AROR UNIVERSITY OF ART, ARCHITECTURE, DESIGN & HERITAGE SUKKUR

Operating System
Lab-04

Compiled by: Aurangzeb Magsi

Course Title: OperatingSystem
Lab-4
Programming using Shell Scripting

SHELL PROGRAMMING

1. **Getting familiar with shells**
2. **Write shell script**
3. **Run shell script**
4.

1. What is Shell Program

When a user enters commands from the command line, he is entering them one at a time and getting a response from the system. From time to time it is required to execute more than one command, one after the other, and get the final result. This can be done with a ***shell program or a shell script***. A shell program is a series of Linux commands and utilities that have been put into a file by using a text editor. When a shell program is executed the commands are interpreted and executed by Linux one after the other.

2. Common Shells.

i. C-Shell - csh :

• Developed by **Bill Joy** at the University of California, Berkeley, in the late 1970s. • It was designed for interactive use, particularly for users familiar with the C programming language.

• Includes useful interactive features like command history and job control.

• **Good for interactive use**: Supports command history (`!!` to repeat the last command, `!n` to execute a specific previous command).

• **Aliases**: Allows users to create shortcuts for frequently used commands.

• **Inferior programmable features**: The scripting capabilities of `csh` are considered weaker compared to other shells like Bourne Shell (`sh`) and Bash.

• It is useful for interactive work but not recommended for complex scripting due to its inconsistent syntax.

• Installation on Ubuntu:

```
sh

sudo apt update
sudo apt install csh
```

• To check if it's installed, type:

```
sh

csh --version
```


• To switch to `csh`, run:

```
sh

csh
```

• To exit, type:

```sh
exit
```

To verify that you are inside `tcsh`:

```sh
echo $SHELL
```

It should return `/usr/bin/tcsh` or `/bin/tcsh`.

2. Displaying the Current Working Directory

```sh
pwd
```

This will show the current directory.

3. Listing Files and Directories

```sh
ls
```

For detailed output:

```sh
ls -la
```

4. Creating and Removing Files

Create a new file:

```sh
touch myfile.txt
```

Remove a file:

```sh
rm myfile.txt
```

5. Creating and Removing Directories

Create a directory:

```sh
mkdir mydir
```

Remove a directory:

```sh
rmdir mydir
```

If the directory contains files, use:

```sh
rm -r mydir
```

6. Command History

To see previously executed commands:

```sh
history
```

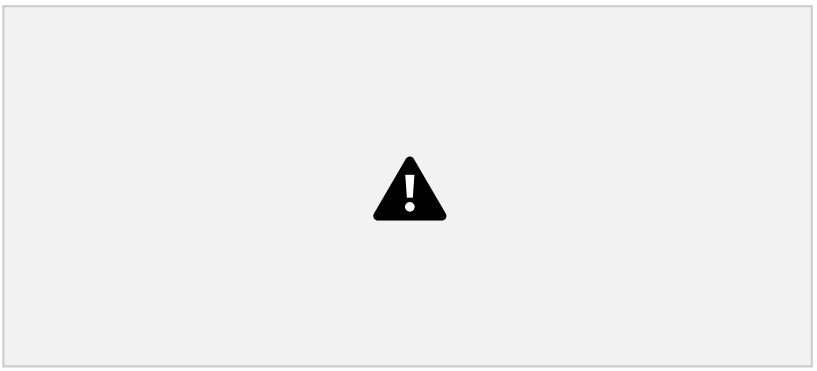
To repeat the last command:

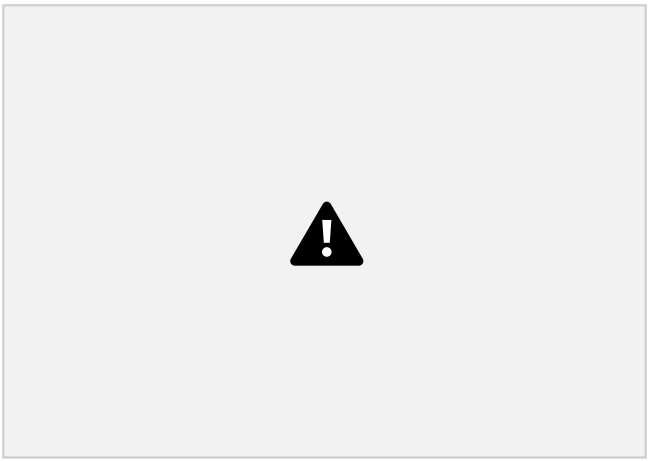
```sh
!!
```

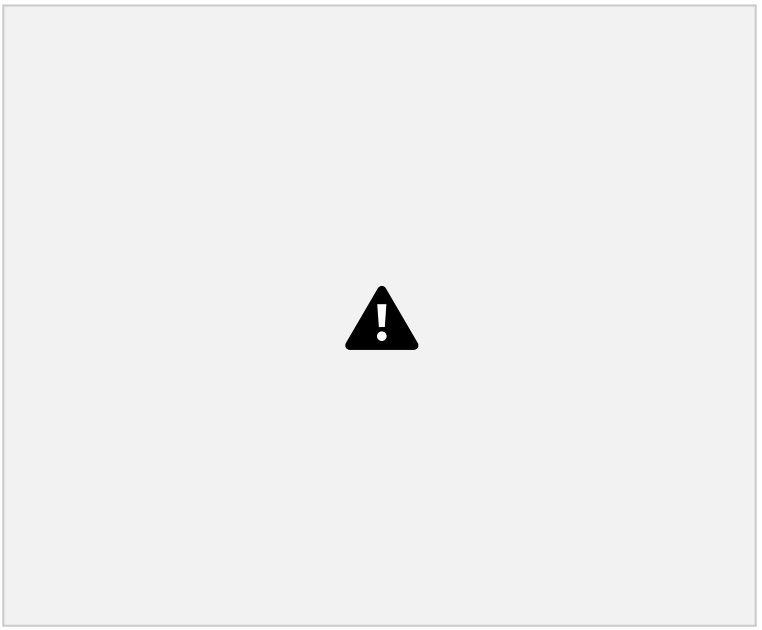
To execute a specific command from history:

