AWK: Complete Programming Guide

1. Basic awk Commands

What is awk?

Definition:[84]

- Named after designers: Alfred V. Aho, Peter J. Weinberger, Brian W. Kernighan
- Powerful programming language for pattern processing
- Processes files line-by-line and takes actions on matched patterns
- GNU version: gawk

Command Syntax

Basic syntax:[84]

```
gawk 'COMMANDS' FILE
gawk <OPTIONS> 'COMMANDS' <FILE(S)>
```

Using external command file:[84]

```
gawk <OPTIONS> -f COMMANDFILE <FILE(S)>
gawk -F: -f script.awk student.txt
```

With field separator option:[84]

```
gawk -F: 'COMMANDS' FILE
```

Records and Fields

Records:[84]

- awk reads input file line-by-line
- Each line is called a record

Fields:[84]

- Each record is split into fields by a separator
- Default field separator: space or tab

- Specify separator with `-F` option
- Current record: `$0`
- Individual fields: `$1`, `$2`, `$3`, ...

Example:[84]

```
File: student.txt
Abhik::Das:UG:10MA20012:Algorithms,OS,Networks,ML:70,90,80

With gawk -F:
$0 = "Abhik::Das:UG:10MA20012:Algorithms,OS,Networks,ML:70,90,80"
$1 = "Abhik"
$2 = ""
$3 = "Das"
$4 = "UG"
$5 = "10MA20012"
$6 = "Algorithms,OS,Networks,ML"
$7 = "70,90,80"
```

---

2. BEGIN and END Sections with Pattern Matching

Program Structure

Complete awk program format:[84]

```
BEGIN { Initial actions }
PATTERN1 { Action1 }
PATTERN2 { Action2 }
...
PATTERNn { Actionn }
END { Final actions }
```

Execution flow:[84]

- BEGIN section executes before any record is read
- Records are processed one by one
- For each record, only matching patterns execute
- Actions taken in sequence as given
- END section executes after all records
- Empty pattern matches every record

BEGIN Section

Purpose:[84]

- Execute before reading any file
- Set initial values and print headers

Example:[84]

```
BEGIN {
    FS = ":"
    print "Reading the student database ..."
}
```

END Section

Purpose:[84]

- Execute after all records processed
- Print summary information

Example:[84]

```
END {
    print "That is all I have. Bye..."
}
```

Pattern Matching

Regular expression patterns:[84]

- Enclosed with delimiters (usually `/`)
- Any regular expression valid

Example - Pattern matching:[84]

```
BEGIN {
    FS = ":"
    nUG = 0
    nPG = 0
}
{
    if ($6 ~ /OS/) {
        if ($4 ~ /UG/) { nUG++ }
        else { nPG++ }
    }
}
END {
    print nUG " UG students have taken OS"
    print nPG " PG students have taken OS"
}
```

Output:

```
2 UG students have taken OS
2 PG students have taken OS
```

Pattern operators:[84]

- `~` : Pattern match
- `!~` : Pattern non-match

3. awk Variables

Built-in Variables

Field variables:[84]

- `$0` : Current record (entire line)
- `$1, $2, $3, ...` : Fields in current record

Record/File variables:[84]

- `NR` : Number of current record (1, 2, 3, ...)
- `NF` : Number of fields in current record
- `RS` : Record Separator (default: newline)
- `FS` : Field Separator
- `FILENAME` : Name of current file (NULL if stdin)

Example - Using NR and NF:[84]

```
{
    print "Record " NR ": has " NF " fields"
}
```

User-Defined Variables

Automatic initialization:[84]

- Not declared before use
- Numeric variables: initialized to 0
- String variables: initialized to empty string
- No fixed types

Type conversion:[84]

- Strings in numerical context: converted to number (or 0 if not numeric)
- Numbers in string context: converted to string
- String comparison: lexicographic ordering
    - As numbers: 9 < 10
    - As strings: "9" > "10"

Example - Variable usage:[84]

```
{
    dept = substr($5, 3, 2)
    if (dept == "MA") {
        nst++
    }
}
```

---

4. awk Arrays

Indexed Arrays

Array declaration and usage:[84]

- Arrays are indexed (1-based indexing)
- Elements accessed as `Array[index]`
- Can also store values while indexing

Example - Indexed array:[84]

```
BEGIN { FS = ":" }
{
    if ($4 ~ /UG/) {
        nUG++
        UG_OS[nUG] = $5
    }
}
END {
    for (i=1; i<=nUG; i++) {
        print UG_OS[i]
    }
}
```

Associative Arrays

Array indexed by strings:[84]

- Indexed with string keys
- Different: `Array[5]` vs `Array["5"]`
- Used for key-value pairs

Example - Associative array:[84]

```
BEGIN { FS = ":" }
{
    if ($4 == "UG") {
        n = split($6, ctaken, ",")
        for (i=1; i<=n; i++) {
            courses[ctaken[i]] = courses[ctaken[i]] " " $5
        }
    }
}
END {
    for (c in courses) {
        printf("%s: %s\n", c, courses[c])
    }
}
```

Output:

```
OS: 10MA20012 10CS20013
AI: 10CS20010
Algorithms: 10CS20010 10MA20012
ML: 10MA20012 10CS20013
Networks: 10CS20010 10MA20012 10CS20013
```

Looping over associative arrays:[84]

```
for (name in Array) {
    # Access entries as Array[name]
}
```

- Iterations not in sorted order of names

---

5. awk Functions

Built-in Functions

Numeric functions:[84]

- `int(x)` - Integer part of x

String functions:[84]

- `length(s)` - Length of string s
- `index(s,t)` - Index of substring t in s (0 if not found)
- `substr(s,b,l)` - Substring of s beginning at index b with length l
- `toupper(s)` - Convert string s to uppercase
- `tolower(s)` - Convert string s to lowercase

Array manipulation:[84]

- `split(s,A,d)` - Split string s by delimiter d, store parts in array A
    - Returns number of parts
    - Array indexing is 1-based

Example - Built-in functions:[84]

```
{
    dept = substr($5, 3, 2)
    if (dept == "MA") {
        printf("%s %s has taken %s\n", $1, $3, $6)
        n = split($6, ctaken, ",")
        nst++
    }
}
END {
    printf("Number of Math students: %d\n", nst)
}
```

Output:

```
Abhik Das has taken Algorithms,OS,Networks,ML
Arvind Srinivasan has taken Algorithms,OS,ML
Gautam Kumar has taken Algorithms,AI
Anusha V Pillai has taken Networks,AI
Number of Math students: 4
```

User-Defined Functions

Function definition:[84]

```
function functionName(parameter1, parameter2, ...)
{
    statements
    return value
}
```

Example - Fibonacci function:[84]

```
function F(n) {
    if (n <= 1) { return n }
    return F(n-1) + F(n-2)
}
BEGIN {
    print "Fib(8) = " F(8)
}
```

Function Variable Scope

Variable scope rules:[84]

- All variables are global
- No local variable declaration provision
- Function parameters act as local variables
- Parameter passing by value only

Making local variables:[84]

- Add local variables to parameter list
- Don't pass values to all parameters
- Unused parameters initialize to 0 or empty string

Example - Local variables:[84]

```
function oddsum(n, i, sum) {
    print "oddsum(" n ") called"
    sum = 0
    term = 1
    for (i=1; i<=n; i++) {
        sum += term
        term += 2
    }
    return sum
}
BEGIN {
    sum = 0
    for (i=1; i<=10; i++) {
        sum += oddsum(i)
    }
    print sum
}
```

Output:

```
oddsum(1) called
oddsum(2) called
...
oddsum(10) called
385
```

Runtime Input

Getting user input:[84]

```
BEGIN {
    printf("Enter a positive integer: ")
    getline n < "-"
    n = int(n)
    print "Fib(" n ") = " F(n)
}
```

Command execution:

```
$ gawk -f fib.awk
Enter a positive integer: 8
Fib(8) = 21
```

Setting Variables with -v Option

Pass values at command line:[84]

```
gawk -v n=6 -f script.awk
```

Example:[84]

```
$ gawk -v n=6 -f fib.awk
Entered argument: 6
Fib(6) = 8
```

---

6. File Operations in awk

Output Redirection

Writing to file:[84]

```
print "text" > "filename"    # Overwrite
print "text" >> "filename"   # Append
```

Example - Redirection:[84]

```
{
    dept = substr($5, 3, 2)
    if (dept == "MA") {
        printf("%s %s with roll no. %s has taken %s\n",
               $1, $3, $5, $6) >> "mathdetails.txt"
        nst++
    }
}
END {
    printf("Number of Math students: %d\n", nst)
}
```

File handling rules:[84]

- No need to open or close file explicitly
- > means overwrite (first print determines mode)
- >> means append (after first print, no distinction)
- Mode determined by first print statement
- After first print, both > and >> behave same
- Filename must be quoted

Verification:[84]

```
$ gawk -F: -f redirection.awk student.txt
Number of Math students: 4

$ cat mathdetails.txt
Abhik Das with roll no. 10MA20012 has taken Algorithms,OS,Networks,ML
Arvind Srinivasan with roll no. 10MA60012 has taken Algorithms,OS,ML
Gautam Kumar with roll no. 10MA60024 has taken Algorithms,AI
Anusha V Pillai with roll no. 09MA10001 has taken Networks,AI
```

C-like Features in awk

Similar syntax to C:[84]

- Comparison operators: `==, !=, <, <=, >, >=`
- New operators: `~` (match), `!~` (non-match), `**` (exponentiation)
- Logical operators: `&&, ||, !`
- Arithmetic operators: `+, -, *, /, %, ++, --`
- Assignment operators: `=, +=, -=, *=, /=, %=`
- Control flow: `if`, `if-else`, `while`, `for`, `break`, `continue`
- Functions: `printf` and `sprintf` work exactly like C

---

7. Quick Reference Examples

Example 1: Print with Pattern

Command:

```
$ gawk -F: '{print $1 "  " $3}' student.txt
```

Output: Names split into fields

Example 2: Conditional Action

Command:

```
$ gawk -F: '{if ($4 == "UG") print $5}' student.txt
```

Output: Roll numbers of UG students only

Example 3: Counting Records

Command:

```
$ gawk -F: '{count++} END {print "Total records: " count}' student.txt
```

Output: Total number of records

Example 4: Field Arithmetic

Command:

```
$ gawk -F: 'BEGIN {sum=0} {sum += NF} END {print "Total fields: " sum}' student.txt
```

Output: Sum of fields in all records

Complete Bash Scripting Guide

1. Shell Variables

Introduction to Variables

Variables in bash follow C-like naming conventions. They store data that can be referenced and manipulated throughout your script.[1]

Basic Syntax:[1]

```
VAR=VALUE
```

Critical Rules:

- **No spaces** allowed before or after `=`
- Access value using `$VAR` or `${VAR}`
- Case-sensitive names
- Cannot start with digits

User-Defined Variables

Creating and Using Variables:[1]

```
$ MY_NAME=Foolan
$ echo $MY_NAME
Foolan

$ MY_FULL_NAME=Foolan Barik
Barik: command not found    # Error! Space without quotes

$ MY_FULL_NAME="Foolan Barik"
$ echo $MY_FULL_NAME
Foolan Barik
```

Deleting Variables:[1]

```
$ MY_NAME="Foolan"
$ echo $MY_NAME
Foolan

$ unset MY_NAME
$ echo $MY_NAME
                    # Empty - variable deleted
```

Understanding Quotes

Bash has three types of quotes with different behaviors:[1]

1. Double Quotes (`"`) - Expand Variables

```
$ MYNAME="Foolan Barik"
$ echo "Welcome $MYNAME"
Welcome Foolan Barik

$ echo "Today is `date`"
Today is Tue Oct 28 19:39:00 IST 2025
```

2. Single Quotes (`'`) - Literal Text (No Expansion)

```
$ echo 'Welcome $MYNAME'
Welcome $MYNAME

$ echo 'Today is `date`'
Today is `date`
```

3. Backticks (`````) or `$()` - Command Substitution

```
$ FILES=`ls /`
$ echo $FILES
bin boot dev etc home lib usr var

$ FILES=$(ls /)
$ echo $FILES
bin boot dev etc home lib usr var
```

Recommendation: Use `$()` instead of backticks for better nesting.[1]

Environment Variables

Bash provides many predefined environment variables:[1]

Common Environment Variables:

Variable	Description	Example
`$HOME`	Home directory	`/home/username`
`$PATH`	Command search paths	`/usr/bin:/bin:/usr/local/bin`
`$USER`	Current username	`foolan`
`$SHELL`	Current shell	`/bin/bash`
`$PWD`	Present working directory	`/home/foolan/documents`
`$HOSTNAME`	Machine name	`mycomputer`
`$RANDOM`	Random number (0-32767)	`15234`

Viewing All Variables:[1]

```
$ set
BASH=/bin/bash
HOME=/home/foolan
PATH=/usr/local/bin:/usr/bin:/bin
USER=foolan
...
```

Special Variables (Positional Parameters)

Built-in Special Variables:[1]

Variable	Meaning
`$0`	Script/command name
`$1, $2, ... $9`	Positional arguments 1-9
`${10}, ${11}, ...`	Arguments 10+ (use braces)
`$#`	Number of arguments
`$*`	All arguments as single string
`$@`	All arguments as separate strings
`$?`	Exit status of last command

Variable	**Meaning**
`$$`	Current process ID
`$!`	Process ID of last background command

Example:[1]

```
$ parameters() {
> echo "Command: $0"
> echo "Number of arguments: $#"
> echo "All arguments (\$*): $*"
> echo "All arguments (\$@): $@"
> echo "First parameter: $1"
> echo "Second parameter: $2"
> }

$ parameters foolan barik kumar
Command: bash
Number of arguments: 3
All arguments ($*): foolan barik kumar
All arguments ($@): foolan barik kumar
First parameter: foolan
Second parameter: barik
```

Difference between `$*` and `$@`:

```
$ show_args() {
> for arg in "$*"; do
>   echo "Argument: $arg"
> done
> }

$ show_args one two three
Argument: one two three    # Treated as single argument

$ show_args2() {
> for arg in "$@"; do
>   echo "Argument: $arg"
> done
> }

$ show_args2 one two three
Argument: one
Argument: two
Argument: three            # Treated as separate arguments
```

Exporting Variables

Export variables to make them available to child processes:[1]

```
$ MYNAME=Foolan
$ bash                        # Start new shell
$ echo $MYNAME
                              # Variable not available

$ exit                        # Return to parent
$ export MYNAME
$ bash
$ echo $MYNAME
Foolan                        # Now available!
```

Combined Declaration and Export:[1]

```
$ export MY_NAME=Foolan
```

Checking Shell Level:[1]

```
$ echo $SHLVL
1
$ bash
$ echo $SHLVL
2
```

Reading User Input

Basic Input:[1]

```
$ echo -n "Enter your name: "
Enter your name: Foolan Barik
$ read MYNAME
$ echo $MYNAME
Foolan Barik
```

Inline Prompt:[1]

```
$ read -p "Enter your name: " MYNAME
Enter your name: Foolan Barik
$ echo $MYNAME
Foolan Barik
```

Multiple Variables:[1]

```
$ read -p "Enter first and last name: " FIRST LAST
Enter first and last name: Foolan Kumar Barik
$ echo $FIRST
Foolan
$ echo $LAST
Kumar Barik    # Extra words go to last variable
```

Silent Input (for passwords):

```
$ read -sp "Enter password: " PASSWORD
Enter password:
$ echo $PASSWORD
mypassword
```

Read-Only Variables

Make variables immutable using `declare -r`:[1]

```
$ MYNAME="Foolan Barik"
$ declare -r MYNAME
$ MYNAME="Someone Else"
bash: MYNAME: readonly variable

$ unset MYNAME
bash: unset: MYNAME: cannot unset: readonly variable
```

---

2. String Operations

String Length

Get the length of a string using `${#VAR}`:[1]

```
$ S="abcdefgh"
$ echo ${#S}
8

$ NAME="Foolan Barik"
$ echo "Name has ${#NAME} characters"
Name has 12 characters
```

Substring Extraction

From Position i to End:[1]

```
$ S="abcdefgh"
$ echo ${S:4}
efgh

$ echo ${S:0}
abcdefgh
```

Last i Characters (Note the space before minus):[1]

```
$ S="abcdefgh"
$ echo ${S: -4}
efgh

$ echo ${S: -1}
h
```

j Characters from Position i:[1]

```
$ S="abcdefgh"
$ echo ${S:4:4}
efgh

$ echo ${S:2:3}
cde
```

From i to j Positions from End:[1]

```
$ S="abcdefgh"
$ echo ${S:4:-2}
ef

$ echo ${S:2:-1}
bcdefg
```

String Concatenation

Simply place variables next to each other:[1]

```
$ S="abcdefgh"
$ T="ghijklmnop"
$ S="$S$T"
$ echo "$S has length ${#S}"
abcdefghghijklmnop has length 18

$ FIRST="Foolan"
$ LAST="Barik"
$ FULL="$FIRST $LAST"
$ echo $FULL
Foolan Barik
```

