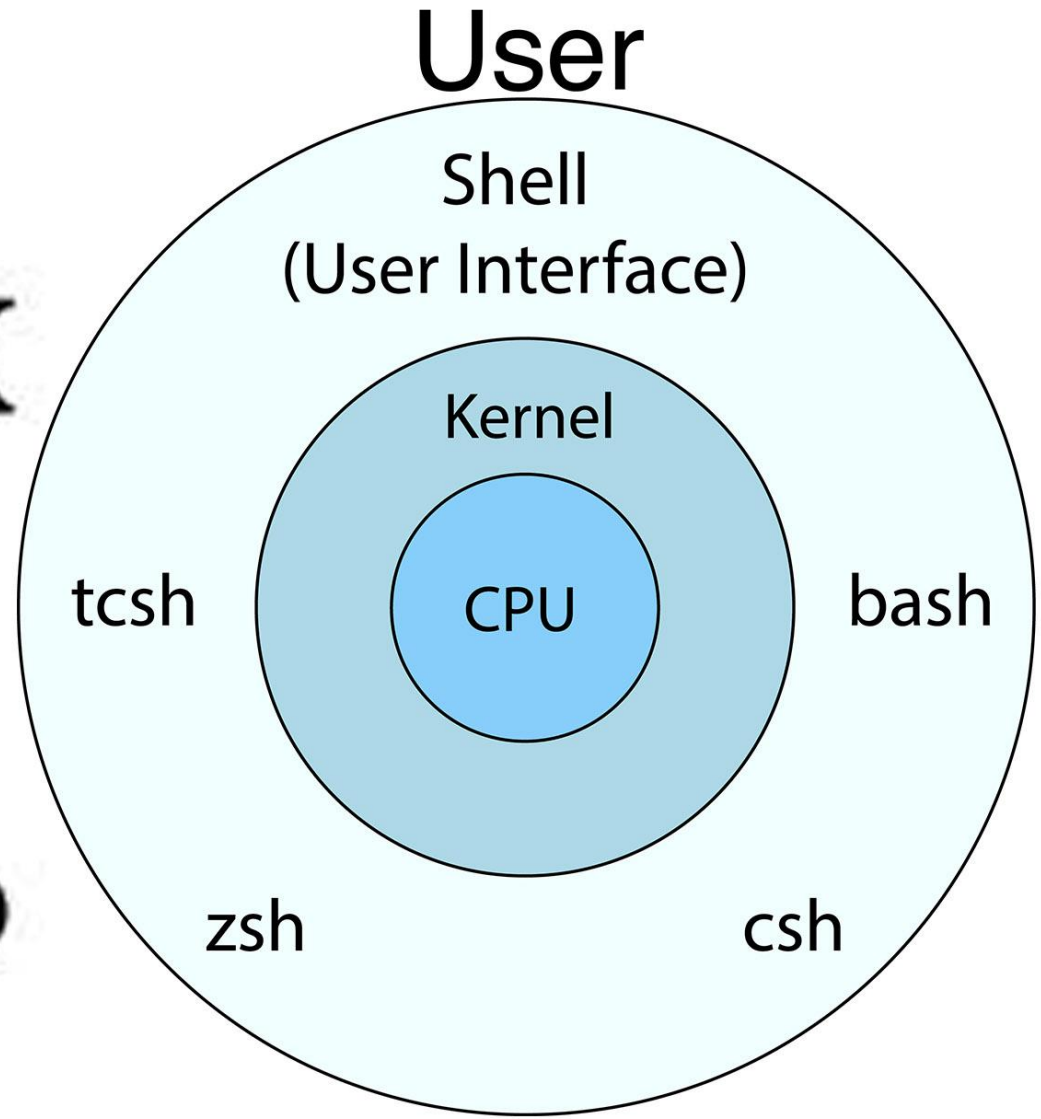
Shell Programming

Linux

UNIX®



Programming or Scripting

- **Linux command line is provided by a program called the shell.**
- **Shell scripting is an important part of process automation in Linux.**
-
- **Scripting helps you write a sequence of commands in a file and then execute them.**
- **bash is not only an excellent command line shell, but a scripting language in itself.**
- **Shell scripting allows us to use the shell's abilities and to automate a lot of tasks that would otherwise require a lot of commands.**

Programming or Scripting

(Contd…)

- **Difference between programming and scripting languages:**
  - **Programming languages are generally a lot more powerful and a lot faster than scripting languages. Programming languages generally start from source code and are compiled into an executable. This executable is not easily ported into different operating systems.**
  - **A scripting language also starts from source code, but is not compiled into an executable. Rather, an interpreter reads the instructions in the source file and executes each instruction. Interpreted programs are generally slower than compiled programs. The main advantage is that you can easily port the source file to any operating system. bash is a scripting language. Other examples of scripting languages are perl, lisp, and tcl.**

Example of a bash program

- **We must know how to use a text editor. There are two major text editors in Linux:**
  - **`vi, emacs (or xemacs)`**
- **So fire up a text editor; for example:**
  - **`bash$ vi`**

  and type the following inside it:
  - ```
    #!/bin/bash
    echo "Hello World"
    ```
- **The first line tells Linux to use the bash interpreter to run this script. We call it hello.sh. Then, make the script executable:**
  - **`bash$ chmod +x hello.sh`**
  - **`bash$ ./hello.sh`**

  `Hello World`
- **The source command**
    ```

Variables

- **We can use variables as in any programming languages. Their values are always stored as strings, but there are mathematical operators in the shell language that will convert variables to numbers for calculations.**
- **We have no need to declare a variable, just assigning a value to its reference will create it.**
- **Example:**
  - ```
    #!/bin/bash

    STR="Hello World!"

    echo $STR
    ```
- **Line 2 creates a variable called STR and assigns the string "Hello World!" to it. Then the value of this variable is retrieved by putting the '$' in at the beginning.**
    ```

Warnings!!!