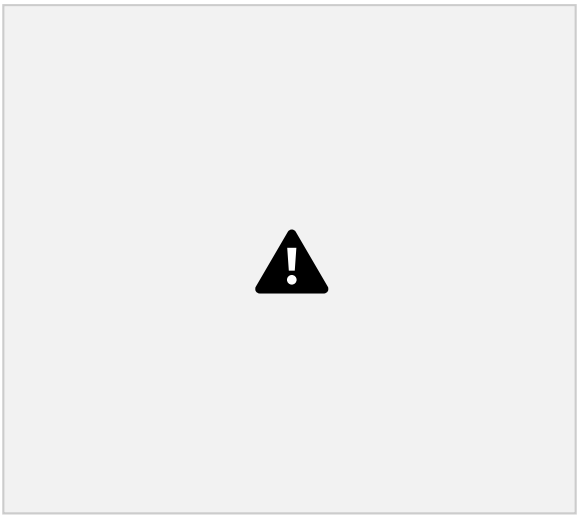
```sh
!5
```

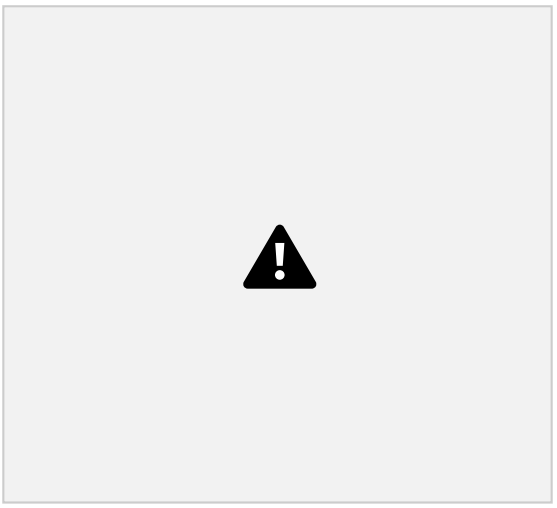
(Executes the 5th command from history)









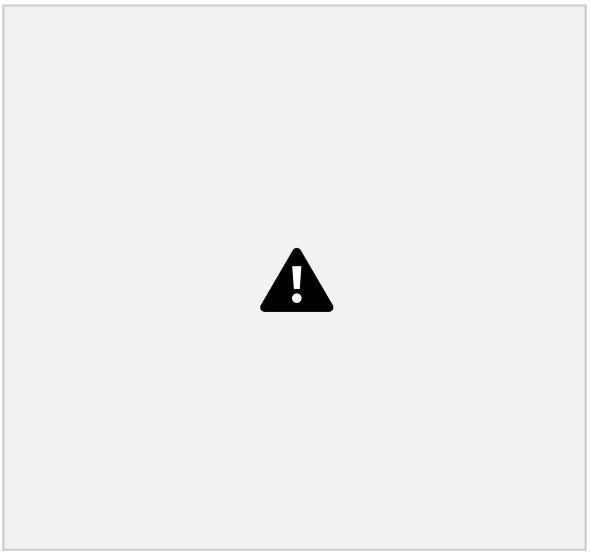


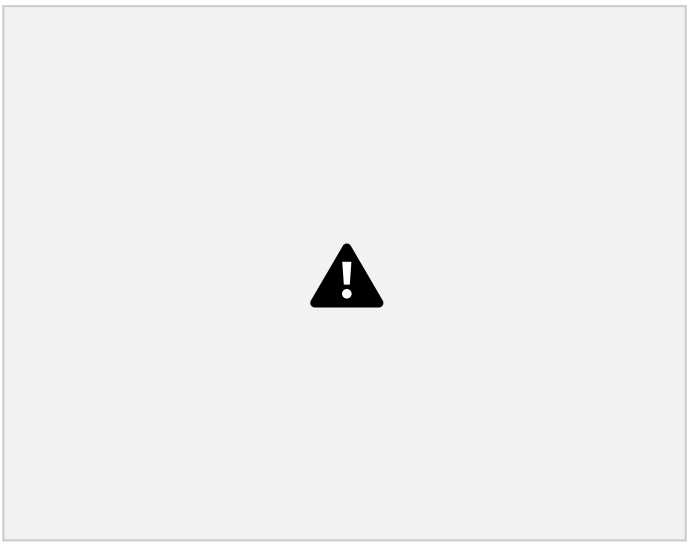
Course Title:

OperatingSystem

Lab-4

Programming using Shell Scripting



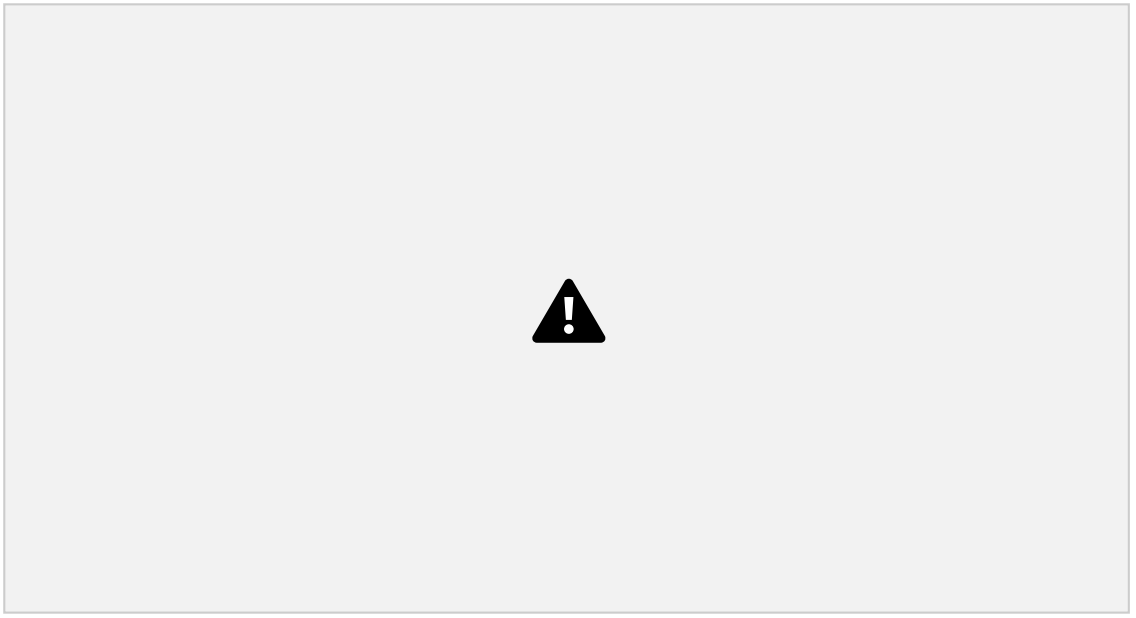


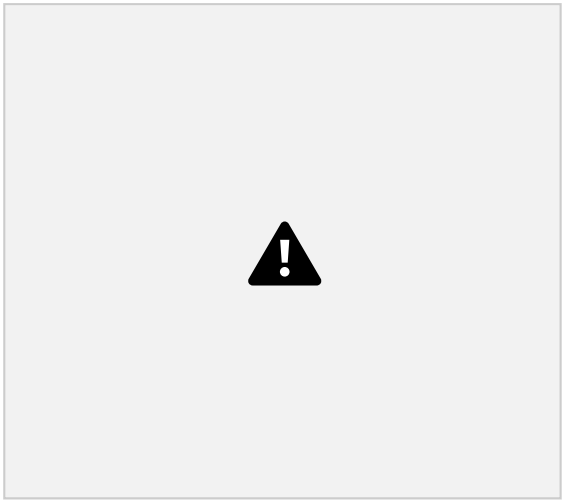
ii. Bourne Shell - bsh or sh - also restricted shell - bsh :

• Developed by **Stephen Bourne** at AT&T Bell Labs in the late 1970s.

• One of the earliest Unix shells, widely used as the default shell in many Unix-like systems. • **Lightweight and efficient**: Uses fewer system resources compared to newer shells. • **Sophisticated pattern matching**: Supports wildcards (`*`, `?`, `[]`) for file name expansion. • **Scripting capabilities**: More powerful and structured than `csh` for writing scripts. • **No command-line history or aliases** (unlike `csh`).

• **Best for scripting** due to its simple, structured syntax.
• It is the **default shell** on many Unix-like systems, ensuring portability of scripts.



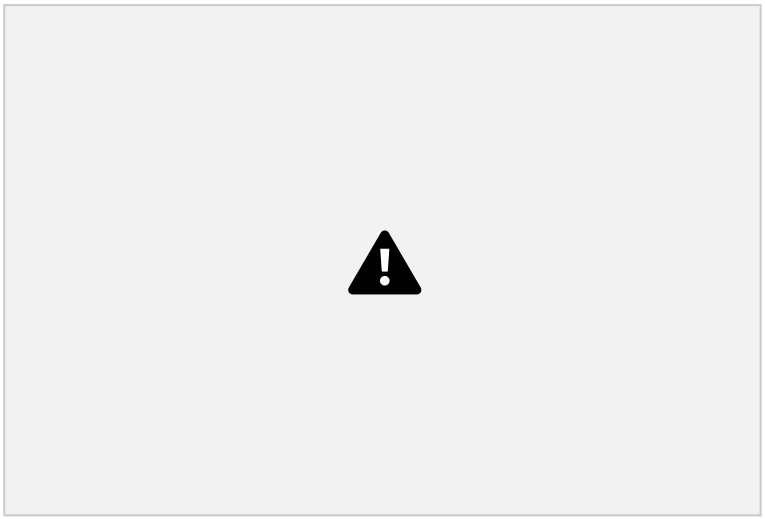


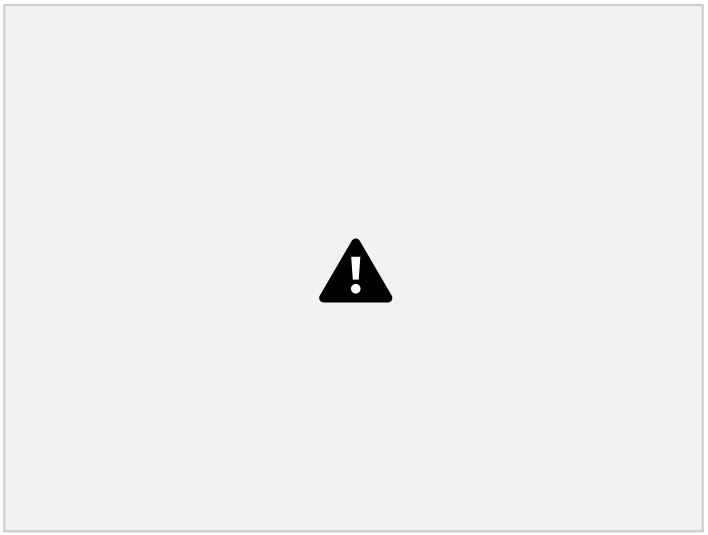
iii. Korn Shell :

• Developed by **David Korn** at AT&T Bell Labs in the early 1980s.