String Replacement

Replace First Occurrence:

```
$ S="hello world hello"
$ echo ${S/hello/hi}
hi world hello
```

Replace All Occurrences:

```
$ S="hello world hello"
$ echo ${S//hello/hi}
hi world hi
```

Remove Pattern:

```
$ FILENAME="document.txt"
$ echo ${FILENAME/.txt/}
document

$ PATH="/usr/local/bin:/usr/bin:/bin"
$ echo ${PATH//:/,}
/usr/local/bin,/usr/bin,/bin
```

String Case Conversion

Uppercase/Lowercase:

```
$ NAME="Foolan Barik"
$ echo ${NAME^^}
FOOLAN BARIK

$ echo ${NAME,,}
foolan barik

$ echo ${NAME^}
Foolan barik    # First character uppercase
```

Practical String Example

```
#!/bin/bash
# Extract filename components
FULLPATH="/home/user/documents/report.pdf"

# Get just filename
FILENAME="${FULLPATH##*/}"
echo "Filename: $FILENAME"           # report.pdf

# Get directory
DIR="${FULLPATH%/*}"
echo "Directory: $DIR"               # /home/user/documents

# Get extension
EXT="${FILENAME##*.}"
echo "Extension: $EXT"               # pdf

# Get name without extension
NAME="${FILENAME%.*}"
echo "Name: $NAME"                   # report
```

---

3. Arrays in Bash

Indexed Arrays

Declaring Arrays:[1]

```
$ declare -a MYARR
```

Setting Elements:[1]

```
$ MYARR[^0]="zero"
$ MYARR[^1]="one"
$ MYARR[^2]="two"
$ MYARR[^4]="four"    # Can skip indices
```

Quick Initialization:[1]

```
$ P=(2 3 5 7 11 13)
$ echo ${P[@]}
2 3 5 7 11 13

$ NAMES=("Alice" "Bob" "Charlie")
$ echo ${NAMES[^1]}
Bob
```

Accessing Array Elements

Individual Elements:[1]

```
$ MYARR[^0]="zero"
$ MYARR[^1]="one"
$ echo ${MYARR[^0]}
zero

$ echo ${MYARR[^1]}
one
```

All Elements:[1]

```
$ P=(2 3 5 7)
$ echo ${P[@]}
2 3 5 7

$ echo ${P[*]}
2 3 5 7
```

Array Indices:[1]

```
$ MYARR[^0]="zero"
$ MYARR[^1]="one"
$ MYARR[^2]="two"
$ MYARR[^4]="four"

$ echo ${!MYARR[@]}
0 1 2 4
```

Array Length:[1]

```
$ P=(2 3 5 7 11)
$ echo ${#P[@]}
5

$ MYARR[^0]="zero"; MYARR[^4]="four"
$ echo ${#MYARR[@]}
2
```

Array Operations

Appending Elements:[1]

```
$ P=(2 3 5 7)
$ P+=(11 13 17 19)
$ echo ${P[@]}
2 3 5 7 11 13 17 19

$ P[${#P[@]}]=23    # Append single element
$ echo ${P[@]}
2 3 5 7 11 13 17 19 23
```

Inserting at Position:[1]

```
$ P=(2 3 5 7 11 13 17 19 23 29 31 37)
$ P=(${P[@]:0:8} 21 23 29 ${P[@]:8})
$ echo ${P[@]}
2 3 5 7 11 13 17 19 21 23 29 23 29 31 37
```

Deleting Elements:[1]

```
$ P=(2 3 5 7 11 13 17 19 21 23 29 31 37)
$ unset P[^8]
$ echo ${P[@]}
2 3 5 7 11 13 17 19 23 29 31 37

$ echo ${!P[@]}
0 1 2 3 4 5 6 7 9 10 11 12    # Index 8 is missing
```

Compacting Array (Re-indexing):[1]

```
$ P=(${P[@]})
$ echo ${!P[@]}
0 1 2 3 4 5 6 7 8 9 10 11    # Continuous indices
```

Concatenating Arrays:[1]

```
$ P=(2 3 5 7 11 13)
$ Q=(17 19 23 29)
$ P=(${P[@]} ${Q[@]})
$ echo ${P[@]}
2 3 5 7 11 13 17 19 23 29
```

Array Slicing:[1]

```
$ P=(2 3 5 7 11 13 17 19 23)
$ echo ${P[@]:2:4}
5 7 11 13

$ echo ${P[@]:5}
13 17 19 23
```

Associative Arrays (Hashes)

Declaration and Initialization:[1]

```
$ declare -A MYINFO
$ MYINFO["name"]="Foolan Barik"
$ MYINFO["fname"]="Foolan"
$ MYINFO["lname"]="Barik"
$ MYINFO["cgpa"]="9.87"

$ echo ${MYINFO[fname]}
Foolan
```

Combined Declaration:[1]

```
$ declare -A MYINFO=(
>   ["name"]="Foolan Barik"
>   ["fname"]="Foolan"
>   ["lname"]="Barik"
>   ["cgpa"]="9.87"
>   ["height"]="5'08''"
> )

$ echo "${MYINFO[fname]} ${MYINFO[lname]}"
Foolan Barik
```

Accessing Keys:[1]

```
$ echo ${!MYINFO[@]}
fname height lname name cgpa

$ for key in ${!MYINFO[@]}; do
>   echo "$key: ${MYINFO[$key]}"
> done
fname: Foolan
height: 5'08''
lname: Barik
name: Foolan Barik
cgpa: 9.87
```

Practical Array Example

File: process_scores.sh

```
#!/bin/bash
# Process student scores

declare -a NAMES=("Alice" "Bob" "Charlie" "David" "Eve")
declare -a SCORES=(85 92 78 95 88)

echo "Student Scores:"
echo "---------------"

for i in ${!NAMES[@]}; do
    echo "${NAMES[$i]}: ${SCORES[$i]}"
done

# Calculate average
TOTAL=0
for score in ${SCORES[@]}; do
    TOTAL=$((TOTAL + score))
done

AVG=$((TOTAL / ${#SCORES[@]}))
echo "---------------"
echo "Average Score: $AVG"
```

Output:

```
Student Scores:
---------------
Alice: 85
Bob: 92
Charlie: 78
David: 95
Eve: 88
---------------
Average Score: 87
```

---

4. Arithmetic Operations

Integer Arithmetic

Use `$((...))` for arithmetic operations:[1]

Basic Operations:[1]

```
$ a=3
$ b=4
$ c=-5

$ echo $((a + b))
7

$ echo $((a + b * c - 6))
-23

$ z=$((a ** 2 + b ** 2))
$ echo $z
25
```

Operators:

Operator	Meaning	Example
`+`	Addition	`$((5 + 3))` → 8
`-`	Subtraction	`$((5 - 3))` → 2
`*`	Multiplication	`$((5 * 3))` → 15
`/`	Division	`$((10 / 3))` → 3
`%`	Modulo	`$((10 % 3))` → 1
`**`	Exponentiation	`$((2 ** 10))` → 1024

Note: Inside `$(( ))`, the `$` before variable names is optional:[1]

```
$ a=3; b=4
$ echo $((a + b))
7

$ echo $(($a + $b))
7
```

Increment and Decrement

Pre and Post Operations:

```
$ n=5
$ echo $((n++))
5
$ echo $n
6

$ echo $((++n))
7
$ echo $n
7

$ echo $((n--))
7
$ echo $n
6

$ echo $((--n))
5
```

Compound Assignment

```
$ a=10
$ ((a += 5))
$ echo $a
15

$ ((a *= 2))
$ echo $a
30

$ ((a /= 3))
$ echo $a
10
```

Comparison in Arithmetic Context

Use `(( ))` for numeric comparisons:

```
$ a=5
$ b=10
$ if ((a < b)); then
>   echo "a is less than b"
> fi
a is less than b

$ ((a > 3)) && echo "a is greater than 3"
a is greater than 3
```

Fibonacci Array Example

Computing Fibonacci Numbers:[1]

```
$ declare -a FIB=([^0]=0 [^1]=1)

$ n=2; FIB[$n]=$((FIB[n-1] + FIB[n-2]))
$ n=3; FIB[$n]=$((FIB[n-1] + FIB[n-2]))
$ n=4; FIB[$n]=$((FIB[n-1] + FIB[n-2]))
$ n=5; FIB[$n]=$((FIB[n-1] + FIB[n-2]))

$ echo ${FIB[@]}
0 1 1 2 3 5
```

Floating-Point Arithmetic

Bash doesn't support floating-point natively. Use `bc` calculator:[1]

Basic bc Usage:[1]

```
$ num=22
$ den=7
$ approxpi=`echo "$num / $den" | bc`
$ echo $approxpi
3
```

Setting Precision with scale:[1]

```
$ approxpi=`echo "scale=10; $num / $den" | bc`
$ echo $approxpi
3.1428571428

$ num=355
$ den=113
$ echo "scale=15; $num / $den" | bc
3.141592920353982
```

Mathematical Functions in bc:

```
# Square root
$ echo "scale=5; sqrt(2)" | bc
1.41421

# Power
$ echo "scale=5; 2^10" | bc
1024

# Sine (requires bc -l)
$ echo "scale=5; s(0)" | bc -l
0
```

Practical Arithmetic Example

File: calculator.sh

```
#!/bin/bash
# Simple calculator

read -p "Enter first number: " num1
read -p "Enter operator (+, -, *, /, %): " op
read -p "Enter second number: " num2

case $op in
    +) result=$((num1 + num2)) ;;
    -) result=$((num1 - num2)) ;;
    \*) result=$((num1 * num2)) ;;
    /)
        if [ $num2 -eq 0 ]; then
            echo "Error: Division by zero"
            exit 1
        fi
        result=$((num1 / num2))
        ;;
    %) result=$((num1 % num2)) ;;
    *)
        echo "Invalid operator"
        exit 1
        ;;
esac

echo "$num1 $op $num2 = $result"
```

---

5. Functions in Bash

Defining Functions

Two Syntaxes:[1]

```
# Syntax 1: With 'function' keyword
function FNAME() {
    commands
}

# Syntax 2: Without 'function' keyword
FNAME() {
    commands
}
```

Simple Example:[1]

```
$ function twopower() {
> echo "Usage: twopower exponent"
> echo "2 to the power $1 is $((2 ** $1))"
> }

$ twopower 10
Usage: twopower exponent
2 to the power 10 is 1024
```

Function Arguments

Functions access arguments like scripts using `$1`, `$2`, etc.:[1]

```
$ greet() {
> echo "Hello, $1!"
> if [ -n "$2" ]; then
>   echo "Welcome to $2"
> fi
> }

$ greet Foolan "Systems Programming"
Hello, Foolan!
Welcome to Systems Programming

$ greet Alice
Hello, Alice!
```

Return Values

Method 1: Using Global Variables[2]

```
#!/bin/bash
function hypotenuse() {
    local a=$1
    local b=$2
    a=$((a * a))
    b=$((b * b))
    csqr=$((a + b))    # Global variable
    c=`echo "scale=10; sqrt($csqr)" | bc`    # Global variable
}

read -p "Enter a and b: " a b
hypotenuse $a $b
echo "a = $a, b = $b, c = $c, csqr = $csqr"
```

Output:

```
$ ./hypo1.sh
Enter a and b: 5 6
a = 5, b = 6, c = 7.8102496759, csqr = 61
```

Method 2: Using Echo (Recommended)[2]

```
#!/bin/bash
function hypotenuse() {
    local a=$1
    local b=$2
    a=$((a * a))
    b=$((b * b))
    local csqr=$((a + b))
    echo `echo "scale=10; sqrt($csqr)" | bc`
}

read -p "Enter a and b: " a b
c=`hypotenuse $a $b`
echo "a = $a, b = $b, c = $c"
```

Output:

```
$ ./hypo2.sh
Enter a and b: 5 6
a = 5, b = 6, c = 7.8102496759
```

Method 3: Using return (Exit Codes Only)

```
$ is_even() {
> local num=$1
> if ((num % 2 == 0)); then
>   return 0    # Success/True
> else
>   return 1    # Failure/False
> fi
> }

$ is_even 4
$ echo $?
0

$ is_even 5
$ echo $?
1

$ if is_even 10; then
>   echo "Even number"
> fi
Even number
```

Variable Scope

Use `local` for function-local variables:[1]

```
$ x=3; y=4; z=5

$ fx() {
> local x=6
> echo "Inside fx: x = $x, y = $y, z = $z"
> }

$ fx
Inside fx: x = 6, y = 4, z = 5

$ echo "Outside: x = $x"
Outside: x = 3
```

Nested Function Calls:[1]

```
$ x=3; y=4; z=5

$ fx() {
> local x=6
> echo "In fx: x = $x, y = $y, z = $z"
> }

$ fxy() {
> local y=7
> local x=9
> echo "In fxy: x = $x, y = $y, z = $z"
> fx
> echo "Back in fxy: x = $x, y = $y"
> }

$ fxy
In fxy: x = 9, y = 7, z = 5
In fx: x = 6, y = 7, z = 5
Back in fxy: x = 9, y = 7

$ echo "Global: x = $x, y = $y, z = $z"
Global: x = 3, y = 4, z = 5
```

Practical Function Examples

Example 1: Factorial Calculator

```
#!/bin/bash
function factorial() {
    local n=$1
    if [ $n -le 1 ]; then
        echo 1
    else
        local prev=`factorial $((n - 1))`
        echo $((n * prev))
    fi
}

read -p "Enter a number: " num
result=`factorial $num`
echo "Factorial of $num is $result"
```

Example 2: String Reversal[2]

```
#!/bin/bash
function reverse() {
    local S=$1
    local Slen=${#S}
    local T

    case $Slen in
        0|1) echo "$S" ;;
        *)
            T=${S:0:-1}
            T=`reverse "$T"`
            echo "${S: -1}$T"
            ;;
    esac
}

read -p "Enter a string: " S
echo -n "reverse($S) = "
S=`reverse "$S"`
echo "$S"
```

Output:

```
$ ./reversal.sh
Enter a string: a bc def ghij klmno pqrstu
reverse(a bc def ghij klmno pqrstu) = utsrqp onmlk jihg fed cb a
```

Example 3: Checking Prime Numbers

```
#!/bin/bash
function is_prime() {
    local n=$1

    if [ $n -lt 2 ]; then
        return 1
    fi

    if [ $n -eq 2 ]; then
        return 0
    fi

    if [ $((n % 2)) -eq 0 ]; then
        return 1
    fi

    local i=3
    local limit=`echo "scale=0; sqrt($n)" | bc`

    while [ $i -le $limit ]; do
        if [ $((n % i)) -eq 0 ]; then
            return 1
        fi
        i=$((i + 2))
    done

    return 0
}

read -p "Enter a number: " num
if is_prime $num; then
    echo "$num is prime"
else
    echo "$num is not prime"
fi
```

---

6. Command Execution

Running Commands

Direct Execution:[1]

```
$ ls /
bin boot dev etc home lib usr var

$ date
Tue Oct 28 19:39:00 IST 2025

$ whoami
foolan
```

Command Substitution

Using Backticks:[1]

```
$ FILES=`ls /`
$ echo $FILES
bin boot dev etc home lib usr var

$ TODAY=`date +%Y-%m-%d`
$ echo $TODAY
2025-10-28
```

Using $() (Preferred):[1]

```
$ FILES=$(ls /)
$ echo $FILES
bin boot dev etc home lib usr var

$ USERS=$(who | wc -l)
$ echo "Number of users logged in: $USERS"
Number of users logged in: 3
```

Nested Command Substitution:

```
# With backticks (difficult to read)
$ echo `echo \`echo hello\``

# With $() (much clearer)
$ echo $(echo $(echo hello))
hello
```

Running Commands Non-Interactively

Using bash -c:[1]

```
$ echo $SHLVL
1

$ bash -c 'cal March 2023'
     March 2023
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

Checking Command Success

Exit Status:[2]

```
$ ls /
bin boot dev etc home lib usr var
$ echo $?
0    # Success

$ ls /nonexistent
ls: cannot access '/nonexistent': No such file or directory
$ echo $?
2    # Failure
```

Logical Operators

AND (&&):

```
$ mkdir testdir && cd testdir
$ pwd
/home/user/testdir

$ false && echo "This won't print"
```

OR (||):

```
$ [ -f myfile.txt ] || echo "File doesn't exist"
File doesn't exist

$ grep "pattern" file.txt || echo "Pattern not found"
```

Combining:

```
$ command1 && echo "Success" || echo "Failure"
```

Background Processes

```
# Run in background
$ long_running_command &
[^1] 12345

# Check jobs
$ jobs
[^1]+ Running    long_running_command &

# Bring to foreground
$ fg %1

# Send to background again
# Press Ctrl+Z
$ bg %1
```

7. Shell Script Basics

Creating Shell Scripts

Basic Script Structure:[2]