- **The shell programming language does not type-cast its variables. This means that a variable can hold number data or character data.**
  - **count=0**
  - **count=Sunday**
- **Switching the TYPE of a variable can lead to confusion for the writer of the script or someone trying to modify it, so it is recommended to use a variable for only a single TYPE of data in a script.**
- **\ is the bash escape character and it preserves the literal value of the next character that follows.**
  - `bash$:  ls \*`

    `ls: *: No such file or directory`

The read command

- **The read command allows you to prompt for input and store it in a variable**

Arithmetic Evaluation

- **The let statement can be used to do mathematical functions:**
  - `bash$: let X=10+2*7`

    `bash$: echo $X`

    `24`

    `bash$: let Y=X+2*4`

    `bash$: echo $Y`

    `32`
- **An arithmetic expression can be evaluated by $[expression] or $((expression))**
  - `bash$: echo "$((123+20))"`

    `143`
  - `bash$: VALORE=$[123+20]`
  - `bash$: echo "$[123*$VALORE]"`

    `17589`

Arithmetic Evaluation

(Contd…)

- **Available operators:** **+, -, /, *, %**
- **Example:**

```
#! /bin/bash
echo -n “Enter the first number: ” ; read x
echo -n “Enter the second number: ”; read y
add=$(($x + $y))
sub=$(($x - $y))
mul=$(($x * $y))
div=$(($x / $y))
mod=$(($x % $y))
# print out the answers:
echo “Sum: $add”
echo “Difference: $sub”
echo “Product: $mul”
echo “Quotient: $div”
echo “Remainder: $mod”
```

Conditional Statements

- **Conditionals let us decide whether to perform an action or not. This decision is taken by evaluating an expression. The most basic form is:**
  - **if [*expression*];**

    then

    statements

    elif [*expression*];

     then

       statements

    else

       statements

    fi
  - **the elif (else if) and else sections are optional**

Simple IF

```
if [ expression ]
then
   statement
fi
```

Simple IF-ELSE

```
if [ expression ]
then
   statement1
else
   statement2
fi
```

Else If ladder

```
if [ expression1 ]
then
   statement1
   statement2
   .
   .
elif [ expression2 ]
then
   statement3
   statement4
   .
   .
else
   statement5
fi
```

Nested if

```
if [ expression1 ]
then
   statement1
   statement2
   .
else
   if [ expression2 ]
   then
      statement3
      .
   fi
fi
```

```
case  in
   Pattern 1) Statement 1;;
   Pattern n) Statement n;;
esac
```

```
#Initializing two variables
a=10
b=20

#Check whether they are equal
if [ $a == $b ]
then
        echo "a is equal to b"
fi

#Check whether they are not equal
if [ $a != $b ]
then
        echo "a is not equal to b"
fi
```

```
#!/bin/bash
echo -n "Enter a number: "
read number
if [ $number -eq 0 ]
then
echo "You entered zero. Zero is an even number."
elif [ $(($number % 2)) -eq 0 ]
then
echo "You entered $number. It is an even number."
else
echo "You entered $number. It is an odd number."
fi
```

```
#Initializing two variables
a=20
b=20

if [ $a == $b ]
then
        #If they are equal then print this
        echo "a is equal to b"
else
        #else print this
        echo "a is not equal to b"
fi
```

```
CARS="bmw"

#Pass the variable in string
case "$CARS" in
        #case 1
        "mercedes") echo "Headquarters - Affalterbach, Germany" ;;

        #case 2
        "audi") echo "Headquarters - Ingolstadt, Germany" ;;

        #case 3
        "bmw") echo "Headquarters - Chennai, Tamil Nadu, India" ;;
esac
```

Looping Statements

- while statement
- for statement
- until statement

To alter the flow of loop statements, two commands are used they are,

- break
- continue

```
while <condition>
do

<command 1>
<command 2>
<etc>
done
```

```
for <var> in <value1 value2 ... valuen>
do
<command 1>
<command 2>
<etc>
done
```

```
until <condition>
do

<command 1>
<command 2>
<etc>
done
```

```
a=0
# -lt is less than operator

#Iterate the loop until a less than 10
while [ $a -lt 10 ]
do
        # Print the values
        echo $a

        # increment the value
        a=`expr $a + 1`
done
```

```
#Start of for loop
for a in 1 2 3 4 5 6 7 8 9 10
do
        # if a is equal to 5 break the loop
        if [ $a == 5 ]
        then
                break
        fi
        # Print the value
        echo "Iteration no $a"
done
```

```
for a in 1 2 3 4 5 6 7 8 9 10
do
        # if a = 5 then continue the loop and
        # don't move to line 8
        if [ $a == 5 ]
        then
                continue
        fi
        echo "Iteration no $a"
done
```

```
a=0
# -gt is greater than operator

#Iterate the loop until a is greater than 10
until [ $a -gt 10 ]
do
        # Print the values
        echo $a

        # increment the value
        a=`expr $a + 1`
done
```

```
COLORS="red green blue"

# the for loop continues until it reads all the values from the COLORS

for COLOR in $COLORS
do
        echo "COLOR: $COLOR"
done
```

The *test* command

- **This command is provided to specify the control statement or condition**

- **It can perform several types of tests like numeric test, string test and file test**

Expressions

- **An expression can be: String comparison, Numeric comparison, File operators and Logical operators**
- **String Comparisons:**
  - **=** compare if two strings are equal
  - **!=** compare if two strings are not equal
  - **-n** evaluate if string length is greater than zero
  - **-z** evaluate if string length is equal to zero

- **Examples:**
  - **[s1 = s2]** (true if s1 same as s2, else false)
  - **[s1 != s2]** (true if s1 not same as s2, else false)
  - **[s1]** (true if s1 is not empty, else false)
  - **[-n s1]** (true if s1 has a length greater then 0, else false)
  - **[-z s2]** (true if s2 has a length of 0, otherwise false)

Expressions

(Contd…)