• It is an improved version of the Bourne Shell with additional scripting features. • **Backward compatibility with Bourne Shell (`sh`)**: Supports all `sh` scripts without modifications. • **Regular expression substitution**: More powerful pattern matching and variable handling. • **Job control and command-line editing**: Allows background and foreground job control. • **Aliases and functions**: Supports aliasing commands and defining functions.

• **More advanced than `sh`**, with features similar to Bash (`bash`).
• Used in **enterprise environments** due to its scripting power and efficiency.

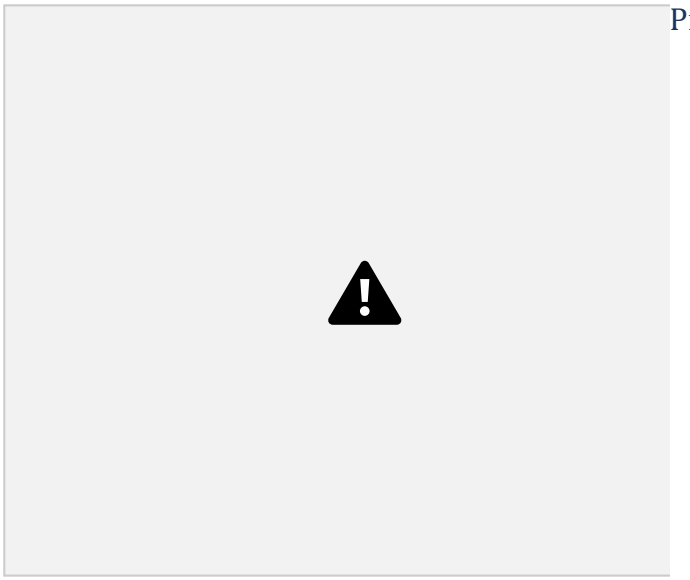


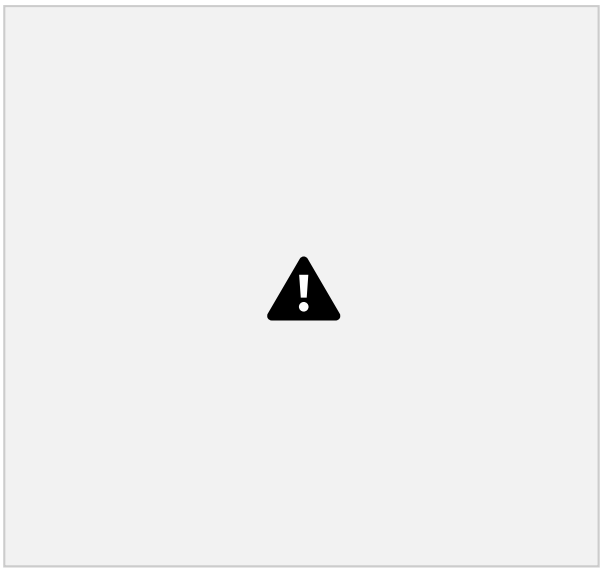


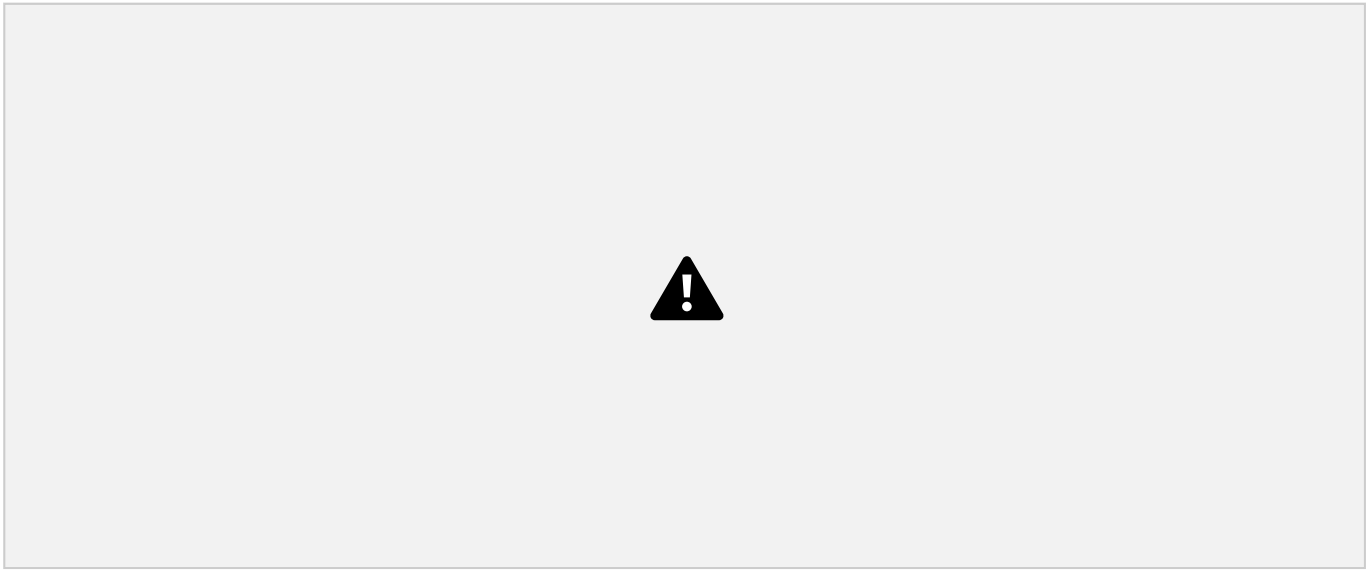
iv. Thomas C-Shell - tcsh :

• An enhanced version of **C-Shell (`csh`)**, developed by Ken Greer.
• It improves upon `csh` with additional interactive features.
• **Command-line editing using Emacs**: Users can navigate and edit the command line efficiently.
• **Word completion**: Automatically completes filenames and commands when pressing `TAB`. • **Spelling correction**: Can suggest and correct mistyped commands.
• **Improved scripting over `csh`**, but still **inferior to `sh` or `bash`**.
• Best suited for **interactive use**, not recommended for complex scripting.