```
#!/bin/bash
# This is a comment
# Script name: hello.sh

echo "Hello, world!"
```

Shebang Line:

- `#!/bin/bash` - Specifies the interpreter
- Must be the first line
- Makes the script executable as a standalone program

Making Scripts Executable:[2]

```
$ chmod 755 hello.sh
# or
$ chmod +x hello.sh

$ ./hello.sh
Hello, world!
```

Script with Arguments

File: greet.sh

```
#!/bin/bash
# Greet a user

if [ $# -eq 0 ]; then
    echo "Usage: $0 NAME"
    exit 1
fi

echo "Hello, $1!"
echo "Welcome to bash scripting."
```

Running:

```
$ ./greet.sh
Usage: ./greet.sh NAME

$ ./greet.sh Foolan
Hello, Foolan!
Welcome to bash scripting.
```

Multi-Argument Scripts

File: sum.sh[2]

```
#!/bin/bash
# Sum multiple numbers

if [ $# -eq 0 ]; then
    echo "Usage: $0 num1 num2 ..."
    exit 1
fi

sum=0
for num in "$@"; do
    sum=$((sum + num))
done

echo "Sum of $# numbers: $sum"
```

Running:

```
$ ./sum.sh 10 20 30 40
Sum of 4 numbers: 100
```

Exit Codes

Setting Exit Codes:

```
#!/bin/bash
# Check if file exists

if [ ! -f "$1" ]; then
    echo "Error: File not found"
    exit 1    # Error exit code
fi

echo "File exists"
exit 0    # Success exit code
```

Standard Exit Codes:

- `0` - Success
- `1` - General error
- `2` - Misuse of shell command
- `126` - Command cannot execute
- `127` - Command not found
- `130` - Script terminated by Ctrl+C

Script Template

File: template.sh

```
#!/bin/bash
################################################################################
# Script Name: template.sh
# Description: Template for bash scripts
# Author: Your Name
# Date: 2025-10-28
# Usage: ./template.sh [options] arguments
################################################################################

# Exit on error
set -e

# Exit on undefined variable
set -u

# Constants
readonly SCRIPT_NAME=$(basename "$0")
readonly SCRIPT_DIR=$(dirname "$0")

# Functions
function usage() {
    echo "Usage: $SCRIPT_NAME [OPTIONS] ARGS"
    echo ""
    echo "Options:"
    echo "  -h, --help    Show this help message"
    echo "  -v, --verbose Verbose output"
    exit 0
}

function error_exit() {
    echo "ERROR: $1" >&2
    exit 1
}

# Parse command-line arguments
VERBOSE=0

while [ $# -gt 0 ]; do
    case $1 in
        -h|--help)
            usage
            ;;
        -v|--verbose)
            VERBOSE=1
            shift
            ;;
        *)
            echo "Unknown option: $1"
            usage
            ;;
    esac
done

# Main script logic
if [ $VERBOSE -eq 1 ]; then
    echo "Running in verbose mode"
fi

echo "Script started successfully"

exit 0
```

8. Interactive Scripts

Reading User Input

Simple Input:[2]

```
#!/bin/bash
echo -n "Enter your name: "
read name
echo "Hello, $name!"
```

Using read with prompt:

```
#!/bin/bash
read -p "Enter your age: " age
echo "You are $age years old"
```

Menu-Based Scripts

File: menu.sh

```
#!/bin/bash
# Interactive menu

while true; do
    echo ""
    echo "===== Main Menu ====="
    echo "1. List files"
    echo "2. Show date"
    echo "3. Show current directory"
    echo "4. Exit"
    echo "====================="
    read -p "Enter your choice [1-4]: " choice

    case $choice in
        1)
            echo "Files in current directory:"
            ls -l
            ;;
        2)
            echo "Current date and time:"
            date
            ;;
        3)
            echo "Current directory:"
            pwd
            ;;
        4)
            echo "Goodbye!"
            exit 0
            ;;
        *)
            echo "Invalid choice. Please try again."
            ;;
    esac

    read -p "Press Enter to continue..."
done
```

File Finder Script

File: findall.sh[2]

```
#!/bin/bash
# Find all files with given extension

echo -n "*** Enter an extension (without the dot): "
read extn

echo "*** Okay, finding all files in your home area with extension $extn"
ls -R ~ | grep "\.$extn$"

echo "*** That's all you have. Bye."
```

Running:[2]

```
$ ./findall.sh
*** Enter an extension (without the dot): tif
*** Okay, finding all files in your home area with extension tif
centralimage-1500.tif
formulas-hires.tif
Crypto.tif
*** That's all you have. Bye.
```

Confirmation Prompts

File: confirm.sh

```
#!/bin/bash
# Delete file with confirmation

if [ $# -eq 0 ]; then
    echo "Usage: $0 filename"
    exit 1
fi

filename=$1

if [ ! -f "$filename" ]; then
    echo "Error: File '$filename' not found"
    exit 1
fi

read -p "Are you sure you want to delete '$filename'? (y/n): " confirm

case $confirm in
    [yY]|[yY][eE][sS])
        rm "$filename"
        echo "File deleted successfully"
        ;;
    [nN]|[nN][oO])
        echo "Operation cancelled"
        ;;
    *)
        echo "Invalid response. Operation cancelled"
        ;;
esac
```

Password Input

File: login.sh

```
#!/bin/bash
# Simple password check

CORRECT_PASSWORD="secret123"

read -p "Username: " username
read -sp "Password: " password
echo ""

if [ "$password" = "$CORRECT_PASSWORD" ]; then
    echo "Login successful! Welcome, $username"
else
    echo "Login failed!"
    exit 1
fi
```

Multi-Field Form

File: registration.sh

```
#!/bin/bash
# User registration form

echo "===== User Registration ====="
read -p "First Name: " firstname
read -p "Last Name: " lastname
read -p "Email: " email
read -p "Age: " age
read -sp "Password: " password
echo ""
read -sp "Confirm Password: " password2
echo ""

if [ "$password" != "$password2" ]; then
    echo "Error: Passwords don't match"
    exit 1
fi

echo ""
echo "===== Registration Summary ====="
echo "Name: $firstname $lastname"
echo "Email: $email"
echo "Age: $age"
echo ""
read -p "Confirm registration? (y/n): " confirm

if [[ $confirm =~ ^[Yy]$ ]]; then
    echo "Registration successful!"
    # Save to file or database here
else
    echo "Registration cancelled"
fi
```

---

9. Running Other Programs

Executing External Commands

Direct Execution:

```
#!/bin/bash
# Run external programs

# List files
ls -la

# Show disk usage
df -h

# Show processes
ps aux | head -n 10
```

Capturing Command Output

Storing in Variables:

```
#!/bin/bash
# Capture output

FILES=$(ls *.txt)
echo "Text files: $FILES"

USER_COUNT=$(who | wc -l)
echo "Users logged in: $USER_COUNT"

DISK_FREE=$(df -h / | awk 'NR==2 {print $4}')
echo "Free disk space: $DISK_FREE"
```

Checking if Commands Exist

```
#!/bin/bash
# Check if command exists

if command -v git &> /dev/null; then
    echo "Git is installed"
    git --version
else
    echo "Git is not installed"
fi

# Alternative method
if which python3 > /dev/null 2>&1; then
    echo "Python3 is available"
else
    echo "Python3 is not available"
fi
```

Running with Specific Environment

```
#!/bin/bash
# Run command with custom environment

# Set temporary environment variable
export LANG=C
date

# Run command in modified PATH
PATH="/custom/path:$PATH" mycommand

# Run with clean environment
env -i HOME="$HOME" bash -c 'echo $PATH'
```

Timeout for Commands

```
#!/bin/bash
# Run command with timeout

timeout 5s long_running_command
if [ $? -eq 124 ]; then
    echo "Command timed out"
fi
```

Piping Between Programs

```
#!/bin/bash
# Chain commands

# Count .txt files
ls *.txt | wc -l

# Find and sort
find . -name "*.log" | sort | head -n 10

# Process CSV
cat data.csv | grep "pattern" | cut -d',' -f1,3 | sort -u
```

Running Programs in Parallel

```
#!/bin/bash
# Run multiple commands in parallel

command1 &
PID1=$!

command2 &
PID2=$!

command3 &
PID3=$!

# Wait for all to complete
wait $PID1 $PID2 $PID3

echo "All commands completed"
```

Compiling and Running C Programs

```
#!/bin/bash
# Compile and run C program

SOURCE="program.c"
BINARY="program"

if [ ! -f "$SOURCE" ]; then
    echo "Error: Source file not found"
    exit 1
fi

echo "Compiling $SOURCE..."
gcc -Wall -g -o "$BINARY" "$SOURCE"

if [ $? -ne 0 ]; then
    echo "Compilation failed"
    exit 1
fi

echo "Compilation successful"
echo "Running $BINARY..."
./"$BINARY"
```

10. Conditionals

if Statements

Basic Syntax:[2]

```
if condition; then
    commands
fi
```

With else:[2]

```
if condition; then
    commands1
else
    commands2
fi
```

With elif:[2]

```
if condition1; then
    commands1
elif condition2; then
    commands2
else
    commands3
fi
```

Numeric Comparisons

Test Operators:[2]

Operator	**Meaning**
`-eq`	Equal to
`-ne`	Not equal to
`-lt`	Less than
`-le`	Less than or equal
`-gt`	Greater than
`-ge`	Greater than or equal

Examples:[2]

```
$ x=3; y=4; z=5

$ if [ $y -gt $x ]; then
>   echo "$y is greater than $x"
> fi
4 is greater than $x

$ if [ $((x**2 + y**2)) -eq $((z**2)) ]; then
>   echo "Pythagorean triple!"
> fi
Pythagorean triple!
```

Arithmetic Context (Preferred for Numbers):

```
#!/bin/bash
a=10
b=20

if ((a < b)); then
    echo "a is less than b"
fi

if ((a > 5 && a < 15)); then
    echo "a is between 5 and 15"
fi
```

String Comparisons

Operators:[2]

Operator	**Meaning**
`=` or `==`	Strings are equal
`!=`	Strings are not equal
`<`	Less than (alphabetically)
`>`	Greater than (alphabetically)
`-z`	String is empty
`-n`	String is not empty

Examples:[2]

```
$ x="Foolan"
$ y="Foolan Barik"

$ if [ "$x" == "$y" ]; then
>   echo "Same"
> else
>   echo "Different"
> fi
Different

$ if [ -z "$z" ]; then
>   echo "Variable z is empty"
> fi
Variable z is empty

$ if [ -n "$x" ]; then
>   echo "Variable x is not empty"
> fi
Variable x is not empty
```

Pattern Matching:

```
#!/bin/bash
filename="document.pdf"

if [[ $filename == *.pdf ]]; then
    echo "PDF file detected"
fi

if [[ $filename =~ ^[a-z]+\. ]]; then
    echo "Starts with lowercase letters"
fi
```

File Tests

Common File Tests:[2]

Test	Meaning
`-e FILE`	File exists
`-f FILE`	Regular file exists
`-d FILE`	Directory exists
`-L FILE`	Symbolic link exists
`-r FILE`	File is readable
`-w FILE`	File is writable
`-x FILE`	File is executable
`-s FILE`	File exists and is not empty
`-N FILE`	File modified since last read
`FILE1 -nt FILE2`	FILE1 is newer than FILE2
`FILE1 -ot FILE2`	FILE1 is older than FILE2

Example Script:[2]

```
#!/bin/bash
# File attribute checker

if [ -z "$1" ]; then
    echo "Usage: $0 filename"
    exit 1
fi

fname=$1

if [ -e "$fname" ]; then
    echo "\"$fname\" exists"
else
    echo "\"$fname\" does not exist"
    exit 0
fi

if [ -f "$fname" ]; then
    echo "\"$fname\" is a regular file"
fi

if [ -d "$fname" ]; then
    echo "\"$fname\" is a directory"
fi

if [ -L "$fname" ]; then
    echo "\"$fname\" is a symbolic link"
fi

echo -n "Permissions:"
[ -r "$fname" ] && echo -n " read"
[ -w "$fname" ] && echo -n " write"
[ -x "$fname" ] && echo -n " execute"
echo ""

if [ -s "$fname" ]; then
    size=$(stat -f%z "$fname" 2>/dev/null || stat -c%s "$fname")
    echo "File size: $size bytes"
else
    echo "File is empty"
fi
```

Logical Operators

AND (`&&` or `-a`):

```
$ x=5
$ if [ $x -gt 0 ] && [ $x -lt 10 ]; then
>   echo "x is between 0 and 10"
> fi
x is between 0 and 10

# Alternative
$ if [ $x -gt 0 -a $x -lt 10 ]; then
>   echo "x is between 0 and 10"
> fi
```

OR (`||` or `-o`):

```
$ x=15
$ if [ $x -lt 0 ] || [ $x -gt 10 ]; then
>   echo "x is outside 0-10 range"
> fi
x is outside 0-10 range
```

NOT (`!`):

```
$ if [ ! -f "nonexistent.txt" ]; then
>   echo "File does not exist"
> fi
File does not exist
```

case Statements

Syntax:[2]

```
case value in
    pattern1)
        commands
        ;;
    pattern2)
        commands
        ;;
    *)
        default_commands
        ;;
esac
```

Example:[2]

```
#!/bin/bash
# File extension checker

if [ $# -eq 0 ]; then
    echo "Usage: $0 filename"
    exit 1
fi

filename=$1

case $filename in
    *.txt)
        echo "Text file"
        ;;
    *.jpg|*.jpeg|*.png|*.gif)
        echo "Image file"
        ;;
    *.pdf)
        echo "PDF document"
        ;;
    *.sh)
        echo "Shell script"
        ;;
    *)
        echo "Unknown file type"
        ;;
esac
```

Pattern Matching in case:

```
#!/bin/bash
# Grade calculator

read -p "Enter your score: " score

case $score in
    [^9][0-9]|100)
        echo "Grade: A"
        ;;
    [^8][0-9])
        echo "Grade: B"
        ;;
    [^7][0-9])
        echo "Grade: C"
        ;;
    [^6][0-9])
        echo "Grade: D"
        ;;
    *)
        echo "Grade: F"
        ;;
esac
```

Practical Conditional Examples

Example: File Backup Script

```
#!/bin/bash
# Backup file with checks

if [ $# -ne 2 ]; then
    echo "Usage: $0 source_file backup_dir"
    exit 1
fi

source=$1
backup_dir=$2

if [ ! -f "$source" ]; then
    echo "Error: Source file doesn't exist"
    exit 1
fi

if [ ! -d "$backup_dir" ]; then
    echo "Backup directory doesn't exist. Creating..."
    mkdir -p "$backup_dir"
fi

backup_name="$backup_dir/$(basename "$source").$(date +%Y%m%d_%H%M%S)"

if cp "$source" "$backup_name"; then
    echo "Backup created: $backup_name"
else
    echo "Error: Backup failed"
    exit 1
fi
```

11. Loops

for Loops

List-Based for Loop:[2]

```
# Simple list
for item in one two three four five; do
    echo $item
done

# Brace expansion
for n in {1..10}; do
    echo $n
done

# Step with brace expansion
for n in {0..100..10}; do
    echo $n
done
```

Iterating Over Arguments:[2]

```
#!/bin/bash
# Process all arguments

for arg in "$@"; do
    echo "Processing: $arg"
done

# Shorter form (implicit)
for arg; do
    echo "Processing: $arg"
done
```

Iterating Over Files:

```
#!/bin/bash
# Process all .txt files

for file in *.txt; do
    if [ -f "$file" ]; then
        echo "Processing $file"
        wc -l "$file"
    fi
done
```

C-Style for Loop:[2]

```
for (( i=0; i<10; i++ )); do
    echo $i
done

# Multiple variables
for (( i=0, j=10; i<10; i++, j-- )); do
    echo "i=$i, j=$j"
done
```

while Loops

Basic while Loop:[2]

```
n=0
while [ $n -lt 10 ]; do
    echo $n
    n=$((n + 1))
done
```

Arithmetic Condition:[2]

```
i=0
while (( i < 10 )); do
    echo $i
    (( i++ ))
done
```

Reading from File:

```
#!/bin/bash
# Read file line by line

while read -r line; do
    echo "Line: $line"
done < input.txt
```

Infinite Loop:

```
while true; do
    echo "Running..."
    sleep 1
done

# Or
while :; do
    echo "Running..."
    sleep 1
done
```

until Loops

Basic until Loop:[2]

```
n=0
until [ $n -eq 10 ]; do
    echo $n
    n=$((n + 1))
done
```

Arithmetic Condition:

```
i=0
until (( i >= 10 )); do
    echo $i
    (( i++ ))
done
```

Loop Control

break - Exit Loop:[2]

```
for i in {1..100}; do
    if [ $i -eq 50 ]; then
        echo "Breaking at $i"
        break
    fi
    echo $i
done
```

continue - Skip Iteration:

```
for i in {1..10}; do
    if [ $((i % 2)) -eq 0 ]; then
        continue    # Skip even numbers
    fi
    echo $i
done
```

Nested Loops with break:

```
#!/bin/bash
# Break out of nested loop

found=0
for i in {1..10}; do
    for j in {1..10}; do
        if [ $((i * j)) -eq 24 ]; then
            echo "Found: $i x $j = 24"
            found=1
            break 2    # Break out of both loops
        fi
    done
    if [ $found -eq 1 ]; then
        break
    fi
done
```

Practical Loop Examples

Example 1: Fibonacci Calculator[2]

```
#!/bin/bash
# Compute Fibonacci numbers iteratively

function computerest() {
    local n=$1
    while [ $n -le $2 ]; do
        F[$n]=$((F[n-1] + F[n-2]))
        n=$((n + 1))
    done
}

declare -ia F=([^0]=0 [^1]=1)
N=1

while true; do
    read -p "Enter n: " n

    if [ $n -lt 0 ]; then
        echo "Enter a positive integer please"
        continue
    fi

    if [ $n -gt $N ]; then
        echo "Computing F($((N+1))) through F($n)"
        computerest $((N+1)) $n
        N=$n
    fi

    echo "F($n) = ${F[$n]}"

    read -p "Repeat (y/n)? " resp
    case $resp in
        [nN]*) echo "Bye..."; exit 0 ;;
    esac
done
```

Example 2: Sum Positive and Negative Separately[2]

```
#!/bin/bash
# Sum positive and negative integers separately

if [ $# -eq 0 ]; then
    echo "Usage: $0 num1 num2 ..."
    exit 1
fi

pos_sum=0
neg_sum=0

for num in "$@"; do
    if [ $num -gt 0 ]; then
        pos_sum=$((pos_sum + num))
    elif [ $num -lt 0 ]; then
        neg_sum=$((neg_sum + num))
    fi
done

echo "Sum of positive numbers: $pos_sum"
echo "Sum of negative numbers: $neg_sum"
```

Example 3: Directory Tree Explorer[2]

```
#!/bin/bash
# Explore directory tree recursively

function exploredir() {
    local currentdir=$1
    local currentlev=$2
    local lev=0

    # Print indentation
    while [ $lev -lt $currentlev ]; do
        echo -n "    "
        lev=$((lev + 1))
    done

    echo -n "$currentdir"

    if [ ! -r "$currentdir" ] || [ ! -x "$currentdir" ]; then
        echo " [Unable to explore further]"
    else
        echo ""
        for entry in "$currentdir"/*; do
            if [ -d "$entry" ]; then
                exploredir "$entry" $((currentlev + 1))
            fi
        done
    fi
}

if [ $# -eq 0 ]; then
    rootdir="."
else
    rootdir="$1"
fi

if [ ! -d "$rootdir" ]; then
    echo "$rootdir is not a directory"
    exit 1
fi

exploredir "$rootdir" 0
```

Example 4: Find Files Modified in Last N Days

```
#!/bin/bash
# Find files modified in last N days

if [ $# -ne 2 ]; then
    echo "Usage: $0 directory days"
    exit 1
fi

dir=$1
days=$2

if [ ! -d "$dir" ]; then
    echo "Error: $dir is not a directory"
    exit 1
fi

echo "Files modified in last $days days:"
find "$dir" -type f -mtime -$days | while read -r file; do
    echo "  $file"
done
```

---

12. File Processing in Bash

Reading Files Line by Line

Method 1: Using while read[2]

```
#!/bin/bash
# Read file into array

if [ $# -eq 0 ]; then
    echo "Usage: $0 filename"
    exit 1
fi

fname=$1

if [ ! -f "$fname" ] || [ ! -r "$fname" ]; then
    echo "Cannot read file: $fname"
    exit 1
fi

echo -n "Reading file $fname: "
L=()
while read -r line; do
    L+=("$line")
done < "$fname"

echo "${#L[@]} lines read"

# Print lines
for i in "${!L[@]}"; do
    echo "Line $((i+1)): ${L[$i]}"
done
```

Method 2: Using mapfile/readarray

```
#!/bin/bash
# Read file using mapfile

mapfile -t lines < filename.txt

for line in "${lines[@]}"; do
    echo "$line"
done
```

Processing Multiple Files

File: file2array.sh[2]

```
#!/bin/bash
# Read multiple files

if [ $# -lt 1 ]; then
    echo "Usage: $0 file1 [file2 ...]"
    exit 1
fi

for fname in "$@"; do
    if [ ! -f "$fname" ] || [ ! -r "$fname" ]; then
        echo "--- Unable to read $fname"
        continue
    fi

    echo -n "+++ Reading file $fname: "
    L=()
    while read -r line; do
        L+=("$line")
    done < "$fname"

    echo "${#L[@]} lines read"
done
```

File Statistics

```bash
#!/bin/bash
# Analyze text file

if [ $# -eq 0 ]; then
    echo "Usage: $0 filename"
    exit 1
fi

file=$1

if [ ! -f "$file" ]; then
    echo "File not found: $file"
    exit 1
fi

lines=$(wc -l < "$file")
words=$(wc -w < "$file")
chars=$(wc -c < "$file")

echo "File: $file"
echo "Lines: $lines"
echo "Words: $words"
echo "Characters: $chars"
```

CSV File Processing

```bash
#!/bin/bash
# Process CSV file

csv_file="data.csv"

# Skip header and process data
tail -n +2 "$csv_file" | while IFS=',' read -r name age city; do
    echo "Name: $name, Age: $age, City: $city"
done
```

File Searching and Filtering

```
#!/bin/bash
# Search for pattern in files

if [ $# -lt 2 ]; then
    echo "Usage: $0 pattern file1 [file2 ...]"
    exit 1
fi

pattern=$1
shift

for file in "$@"; do
    if [ -f "$file" ]; then
        count=$(grep -c "$pattern" "$file")
        echo "$file: $count matches"
    fi
done
```

File Backup Script

```
#!/bin/bash
# Backup files with specific extension

if [ $# -ne 2 ]; then
    echo "Usage: $0 extension backup_dir"
    exit 1
fi

ext=$1
backup_dir=$2

mkdir -p "$backup_dir"

for file in *."$ext"; do
    if [ -f "$file" ]; then
        cp "$file" "$backup_dir/"
        echo "Backed up: $file"
    fi
done

echo "Backup complete to $backup_dir"
```

Log File Analysis

```bash
#!/bin/bash
# Analyze log file

logfile="/var/log/syslog"

if [ ! -r "$logfile" ]; then
    echo "Cannot read log file"
    exit 1
fi

echo "=== Log Analysis ==="
echo "Total lines: $(wc -l < "$logfile")"
echo "Error count: $(grep -c ERROR "$logfile")"
echo "Warning count: $(grep -c WARNING "$logfile")"
echo ""
echo "Recent errors:"
grep ERROR "$logfile" | tail -n 5
```

File Comparison

```bash
#!/bin/bash
# Compare two files

if [ $# -ne 2 ]; then
    echo "Usage: $0 file1 file2"
    exit 1
fi

file1=$1
file2=$2

if [ ! -f "$file1" ] || [ ! -f "$file2" ]; then
    echo "One or both files not found"
    exit 1
fi

if cmp -s "$file1" "$file2"; then
    echo "Files are identical"
else
    echo "Files are different"
    echo ""
    echo "Differences:"
    diff "$file1" "$file2"
fi
```

Find and Process Files

```
#!/bin/bash
# Find files and process them

# Find all .txt files and count lines
find . -name "*.txt" -type f | while read -r file; do
    lines=$(wc -l < "$file")
    echo "$file: $lines lines"
done

# Find large files
echo ""
echo "Files larger than 1MB:"
find . -type f -size +1M -exec ls -lh {} \; | awk '{print $9, $5}'
```

File Renaming

```
#!/bin/bash
# Rename files by replacing spaces with underscores

for file in *\ *; do
    if [ -f "$file" ]; then
        newname=$(echo "$file" | tr ' ' '_')
        mv "$file" "$newname"
        echo "Renamed: $file -> $newname"
    fi
done
```

Directory Processing

```
#!/bin/bash
# List only subdirectories

for entry in *; do
    if [ -d "$entry" ]; then
        echo "$entry"
    fi
done

# Alternative using find
find . -maxdepth 1 -type d -not -name "."
```

File Content Transformation

```
#!/bin/bash
# Convert file to uppercase

if [ $# -ne 1 ]; then
    echo "Usage: $0 filename"
    exit 1
fi

input=$1
output="${input%.txt}_upper.txt"

tr '[:lower:]' '[:upper:]' < "$input" > "$output"
echo "Created: $output"
```

Monitoring File Changes

```
#!/bin/bash
# Monitor file for changes

if [ $# -ne 1 ]; then
    echo "Usage: $0 filename"
    exit 1
fi

file=$1

if [ ! -f "$file" ]; then
    echo "File not found: $file"
    exit 1
fi

old_md5=$(md5sum "$file" | cut -d' ' -f1)

while true; do
    sleep 5
    new_md5=$(md5sum "$file" | cut -d' ' -f1)

    if [ "$old_md5" != "$new_md5" ]; then
        echo "File changed at $(date)"
        old_md5=$new_md5
    fi
done
```

Summary

This comprehensive guide covers all essential aspects of Bash scripting:

Variables - User-defined, environment, and special variables with proper quoting and scoping

String Operations - Length, substring extraction, concatenation, replacement, and case conversion

Arrays - Indexed and associative arrays with full manipulation capabilities

Arithmetic - Integer operations with `$(( ))` and floating-point with `bc`

Functions - Reusable code blocks with local/global scope and multiple return methods

Command Execution - Running programs, command substitution, piping, and parallel execution

Scripts - Proper structure with shebang, arguments, and exit codes

Interactive Scripts - User input, menus, confirmations, and forms

Running Programs - External command execution, output capture, and environment control

Conditionals - Numeric, string, and file tests with `if`, `elif`, `else`, and `case`

Loops - `for`, `while`, `until` loops with control flow (`break`, `continue`)

File Processing - Reading, writing, analyzing, transforming, and monitoring files

By mastering these concepts and practicing with the provided examples, you can write robust, maintainable bash scripts for system administration, automation, data processing, and complex workflows.[1]

⁂

perplexity

Complete Valgrind Guide: Memory Debugging Tool

Introduction to Valgrind

What is Valgrind?

Valgrind is **not** a "value grinder" - it's named after the gate to Valhalla (Hall of the Slain) in Norse mythology. It is a powerful memory debugging tool capable of detecting many common memory-related errors and problems.[1]

Primary Uses:[1]

- Memory leak detection
- Invalid memory access detection
- Memory profiling
- In this guide, we focus on the **memcheck** feature

Basic Usage

Running Valgrind

Basic Syntax:[1]

```
valgrind executable [command line options]
```

Example:[1]

```
valgrind ./a.out 2022 -name "Sad Tijihba"
```

Understanding Valgrind Output

Sample Output (Clean Program):[1]

```
==4825== Memcheck, a memory error detector
==4825== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4825== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4825== Command: ./a.out
==4825==
[Your Program's Input/Output Here]
==4825==
==4825== HEAP SUMMARY:
==4825==     in use at exit: 0 bytes in 0 blocks
==4825==   total heap usage: 23 allocs, 23 frees, 1,376 bytes allocated
==4825==
==4825== All heap blocks were freed -- no leaks are possible
==4825==
==4825== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Key Points:

- The number `==4825==` is the process ID[1]
- `23 allocs, 23 frees` means all allocated memory was freed (perfect!)
- `0 errors` indicates no memory problems detected

Common Command-Line Options

1. Leak Check Options

`--leak-check=full`

Shows detailed information about memory leaks:[3]

```
valgrind --leak-check=full ./myprogram
```

`--show-leak-kinds=all`

Displays all types of memory leaks:[4]

```
valgrind --leak-check=full --show-leak-kinds=all ./myprogram
```

2. Tracking Uninitialized Values

`--track-origins=yes`

Tracks where uninitialized values come from:[5][6]

```
valgrind --track-origins=yes ./myprogram
```

Example Output:

```
==30384== Conditional jump or move depends on uninitialised value(s)
==30384==    at 0x400580: main (foo.c:10)
==30384==  Uninitialised value was created by a heap allocation
==30384==    at 0x4C2C66F: malloc (vg_replace_malloc.c:270)
==30384==    by 0x400551: func1 (foo.c:4)
```

This helps you find exactly where the uninitialized value was created.

3. Output Control

`--log-file=filename`

Saves Valgrind output to a file:[3]

```
valgrind --log-file=output.txt ./myprogram
```

Using special patterns:

```
valgrind --log-file="valgrind-%p.log" ./myprogram
```

Where `%p` is replaced by the process ID.

`-q` or **`--quiet`**

Suppresses informational messages, shows only errors:[7]

```
valgrind -q --leak-check=full ./myprogram
```

4. Error Handling

`--error-exitcode=number`

Returns specified exit code when errors are found:[8]

```
valgrind --error-exitcode=99 ./myprogram
```

Useful for automated testing and CI/CD pipelines.

`--num-callers=number`

Controls stack trace depth (default: 12):[9]

```
valgrind --num-callers=20 ./myprogram
```

Memory Leak Categories

Valgrind classifies unfreed memory into four categories:[101]

1. **Definitely Lost**

Memory that is no longer accessible - **true memory leaks**[1]

2. **Indirectly Lost**

Memory accessible only through pointers in "definitely lost" blocks[1]

3. **Possibly Lost**

Memory accessible only via interior pointers (not pointing to the start)[1]

4. **Still Reachable**

Memory that wasn't freed but is still accessible at program exit - not errors, but cleanup opportunities[1]

Practical Examples from PDF

Example 1: Linked List - Delete Without Freeing (Memory Leak)

Problematic Code:[1]

```
node *ldel(node *L, int x) {
    node *p;
    p = L;
    while (p->next) {
        if (p->next->data == x) {
            p->next = p->next->next;  // Memory leak! Node not freed
            return L;
        }
        if (p->next->data > x) break;
        p = p->next;
    }
    return L;
}
```

Valgrind Output:[1]

```
==9837== HEAP SUMMARY:
==9837==     in use at exit: 144 bytes in 9 blocks
==9837==   total heap usage: 10 allocs, 1 frees, 1,168 bytes allocated
==9837==
==9837== LEAK SUMMARY:
==9837==   definitely lost: 48 bytes in 3 blocks
==9837==   indirectly lost: 32 bytes in 2 blocks
==9837==      possibly lost: 0 bytes in 0 blocks
==9837==    still reachable: 64 bytes in 4 blocks
```

Analysis: The program allocated 10 blocks but freed only 1, resulting in definitely lost and indirectly lost memory.

Example 2: Linked List - Delete With Proper Freeing

Corrected Code:[1]

```
node *ldel(node *L, int x) {
    node *p, *q;
    p = L;
    while (p->next) {
        if (p->next->data == x) {
            q = p->next;
            p->next = q->next;
            free(q);  // Properly freed!
            return L;
        }
        if (p->next->data > x) break;
        p = p->next;
    }
    return L;
}
```

Valgrind Output:[1]

```
==10012== HEAP SUMMARY:
==10012==     in use at exit: 64 bytes in 4 blocks
==10012==   total heap usage: 10 allocs, 6 frees, 1,168 bytes allocated
==10012==
==10012== LEAK SUMMARY:
==10012==   definitely lost: 0 bytes in 0 blocks
==10012==   indirectly lost: 0 bytes in 0 blocks
==10012==      possibly lost: 0 bytes in 0 blocks
==10012==    still reachable: 64 bytes in 4 blocks
```

Analysis: No definitely/indirectly lost memory! The remaining blocks are "still reachable" (the list itself).

Example 3: Freeing All Nodes - Perfect Cleanup

Complete Cleanup Code:[1]

```
void ldestroy(node *L) {
    node *p;
    while (L) {
        p = L->next;
        free(L);
        L = p;
    }
}
```

Valgrind Output:[1]

```
==10160== HEAP SUMMARY:
==10160==     in use at exit: 0 bytes in 0 blocks
==10160==   total heap usage: 10 allocs, 10 frees, 1,168 bytes allocated
==10160==
==10160== All heap blocks were freed -- no leaks are possible
==10160==
==10160== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Analysis: Perfect! All allocated memory was freed. This is the ideal output.

Example 4: Possibly Lost Memory

Problematic Code:[1]

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p, *q;
    p = (int *)malloc(10 * sizeof(int));
    q = p + 5;  // q points to middle of the block
    p = (int *)malloc(5 * sizeof(int));  // p now points elsewhere!
    exit(0);
}
```

Valgrind Output:[1]

```
==4155== HEAP SUMMARY:
==4155==     in use at exit: 60 bytes in 2 blocks
==4155==   total heap usage: 2 allocs, 0 frees, 60 bytes allocated
==4155==
==4155== LEAK SUMMARY:
==4155==   definitely lost: 0 bytes in 0 blocks
==4155==   indirectly lost: 0 bytes in 0 blocks
==4155==      possibly lost: 40 bytes in 1 blocks
==4155==    still reachable: 20 bytes in 1 blocks
```

Analysis: The first block (10 ints = 40 bytes) is "possibly lost" because only an interior pointer `q` points to it. Valgrind can't be sure if this is intentional.