- **Number Comparisons:**
  - **-eq** compare if two numbers are equal
  - **-ge** compare if one number is greater than or equal to a number
  - **-le** compare if one number is less than or equal to a number
  - **-ne** compare if two numbers are not equal
  - **-gt** compare if one number is greater than another number
  - **-lt** compare if one number is less than another number

- **Examples:**
  - **[n1 -eq n2]** (true if n1 same as n2, else false)
  - **[n1 -ge n2]** (true if n1greater then or equal to n2, else false)
  - **[n1 -le n2]** (true if n1 less then or equal to n2, else false)
  - **[n1 -ne n2]** (true if n1 is not same as n2, else false)
  - **[n1 -gt n2]** (true if n1 greater then n2, else false)
  - **[n1 -lt n2]** (true if n1 less then n2, else false)

Example of test command

```
clear
echo
echo -n "Enter a number: "
read num
if test $num  -eq  0
then
       echo "The number entered by you is zero"
elif test $num  -lt 0
then
       echo "The number entered by you is negative"
else
       echo "The number entered by you is positive"
fi
```

Without the *test* command

- **Instead of specifying *test* command explicitly whenever we want to check for condition, the condition can be enclosed in square brackets**
- **Example is given on the next slide ➔ ➔**

Example (without using test)

```
clear
echo
echo -n "Enter a number: "
read num
if [ num -eq 0 ]
then
        echo "The number entered by you is zero"
elif [ num -lt 0 ]
then
        echo "The number entered by you is negative"
else
        echo "The number entered by you is positive"
fi
```

Some more examples

```
#!/bin/bash
clear
echo
echo -n "Enter two names: "
read name1
read name2
if [ $name1 = $name2 ]
then
        echo "The names entered by you are the same"
else
        echo "The names are different"
fi
```

Expressions

(Contd…)

- **Files operators:**
  - **-d** check if path given is a directory
  - **-f** check if path given is a file
  - **-e** check if file name exists
  - **-r** check if read permission is set for file or directory
  - **-s** check if a file has a length greater than 0
  - **-w** check if write permission is set for a file or directory
  - **-x** check if execute permission is set for a file or directory

Expressions

(Contd…)

- **Examples:**
  - **[-d fname]** (true if fname is a directory, otherwise false)
  - **[-f fname]** (true if fname is a file, otherwise false)
  - **[-e fname]** (true if fname exists, otherwise false)
  - **[-s fname]** (true if fname length is greater then 0, else false)
  - **[-r fname]** (true if fname has the read permission, else false)
  - **[-w fname]** (true if fname has the write permission, else false)
  - **[-x fname]** (true if fname has the execute permission, else false)

```
read -p 'Enter a : ' a
read -p 'Enter b : ' b

if(($a == "true" && $b == "true" ))
then
        echo Both are true.
else
        echo Both are not true.
fi

if(($a == "true" || $b == "true" ))
then
        echo Atleast one of them is true.
else
        echo None of them is true.
fi

if(( ! $a == "true" ))
then
        echo "a" was initially false.
else
        echo "a" was initially true.
fi
```

```
#!/bin/bash

#reading data from the user
read -p 'Enter a : ' a
read -p 'Enter b : ' b

bitwiseAND=$(( a&b ))
echo Bitwise AND of a and b is $bitwiseAND

bitwiseOR=$(( a|b ))
echo Bitwise OR of a and b is $bitwiseOR

bitwiseXOR=$(( a^b ))
echo Bitwise XOR of a and b is $bitwiseXOR

bitiwiseComplement=$(( ~a ))
echo Bitwise Compliment of a is $bitiwiseComplement

leftshift=$(( a<<1 ))
echo Left Shift of a is $leftshift

rightshift=$(( b>>1 ))
echo Right Shift of b is $rightshift
```

Expressions

(Contd…)

Logical operators:

- **!** **negate (NOT) a logical expression**
- **-a** **logically AND two logical expressions**
- **-o** **logically OR two logical expressions**

Example

```
#!/bin/bash
echo -n "Enter a number 1 < x < 10:"
read num
if [ "$num" -gt 1 -a "$num" -lt 10 ];
then
    echo "$num*$num=$(($num*$num))"
else
    echo "Wrong insertion !"
fi
```

Expressions

(Contd…)

Logical operators:

- && logically **AND** two logical expressions
- || logically **OR** two logical expressions

Expressions

(Contd…)

```
#!/bin/bash
echo -n "Enter a number 1 < x < 10: "
read num
if [ “$num” -gt 1 ] && [ “$num” -lt 10 ];
then
    echo “$num*$num=$(($num*$num))”
else
    echo “Wrong insertion !”
fi
```

Iteration Statements

- **The for structure is used when you are looping through a range of variables.**
-

```
for var in list
do
   statements
done
```

```
while <condition>
do

<command 1>
<command 2>
<etc>
done
```

```
for <var> in <value1 value2 ... valuen>
do
<command 1>
<command 2>
<etc>
done
```

```
until <condition>
do

<command 1>
<command 2>
<etc>
done
```

```
#Start of for loop
for a in 1 2 3 4 5 6 7 8 9 10
do
        # if a is equal to 5 break the loop
        if [ $a == 5 ]
        then
                break
        fi
        # Print the value
        echo "Iteration no $a"
done
```

```
for a in 1 2 3 4 5 6 7 8 9 10
do
        # if a = 5 then continue the loop and
        # don't move to line 8
        if [ $a == 5 ]
        then
                continue
        fi
        echo "Iteration no $a"
done
```

```
a=0
# -lt is less than operator

#Iterate the loop until a less than 10
while [ $a -lt 10 ]
do
        # Print the values
        echo $a

        # increment the value
        a=`expr $a + 1`
done
```

```
a=0
# -gt is greater than operator

#Iterate the loop until a is greater than 10
until [ $a -gt 10 ]
do
        # Print the values
        echo $a

        # increment the value
        a=`expr $a + 1`
done
```

```
COLORS="red green blue"

# the for loop continues until it reads all the values from the COLORS

for COLOR in $COLORS
do
        echo "COLOR: $COLOR"
done
```

