looks for the six-letter string **murthy** in the default variable **\$\_**
- If the pattern is found then a true value is returned, else a false value will be returned.
- Perl also has certain metacharacters, using which flexible and powerful search patterns (regular expressions) can be built.



<i>Metacharacter</i>	<i>Meaning</i>	<i>Example</i>
<b>.</b> (dot)	Matches exactly any one character in that position.	/murth./ matches with patterns like murthy, murthi, etc.
<b>*</b> (asterisk)	Matches zero or any number of times the character in the preceding position.	/ab*c/ matches with patterns like ac, abc, abbc and so on.
<b>+</b> (plus)	Matches one or any number of times the character in the preceding position.	/ab+c/ matches with patterns like abc, abbc and so on. (observe that ac is missing)
<b>?</b> (question mark)	Matches zero or one time the character in the preceding position.	/ab?c/ matches with ac or abc. (just these two only )

### **Character Classes and their Short Cuts:**

- Regular expressions can include character classes.
- The character class is a list of possible characters taken inside a pair of square brackets ([ ]).
- Such character classes are used to match any one character from within the class. For example, the character class **[uvwxyz]** may match any one of these six characters.
- For convenience, characters inside a class may be represented as a range, using the hyphen (-) character. Thus, the character class **[uvwxyz]** is equivalent to **[u-z]**, the character class of all decimal digits may be written as **[0-9]** and so on.
- In Perl there are shortcuts using which, certain classes can be represented. For Example , the character class of any digit [0-9] is abbreviated as **\d**.
- The character class made up of all the English letters (both uppercase and lowercase), digits and underscore is abbreviated as **\w**. This is equivalent to **[a-zA-Z0-9\_]**.
- The character class made up of all the white space characters is abbreviated as **\s**, which is equivalent to **[\t\n\f\r]**
- The shortcuts **\S**, **\W** and **\D** work just as opposite way to that of **\s**, **\w** and **\d**, respectively.
- The character class is negated by using the caret character (^) as the very first character inside it. For example, the regular expression **[^0-9]** matches with all the characters except the decimal digits.

### **The Match Operator (m/ /)**

- This operator is used to check whether a variable contains the specified data (the search-pattern) or not.
- The use of **m** along with forward slashes is optional.
- Pattern searching can also be made on any string by binding the search pattern to that string. For this the binding operator (**=~**) is used as shown in the statement below:

**`$variable =~ m/search_pattern/;`**

- When the above statement is executed, the search pattern on the right-hand side travels through the contents of the scalar **\$variable** and returns a true value if the pattern is found.
- If the search pattern itself contains a lot of forward slashes one can use other delimiters. For example, one could use **m#search\_pattern#**. Here the use of **m** is mandatory.
- It is possible to modify the default search behavior, using single-character modifiers such as **g**, **i** and **o**.
- The modifier **g** is used for global searching.
- The modifier **i** is used for ignoring the case
- The modifier **o** is used to search the pattern only once.

### **The Substitute Operator:**

- The substitute operator searches for a pattern and replaces and substitutes it with the replacement string.
- The general format of this operator is

**`s/search_pattern/replacement_pattern/;`**

### **Multiple Search Patterns:**

- In Perl, it is possible to search for a pattern among two or more alternate search patterns.
- For example, the search pattern **/sachin/david/kaif/** matches any string that contains the subpatterns **sachin, david and kaif**.
- The pipe character (**|**) behaves like a logical or and hence helps in selecting one of the many alternate patterns mentioned inside the search pattern.

### **Anchors:**

- In Perl, it is possible to look for a required pattern at specific positions in the string, that is at the beginning of the string, end of the string, the beginning of words in the string, end of the words in the string and so on, using certain metacharacters.
- A metacharacter that is used to fix the position of search in a string is known as an **anchor**.
- The two most popular anchors are metacharacters **^** (caret) and the **\$** (dollar).

- The anchor character `^` (caret) is used for searching a pattern in the beginning of the string.
- The anchor character `$` (dollar) is used for searching a pattern at the end of the string.

**Ex:**

Given the two strings “**Asoka the great**” and “**The great Asoka**”, the pattern `/^Asoka/` matches with the former string whereas the pattern `/Asoka$/` matches with the latter string.

### Word Anchors:

- The word anchors work with the group of `\w` class characters.
- The most popular word anchor is the word –boundary anchor `\b`. By using this anchor, only whole words are matched.
- For example, the pattern `/bAsoka\b/` matches not only both the strings given above but also matches a string like “**King Asoka and the Buddhism**”.
- Another interesting word anchor is the nonword-boundary anchor `\b`.
- The pattern `/bsearch\b/` matches with the words such as searching, searchers, searched and so on. However it does not match with words like research and search.

### Sub-Routines:

- In Perl, sub-routines are user defined functions.

**Ex:**

```
sub triangle_area
{
    $base = ...
    $height = ...
    return 1/2*base*height;
}
```

`triangle_area` is the name of the sub-routine and the return statement would be returning the computed result back to the calling program.

A sub-routine is used or called from within a Perl script by using its name along with the actual arguments, if any.

**For example:**

```
$area = &triangle_area($b, $h);
```

- It is not mandatory to use a return statement in the body of a sub-routine.
- Whenever a return statement is not explicitly used, the result of the last performed calculation is returned automatically.
- The arguments are passed via a special array variable called `@_`