Example 5: Array Overflow - Invalid Write

Buggy Code:[1]

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n = 16, i, *A;
    A = (int *)malloc(n * sizeof(int));
    printf("A starts at %p, and ends at %p\n", A, A+n-1);

    // Off-by-one error: should be i < n or i = 0 to n-1
    for (i = 1; i <= n; ++i) A[i] = i * i;
    for (i = 1; i <= n; ++i) printf("%d ", A[i]);

    printf("\n");
    free(A);
    exit(0);
}
```

Valgrind Output:[1]

```
==13180== Invalid write of size 4
==13180==    at 0x109240: main (in /home/abhij/.../a.out)
==13180==  Address 0x4a5a080 is 0 bytes after a block of size 64 alloc'd
==13180==    at 0x483B7F3: malloc (in .../vgpreload_memcheck-amd64-linux.so)
==13180==    by 0x1091EC: main (in /home/abhij/.../a.out)
==13180==
==13180== Invalid read of size 4
==13180==    at 0x10926B: main (in /home/abhij/.../a.out)
==13180==  Address 0x4a5a080 is 0 bytes after a block of size 64 alloc'd
```

Analysis: Array indices go from `A` to `A[^14]`, but the loop uses `A[^1]` to `A[^12]`. Writing to `A[^12]` is out of bounds!

Fix:

```
for (i = 0; i < n; ++i) A[i] = (i+1) * (i+1);
for (i = 0; i < n; ++i) printf("%d ", A[i]);
```

Example 6: Buffer Overflow - String Operations

Vulnerable Code:[1]

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char *wnote = malloc(32);
    if (argc == 1) exit(1);

    printf("The input has size %ld\n", strlen(argv[^1]));
    printf("wnote starts at %p and ends at %p\n", wnote, wnote + 31);

    sprintf(wnote, "Welcome to %s", argv[^1]);  // Potential overflow!
    printf("%s\n", wnote);

    free(wnote);
    exit(0);
}
```

Running with long input:

```
valgrind ./a.out "Systems Programming Laboratory"
```

Valgrind Output:[1]

```
The input has size 30
wnote starts at 0x4a5a040 and ends at 0x4a5a05f
==12432== Invalid write of size 1
==12432==    at 0x483EF64: sprintf (in .../vgpreload_memcheck-amd64-linux.so)
==12432==  Address 0x4a5a060 is 0 bytes after a block of size 32 alloc'd
==12432==
==12432== Invalid write of size 1
==12432==  Address 0x4a5a069 is 9 bytes after a block of size 32 alloc'd
```

Analysis: "Welcome to " = 11 chars + "Systems Programming Laboratory" = 30 chars + null terminator = 42 bytes total, but only 32 bytes allocated!

Fix:

```
char *wnote = malloc(strlen(argv[^1]) + 12);  // "Welcome to " + input + '\0'
sprintf(wnote, "Welcome to %s", argv[^1]);
```

Or use safer functions:

```
snprintf(wnote, 32, "Welcome to %s", argv[^1]);  // Prevents overflow
```

Additional Useful Options

Compilation Best Practices

Compile with debugging symbols for better error reports:[11]

```
gcc -g -O1 myprogram.c -o myprogram
```

- `-g`: Adds debugging information (line numbers, function names)
- `-O1`: Light optimization that doesn't interfere with debugging

Combining Options

Comprehensive memory check:

```
valgrind --leak-check=full \
         --show-leak-kinds=all \
         --track-origins=yes \
         --log-file=valgrind-report.txt \
         ./myprogram
```

Automated testing:

```
valgrind -q --error-exitcode=1 --leak-check=full ./myprogram
if [ $? -eq 1 ]; then
    echo "Memory errors detected!"
fi
```

Quick Reference Table

Option	Purpose	Example
`--leak-check=full`	Detailed leak information	`valgrind --leak-check=full ./prog`
`--track-origins=yes`	Track uninitialized values	`valgrind --track-origins=yes ./prog`
`--log-file=<file>`	Save output to file	`valgrind --log-file=out.txt ./prog`
`-q` or `--quiet`	Show only errors	`valgrind -q ./prog`
`--error-exitcode=N`	Exit code on errors	`valgrind --error-exitcode=99 ./prog`
`--num-callers=N`	Stack trace depth	`valgrind --num-callers=20 ./prog`
`--show-leak-kinds=all`	Show all leak types	`valgrind --show-leak-kinds=all ./prog`

Best Practices

1. **Always compile with `-g`** for meaningful error messages[12]
2. **Fix errors in order** - later errors may be caused by earlier ones[2]

3. **Aim for zero errors** - especially "definitely lost" and "invalid" errors[1]
4. **Free all allocated memory** before program exits[1]
5. **Use array bounds carefully** - respect allocated sizes[1]
6. **Be careful with string operations** - check buffer sizes[1]

Common Pitfalls to Avoid

Off-by-one errors:

```
// Wrong: accesses array[n]
for (i = 0; i <= n; i++) array[i] = value;

// Correct: accesses array[^0] to array[n-1]
for (i = 0; i < n; i++) array[i] = value;
```

Losing pointer references:

```
// Wrong: original pointer lost
int *p = malloc(100);
p = malloc(200);  // First block leaked!

// Correct: free before reassigning
int *p = malloc(100);
free(p);
p = malloc(200);
```

Interior pointers:

```
// Risky: only interior pointer remains
int *p = malloc(10 * sizeof(int));
p = p + 5;  // Lost beginning of block!

// Better: keep original pointer
int *p = malloc(10 * sizeof(int));
int *q = p + 5;  // Use q, keep p for free()
```

Summary

Valgrind is an essential tool for C/C++ programmers to detect and fix memory errors. It helps identify:[1]

- **Memory leaks** (allocated but not freed memory)
- **Invalid memory access** (reading/writing outside allocated bounds)
- **Uninitialized value usage**

- **Double-free errors**

By running your programs through Valgrind and fixing all reported errors, you can write more robust, reliable, and correct code. The goal is to achieve the perfect output: **"All heap blocks were freed -- no leaks are possible"** with **"0 errors from 0 contexts"**.[1]

Start simple with `valgrind ./myprogram`, then add options like `--leak-check=full` as needed. Always remember: **a Valgrind-clean program is a well-written program**.[2]

⁂

GDB: Complete Debugging Guide

1. Introduction to gdb

What is gdb?

gdb (GNU Debugger) is a powerful debugging tool for C/C++ programs[86]. It allows you to:

- Step through code line by line
- Watch variable values interactively
- Set breakpoints to pause execution
- Analyze program state when errors occur
- No need to write diagnostic printf statements[86]

Why Use a Debugger?

Types of errors debuggers help with:[86]

- Compilation errors (caught by compiler)
- Logical errors (program runs but wrong output)
- Runtime errors (segmentation faults, crashes)

Remember: The compiler is never faulty. Question your understanding and code first[86].

---

2. Starting gdb

Compilation with Debug Symbols

Compile with -g flag:[86]

```
$ gcc -Wall -g tarea.c
```

The `-g` flag includes debugging symbols in the executable, allowing gdb to show source code and variable names.

Example program (tarea.c):[86]

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int x1, y1, x2, y2, x3, y3;
    double area;

    printf("Program to calculate the area of a triangle\n");
    printf("Enter the coordinates of the first corner: ");
    scanf("%d%d", &x1, &y1);
    printf("Enter the coordinates of the second corner: ");
    scanf("%d%d", &x2, &y2);
    printf("Enter the coordinates of the third corner: ");
    scanf("%d%d", &x3, &y3);

    area = abs(x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1 - y2)) / 2.0;

    printf("The area of the triangle = %lf\n", area);
    exit(0);
}
```

Starting gdb

Launch gdb:[86]

```
$ gdb ./a.out
```

gdb prompt:

```
GNU gdb (GDB) 8.1.1
...
Reading symbols from ./a.out...done.
(gdb)
```

---

3. Basic Navigation Commands

list Command - View Source Code

Basic list:[86]

```
(gdb) list
```

Shows 10 lines at a time, starting from main function.

Output example:[86]

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    int main()
5    {
6        int x1, y1, x2, y2, x3, y3;
7        double area;
8
9        printf("Program to calculate the area of a triangle\n");
10       printf("Enter the coordinates of the first corner: ");
```

Subsequent list commands:[86]

```
(gdb) list
```

Shows next 10 lines.

List specific lines:[86]

```
(gdb) list 18,22
```

Output:

```
18       printf("Enter the coordinates of the third corner: ");
19       scanf("%d%d", &x3, &y3);
20
21       area = abs(x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1 - y2)) / 2.0;
22
```

List around a line:[86]

```
(gdb) list 18
```

Shows 10 lines centered around line 18.

List specific function:[86]

```
(gdb) list main
(gdb) list function_name
```

Repeating Commands

Press Enter/Return:[86] Repeats the last command. Very useful with `list`, `step`, `next`.

---

4. Running Programs in gdb

run Command - Start Execution

Basic run:[86]

```
(gdb) run
```

Output:[86]

```
Starting program: /home/user/a.out
Program to calculate the area of a triangle
Enter the coordinates of the first corner: 0 0
Enter the coordinates of the second corner: 4 0
Enter the coordinates of the third corner: 2 3
The area of the triangle = 6.000000
[Inferior 1 (process 12345) exited normally]
```

Run with command-line arguments:[86]

```
(gdb) run arg1 arg2 arg3
```

Run with input redirection:[86]

```
(gdb) run < inputfile.txt
```

---

5. Breakpoints

Setting Breakpoints

Break at line number:[86]

```
(gdb) break 18
```

Output:

```
Breakpoint 1 at 0x400632: file tarea.c, line 18.
```

Break at function:[86]

```
(gdb) break main
```

Output:

```
Breakpoint 2 at 0x400580: file tarea.c, line 6.
```

Break at function in specific file:[86]

```
(gdb) break filename.c:function_name
(gdb) break filename.c:25
```

Viewing Breakpoints

List all breakpoints:[86]

```
(gdb) info break
```

Output:

```
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400632 in main at tarea.c:18
2       breakpoint     keep y   0x0000000000400580 in main at tarea.c:6
```

Managing Breakpoints

Disable breakpoint:[86]

```
(gdb) disable 1
```

Enable breakpoint:[86]

```
(gdb) enable 1
```

Delete breakpoint:[86]

```
(gdb) delete 1
```

Delete all breakpoints:[86]

```
(gdb) delete
```

Running to Breakpoint

Continue execution:[86]

```
(gdb) run
Starting program: /home/user/a.out
Program to calculate the area of a triangle

Breakpoint 1, main () at tarea.c:6
6         int x1, y1, x2, y2, x3, y3;
```

Continue to next breakpoint:[86]

```
(gdb) continue
```

Output:

```
Continuing.
Enter the coordinates of the first corner: 0 0
Enter the coordinates of the second corner: 4 0

Breakpoint 2, main () at tarea.c:18
18        printf("Enter the coordinates of the third corner: ");
```

---

6. Stepping Through Code

next Command - Execute Next Line

Execute current line, stay in same function:[86]

```
(gdb) next
```

Short form:

```
(gdb) n
```

Example execution:[86]

```
18       printf("Enter the coordinates of the third corner: ");
(gdb) next
Enter the coordinates of the third corner: 19       scanf("%d%d", &x3, &y3);
```

step Command - Step Into Functions

Step into function calls:[86]

```
(gdb) step
```

Short form:

```
(gdb) s
```

Difference from next:

- `next`: Executes function call and stops at next line
- `step`: Enters the function and stops at first line inside

finish Command - Run Until Return

Complete current function:[86]

```
(gdb) finish
```

Runs until current function returns, then stops.

Example:

```
(gdb) finish
Run till exit from #0  factorial (n=5) at factorial.c:10
0x0000000000400632 in main () at factorial.c:25
25       result = factorial(num);
Value returned is $1 = 120
```

return Command - Force Return

Return immediately from function:[86]

```
(gdb) return
(gdb) return value
```

Forces immediate return, skipping remaining code in function.

---

7. Examining Variables

print Command - Display Values

Print variable:[86]

```
(gdb) print x1
```

Output:

```
$1 = 0
```

Print expression:[86]

```
(gdb) print x1 + x2
```

Output:

```
$2 = 4
```

Print with different formats:[86]

```
(gdb) print x1          # Decimal (default)
(gdb) print/x x1        # Hexadecimal
(gdb) print/o x1        # Octal
(gdb) print/t x1        # Binary
(gdb) print/c x1        # Character
```

Value history:[86] Results are stored as $1, $2, $3, etc.

```
(gdb) print $1 + $2
```

display Command - Auto-Display Variables

Automatically display after each step:[86]

```
(gdb) display area
```

Output:

```
1: area = 0
```

After each `next` or `step`, gdb shows:

```
1: area = 6
```

View all displays:[86]

```
(gdb) info display
```

Remove display:[86]

```
(gdb) undisplay 1
```

Disable/enable display:[86]

```
(gdb) disable display 1
(gdb) enable display 1
```

set Command - Modify Variables

Change variable value:[86]

```
(gdb) set var x1 = 10
(gdb) set var area = 0.0
```

Example:[86]

```
(gdb) print area
$3 = 5.5
(gdb) set var area = 10.0
(gdb) print area
$4 = 10
```

---

8. Watchpoints

watch Command - Break on Variable Change

Set watchpoint:[86]

```
(gdb) watch x1
```

Output:

```
Hardware watchpoint 3: x1
```

Program stops whenever `x1` changes value.

View watchpoints:[86]

```
(gdb) info watch
```

Delete watchpoint:[86]

```
(gdb) delete 3
```

Example output when watchpoint triggers:[86]

```
Hardware watchpoint 3: x1
Old value = 0
New value = 5
main () at tarea.c:12
12       printf("Enter the coordinates of the second corner: ");
```

---

9. Conditional Breakpoints

Setting Conditions on Breakpoints

Break only if condition true:[86]

```
(gdb) break 234 if p == 0
```

Stops at line 234 only when pointer `p` is NULL.

Add condition to existing breakpoint:[86]

```
(gdb) condition 2 i == 100
```

Breakpoint 2 now triggers only when `i` equals 100.

Remove condition:[86]

```
(gdb) condition 2
```

Example - Finding NULL pointer:[86]

```
(gdb) break 234 if p == 0
Breakpoint 1 at 0x400632: file prog.c, line 234.
(gdb) run
...
Breakpoint 1, main () at prog.c:234
234      p->data = value;
(gdb) print p
$1 = (node *) 0x0
```

ignore Command - Skip Breakpoint Hits

Ignore next N hits:[86]

```
(gdb) ignore 2 5
```

Breakpoint 2 will be ignored for the next 5 hits.

Example:[86]

```
(gdb) break 15
Breakpoint 1 at 0x400580: file loop.c, line 15.
(gdb) ignore 1 99
Will ignore next 99 crossings of breakpoint 1.
(gdb) run
...
Breakpoint 1, main () at loop.c:15  # Stops on 100th iteration
15       sum += i;
```

---

10. Call Stack and Frames

backtrace Command - View Call Stack

Show call stack:[86]

```
(gdb) backtrace
(gdb) bt
```

Output example:[86]

```
#0  factorial (n=3) at factorial.c:10
#1  0x0000000000400625 in factorial (n=4) at factorial.c:12
#2  0x0000000000400625 in factorial (n=5) at factorial.c:12
#3  0x0000000000400655 in main () at factorial.c:25
```

Shows function call hierarchy from current function to main.

frame Command - Navigate Frames

Show current frame:[86]

```
(gdb) frame
```

Output:

```
#0  factorial (n=3) at factorial.c:10
10       if (n <= 1) return 1;
```

Switch to specific frame:[86]

```
(gdb) frame 2
```

Output:

```
#2  0x0000000000400625 in factorial (n=5) at factorial.c:12
12       return n * factorial(n - 1);
```

Move up/down stack:[86]

```
(gdb) up        # Move to calling function
(gdb) down      # Move to called function
```

Get frame info:[86]

```
(gdb) info frame
```

Shows detailed information about current frame.

---

11. Memory Examination

x Command - Examine Memory

Examine memory at address:[86]

```
(gdb) x/5wx A
```

Format: x/[count][format][size] address

Sizes:[86]

- `b` - byte (1 byte)
- `h` - halfword (2 bytes)
- `w` - word (4 bytes)
- `g` - giant (8 bytes)

Formats:[86]

- `x` - hexadecimal
- `d` - decimal
- `u` - unsigned decimal
- `o` - octal
- `t` - binary
- `c` - character
- `s` - string

Examples:[86]

```
(gdb) x/5wx A           # 5 words in hex starting at A
(gdb) x/1wx &i          # 1 word in hex at address of i
(gdb) x/10bd array      # 10 bytes in decimal from array
(gdb) x/s str           # String at str
```

Output example:[86]

```
(gdb) x/5wx A
0x7fffffffe420: 0x00000001  0x00000002  0x00000003  0x00000004
0x7fffffffe430: 0x00000005
```

12. Advanced Features

Working with Arrays

Print array:[86]

```
(gdb) print A[0]@10
```

Prints 10 elements starting from A[0].