Examples

- statements are executed with **var** set to each value in the **list**

```
#!/bin/bash
let sum=0
for num in 1 2 3 4 5
do
  let "sum = $sum + $num"
done
echo $sum
```

Some more examples

```
#!/bin/bash
for x in paper pencil pen; do
  echo "The value of variable x is: $x"
  sleep 1
done
```

Some more examples

(Contd…)

If the `list` part is left off, `var` is set to each parameter passed to the script ($1, $2, $3,…)

```
#!/bin/bash
for x
do
  echo "The value of variable x is: $x"
  sleep 1
done
```

OUTPUT:

```
./file.sh alpha beta
The value of variable x is: alpha
The value of variable x is: beta
```

Functions

- **Functions make scripts easier to maintain. Basically it breaks up the program into smaller pieces. A function performs an action defined by you, and it can return a value if you wish.**

```
#!/bin/bash
hello()
{
echo "You are in function hello()"
}

echo "Calling function hello()…"
hello
echo "You are now out of function hello()"
```

What does this script do?

(Contd…)

```
#!/bin/bash
function check() {
if [ -e "$PWD/$1" ]
then
  return 0
else
  return 1
fi
}

echo "Enter the name of the file: " ; read x
if check $x
then
  echo "$x exists!"
else
  echo "$x does not exists!"
fi
```

Reading from a file

```
clear
echo
echo -n "Enter the name of a file: "
read  file_name
exec  <  $file_name
while read line
do
        echo $line
done
echo
```

Shell Keywords

echo	**read**	**set**	**unset**
readonly	**shift**	**export**	**if**
else	**fi**	**while**	**do**
done	**for**	**until**	**case**
esac	**break**	**continue**	**exit**
return	**tap**	**wait**	**eval**
exec	**ulimit**	**umask**	

Arrays- Declaration

ARRAY=(apple banana cherry date fig)

ARRAY=()

ARRAY=([0]=apple [2]=cherry [4]=fig)

ARRAY[0]=apple

ARRAY[1]=banana

ARRAY[2]=cherry

ARRAY[3]=date

ARRAY[4]=fig

ARRAYS IN SHELL PROGRAMMING

Declaring and initializing arrays

```
# Declare an array
declare -a fruits

# Initialize an array
fruits=("apple" "banana" "orange")

# Alternative initialization
fruits=(
  "apple"
  "banana"
  "orange"
)
```

Accessing array elements

```
echo ${fruits[0]}  # Output: apple
echo ${fruits[1]}  # Output: banana
echo ${fruits[@]}  # Output: apple banana orange
echo ${#fruits[@]}  # Output: 3 (number of elements)
```

Adding elements to an array

```
fruits+=("grape")
fruits[4]="mango"
```

Iterating over array elements

```
for fruit in "${fruits[@]}"; do
  echo "I like $fruit"
done
```

Slicing arrays

```
echo ${fruits[@]:1:2}  # Output: banana orange
```

Associative arrays

```
declare -A capital_cities
capital_cities[France]="Paris"
capital_cities[Germany]="Berlin"
capital_cities[Italy]="Rome"

echo ${capital_cities[France]}  # Output: Paris
```

Array operations

```
# Copy an array
new_fruits=("${fruits[@]}")

# Remove an element (3rd element)
unset fruits[2]

# Clear the entire array
unset fruits
```

Using arrays with command substitution

```
files=($(ls *.txt))
for file in "${files[@]}"; do
  echo "Processing $file"
done
```

Array of numbers and basic arithmetic

```
numbers=(1 2 3 4 5)
sum=0
for num in "${numbers[@]}"; do
  sum=$((sum + num))
done
echo "Sum: $sum"  # Output: Sum: 15
```

STRING MANIPULATIONS IN SHELL

String Length

- To get the length of a string, you can use the ${#variable} syntax.

```
string="Hello, World!"
echo ${#string}  # Output: 13
```

String Concatenation

- concatenate strings by simply placing them next to each other or using the += operator.

```
str1="Hello"
str2="World"
result="$str1 $str2"
echo $result  # Output: Hello World

str1+=" there"
echo $str1  # Output: Hello there
```

Substring Extraction

- To extract a portion of a string, use ${string:start_position:length}.

```
string="Hello, World!"
echo ${string:7:5}  # Output: World
echo ${string:7}    # Output: World! (omitting length extracts to the end)
```

String Replacement

- To replace parts of a string, use ${string/pattern/replacement}.

```
string="The quick brown fox"
echo ${string/quick/slow}  # Output: The slow brown fox

# Replace all occurrences
echo ${string//o/0}  # Output: The quick br0wn f0x

# Replace at the beginning
echo ${string/#The/A}  # Output: A quick brown fox

# Replace at the end
echo ${string/%fox/dog}  # Output: The quick brown dog
```

String Trimming

- To remove characters from the beginning or end of a string

```
string="   Hello, World!   “

echo "${string##*( )}"
# Output: Hello, World!   (left trim)

echo "${string%%*( )}"
 # Output:    Hello, World! (right trim)

echo "${string##*( )}" | sed 's/[[:space:]]*$//'
# Output: Hello, World! (trim both sides)
```

Converting Case

- Bash doesn't have built-in functions for case conversion, but you can use tr:

```
string="Hello, World!“

echo "$string" | tr '[:lower:]' '[:upper:]'
# Output: HELLO, WORLD!

echo "$string" | tr '[:upper:]' '[:lower:]'
 # Output: hello, world!
```

String Comparison

- Compare strings using operators like = (equal) and != (not equal) in conditional statements.

```
str1="hello"
str2="world"

if [ "$str1" = "$str2" ]; then
    echo "Strings are equal"
else
    echo "Strings are not equal"
fi
# Output: Strings are not equal
```

String Contains

- To check if a string contains a substring.

```
string="Hello, World!"
if [[ $string == *"World"* ]]; then
    echo "String contains 'World'"
fi
# Output: String contains 'World'
```

