***Chapter 7 Input/Output and Command-Line Processing***

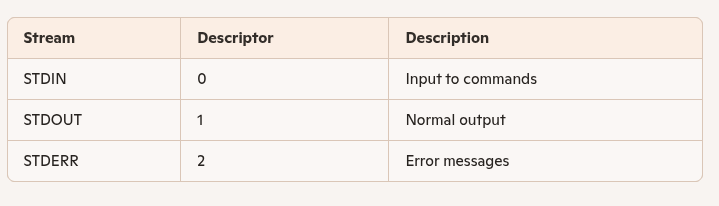
*Introduction:*

1. **Shift in Focus**
   * Previous chapters emphasized *data flow* and *control flow* in shell scripts.
   * This chapter shifts to **input/output (I/O)** mechanisms and **command-line processing**.
2. **Part 1: File-Oriented I/O**
   * Explores the shell’s advanced **I/O redirectors**.
   * Builds upon basic knowledge of <, >, >>, and |.
   * Covers handling of **files**, **streams**, and **descriptors**.
3. **Part 2: Line & Word-Level I/O**
   * Focuses on moving data **between files/terminals and shell variables**.
   * This is different from just redirecting files.
   * Involves tools like:
     + echo
     + **Command substitution** ($(...), `...`)
     + read, printf, here-strings and here-documents
4. **Command Line Processing**
   * Explains **how the shell interprets command lines**.
   * Discusses:
     + **Quotation rules**
     + **Tokenization**
     + **Expansion order**
     + Shell **parsing behavior**
5. **Advanced Feature – eval**
   * Introduces the eval command.
   * Shows how to **dynamically construct and execute** shell code.
   * Useful but requires understanding of how the shell parses and executes commands.

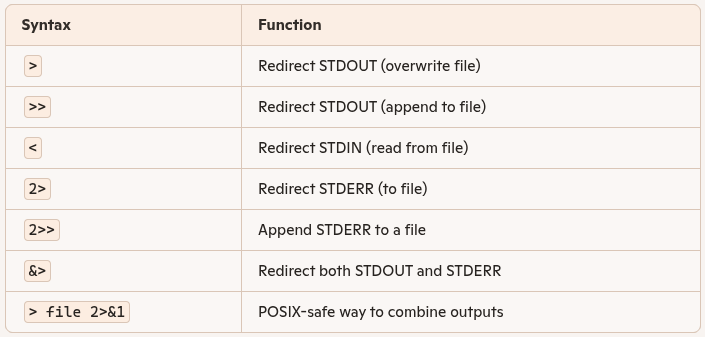
I/O Redirectors in Bash:I/O redirectors allow you to send output to files, read input from files, and control where errors and results go. Bash provides both simple and advanced redirection mechanisms.Bash redirectors allow managing input and output between:

* Commands
* Files
* The terminal
* Shell variables

Standard Streams in Bash:-

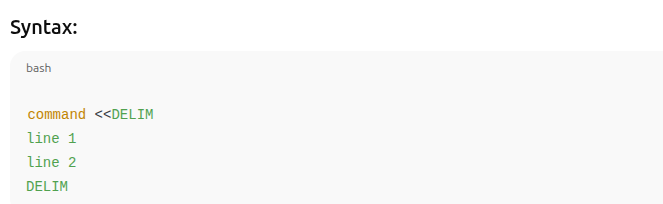


Basic Redirectors:-



## **Here Documents (<<)**

* Feed **multi-line** input directly into a command
* Useful for scripts and automation



exec for Persistent Redirection ->Use exec to **permanently redirect** streams for a script or session:



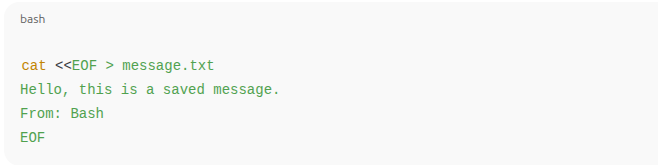
## **What is a Here-Document?**

A **Here-Document** is a way to **pass multi-line input directly** to a command from within a script or terminal, without needing to create a separate file.It simulates feeding input via **standard input (STDIN)**.