Example:[86]

```
(gdb) print A[0]@5
$1 = {1, 2, 3, 4, 5}
```

Working with Pointers

Print dereferenced pointer:[86]

```
(gdb) print *ptr
```

Print pointer address:[86]

```
(gdb) print ptr
```

Print structure through pointer:[86]

```
(gdb) print *node_ptr
```

Type Information

Check variable type:[86]

```
(gdb) ptype variable
```

Example:[86]

```
(gdb) ptype area
type = double
(gdb) ptype x1
type = int
```

---

13. Multi-File Debugging

Debugging Programs with Multiple Files

Scenario 1: Files included with #include:[86]

Compile:

```
$ gcc -Wall -g allparts.c
```

In gdb, list specific file:[86]

```
(gdb) list part1.c:1
```

Scenario 2: Separately compiled files:[86]

Compile:

```
$ gcc -Wall -g -c part1.c
$ gcc -Wall -g -c part2.c
$ gcc -Wall -g -c allparts.c
$ gcc -g -o a.out allparts.o part1.o part2.o
```

In gdb:[86]

```
$ gdb ./a.out
(gdb) list part1.c:1
(gdb) list part2.c:function_name
(gdb) break part1.c:25
(gdb) break part2.c:function_name
```

Loading Different Executable

Load new executable without restarting gdb:[86]

```
(gdb) file newprog
```

Useful when you recompile and want to debug new version.

---

14. Help System

Getting Help in gdb

General help:[86]

```
(gdb) help
```

Help on specific command:[86]

```
(gdb) help break
(gdb) help print
(gdb) help run
```

Search help topics:[86]

```
(gdb) apropos keyword
```

Example:[86]

```
(gdb) apropos breakpoint
```

Lists all commands related to breakpoints.

List command categories:[86]

```
(gdb) help all
```

---

15. Quitting gdb

Exiting gdb Session

Quit gdb:[86]

```
(gdb) quit
(gdb) q
```

If program is running:

```
A debugging session is active.
Inferior 1 [process 12345] will be killed.
Quit anyway? (y or n) y
```

---

16. Common Debugging Scenarios

Example 1: Segmentation Fault

Program crashes:

```
$ ./a.out
Segmentation fault (core dumped)
```

Debug:[86]

```
$ gdb ./a.out
(gdb) run
...
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400632 in main () at prog.c:234
234      p->data = value;
(gdb) print p
$1 = (node *) 0x0
```

p is NULL, causing segfault.

Example 2: Infinite Loop

Set breakpoint in loop:[86]

```
(gdb) break 45
(gdb) run
...
Breakpoint 1, main () at prog.c:45
45       while (i < n) {
(gdb) display i
(gdb) display n
(gdb) continue
```

Watch i and n values to identify why loop doesn't terminate.

Example 3: Wrong Calculation

Use print to check intermediate values:[86]

```
(gdb) break 21
(gdb) run
...
Breakpoint 1, main () at tarea.c:21
21        area = abs(x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1 - y2)) / 2.0;
(gdb) print x1
$1 = 0
(gdb) print y2 - y3
$2 = -3
(gdb) print x1 * (y2 - y3)
$3 = 0
```

Step through calculation to find error.

---

17. Command Summary

Command	Shortcut	Description
`run [args]`	`r`	Start program execution
`break [location]`	`b`	Set breakpoint
`continue`	`c`	Continue to next breakpoint
`next`	`n`	Execute next line (step over)
`step`	`s`	Step into function
`finish`	-	Run until function returns
`print [expr]`	`p`	Print value/expression
`display [expr]`	-	Auto-display after each step
`watch [var]`	-	Break when variable changes
`backtrace`	`bt`	Show call stack
`frame [n]`	`f`	Select stack frame
`list [loc]`	`l`	Show source code
`info break`	`i b`	List breakpoints
`delete [n]`	`d`	Delete breakpoint
`set var x=val`	-	Set variable value
`quit`	`q`	Exit gdb
`help [cmd]`	`h`	Get help

This comprehensive guide covers all gdb features from the PDF with practical examples for each command and use case.

gprof: Performance Profiling Guide

1. Introduction and Basic Usage

What is gprof?

Purpose:[81]

- gprof is a profiler that monitors performance of your program
- Measures relative performance of functions
- Helps detect performance bottlenecks

Why Profile?[81] A function's performance may be poor for two reasons:

1. Each invocation takes too much time
2. The function is called too many times

Three-Step Workflow

Step 1: Compile with -pg flag[81]

```
$ gcc -Wall -pg -o myprog myprog.c
```

Creates executable with profiling instrumentation

Step 2: Run the program[81]

```
$ ./myprog
```

Executes normally and creates `gmon.out` profile data file

Step 3: Analyze with gprof[81]

```
$ gprof ./myprog
```

Displays flat profile and call graph

---

2. Understanding Profiling Output

Flat Profile (Timing Profile)

Purpose:[81] Lists functions with detailed profiling information on running times

Example Output:[81]

```
$ gprof -b -p -z ./a.out
Flat profile:
Each sample counts as 0.01 seconds.
%         cumulative    self          self      total
time      seconds       seconds  calls  ns/call   ns/call      name
81.05     0.58          0.58     93324100  6.25      6.25      nextnum
12.69     0.67          0.09     10000000  9.13      61.24     ishappy
4.23      0.71          0.03            -          -        main
0.70      0.71          0.01            -          -        frame_dummy
0.00      0.71          0.00            -          -        __do_global_dtors_aux
```

Column Meanings:[81]

Column	Meaning	Example
% time	Percentage of time in this function (excluding called functions)	81.05%
Self seconds	Time spent inside this function only	0.58s
Cumulative seconds	Running total of self times	0.67s
Calls	Number of times function called	93324100
Self ns/call	Average self time per call in nanoseconds	6.25 ns
Total ns/call	Self time plus called functions per invocation	6.25 ns
Name	Function name	nextnum

Call Graph

Purpose:[81] Shows which functions call which functions and call counts

Example Output:[81]

```
index % time   self    children   called      name
                        <spontaneous>
[1]   100.0  0.01    0.25                   main [1]
                        1000000/1000000
                        ishappy [2]

                        1000000/1000000
[2]   96.1   0.03    0.22      1000000     ishappy [2]
                        12469250/12469250
                        isvisited [3]
                        12469250/12469250
                        nextnum [4]
```

Reading Call Graph:[81]

- [index] is function index
- Primary line shows total calls
- Above: caller functions

- Below: called functions
- Format `count1/count2` shows count1 calls out of total count2

---

3. gprof Options and Commands

Compilation and Execution

Compile with profiling:[81]

```
$ gcc -Wall -pg myprog.c              # Creates a.out
$ gcc -Wall -pg -o myprog myprog.c    # Creates myprog
```

Run program with arguments:[81]

```
$ ./a.out arg1 arg2 arg3
```

Display Options

Basic analysis:[81]

```
$ gprof ./a.out                 # Full output (flat + call graph)
$ gprof ./a.out gmon.out        # Explicit profile file
```

Option: -b (Compact output)[81]

```
$ gprof -b ./a.out
```

Removes explanatory text, shows only data

Option: -p (Flat profile only)[81]

```
$ gprof -p ./a.out
```

Option: -q (Call graph only)[81]

```
$ gprof -q ./a.out
```

Option: -pfunctionname (Specific function)[81]

```
$ gprof -pnextnum ./a.out
```

Shows flat profile for `nextnum` function only

Option: -z (Include all functions)[81]

```
$ gprof -z ./a.out
```

Includes functions with zero time and system functions

Combined options:[81]

```
$ gprof -b -p -z ./a.out           # Compact flat profile with all functions
$ gprof -b -pfishappy -z ./a.out   # Compact profile of specific function
```

---

4. Case Studies

Case Study 1: Happy Numbers

Problem:[81] Check if numbers are happy (repeatedly sum squares of digits until reaching 1 or cycle)

Examples:[81]

- 2026 is happy: 2026 → 44 → 32 → 13 → 10 → 1 ✓
- 2024 is unhappy: 2024 → 24 → 20 → 4 → 16 → 37 → 58 → 89 → 145 → 42 → 20 (cycle)

Four Optimization Attempts:

Attempt 1: Array initialization bottleneck[81]

```
$ gprof -b -p -z ./a.out
%        cumulative    self           self      total
time     seconds       seconds  calls  us/call   us/call      name
99.15    9.32          9.32     100000  93.20     93.20      init
0.11     9.33          0.01     1246773 0.01      0.01       isvisited
```

Problem: init() takes 99.15% initializing large array for each call

Attempt 2: Smaller arrays (math optimization)[81]

```
%        cumulative    self           self      total
time     seconds       seconds  calls  us/call   us/call      name
90.17    1.51          1.51     1000000 1.51      1.51       init
4.84     1.59          0.08     12469340 0.01     0.01       nextnum
```

Improvement: 9.32s → 1.69s (5.5× faster) Problem: init() still 90.17%

Attempt 3: Dictionary approach[81]

```
%        cumulative    self           self      total
time     seconds       seconds  calls  ns/call   ns/call      name
50.53    0.13          0.13     12469250 10.54     10.54      isvisited
23.32    0.19          0.06     12469250 4.86      4.86       nextnum
11.66    0.22          0.03     1000000  30.32     247.62     ishappy
```

Improvement: 1.69s → 0.26s (6.5× faster) Problem: isvisited() now bottleneck at 50.53%

Attempt 4: Algorithmic breakthrough[81]

```
%        cumulative    self           self      total
time     seconds       seconds  calls  ns/call   ns/call      name
82.27    0.54          0.54     93324100 5.82      5.82       nextnum
15.38    0.64          0.10     10000000 10.15     58.63      ishappy
```

Key insight: Happy numbers → 1, unhappy numbers → cycle containing 4 No data structure needed, just check if reaches 1 or 4

Case Study 2: Recursive Fibonacci

Naive Recursion:[81]

```
int Fib(int n) {
    if (n < 0) return -1;
    if (n == 0) return 0;
    if (n == 1) return 1;
    return Fib(n-1) + Fib(n-2);
}
// Call: Fib(32)
```

Call graph output:[81]

```
index % time  self    children  called           name
                      7049154
[1]   100.0  0.01    0.00      1+7049154        Fib [1]
```

Calls: 1 + 7,049,154 recursive calls

With Memoization:[81]

```
int Fib(int n, int F[]) {
    if (F[n] >= 0) return F[n];
    if (n == 0) F[n] = 0;
    else if (n == 1) F[n] = 1;
    else F[n] = Fib(n-1, F) + Fib(n-2, F);
    return F[n];
}
// Call: Fib(32, F)
```

Call graph output:[81]

```
index % time  self    children  called       name
                      62
[1]   0.0    0.00    0.00      1+62         Fib [1]
```

Calls: 1 + 62 recursive calls Reduction: 7,049,154 → 62 calls (113,373× improvement!)

---

5. Limitations and Important Notes

Sampling-Based Approach[81]

How sampling works:

- gprof samples execution every 0.01 seconds (default)
- Based on samples, makes statistical analysis
- Percentages are estimates, not exact

Accuracy Requirements:[81]

- Program must run for at least a few seconds for meaningful results
- Insufficient samples lead to inaccurate estimates
- Sampling rate cannot be changed

Percentage Limitations:[81]

- Percentages may not sum to exactly 100%
- Sum may be less than or even larger than 100%
- Normal limitation of sampling-based profiling

Functions in Output[81]

Functions not listed:

- Functions not called during profiling
- Missed all samples
- Use `-z` option to include them

Unexpected system functions:[81]

- Functions like `frame_dummy`, `__do_global_dtors_aux`
- Called by runtime system
- Usually account for small percentage

Call Count Notation:[81]

- Regular: Single call count
- Recursive: `count1+count2` format (count1=non-recursive, count2=recursive)
- Caller/called lines: `count1/count2` format

Profiling Limitations[81]

Function-level only:

- gprof handles function-level profiling
- For line-by-line profiling, use `gcov`

No line-by-line in modern systems:[81]

- Line-by-line profiling option `-l` works with old gcc
- Recommended to use `gcov` for modern systems

grep Command: Usage and Options

Introduction to grep

grep stands for "Global Regular Expression Print"[78]. It locates lines that contain matches of regular expression(s) and may or may not highlight the match depending on terminal capabilities[78].

Basic Syntax:[78]

```
grep [OPTIONS] [PATTERN] [FILE(S)]
```

Important Points:[78]

- The pattern is a regular expression
- Regular expressions may contain special characters (like `*`) having special meanings to the shell
- Single quotes are recommended for quoting the pattern
- Quoting enables searching for patterns containing spaces

---

grep Command Usage Examples

Basic Search

Search for word "method":[78]

````
$ grep method textfile.txt
````

Output:

~~~
trial division and Pollard's rho method. This is followed by two
sub-exponential algorithms based upon Fermat's factoring method. Dixon's
method uses random squares, but illustrates the basic concepts of the
and its variants. As a representative of the square-root methods for solving
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
index calculus method (ICM) as a general paradigm for solving the DLP.
```[78]

**Search with space in pattern (must quote):**[78]
```bash
$ grep 'method ' textfile.txt
~~~

Output:

~~~
method uses random squares, but illustrates the basic concepts of the
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
index calculus method (ICM) as a general paradigm for solving the DLP.
```[78]

**Correct quoting for pattern with special characters:**[78]
```bash
$ grep 'method[ \.]' textfile.txt
~~~

Output:

~~~
trial division and Pollard's rho method. This is followed by two
sub-exponential algorithms based upon Fermat's factoring method. Dixon's
method uses random squares, but illustrates the basic concepts of the
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
index calculus method (ICM) as a general paradigm for solving the DLP.
```[78]

**Error when pattern not quoted properly:**[78]
```bash
$ grep method[ \.] textfile.txt
~~~

Output:

```
grep: Invalid regular expression
```

---

grep Options

-e Option: Multiple Patterns

Syntax:[78]

```
grep -e pattern1 -e pattern2 file
```

Search for multiple patterns:[78]

```
$ grep -e 'method' -e 'algorithm' textfile.txt
```

Output:

~~~
This tutorial focuses on algorithms for factoring large composite integers
public-key algorithms for encryption, key exchange, and digital signatures.
These algorithms highlight the roles played by the apparent difficulty of
Two exponential-time integer-factoring algorithms are first covered:
trial division and Pollard's rho method. This is followed by two
sub-exponential algorithms based upon Fermat's factoring method. Dixon's
method uses random squares, but illustrates the basic concepts of the
and its variants. As a representative of the square-root methods for solving
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
index calculus method (ICM) as a general paradigm for solving the DLP.
```[78]

**Handling patterns starting with hyphen:**[78]

Without `-e`, patterns starting with `-` are interpreted as options:
```bash
$ grep '-key' textfile.txt
~~~

Output:

```
grep: invalid option -- 'k'
Usage: grep [OPTION]... PATTERNS [FILE]...
Try 'grep --help' for more information.
```

With `-e` option:

```
$ grep -e '-key' textfile.txt
```

Output:

~~~
public-key algorithms for encryption, key exchange, and digital signatures.
public-key protocols.
```[78]

### -v Option: Inverted Search

**Syntax:**[78]
```bash
grep -v pattern file
~~~

Shows lines NOT containing the pattern:[78]

```
$ grep -v '[A-Z]' textfile.txt
```

Output:

~~~
public-key algorithms for encryption, key exchange, and digital signatures.
solving the factoring and discrete-logarithm problems, for designing
public-key protocols.
method uses random squares, but illustrates the basic concepts of the
candidates for smoothness testing and of sieving.
for extension fields of characteristic two.
```[78]

### -i Option: Case-Insensitive Search

**Syntax:**[78]
```bash
grep -i pattern file
~~~

Case-sensitive search:[78]

```
$ grep 'method' textfile.txt
```

Output:

```
trial division and Pollard's rho method. This is followed by two
sub-exponential algorithms based upon Fermat's factoring method. Dixon's
method uses random squares, but illustrates the basic concepts of the
and its variants. As a representative of the square-root methods for solving
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
index calculus method (ICM) as a general paradigm for solving the DLP.
```

Case-insensitive search:[78]

```
$ grep -i 'method' textfile.txt
```

Output:

~~~
trial division and Pollard's rho method. This is followed by two
sub-exponential algorithms based upon Fermat's factoring method. Dixon's
method uses random squares, but illustrates the basic concepts of the
Quadratic Sieve Method (QSM) which brings the benefits of using small
and its variants. As a representative of the square-root methods for solving
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
index calculus method (ICM) as a general paradigm for solving the DLP.
```[78]