String Splitting

- can split a string into an array using IFS (Internal Field Separator):

```
string="apple,banana,cherry"
IFS=',' read -ra ADDR <<< "$string"
for i in "${ADDR[@]}"; do
    echo "$i"
done
# Output:
# apple
# banana
# cherry
```

Advanced Pattern Matching and Replacement

```
#!/bin/bash

# Sample text
text="The quick brown fox jumps over the lazy dog. The FOX is quick!"

# Replace 'quick' with 'slow', case-insensitive, all occurrences
echo "${text//[Qq][Uu][Ii][Cc][Kk]/slow}"

# Replace the first word that starts with 'f' or 'F' and ends with 'x' or 'X'
echo "${text/[fF][a-zA-Z]*[xX]/cat}"

# Remove all words of 3 letters or less
echo "$text" | sed 's/\b[a-zA-Z]\{1,3\}\b//g'

# Output:
# The slow brown fox jumps over the lazy dog. The FOX is slow!
# The quick brown cat jumps over the lazy dog. The FOX is quick!
# quick brown jumps over lazy dog. FOX quick!
```

Date and Time Manipulation

```
#!/bin/bash

# Current date and time
now=$(date +"%Y-%m-%d %H:%M:%S")
echo "Current date and time: $now"

# Date 30 days from now
future_date=$(date -d "+30 days" +"%Y-%m-%d")
echo "Date 30 days from now: $future_date"

# Convert timestamp to human-readable format
timestamp=1609459200
readable_date=$(date -d @$timestamp +"%Y-%m-%d %H:%M:%S")
echo "Converted timestamp: $readable_date"

# Calculate time difference
start_date="2023-01-01 00:00:00"
end_date="2023-12-31 23:59:59"
difference=$(( $(date -d "$end_date" +%s) - $(date -d "$start_date" +%s) ))
echo "Time difference: $((difference / 86400)) days"

# Output:
# Current date and time: 2023-05-15 10:30:45
# Date 30 days from now: 2023-06-14
# Converted timestamp: 2021-01-01 00:00:00
```

Pattern matching in shell programming

- Basic Wildcards

* Matches any sequence of characters (except leading .)

? Matches any single character

[abc] Matches any one character listed in brackets

[!abc] or [^abc] Matches any one character NOT listed in brackets

[a-z] Matches any one character in the given range

Examples

```
# List all .txt files
ls *.txt

# List files with single character before .txt
ls ?.txt

# List files starting with either a, b, or c
ls [abc]*

# List files NOT starting with vowels
ls [!aeiou]*

# Match files with numbers
ls [0-9]*
```

Basic Function Syntax

```
# Method 1
function_name() {
   commands
}

# Method 2
function function_name {
   commands
}

# Example of a simple function
hello() {
   echo "Hello, World!"
}
```

Functions with Parameters

```
# $1, $2, etc. represent positional parameters

greet() {
    echo "Hello, $1!"
    echo "Age: $2"
}

# Call: greet "John" "25"

# $# gives number of parameters
# $* or $@ gives all parameters

check_params() {
    echo "Number of parameters: $#"
    echo "All parameters: $*"
    echo "Parameter list: $@"
}
```

Return Values

```
# Using return (0-255 only)

is_number() {
    if [[ $1 =~ ^[0-9]+$ ]]; then
        return 0    # Success
    else
        return 1    # Failure
    fi
}

# Using echo for string return
get_date() {
    echo $(date +%Y-%m-%d)
}
# Usage: today=$(get_date)
```

Local Variables

```
calculate() {
    local result    # Local variable
    result=$(($1 + $2))
    echo $result
}

# Variable scope example
demo_scope() {
    local local_var="I'm local"
    global_var="I'm global"
    echo "Inside function: $local_var"
}
```

Obtaining the Entire Array's Values or Its Size

Notation	Meaning	Result
${#ARRAY[@]} ${#ARRAY[*]}	Return the number of elements in the array (NULL elements are not counted).	3
${ARRAY[@]} ${ARRAY[*]}	Return all elements of the array as a list.	`apple cherry fig`
"${ARRAY[@]}"	Return all elements of the array as a list where each element is individually quoted.	`"apple" "cherry" "fig"`
"${ARRAY[*]}"	Return all elements as a single quoted string.	`"apple cherry fig"`

Arrays- Declaration

```
for item in ${ARRAY[@]}
do
echo $item
done
```

```
for (( i=0; i<${#ARRAY[@]}; i++ ))
do
echo ${ARRAY[i]}
done
```

Accessing Array Elements

```
myArray[element]="Hey Ninjas"
echo ${myArray[element]}
```

```
Hey Ninjas
```

```
myArray=(1 2 3 4 5)
echo ${myArray[2]}
```

```
3
```

Reading Array Elements

[@] symbol is used to print all the elements at the same time.

```
myArray=(1 2 3 4 5 6 7 8)

for i in ${myArray[@]}
do
    echo $i
done
```

```
echo ${arrrayName[whichElement]:startingIndex}
```

```
myArray=(I really love Coding Ninjas)
echo ${myArray[@]:0}
echo ${myArray[@]:1}
echo ${myArray[@]:2}
echo ${myArray[0]:1}
```

Output:

```
I really love Coding Ninjas
really love Coding Ninjas
love Coding Ninjas
```

```
echo ${arrayName[whichElement]:startingIndex:countElement}
```

```
myArray=(I really love Coding Ninjas)
echo ${myArray[@]:1:4}
echo ${myArray[@]:2:3}
echo ${myArray[@]:3:4}
echo ${myArray[0]:1:3}
```

Output:

```
really love Coding Ninjas
love Coding Ninjas
Coding Ninjas
```

Counting the Number of Elements

```
myArray=(1 2 3 4 5)
echo ${#myArray[@]}
```

Output:

5

Delete a Single Array Element

```
myArray=(1 2 3 4 5 6 7 8)
printf "Array before delete:\n"
for i in ${myArray[@]}
do
   echo $i
done
unset myArray[2]
printf "Array after delete:\n"
for i in ${myArray[@]}
do
   echo $i
done
```

Array before delete:
1
2
3
4
5
6
7
8
Array after delete:
1
2
4
5
6
7
8

Search and Replace Array Element

echo ${arrayName[@]//character/replacement}

myArr=(My fav learning website is Coding Ninjas)
echo ${myArr[@]//fav/favourite} #changing a complete element
echo ${myArr[@]//d/D} #changing a character in an element

Output:

```
My favourite learning website is Coding Ninjas
My fav learning website is CoDing Ninjas
```

String Manipulation in Shell Scripting

Syntax:

VariableName='value'

echo $VariableName

VariableName="value"

echo ${VariableName}

VariableName=value

echo "$VariableName"

```
str1='welcome'
str2="to"
str3=GeeksForGeeks
echo $str1
echo ${str2}
echo "$str3"
```

Read Only variable declaration

$ declare -r VAR1='Hello world'

print length of string inside Bash Shell

variableName=value

echo ${#variablename}

Eg:

Str=GodIsGreat

Echo ${#str}

10

Concatenate strings inside Bash Shell using variables

```
var=${var1}${var2}${var3}
```

Or

```
var=$var1$var2$var3
```

Or

```
var="$var1""$var2""$var3"
```

Following will insert "**" between the strings

```
var=${var1}**${var2}**${var3}
```

or

```
var=$var1**$var2**$var3
```

or

```
var="$var1"**"$var2"**"$var3"
```

Following concatenate the strings using space:

```
var=${var1} ${var2} ${var3}
```

or

```
var="$var1" "$var2" "$var3"
```

or

```
echo ${var1} ${var2} ${var3}
```

```
yash9274@YASH-PC:~/GFG$ cat geekfile.sh
str1='welcome'
str2="to"
str3=GeeksForGeeks
res=$str1$str2$str3

echo $res
echo $str1**$str2**$str3
echo ${str1}--${str2}--${str3}
echo "$str1" "$str2" "$str3"

yash9274@YASH-PC:~/GFG$ ./geekfile.sh
welcometoGeeksForGeeks
welcome**to**GeeksForGeeks
welcome--to--GeeksForGeeks
welcome to GeeksForGeeks
```

Concatenate strings inside Bash Shell using an array:

To create an array:
arr=("value1" value2 $value3)

To print an array:
echo ${arr[@]}

To print length of an array:
echo ${#arr[@]}

Using indices (index starts from 0):
echo ${arr[index]}

Note: echo ${arr} is the same as echo ${arr[0]}

```
yash9274@YASH-PC:~/GFG$ cat geekfile.sh
str1='welcome'
str2="to"
str3=GeeksForGeeks
str4="Programming"

arr1=($str1 ${str2} "$str3")
arr2=($str4 "is" "fun" 10 20 30)

echo "arr1: " ${arr1[@]}
echo "arr2: " ${arr2[@]}
echo "length of arr1: " ${#arr1[@]}
echo "length of arr2: " ${#arr2[@]}
echo "element at arr1[2]: " ${arr1[2]}
echo "element at arr2[2]: " ${arr2[2]}
echo "This gets printed if index is OutOfBound: " ${arr1[10]}
yash9274@YASH-PC:~/GFG$ ./geekfile.sh
arr1:  welcome to GeeksForGeeks
arr2:  Programming is fun 10 20 30
length of arr1:  3
length of arr2:  6
element at arr1[2]:  GeeksForGeeks
element at arr2[2]:  fun
This gets printed if index is OutOfBound:
yash9274@YASH-PC:~/GFG$
```

Extract a substring from a string

- ${string:position} --> returns a substring starting from $position till end
- ${string:position:length} --> returns a substring of $length characters starting from $position.

Note:

- **$length** and **$position** must be always greater than or equal to zero.
- If the **$position** is less than 0, it will print the complete string.
- If the $**length** is less than 0, it will raise an error and will not execute.

```
yash9274@YASH-PC:~/GFG$ cat geekfile.sh
str="welcome to GeeksForGeeks"
echo ${str:-100}
echo ${str:7}
echo ${str:0:10}

yash9274@YASH-PC:~/GFG$ ./geekfile.sh
welcome to GeeksForGeeks
to GeeksForGeeks
welcome to
yash9274@YASH-PC:~/GFG$
```

Substring matching:

- In Bash, the shortest and longest possible match of a substring can be found and deleted from either front or back.

Syntax:

- To delete the shortest substring match from front of $string:

  ${string#substring}

- To delete the shortest substring match from back of $string:

  ${string%substring}

- To delete the longest substring match from front of $string:

  ${string##substring}

- To delete the shortest substring match from back of $string of $string:

  ${string%%substring}

```
yash9274@YASH-PC:~/GFG$ cat geekfile.sh
match="Welcome.to.GeeksForGeeks"
echo "This will delete the shortest substring that matches *. from front: " ${match#*.}
echo "This will delete the shortest substring that matches .* from back: " ${match%.*}
echo "This will delete the longest substring that matches *. from front: " ${match##*.}
echo "This will delete the longest substring that matches .* from back: " ${match%%.*}

yash9274@YASH-PC:~/GFG$ ./geekfile.sh
This will delete the shortest substring that matches *. from front:  to.GeeksForGeeks
This will delete the shortest substring that matches .* from back:  Welcome.to
This will delete the longest substring that matches *. from front:  GeeksForGeeks
This will delete the longest substring that matches .* from back:  Welcome
yash9274@YASH-PC:~/GFG$
```

In the above example:

- The first echo statement substring ‘***.**‘ matches the characters ending with a dot, and **#** deletes the shortest match of the substring from the front of the string, so it strips the substring ‘**Welcome.**‘.
- The second echo statement substring ‘**.***‘ matches the substring starting with a dot and ending with characters, and **%** deletes the shortest match of the substring from the back of the string, so it strips the substring ‘**.GeeksForGeeks**‘
- The third echo statement substring ‘***.**‘ matches the characters ending with a dot, and **##** deletes the longest match of the substring from the front of the string, so it strips the substring ‘**Welcome.to.**‘
- The fourth echo statement substring ‘**.***‘ matches the substring starting with a dot and ending with characters, and **%%** deletes the longest match of the substring from the back of the string, so it strips the substring ‘**.to.GeeksForGeeks**‘.