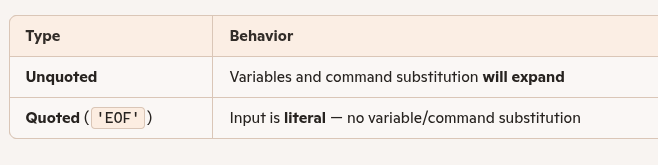


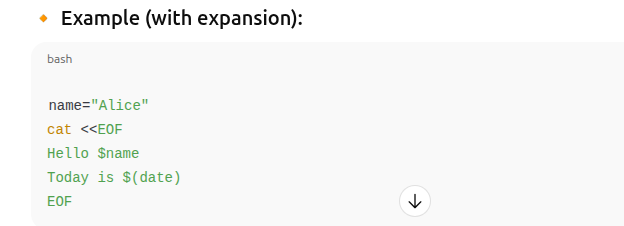
* The DELIMITER can be any word (often EOF, END, etc.)
* Input lines are passed to the command as if typed into the terminal
* Input ends when the **ending delimiter appears alone on a line**

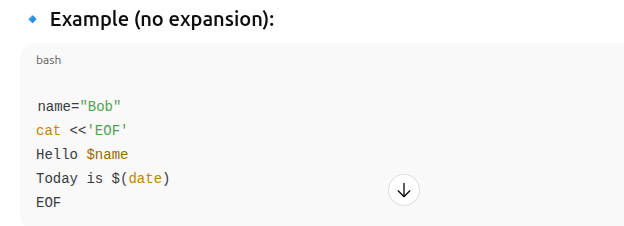
Saving the output:-



Delimiter:-



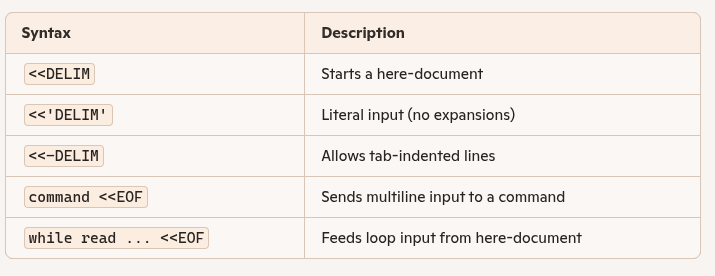




## **Tips and Best Practices**

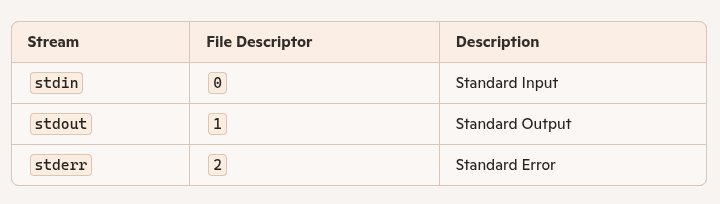
* **Use quoted delimiters** if you want literal text (no substitution).
* Be careful when using <<-; it only works with **tabs**, not spaces.
* Use meaningful delimiter names like EOF, END, INPUT, etc.
* Prefer here-documents for **structured or multi-line** inputs directly in your script.

Here-Document Syntax Reference:-



What Are File Descriptors?

* A **file descriptor (FD)** is an integer used to access input/output streams in UNIX-like systems.
* Every process starts with **three default descriptors**:



## **1.Redirecting Output Using File Descriptors**

You can **redirect individual streams**:



## **2.Redirecting Both stdout and stderr to the Same File**

Use 2>&1 to merge stderr with stdout:



### **Explanation:**

* > redirects stdout to all.log.
* 2>&1 redirects stderr (FD 2) **to wherever FD 1 is currently pointing** (i.e., all.log).

## **3. Running Background Jobs and Logging Output**

## **Task 7-2: Script to run jobs in the background and log all output**



* $@ expands to all arguments (i.e., the full command).
* > sends stdout to logfile.
* 2>&1 sends stderr to the same location.
* & runs the job **in the background**.

**4.Piping stdout and stderr Together**

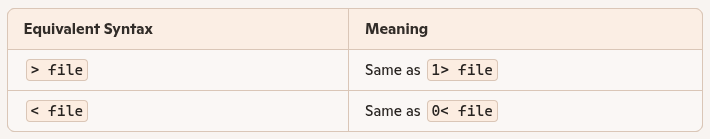
****

* **2>&1: Combine stderr with stdout.**
* **| tee logfile: Send output to both terminal and logfile.**
* **&: Run in background.**

**5.Closing File Descriptors**

****

**6.FD Shorthand Equivalence**

****

***String I/O :-***

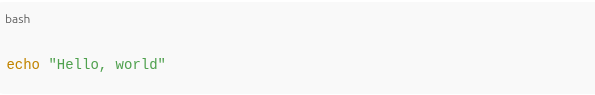
* String I/O refers to line- and word-level input/output.
* In Bash, this is primarily handled using:
  + echo – for outputting text
  + read – for accepting user or string input

These commands give shell scripts I/O capabilities similar to traditional programming languages.Advanced Redirectors (Rarely Used in Scripts)

* <&n : Read input from another file descriptor.
* >&n : Duplicate output to another FD.