Note: Additional match "Quadratic Sieve Method (QSM)" with capital M

### -w Option: Word-Based Search

**Syntax:**[78]
```bash
grep -w pattern file
~~~

All lines containing uppercase letters:[78]

```
$ grep '[A-Z]' textfile.txt
```

Output:

```
Abstract
This tutorial focuses on algorithms for factoring large composite integers
and for computing discrete logarithms in large finite fields. In order to
make the exposition self-sufficient, I start with some common and popular
These algorithms highlight the roles played by the apparent difficulty of
Two exponential-time integer-factoring algorithms are first covered:
trial division and Pollard's rho method. This is followed by two
sub-exponential algorithms based upon Fermat's factoring method. Dixon's
relation-collection and the linear-algebra stages. Next, I introduce the
Quadratic Sieve Method (QSM) which brings the benefits of using small
As the third module, I formally define the discrete-logarithm problem (DLP)
and its variants. As a representative of the square-root methods for solving
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
index calculus method (ICM) as a general paradigm for solving the DLP.
Various stages of the basic ICM are explained both for prime fields and
```

Only whole-word single-letter uppercase words:[78]

```
$ grep -w '[A-Z]' textfile.txt
```

Output:

~~~
make the exposition self-sufficient, I start with some common and popular
relation-collection and the linear-algebra stages. Next, I introduce the
As the third module, I formally define the discrete-logarithm problem (DLP)
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
```[78]

### -n Option: Print Line Numbers

**Syntax:**[78]
```bash
grep -n pattern file
~~~

Without line numbers:[78]

```
$ grep '[^a-zA-Z]$' textfile.txt
```

Output:

```
public-key algorithms for encryption, key exchange, and digital signatures.
public-key protocols.
Two exponential-time integer-factoring algorithms are first covered:
candidates for smoothness testing and of sieving.
As the third module, I formally define the discrete-logarithm problem (DLP)
index calculus method (ICM) as a general paradigm for solving the DLP.
for extension fields of characteristic two.
```

With line numbers:[78]

```
$ grep -n '[^a-zA-Z]$' textfile.txt
```

Output:

~~~
6:public-key algorithms for encryption, key exchange, and digital signatures.
9:public-key protocols.
11:Two exponential-time integer-factoring algorithms are first covered:
17:candidates for smoothness testing and of sieving.
19:As the third module, I formally define the discrete-logarithm problem (DLP)
22:index calculus method (ICM) as a general paradigm for solving the DLP.
24:for extension fields of characteristic two.
```[78]

### -c Option: Count Matches

**Syntax:**[78]
```bash
grep -c pattern file
~~~

Counts matching lines only:[78]

```
$ grep -c '[^a-zA-Z]$' textfile.txt
```

Output:

~~~
7
```[78]

### -r or -R Option: Recursive Search

**Syntax:**[78]
```bash
grep -r pattern directory/
grep -R pattern directory/
~~~

Search recursively in subdirectories:[78]

```
$ grep -r 'nodep' .
```

Output:

~~~
./libstaque/static/defs.h:typedef node *nodep;
./libstaque/static/stack.h:typedef nodep stack;
./libstaque/static/queue.h:
nodep front;
./libstaque/static/queue.h:
nodep back;
./libstaque/shared/defs.h:typedef node *nodep;
./libstaque/shared/stack.h:typedef nodep stack;
./libstaque/shared/queue.h:
nodep front;
./libstaque/shared/queue.h:
nodep back;
```[78]

### -l Option: List Filenames Only

**Syntax:**[78]
```bash
grep -l pattern file(s)
grep -r -l pattern directory/
~~~

Print only filenames with matches:[78]

```
$ grep -r -l 'nodep' .
```

Output:

~~~
./libstaque/static/defs.h
./libstaque/static/stack.h
./libstaque/static/queue.h
./libstaque/shared/defs.h
./libstaque/shared/stack.h
./libstaque/shared/queue.h
```[78]

---

## grep Options Summary Table

| Option | Description | Example |
|--------|-------------|---------|
| `-e pattern` | Specify multiple patterns | `grep -e 'foo' -e 'bar' file` |
| `-v` | Invert match (non-matching lines) | `grep -v 'pattern' file` |
| `-i` | Case-insensitive matching | `grep -i 'Pattern' file` |
| `-w` | Match whole words only | `grep -w 'word' file` |
| `-n` | Show line numbers | `grep -n 'pattern' file` |
| `-c` | Count matching lines | `grep -c 'pattern' file` |
| `-l` | List filenames only | `grep -l 'pattern' *.txt` |
| `-r, -R` | Recursive search | `grep -r 'pattern' dir/` |

---

## Combining Options

**Multiple options can be used together:**

```bash
$ grep -n -c 'pattern' file
~~~

Counts lines and shows line numbers[78]

```
$ grep -r -l -i 'pattern' .
```

Searches recursively, lists files only, case-insensitive[78]

```
$ grep -v -c 'pattern' file
```

Counts lines that do NOT match[78]

```
$ grep -w -n 'word' file
```

Shows matching whole words with line numbers[78]

Important Notes from PDF

Pattern Quoting:[78]

- Always quote patterns with single quotes to prevent shell expansion
- Quoting is especially important when pattern contains spaces or special characters

Multiple Pattern Matching:[78]

- Use `-e` option for each pattern when searching for multiple patterns
- Without `-e`, patterns starting with `-` will be misinterpreted as options

File Names in Output:[78]

- When searching multiple files with `-r`, grep shows filename before match
- Use `-l` to show only filenames
- Use `-h` (not in exercises) to suppress filename display

Line Counting vs Output:[78]

- `-c` shows count only (no lines displayed)
- `-n` shows both lines and their numbers
- Both can be used together for different purposes

PART 2: LINUX COMMANDS AND FILE SYSTEM

2. Linux Shell

Definition

- Command interpreter waiting for user to type commands from keyboard
- OS executes commands and shows results back to user
- Just another program written on top of OS

Different Shells

- Bourne shell (sh): original shell by Steve Bourne
- GNU Bourne-again shell (bash): sh with more features
- C shell (csh)
- Korn shell (ksh)

Used in Course

- bash shell only

- `$` prompt: bash waiting for command entry

3. Linux Command Structure

Parts of a Command

1. The actual command (required)
2. Command option(s) (optional, start with `-` or `--`)
3. Argument(s) (optional for some, mandatory for others)

Examples

```
$ ls
Output: file1.txt  file2.txt  directory1/
```

Command only, no option or argument

```
$ ls -l
Output: -rw-r--r-- 1 user group 1024 Oct 28 10:05 file1.txt
         -rw-r--r-- 1 user group 2048 Oct 28 10:10 file2.txt
         drwxr-xr-x 2 user group 4096 Oct 28 09:50 directory1
```

Command with option, no argument

```
$ gcc myfile.c
Output: (creates a.out if successful)
```

Command and argument, no option

```
$ gcc -Wall myfile.c
Output: (same as above, with additional warnings)
```

All three present: command, option, and argument

```
$ ls -lar
Output: total 24
        drwxr-xr-x 3 user user 4096 Oct 28 10:15 .
        drwxr-xr-x 4 root root 4096 Oct 27 09:00 ..
        -rw-r--r-- 1 user user 1024 Oct 28 09:50 .bashrc
        -rw-r--r-- 1 user user 2048 Oct 28 10:05 file1.txt
```

Command with multiple options (-l, -a, and -r)

4. Linux Directory Structure

Tree Organization

- Root directory: `/`
- Subdirectories within directories
- Files at leaf nodes
- Directory can contain subdirectories and files
- Every directory contains: `.` (current) and `..` (parent)
- Hidden files/directories: names starting with `.`
- Home directory on login: `/home/username`

Example Directory Tree

```
/
├── bin/
├── boot/
├── etc/
├── home/
│   └── foobar/
│       └── my_courses/
│           ├── SysProgLab/
│           │   ├── Assignments/
│           │   ├── Materials/
│           │   └── readme.txt
│           └── AdvancedOS/
├── lib/
├── tmp/
├── usr/
│   └── local/
│       ├── bin/
│       └── lib/
└── var/
```

5. Identifying Files/Directories

Absolute Names (from root `/`)

```
/usr/local/lib/
/usr/local/lib/libstaque.so
/home/foobar/spl/prog/assignments/A1/src/
/home/foobar/spl/prog/assignments/A1/src/Makefile
```

Relative Names (from current directory, assume current is `/home/foobar`)

```
spl/prog/assignments/A2/myprog.c
./spl/prog/assignments/
../artim/SPL/tests/T1/questions.pdf
```

Home Directory Relative Names (using ~)

```
~/spl/prog/assignments/A3/
~/sad/SPL/doc/T1soln.pdf
~other_user/shared/file.txt
```

6. File and Directory Permissions

Three Types of Users

- Owner (u): User who owns the file
- Group (g): Users in the file's group
- Others (o): All other users

Three Types of Permissions

- Read (r)
- Write (w)
- Execute (x)

Meanings for Files

- Read: Can read contents
- Write: Can modify contents
- Execute: Can run as program

Meanings for Directories

- Read: Can read contents (ls command)
    - With only read permission, cannot access files in directory
- Write: Can create new files in directory
- Execute: Can go to directory, open/execute files in directory (if you know names)
    - With only execute permission, cannot see directory contents

Permission Examples

```
rwxr-xr-x
├─ Owner (u): rwx - read, write, execute
├─ Group (g): r-x - read, execute
└─ Others (o): r-x - read, execute

rw-r--r--
├─ Owner (u): rw- - read, write
├─ Group (g): r-- - read
└─ Others (o): r-- - read

rwx------
├─ Owner (u): rwx - full permissions
├─ Group (g): --- - no permissions
└─ Others (o): --- - no permissions
```

PART 3: FILE AND DIRECTORY ORGANIZATION COMMANDS

cd - Change Directory

Syntax

```
cd <dirname>
```

Description Changes current working directory to directory named `<dirname>`. Name can be absolute or relative.

Examples

```
$ pwd
/home/foobar

$ cd /usr/local/bin
$ pwd
/usr/local/bin

$ cd SysProgLab
$ pwd
/usr/local/bin/SysProgLab

$ cd ..
$ pwd
/usr/local/bin

$ cd ~
$ pwd
/home/foobar

$ cd -
$ pwd
/usr/local/bin
```

Goes back to previous directory

pwd - Print Working Directory

Syntax

```
pwd
```

Description Shows current working directory (full absolute path).

Examples

```
$ pwd
/home/foobar

$ cd /usr
$ pwd
/usr

$ cd /usr/local/bin
$ pwd
/usr/local/bin
```

mkdir - Make Directory

Syntax

```
mkdir <dirname>
mkdir -p <path/to/nested/directories>
```

Description Creates directory. You need write permission in parent directory.

Option: -p Creates parent directories as needed. Creates entire path if it doesn't exist.

Examples

```
$ mkdir mynewdir
$ ls -l
drwxr-xr-x  2  user  group  4096  Oct 28 10:05  mynewdir
```

```
$ mkdir nested/dir/structure
mkdir: cannot create directory 'nested/dir/structure': No such file or directory
```

```
$ mkdir -p nested/dir/structure
$ pwd
/home/user/nested/dir/structure
```

```
$ mkdir -p my_courses/SysProgLab/Assignments
$ mkdir -p my_courses/SysProgLab/Materials
$ ls -lR my_courses/
my_courses/:
drwxr-xr-x  SysProgLab/

my_courses/SysProgLab/:
drwxr-xr-x  Assignments/
drwxr-xr-x  Materials/
```

rmdir - Remove Directory

Syntax

```
rmdir <dirname>
```

Description Removes directory (must be empty). You need write permission in parent.

Examples

```
$ rmdir emptydir
$ ls
directory1/  directory2/
```

emptydir is gone

```
$ rmdir dirwithfiles
rmdir: failed to remove 'dirwithfiles': Directory not empty
```

cp - Copy File/Directory

Syntax

```
cp <file1> <file2>
cp -r <source_dir> <dest_dir>
cp -f <file> <dest>
```

Options

- `-r`: Recursively copy entire directory tree
- `-f`: Force overwrite without asking

Examples

```
$ ls
file1.txt  file2.txt

$ cp file1.txt file1_backup.txt
$ ls
file1.txt  file1_backup.txt  file2.txt
```

```
$ cp file1.txt /tmp/
$ ls /tmp/
file1.txt
```

```
$ ls
mydir/  myfile.txt

$ cp -r mydir/ mydir_backup/
$ ls
mydir/  mydir_backup/  myfile.txt

$ ls mydir_backup/
(same contents as mydir/)
```

```
$ cp file1.txt file2.txt file3.txt ./targetdir/
$ ls ./targetdir/
file1.txt  file2.txt  file3.txt
```

mv - Move/Rename File

Syntax

```
mv <file1> <file2>
mv <file> <directory>
mv <dir1> <dir2>
```

Description Moves or renames files. Can move to different directory or rename in same directory.

Examples

```
$ ls
oldname.txt

$ mv oldname.txt newname.txt
$ ls
newname.txt
```

```
$ ls
newname.txt

$ mv newname.txt /tmp/
$ ls /tmp/
newname.txt
```

```
$ cd /tmp
$ ls
newname.txt

$ mv newname.txt .
$ pwd
/tmp
$ ls
newname.txt
```

```
$ ls
file1.txt  file2.txt  file3.txt  targetdir/

$ mv file1.txt file2.txt file3.txt ./targetdir/
$ ls
targetdir/

$ ls ./targetdir/
file1.txt  file2.txt  file3.txt
```

```
$ ls
dir1/  dir2/

$ mv dir1/ newdirname/
$ ls
newdirname/  dir2/
```

rm - Remove File

Syntax

```
rm <file1> <file2> ...
rm -i <file>
rm -r <directory>
rm -d <empty_dir>
```

Options

- `-i`: Interactive - asks for confirmation before deletion
- `-r`: Recursive - delete entire directory tree
- `-d`: Delete empty directory

Examples

```
$ ls
file1.txt  file2.txt  dir1/

$ rm file1.txt
$ ls
file2.txt  dir1/
```

```
$ ls
file2.txt

$ rm -i file2.txt
remove file2.txt? y
$ ls
(empty)
```

```
$ ls
file2.txt

$ rm -i file2.txt
remove file2.txt? n
$ ls
file2.txt
```

File not deleted

```
$ ls
dir1/  dir2/

$ rm -r dir1/
$ ls
dir2/
```

```
$ ls
dir_with_files/  (contains file1.txt and file2.txt)

$ rm -r dir_with_files/
$ ls
(empty)
```

PART 4: LISTING FILES AND DIRECTORIES

ls - List Directory Contents

Syntax

```
ls [options] [file/directory]
```

Options

Option	**Meaning**
`-l`	Long listing (detailed format)
`-a`	Show hidden files (names starting with `.`)
`-R`	Recursively list all subdirectories
`-t`	Sort by modification time (newest first)
`-r`	Reverse sorting order
`-d`	List directory itself, not its contents

Examples

```
$ ls
AdvancedOS/  Materials/  readme.txt  SysProgLab/
```

Basic listing

```
$ ls -a
.  ..  .bashrc  .profile  AdvancedOS/  Materials/  readme.txt  SysProgLab/
```

Shows hidden files (`.bashrc`, `.profile`)

```
$ ls -l
total 20
drwxr-xr-x  2  user  group  4096  Oct 28 19:46  AdvancedOS/
drwxr-xr-x  2  user  group  4096  Oct 28 19:48  Materials/
-rw-r--r--  1  user  group    47  Oct 28 19:47  readme.txt
drwxr-xr-x  2  user  group  4096  Oct 28 19:46  SysProgLab/
```

Long detailed listing

```
$ ls -l *.txt
-rw-r--r--  1  user  group  47  Oct 28 19:47  readme.txt
```

List specific pattern

```
$ ls -ld SysProgLab/
drwxr-xr-x  2  user  group  4096  Oct 28 19:46  SysProgLab/
```

List directory itself, not contents

```
$ ls -lR
AdvancedOS/:
total 8
-rw-r--r-- file1.txt

Materials/:
total 12
-rw-r--r-- list.txt
-rw-r--r-- readme.txt

SysProgLab/:
total 16
-rw-r--r-- assignment1.txt
-rw-r--r-- assignment2.txt
```

Recursive listing

```
$ ls -lart
total 24
-rw-r--r-- 1 user group  47 Oct 28 19:46 readme.txt
drwxr-xr-x 2 user group 4096 Oct 28 19:47 Materials/
drwxr-xr-x 2 user group 4096 Oct 28 19:48 AdvancedOS/
drwxr-xr-x 2 user group 4096 Oct 28 19:50 SysProgLab/
```

Long listing, all files, reverse order, sorted by time (oldest first)

Explanation of ls -l Output

```
-rw-r--r--  1  user  group  47  Oct 28 19:47  readme.txt
```

Component	Meaning
`-`	Type: `-` = regular file, `d` = directory
`rw-r--r--`	Permissions (user-group-others)
`1`	Number of hard links
`user`	Owner username
`group`	Group name
`47`	File size in bytes
`Oct 28 19:47`	Last modification date/time

Component	Meaning
`readme.txt`	Filename

cat - Concatenate and Print Files

Syntax

```
cat <file1> <file2> ...
```

Description Prints contents of files to screen. Concatenates multiple files.