Shell Functions

- Function is a collection of statements that execute a specified task.
- Its main goal is to break down a complicated procedure into simpler subroutines that can subsequently be used to accomplish the more complex routine.

Functions are popular:

- Assist with code reuse.
- Enhance the program’s readability.
- Modularize the software.
- Allow for easy maintenance.

Basic structure of a function in shell scripting looks as follows:

```
function_name(){
    // body of the function
}
```

```
function_name () { list of commands }
```

Shell functions are similar to subroutines, procedures, and functions in other programming languages.

Creating Functions

```
function_name () {
   list of commands
}
```

Example :-

```
# Create function

Hello ()
{
Echo "hello world"
}

# invoking a function

Hello
```

You can define a function that will accept parameters while calling the function.
These parameters would be represented by $1, $2 and so on.

EXAMPLE :-

```
# Define your function here
Hello () {
  echo "Hello World $1 $2"
}

# Invoke your function
Hello  SAM TAM
```

Returning Values from Functions

If you execute an exit command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function.

Based on the situation you can return any value from your function using the return command whose syntax is as follows −

return code

Here code can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

EXAMPLE:

```
Hello () {
  echo "Hello World $1 $2"
  return 10
}

# Invoke your function
Hello SAM TAM

# Capture value returned by last command
ret=$?

echo "Return value is $ret"
```

Recursion

Functions can be recursive - here's a simple example of a factorial function:

factorial.sh

```
#!/bin/sh

factorial()
{
  if [ "$1" -gt "1" ]; then
    i=`expr $1 - 1`
    j=`factorial $i`
    k=`expr $1 \* $j`
    echo $k
  else
    echo 1
  fi
}

while :
do
  echo "Enter a number:"
  read x
  factorial $x
done
```

Creating Accounts and Groups- Managing Users and Groups- Passwords

- Computer is used by many people it is usually necessary to differentiate between the users, for example, so that their private files can be kept private.
- An account is all the files, resources, and information belonging to one user.

Types of user account

- Root account: This is also called super user and would have complete and unfettered control of the system. A super user can run any commands without any restriction. This user should be assumed as a system administrator.
- System accounts: System accounts are those needed for the operation of system specific components for example mail accounts and the sshd accounts. These accounts are usually needed for some specific function on your system, and any modifications to them could adversely affect the system.
- User accounts: User accounts provide interactive access to the system for users and groups of users. General users are typically assigned to these accounts and usually have limited access to critical system files and directories.

Managing Users and Groups

- /etc/passwd − Keeps the user account and password information. This file holds the majority of information about accounts on the Unix/Linux system.
- /etc/shadow − Holds the encrypted password of the corresponding account. Not all the systems support this file.
- /etc/group − This file contains the group information for each account.
- /etc/gshadow − This file contains secure group account information.

Following are commands available on the majority of Linux systems to create and manage accounts and groups:

Command	Description
useradd	Adds accounts to the system.
usermod	Modifies account attributes.
userdel	Deletes accounts from the system.
groupadd	Adds groups to the system.
groupmod	Modifies group attributes.
groupdel	Removes groups from the system.

useradd

- useradd [options] username

- sudo useradd johndoe

- sudo useradd -m -d /home/janedoe -s /bin/bash janedoe

Common options:
- -m: Create the user's home directory
- -d: Specify the home directory
- -s: Specify the login shell
- -G: Add the user to additional groups

usermod

- usermod [options] username

- sudo usermod -d /newhome/johndoe johndoe

- sudo usermod -aG sudo,developers janedoe

Common options:
- -d: Change the home directory
- -s: Change the login shell
- -l: Change the username
- -G: Set supplementary groups
- -aG: Add to supplementary groups without removing from existing ones

userdel

- userdel [options] username

- sudo userdel johndoe

- sudo userdel -r janedoe

Common options:

- -r: Remove the user's home directory and mail spool

groupadd

- groupadd [options] group_name
- sudo groupadd developers
- sudo groupadd -g 1500 project_team

groupmod

- groupmod [options] group_name
- sudo groupmod -n new_developers developers
- sudo groupmod -g 2000 project_team

groupdel

- groupdel group_name

- sudo groupdel project_team

getent group

- To view existing groups on the system.

How ownership works

- You can specify a different user and/or group as the owner of a given file or directory. To change the user who owns a file, you must be logged in as root. To change the group that owns a file, you must be logged in as root or as the user who currently owns the file.

✓ Using **chown**

✓ Using **chgrp**

✓ You can also view file ownership from the command line using the **ls –l** command

chown

- chown command in Linux is used to change the ownership of files and directories.
- chown [OPTIONS] USER[:GROUP] FILE(s)

Where:

- USER is the new owner's username
- GROUP (optional) is the new group
- FILE(s) are the target files or directories

- chown john file.txt
- chown john:users file.txt
- chown -R alice:staff /home/project
- chown 1000:1000 file.txt
- chown :developers file.txt

chgrp

- chgrp command in Linux is used to change the group ownership of files and directories.