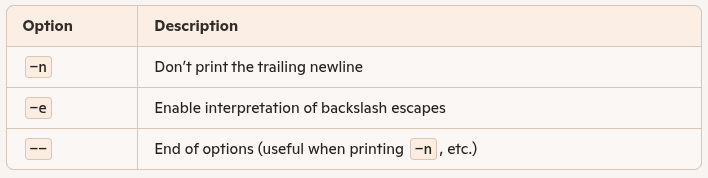
Mostly used in systems programming or advanced I/O control, rarely in general scripting.

***echo – Output Strings:***

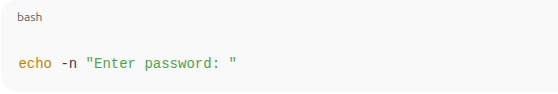


* Outputs a string to **stdout**
* Arguments are separated by spaces

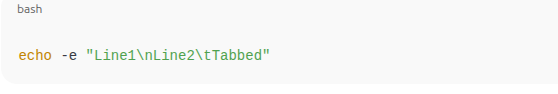
***Key Behaviors:-***



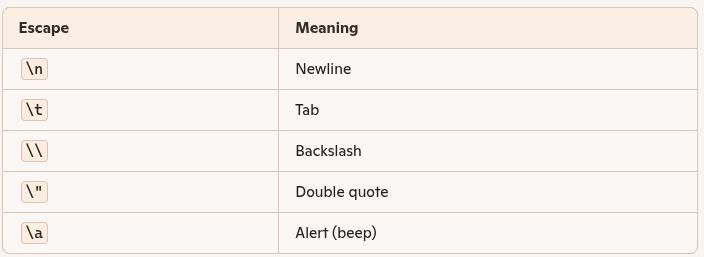
Example with -n:



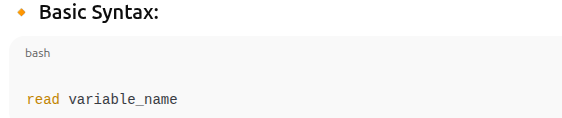
Example with -e (escape characters):



Common Escapes:-

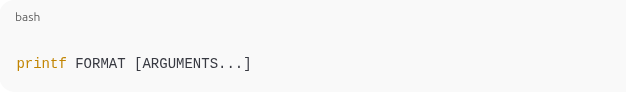


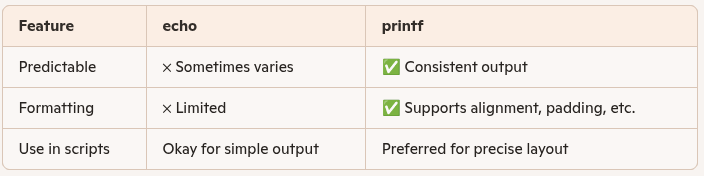
**read – Input Strings:-**

****

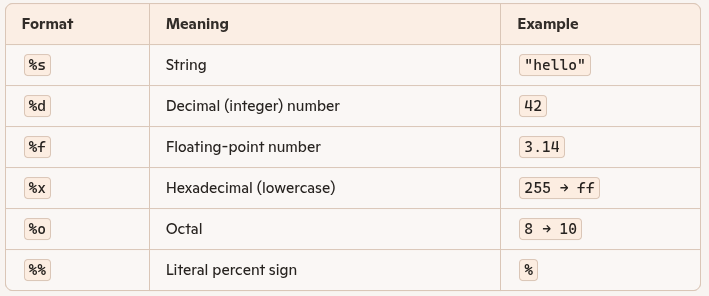
* Takes input from user or STDIN
* Stores it in a shell variable

**What is printf:-**

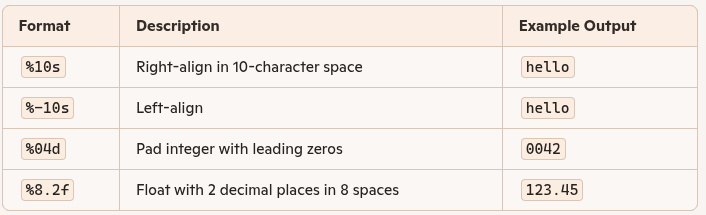
* printf is a more powerful and precise alternative to echo.
* It follows the formatting conventions of the C printf() function.
* Ideal for producing structured, aligned, and formatted output.  
  