Lab-4
Programming using Shell Scripting







Course Title: OperatingSystem Lab-4

Programming using Shell Scripting

3. Understanding General Concepts in Shell Scripting

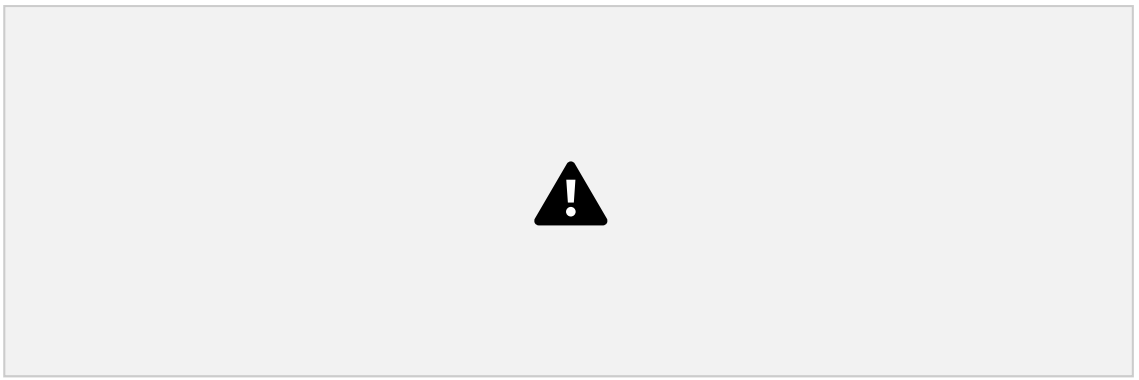
i. The Shbang Line (#!)

- The **shbang** (also called shebang) is the very first line of a shell script.
- It **informs the operating system** which interpreter (shell) should be used to execute the script.
- It **starts with #! (hash and exclamation mark)** followed by the absolute path of the shell.

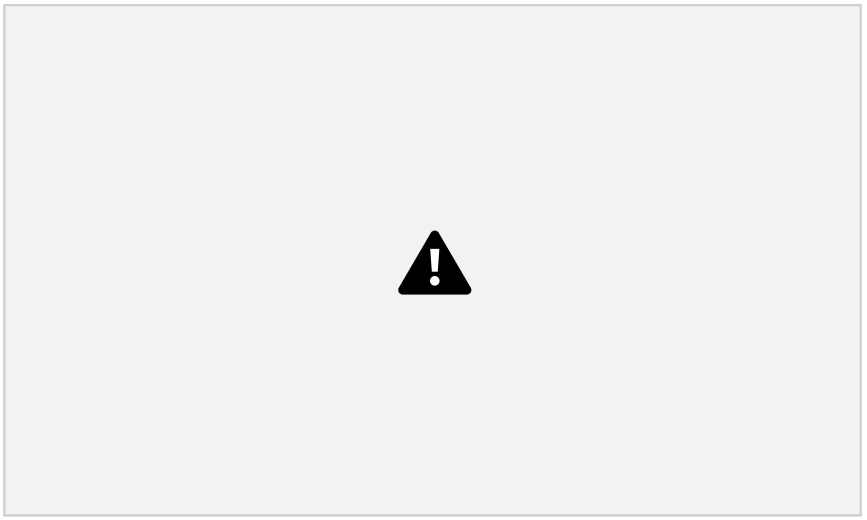
E*XAMPLE:*

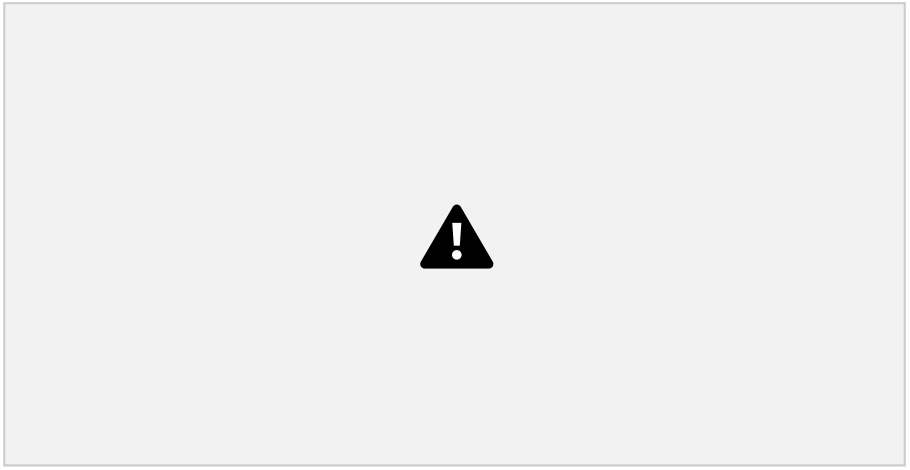
#!/bin/sh

- Without the shbang line, the script would be executed using the **default shell**, which may not be what you intend.
- Ensures portability, allowing the script to run consistently across different environments.



How to Run a Script with the Shbang Line





Comments

Comments are descriptive material preceded by a # sign. They are in effect until the end of a line and can be started anywhere on the line.