- chgrp [OPTIONS] GROUP FILE(S)

Change the group of a single file:

chgrp developers myfile.txt

Change the group of multiple files:

chgrp staff file1.txt file2.txt file3.txt

Recursively change group ownership of a directory and its contents

```
chgrp -R project_team /path/to/project/
```

Change group ownership using a numeric group ID

```
chgrp 1001 document.pdf
```

Change group and display what's being changed:

```
chgrp -v marketing report.docx
```

Change group only if it matches a specific group:

```
chgrp --from=oldgroup newgroup shared_file.txt
```

Change group and preserve root directory group:

```
chgrp --preserve-root team /some/directory
```

su

- Short for *substitute* or *switch user*
- Syntax: `su [options] [username]`
  - If `username` is omitted, `root` is assumed
- After issuing command, prompted for that user's password
- A new shell opened with the privileges of that user
- Once done issuing commands, must type exit

sudo

- Allows you to issue a single command as another user
- Syntax:
  `sudo [options] [-u user] command`
- Again, if no user specified, root assumed
- New shell opened with user's privileges
- Specified command executed
- Shell exited

Create a Group

- All the default groups would be system account specific groups and it is not recommended to use them for ordinary accounts.

Syntax: sudo groupadd [-g gid [-o]] [-r] [-f] groupname

groupadd IMCAGEN

Option	Description
-g GID	The numerical value of the group's ID.
-o	This option permits to add group with non-unique GID
-r	This flag instructs groupadd to add a system account
-f	This option causes to just exit with success status if the specified group already exists. With -g, if specified GID already exists, other (unique) GID is chosen.
Groupname	Actaul group name to be created.

Modify a Group

- To modify a group, use the groupmod syntax:

```
$groupmod –n newgroupname oldgroupame
```

Delete a Group

- To delete an existing group, all you need are the groupdel command and the group name. To delete the ‘networking’ group, the command is:

$goupdel IMCAGEN

Create an Account

$ sudo useradd -d homedir -g groupname -m -s shell -u userid accountname

$ useradd -d /home/gobi -g developers -s /bin/ksh gobi

Option	Description
- d homedir	Specifies home directory for the account.
-g groupname	Specifies a group account for this account.
-m	Creates the home directory if it doesn't exist.
-s shell	Specifies the default shell for this account.
-u userid	You can specify a user id for this account.
accountname	Actual account name to be created

- creates an account gobi, setting its home directory to /home/gobi and the group as developers. This user would have Korn Shell assigned to it.

- Once an account is created you can set its password using the passwd command as follows:

$ passwd **gobi**

Changing password for user gobi

New UNIX password:

Retype new UNIX password:

passwd: all authentication tokens updated successfully.

Modify an Account

- usermod command enables you to make changes to an existing account from the command line. It uses the same arguments as the useradd command, plus the -l argument, which allows you to change the account name.

$ usermod -d /home/ram -m -l gobi ram

Delete an Account

- userdel command can be used to delete an existing user. This is a very dangerous command if not used with caution.
- There is only one argument or option available for the command .r, for removing the account's home directory and mail file.

```
$ userdel -r ram
```

Changing user properties

- There are a few commands for changing various properties of an account (i.e., the relevant field in /etc/passwd):

- chfn: to change the full name field.
- chsh: to change the login shell.
- passwd: to change the password.

- super−user may use these commands to change the properties of any account.
- Normal users can only change the properties of their own account.

/etc/passwd and other informative files

- Username
- Previously this was where the user's password was stored.
- Numeric user id.
- Numeric group id.
- Full name or other description of account.
- Home directory.
- Login shell (program to run at login)

passwd

- Changes a user's password.

passwd [options] [LOGIN]

- passwd command changes passwords for user accounts.
- A normal user may only change the password for his or her own account, while the superuser may change the password for any account.
- passwd also changes the account or associated password validity period.

passwd : Change your own password.

passwd username : Change the password for the user named username.

General guideline-Password

- passwords should consist of 6 to 8 characters including one or more characters from each of the following sets:
- lower case letters
- digits 0 through 9
- punctuation marks

Options:

-a, --all	This option can be used only with -S and causes show status for all users.
-d, --delete	Delete a user's password (make it empty). This is a quick way to disable a password for an account. It will set the named account passwordless.
-e, --expire	Immediately expire an account's password. This in effect can force a user to change his/her password at the user's next login.
-h, --help	Display a help message, and exit.

passwd exits with one of the following status codes, depending on what occurred:

0	Success.
1	Permission denied.
2	Invalid combination of options.
3	Unexpected failure; nothing done.
4	Unexpected failure; passwd file missing.
5	passwd file busy; try again.
6	Invalid argument to one or more options.

Set Password Expiry Date for an user using chage option -M

- Root user (system administrators) can set the password expiry date for any user

```
# chage -M number-of-days username
```

sudo chage –M 30 gobi


pi@raspberrypi:~ $ chage -h
Usage: chage [options] LOGIN
Options:
-d, --lastday LAST_DAY set date of last password change to LAST_DAY
-E, --expiredate EXPIRE_DATE set account expiration date to EXPIRE_DATE
-h, --help display this help message and exit
-I, --inactive INACTIVE set password inactive after expiration
to INACTIVE
-l, --list show account aging information
-m, --mindays MIN_DAYS set minimum number of days before password
change to MIN_DAYS
-M, --maxdays MAX_DAYS set maximim number of days before password
change to MAX_DAYS
-R, --root CHROOT_DIR directory to chroot into
-W, --warndays WARN_DAYS set expiration warning days to WARN_DAYS
pi@raspberrypi:~ $

Using chown

- The chown utility can be used to change the **user** or **group** that owns a file or **directory**.

  Syntax **chown** user.group file or directory.

  Example: If I wanted to change the file's owner to the **ken1** user, I would enter

  chown ken1 /tmp/myfile.txt

  –If I wanted to change this to the users group, of which **users** is a member, I would enter

  chown .users /tmp/myfile.txt

  Notice that I used a period **(.)** before the group name to tell chown that the entity specified is a group, not a user account.

  Ex: **chown** student.users /tmp/myfile.txt

  Note: You can use the –R option with chown to change ownership on many files at once recursively.

Using chgrp

- In addition to chown, you can also use **chgrp** to change the group that owns a file or directory.
- Syntax: **chgrp group file (or directory)**
- Example: **chgrp student /tmp/newfile.txt.**