* FORMAT: A **format string** with placeholders (e.g., %s, %d)
* ARGUMENTS: Values to fill into the format placeholders

Comparison: echo vs printf:  


Common Format Specifiers:-

****

**Padding and Alignment:-**

****

**Practical Use in Scripts:-**

****

# **Key Points: Additional Bash printf Specifiers:-**

Overview

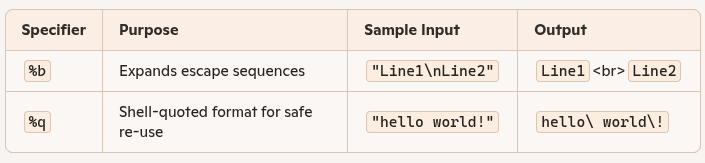
* In addition to standard format specifiers (%s, %d, %f, etc.), **Bash** (and other **POSIX-compliant shells**) supports **two special specifiers**:
  + %b
  + %q
* These are **not portable** across all UNIX environments—mainly specific to Bash.

## **🔸 %b – Escape Sequence Expansion**

* Works like %s, **but interprets backslash-escaped characters** (like in echo -e).
* Useful for converting **string literals into formatted text**.

## %q – Shell-Quoted Output

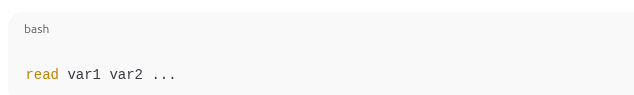
* Outputs the string quoted in a way that makes it safe for re-use in shell input.
* Escapes spaces, quotes, and special characters, making the string reusable in scripts or command-line commands.

****

**read in Bash:-**

## What is read?

* read is used to read a line of input from standard input and assign words to shell variables.
* It is the input counterpart to echo/printf, enabling interactive or file-based data intake in scripts.



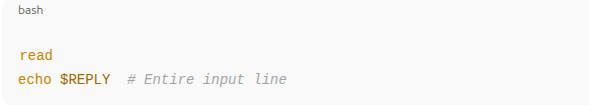
* Reads a full line and splits it into **words**, assigning them to variables.
* Word separation is controlled by the IFS variable (defaults: space, tab, newline).

Example:-



## Special Behavior

* Extra words are assigned to the last variable.
* If no variable is given, the full input is stored in the built-in variable REPLY.



## **End-of-File (EOF) Detection**

* read returns **non-zero (1)** on EOF — ideal for use in while loops:



# **I/O Redirection and Multiple Commands in Bash:**

## 

## **Problem Context**

* Sometimes we need to **redirect input** (like a config file) to **multiple commands** or control structures in a script.
* Goal: read from a file (like /etc/terms) **line by line**, **without using a subshell** (which creates a separate process and is less efficient).

## **🔸 Why Not Use a Subshell?**

* Subshells are **covered in the next chapter**, but:  
  + They **create a new process**, separating variables and side effects from the parent shell.
  + This can make **variable changes ineffective** outside the subshell.
* **Better solution**: use **in-process redirection** with Bash's built-in mechanisms.

## **🔹 Bash Provides 3 Methods for Redirection with Multiple Commands**

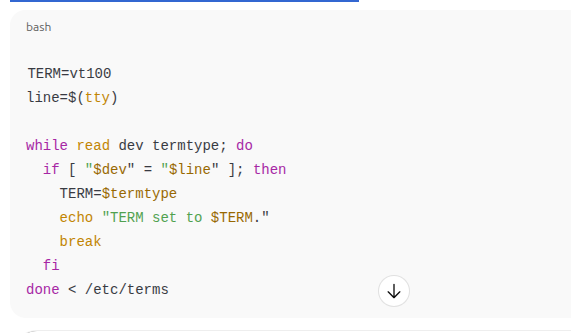
### 1. Redirect Input When Calling a Function:



* The function behaves like a standalone script.
* The redirect < /etc/terms applies to everything inside the function.
* Keeps read working correctly line by line.

2. Redirect Input at the Function Definition:



3. Redirect Input at the End of a Loop:-

Redirects input only to the while loop.

Most **common and readable** pattern.

Works for **any compound statement**, like:

* if ... fi
* case ... esac
* until ... done
* select ... done