E*XAMPLE*

```
# this text is not
# interpreted by the shell
```

4. Assigning values to variables

A value is assigned to a variable simply by typing the variable name followed by an equal sign and the value that is to be assigned to the variable. For example, if you wanted to assign a value of 5 to the variable count, you would enter the following command:

```
count=5
```

5. Accessing variable values

To access the value stored in a variable precede the variable name with a dollar sign ($). If you wanted to print the value stored in the count variable to the screen, you would do so by entering the following command:

```
echo $count
```

If you omitted the $ from the preceding command, the echo command would display the word count on screen.

6. Positional parameters

What are Positional Parameters?

The shell has knowledge of a special kind of variable called a positional parameter. Positional parameters are used to refer to the parameters that were passed to a shell program on the command line or a shell function by the shell script that invoked the function. When you run a shell program that requires or supports a number of command-line options, each of these options is stored into a positional parameter. The first parameter is stored into a variable named 1, the second parameter is stored into a variable named 2, and so forth. These variable names are reserved by the shell so that you can't use them as variables you define. To access the values stored in these variables, you must precede the variable name with a dollar sign ($) just as you do with variables you define.

The following shell program expects to be invoked with two parameters. The program takes the two parameters and prints the second parameter that was typed on the command line first and the first parameter that was typed on the command line second.

```
#program reverse, prints the command line parameters out in reverse #order
        echo "$2" "$1"
```

If you invoked this program by entering

```
        reverse hello there
```

The program would return the following output:

```
        there hello
```

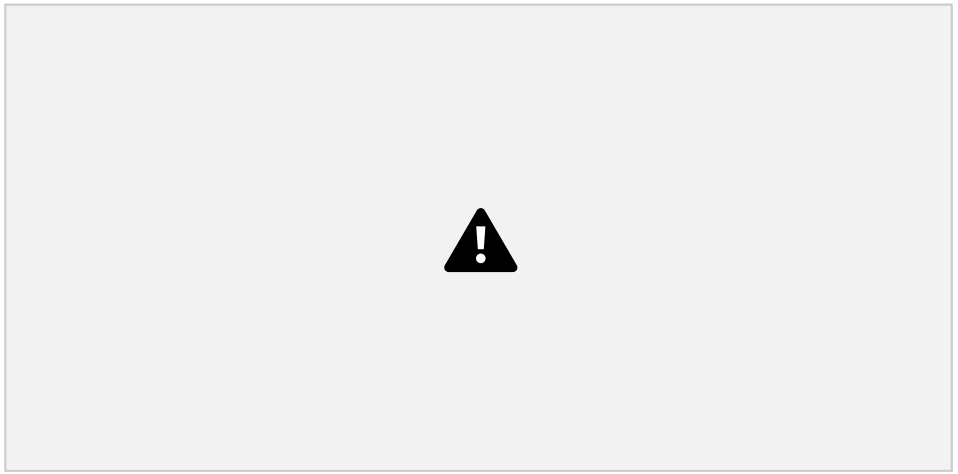
Positional parameters are **special variables in a shell script** that allow access to arguments passed to the script via the command line. These parameters enable **dynamic execution** of scripts based on user input.

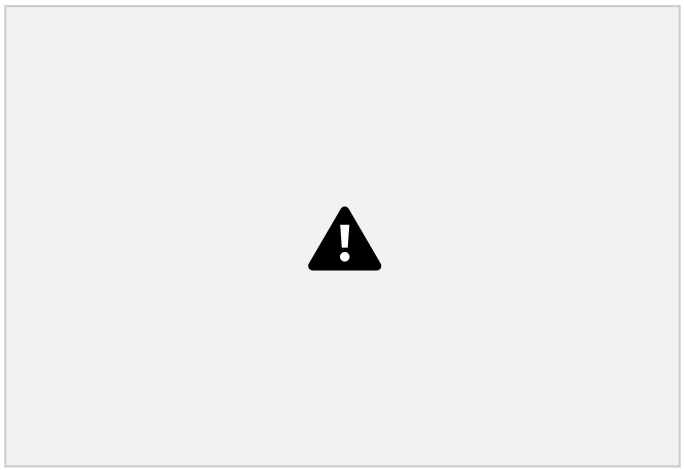
- Each parameter is referenced **numerically**, starting from `$1`, `$2`, `$3`, etc.
- `$0` stores the **script name** or **command itself**.
- `$#` represents the **total number of arguments** passed.
- `$*` and `$@` hold **all command-line arguments**.

Course Title:
OperatingSystem

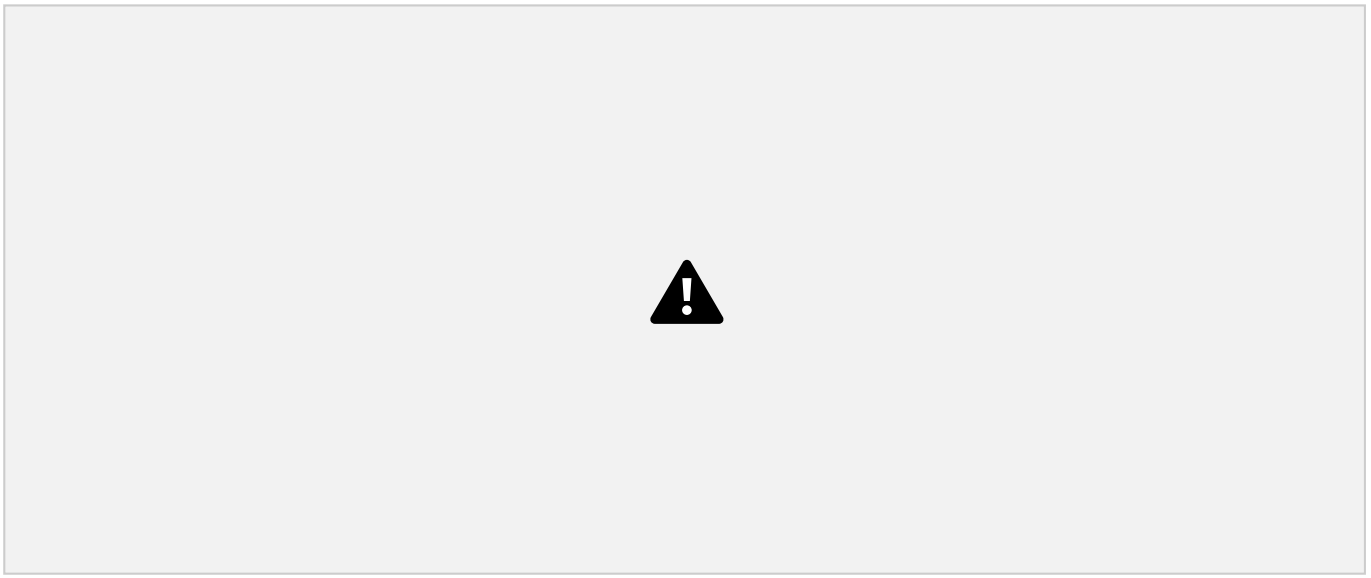
How Positional Parameters Work?

Lab-4



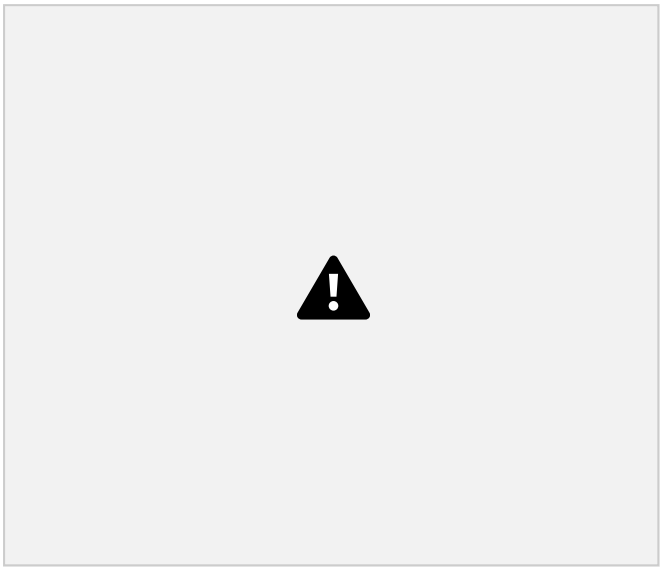


Special Positional Parameters:

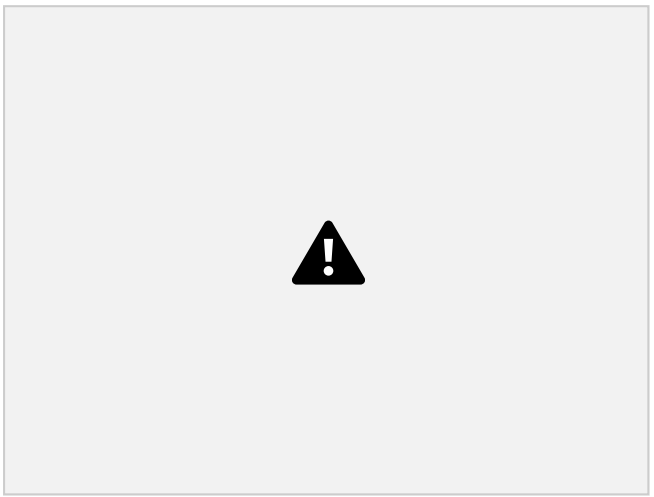


Difference Between $* and $@: `$*` treats all arguments as a single string.

`$@` treats each argument separately.



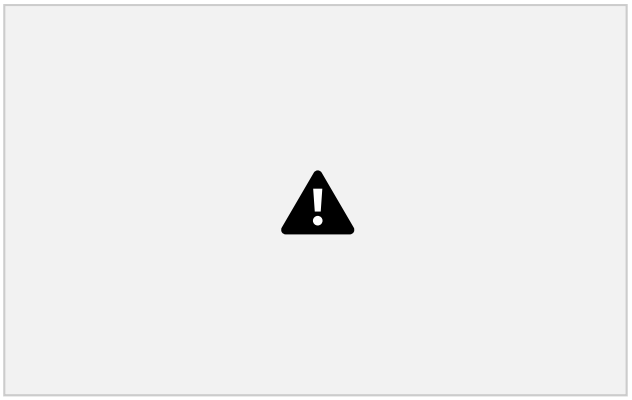
Reversing Command-Line Parameters:



Lab-4

Course Title:
OperatingSystem

Looping Through Arguments: To iterate over all command-line arguments:



7. Built-in Variables

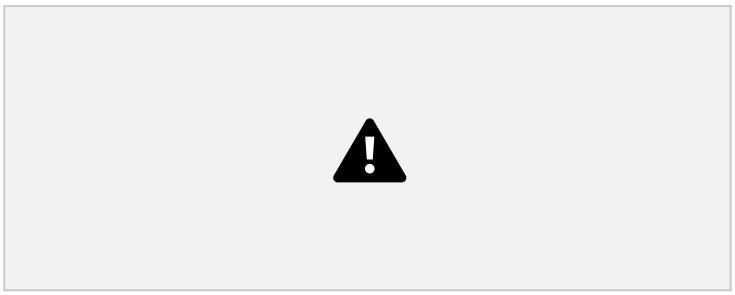
These are special variables that Linux provides that can be used to make decisions in a program. Their values cannot be modified. Several other built-in shell variables are important to know about when you are doing a lot of shell programming. The following table lists these variables and gives a brief description of what each is used for.

Variable	Use
$#	Stores the number of command-line arguments that were passed to the shell program.
$?	Stores the exit value of the last command that was executed.
$0	Stores the first word of the entered command (the name of the shell program).
$*	Stores all the arguments that were entered on the command line ($1 $2 ...).
"$@"	Stores all the arguments that were entered on the command line, individually quoted ("$1" "$2" ...).

Built-in variables in Linux shell scripting are **special variables provided by the shell**. These variables **store system-related and script execution information** and **cannot be modified directly**.

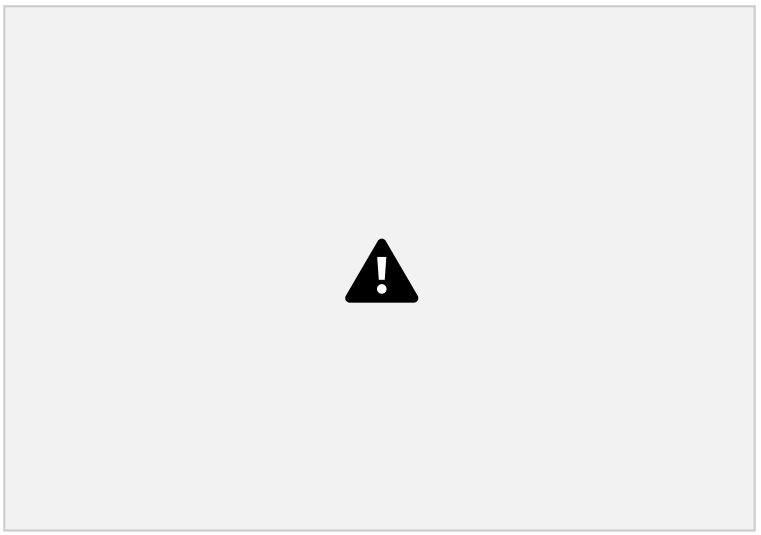
i. `$#` (Number of Command-Line Arguments)

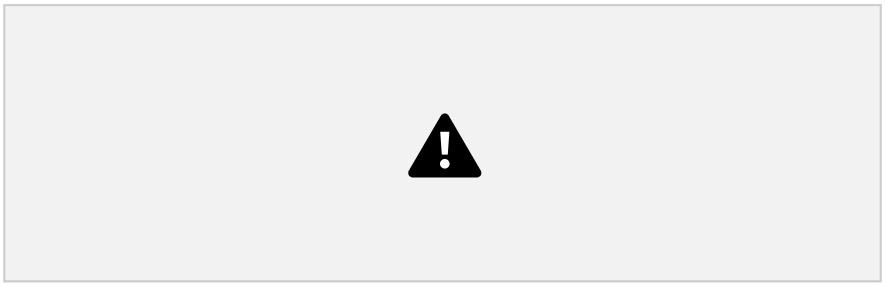
- Stores the **total number of arguments** passed to a script.
- Useful for **validating user input** and **checking argument count**.



b. `$?` (Exit Status of Last Command)

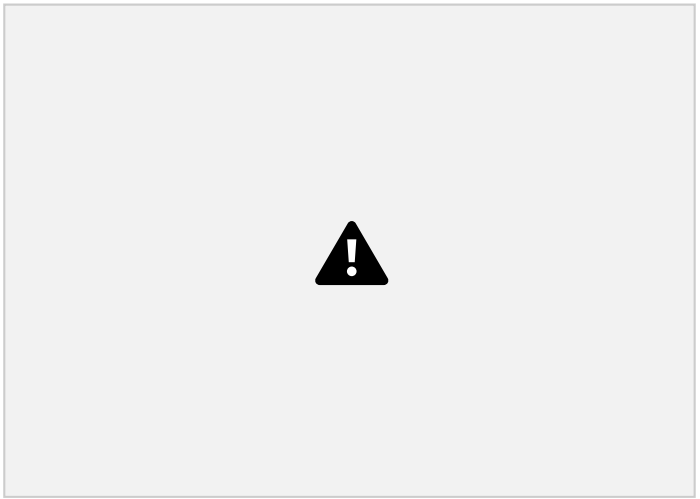
- Stores the **exit status (return value)** of the **last executed command**.
- In Linux, an exit status of:
  - 0 **means success**.
  - Any **nonzero value indicates an error**.





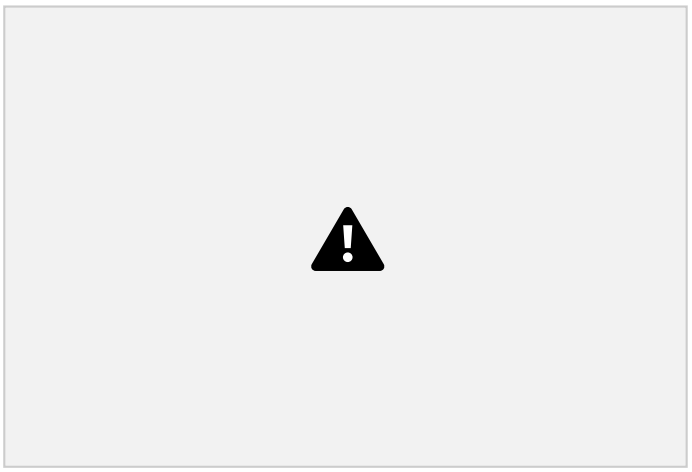
c. $0 (Script Name)

- Stores the **name of the script itself**.
- Useful for **debugging and logging**.



d. $* (All Arguments as a Single String)

- Stores **all command-line arguments as one string**.
- Treats all arguments **as a single entity**.



e. "$@" (All Arguments as Individual Strings)

- Similar to `$*`, but **treats each argument separately**.
- Maintains **individual argument separation**.