Examples

```
$ cat readme.txt
This is the readme file.
It contains important information.
Please read carefully.
```

```
$ cat file1.txt file2.txt
Contents of file1
More contents of file1

Contents of file2
More contents of file2
```

```
$ cat > newfile.txt
hello
world
Ctrl+D
$ cat newfile.txt
hello
world
```

head - Print Beginning of File

Syntax

```
head [options] <file>
```

Options

Option	Meaning
`-n N`	Print first N lines (default 10)
`-c N`	Print first N bytes

Examples

```
$ cat readme.txt
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12

$ head readme.txt
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
```

```
$ head -n 3 readme.txt
Line 1
Line 2
Line 3
```

```
$ head -c 50 readme.txt
Line 1
Line 2
Line 3
Line 4
Line
```

tail - Print End of File

Syntax

```
tail [options] <file>
```

Options

Option	**Meaning**
`-n N`	Print last N lines (default 10)
`-c N`	Print last N bytes

Examples

```
$ cat readme.txt
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12

$ tail readme.txt
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12
```

```
$ tail -n 3 readme.txt
Line 10
Line 11
Line 12
```

```
$ tail -c 20 readme.txt
e 10
Line 11
Line 12
```

more - Page-by-Page Display

Syntax

```
more <file>
```

Description Shows file one screen at a time. Waits for user interaction.

Navigation Keys

- Space: Next page
- Enter: Next line
- q: Quit
- /pattern: Search for pattern
- n: Find next search result

Example

```
$ more largefile.txt
(displays first screen)
(press space for next screen)
```

PART 5: FILE COMPARISON AND INFORMATION COMMANDS

wc - Word/Character/Line Count

Syntax

```
wc [options] <file>
```

Options

Option	Meaning
`-l`	Count lines only
`-w`	Count words only
`-c`	Count bytes only

Option	Meaning
`-m`	Count characters

Default Output: `lines words bytes filename`

Examples

```
$ cat readme.txt
This is line one
This is line two
This is line three

$ wc readme.txt
3 9 47 readme.txt
```

3 lines, 9 words, 47 bytes

```
$ wc -l readme.txt
3 readme.txt
```

```
$ wc -w readme.txt
9 readme.txt
```

```
$ wc -c readme.txt
47 readme.txt
```

```
$ ls *.txt
file1.txt  file2.txt  file3.txt

$ wc -l *.txt
10 file1.txt
15 file2.txt
8 file3.txt
33 total
```

diff - Compare Files Line by Line

Syntax

```
diff [options] <file1> <file2>
```

Options

Option	Meaning
`-y`	Side-by-side comparison
`-u`	Unified format (shows context)

Notation

- `<`: Lines in file1
- `>`: Lines in file2
- `d`: delete line
- `a`: add line
- `c`: change line

Examples

```
$ cat file1.txt
apple
banana
cherry
date

$ cat file2.txt
apple
blueberry
cherry
date

$ diff file1.txt file2.txt
2c2
< banana
---
> blueberry
```

Line 2: change banana to blueberry

```
$ diff -y file1.txt file2.txt
apple                          apple
banana                       |  blueberry
cherry                         cherry
date                           date
```

PART 6: FILE PERMISSIONS

chmod - Change File Permissions

Syntax

```
chmod <mode> <file/directory>
```

Two Methods

1. Symbolic Mode

```
chmod [who][+/-][permission] file
```

- **who**: `u` (user/owner), `g` (group), `o` (others), `a` (all)
- **+/-**: `+` (add), `-` (remove)
- **permission**: `r` (read), `w` (write), `x` (execute)

2. Numeric Mode

```
chmod [octal] file
```

Number	Permission
4	read (r)
2	write (w)
1	execute (x)

Combine for each category: user-group-others

Examples

```
$ ls -l file.txt
-rw-r--r-- file.txt

$ chmod g+x file.txt
$ ls -l file.txt
-rw-r-xr-- file.txt
```

```
$ chmod o-r file.txt
$ ls -l file.txt
-rw-r-x--- file.txt
```

```
$ chmod 755 file.txt
$ ls -l file.txt
-rwxr-xr-x file.txt
```

```
$ chmod 644 file.txt
$ ls -l file.txt
-rw-r--r-- file.txt
```

```
$ chmod 600 file.txt
$ ls -l file.txt
-rw------- file.txt
```

```
$ chmod 700 file.txt
$ ls -l file.txt
-rwx------ file.txt
```

```
$ chmod u+x script.sh
$ ls -l script.sh
-rwxr--r-- script.sh
```

Add execute for owner

```
$ chmod a+w file.txt
$ ls -l file.txt
-rw-rw-rw- file.txt
```

Add write for everyone

Permission Conversion Table

```
rwx = 4+2+1 = 7  (read, write, execute)
rw- = 4+2 = 6    (read, write)
r-x = 4+1 = 5    (read, execute)
r-- = 4 = 4      (read only)
-wx = 2+1 = 3    (write, execute)
-w- = 2 = 2      (write only)
--x = 1 = 1      (execute only)
--- = 0 = 0      (no permissions)

Examples:
755 = rwxr-xr-x  (owner: all, group: rx, others: rx)
644 = rw-r--r--  (owner: rw, group: r, others: r)
700 = rwx------  (owner: all, group: none, others: none)
777 = rwxrwxrwx  (everyone: all permissions)
600 = rw-------  (owner: rw, others: none)
```

PART 7: WILDCARDS AND PATTERN MATCHING

Wildcard Characters

`*` (Asterisk) - Any Character Sequence

Matches any sequence of characters (including no characters).

Examples

```
$ ls *.txt
file1.txt  file2.txt  readme.txt

$ ls *.c
program.c  util.c  test.c

$ ls start*
start_file.txt  startup.sh  started.log

$ cat *.txt
```

Prints all .txt files

`?` (Question Mark) - Single Character

Matches exactly ONE character.

Examples

```
$ ls ?.txt
a.txt  b.txt  z.txt

$ ls ???.c
main.c  util.c  foo.c

$ ls file?.txt
file1.txt  file2.txt
```

Does NOT match file10.txt or file.txt

`[...]` (Bracket Expression) - Character Range

Matches any single character within brackets.

Examples

```
$ ls file[1-3].txt
file1.txt  file2.txt  file3.txt

$ ls [a-c].txt
a.txt  b.txt  c.txt

$ ls [[:digit:]]*.txt
0data.txt  1data.txt  9notes.txt

$ ls [[:alpha:]]*
apple.txt  books.pdf  config.sh
```

PART 8: REDIRECTION AND PIPING

Input Redirection (<)

Syntax

```
command < input_file
```

Description Command reads input from file instead of keyboard.

Use Case Instead of typing large inputs every time, put them in file and redirect.

Examples

```
$ cat > input.txt
10
20
30
Ctrl+D

$ ./calculate < input.txt
30
40
50
```

Program reads from input.txt instead of keyboard

```
$ grep "pattern" < file.txt
pattern1 found
pattern2 found
```

Output Redirection (>)

Syntax

```
command > output_file
command >> output_file
```

Description

- >: Overwrites file if it exists, creates if it doesn't
- >>: Appends to file if it exists, creates if it doesn't

Examples

```
$ ls -l > file_list.txt
$ cat file_list.txt
total 12
-rw-r--r-- 1 user group 1024 Oct 28 09:50 file1.txt
-rw-r--r-- 1 user group 2048 Oct 28 10:05 file2.txt
drwxr-xr-x 2 user group 4096 Oct 28 10:10 directory1
```

Nothing printed on screen

```
$ echo "First line" > output.txt
$ cat output.txt
First line

$ echo "Second line" >> output.txt
$ cat output.txt
First line
Second line
```

```
$ date > timestamp.txt
$ cat timestamp.txt
Tue Oct 28 10:15:30 IST 2025
```

Piping (|)

Syntax

```
command1 | command2
```

Description Output of command1 becomes input to command2.

Examples

```
$ ls -l | wc -l
15
```

Counts number of lines in ls output

```
$ cat readme.txt | grep "important"
This is an important file
```

```
$ ls -l | grep "\.txt$"
-rw-r--r-- readme.txt
-rw-r--r-- notes.txt
```

```
$ cat data.txt | sort | uniq
apple
banana
cherry
```

```
$ ps aux | grep bash
user  12345  0.0  0.1  4124  2048 pts/0 S+  10:15  0:00 bash
user  12346  0.0  0.1  4124  2048 pts/1 S+  10:20  0:00 bash
```

Complex Piping

```
$ cat input.txt | head -n 5 | wc -l
5
```

Gets first 5 lines, counts them

```
$ ls -l | grep "\.c$" | wc -l
3
```

Counts C source files

PART 9: USING MANPAGES

man Command

Syntax

```
man <command>
man <section> <command>
```

Description Displays manual pages for commands and functions.

Sections

Section	Type
1	Commands (default)
2	System calls
3	Library functions
4	Device files
5	File formats

Examples

```
$ man ls
(opens manual page for ls command)
```

```
$ man 3 printf
(opens manual for printf() function, section 3)
```

```
$ man printf
(opens manual for printf command, section 1)
```

```
$ man -k permission
chgrp (1)           - change group ownership
chmod (1)           - change file mode bits
(searches for pages related to "permission")
```

Navigation in man

- Space/Page Down: Next page
- b/Page Up: Previous page
- q: Quit
- /pattern: Search
- n: Next search result

PART 10: PRACTICE EXERCISES FROM PDF

Exercise 1: Create Directory Structure

Objective: Create directory tree and verify

Directory Structure to Create:

```
my_courses/
├── readme.txt
├── SysProgLab/
│   ├── Assignments/
│   │   ├── Assgn-1.txt
│   │   └── Assgn-2.txt
│   ├── Materials/
│   │   └── Slides/
│   └── Marks/
├── AdvancedOS/
└── additional/
    └── AGT/
```

Commands:

```
$ mkdir -p my_courses/SysProgLab/Assignments
$ mkdir -p my_courses/SysProgLab/Materials/Slides
$ mkdir -p my_courses/SysProgLab/Marks
$ mkdir -p my_courses/AdvancedOS
$ mkdir -p my_courses/additional/AGT

$ echo "readme content" > my_courses/readme.txt
$ echo "Assignment 1" > my_courses/SysProgLab/Assignments/Assgn-1.txt
$ echo "Assignment 2" > my_courses/SysProgLab/Assignments/Assgn-2.txt

$ ls -lR my_courses/
my_courses/:
-rw-r--r-- readme.txt
drwxr-xr-x AdvancedOS/
drwxr-xr-x SysProgLab/
drwxr-xr-x additional/

my_courses/AdvancedOS:
(empty)

my_courses/SysProgLab:
drwxr-xr-x Assignments/
drwxr-xr-x Materials/
drwxr-xr-x Marks/

my_courses/SysProgLab/Assignments:
-rw-r--r-- Assgn-1.txt
-rw-r--r-- Assgn-2.txt

my_courses/SysProgLab/Materials:
drwxr-xr-x Slides/

my_courses/SysProgLab/Materials/Slides:
(empty)

my_courses/SysProgLab/Marks:
(empty)

my_courses/additional:
drwxr-xr-x AGT/

my_courses/additional/AGT:
(empty)
```

Exercise 2: List Contents with Different Options

```
$ ls my_courses/
AdvancedOS/  SysProgLab/  additional/  readme.txt

$ ls -l my_courses/
drwxr-xr-x 2 user group 4096 Oct 28 10:10 AdvancedOS/
-rw-r--r-- 1 user group   15 Oct 28 10:15 readme.txt
drwxr-xr-x 4 user group 4096 Oct 28 10:20 SysProgLab/
drwxr-xr-x 2 user group 4096 Oct 28 10:25 additional/

$ ls -lR my_courses/ | head -n 20
my_courses/:
drwxr-xr-x 2 user group 4096 Oct 28 10:10 AdvancedOS/
-rw-r--r-- 1 user group   15 Oct 28 10:15 readme.txt
drwxr-xr-x 4 user group 4096 Oct 28 10:20 SysProgLab/
drwxr-xr-x 2 user group 4096 Oct 28 10:25 additional/

my_courses/AdvancedOS:
total 0

my_courses/SysProgLab:
drwxr-xr-x 2 user group 4096 Oct 28 10:20 Assignments/
drwxr-xr-x 2 user group 4096 Oct 28 10:20 Materials/
drwxr-xr-x 2 user group 4096 Oct 28 10:20 Marks/
```

Exercise 3: File Operations

```
$ cp my_courses/readme.txt my_courses/SysProgLab/Materials/readme-copy.txt

$ ls my_courses/SysProgLab/Materials/
readme-copy.txt  Slides/

$ cat my_courses/readme.txt
readme content

$ head -n 1 my_courses/readme.txt
readme content

$ tail -n 1 my_courses/readme.txt
readme content

$ wc my_courses/SysProgLab/Materials/readme-copy.txt
1 2 15 my_courses/SysProgLab/Materials/readme-copy.txt
```

Exercise 4: File Comparison

```
$ echo "modified content here" >> my_courses/SysProgLab/Materials/readme-copy.txt

$ wc my_courses/SysProgLab/Materials/readme-copy.txt
2 5 36 my_courses/SysProgLab/Materials/readme-copy.txt

$ diff my_courses/readme.txt my_courses/SysProgLab/Materials/readme-copy.txt
1a2
> modified content here
```

Exercise 5: Concatenation and Redirection

```
$ cat my_courses/SysProgLab/Assignments/Assgn-1.txt \
      my_courses/SysProgLab/Assignments/Assgn-2.txt > \
      my_courses/SysProgLab/Materials/Assgns.txt

$ cat my_courses/SysProgLab/Materials/Assgns.txt
Assignment 1
Assignment 2

$ ls -l my_courses | wc -l
5
(output includes total line + 4 items)

$ head -n 1 my_courses/readme.txt
readme content
```

Exercise 6: Copy Directories

```
$ cp -r my_courses/SysProgLab my_courses/SysProgLab-Copy

$ ls my_courses/
AdvancedOS/  SysProgLab/  SysProgLab-Copy/  additional/  readme.txt

$ ls my_courses/SysProgLab-Copy/
Assignments/  Materials/  Marks/

$ ls my_courses/SysProgLab-Copy/Assignments/
Assgn-1.txt  Assgn-2.txt
```

Exercise 7: Permission Changes

Create test directory

```
$ mkdir my_courses/test_perms

$ ls -ld my_courses/test_perms/
drwxr-xr-x 2 user group 4096 Oct 28 11:00 test_perms/
```

Test Case 1: Only Read and Execute

```
$ chmod u=rx my_courses/test_perms/
$ ls -ld my_courses/test_perms/
dr-xr-xr-x 2 user group 4096 Oct 28 11:00 test_perms/

$ cd my_courses/test_perms/
$ pwd
my_courses/test_perms

$ mkdir newdir
mkdir: cannot create directory 'newdir': Permission denied
```

Can go to directory (execute) and see contents (read), but cannot create files/directories (no write)

Test Case 2: Only Read and Write

```
$ chmod u=rw my_courses/test_perms/
$ ls -ld my_courses/test_perms/
drw-r--r-x 2 user group 4096 Oct 28 11:00 test_perms/

$ cd my_courses/test_perms/
bash: cd: my_courses/test_perms/: Permission denied
```

Cannot go to directory (no execute), so cannot access anything

Test Case 3: Only Read

```
$ chmod u=r my_courses/test_perms/
$ ls -ld my_courses/test_perms/
dr--r--r-x 2 user group 4096 Oct 28 11:00 test_perms/

$ ls my_courses/test_perms/
(shows contents, but no write or execute)

$ cd my_courses/test_perms/
bash: cd: my_courses/test_perms/: Permission denied
```

Test Case 4: Only Execute

```
$ chmod u=x my_courses/test_perms/
$ ls -ld my_courses/test_perms/
d--x--x--x 2 user group 4096 Oct 28 11:00 test_perms/

$ ls my_courses/test_perms/
ls: cannot open directory 'my_courses/test_perms/': Permission denied
```

Can go to directory but cannot see contents

Test Case 5: Only Write

```
$ chmod u=w my_courses/test_perms/
$ ls -ld my_courses/test_perms/
d-w------- 2 user group 4096 Oct 28 11:00 test_perms/

$ cd my_courses/test_perms/
bash: cd: my_courses/test_perms/: Permission denied

$ ls my_courses/test_perms/
ls: cannot open directory 'my_courses/test_perms/': Permission denied
```

Revert to rwx

```
$ chmod u=rwx my_courses/test_perms/
$ ls -ld my_courses/test_perms/
drwxr--r-x 2 user group 4096 Oct 28 11:00 test_perms/
```

Exercise 8: Numeric Permission Mode

```
$ chmod 755 my_courses/readme.txt
$ ls -l my_courses/readme.txt
-rwxr-xr-x user group readme.txt

$ chmod 644 my_courses/readme.txt
$ ls -l my_courses/readme.txt
-rw-r--r-- user group readme.txt

$ chmod 700 my_courses/additional/
$ ls -ld my_courses/additional/
drwx------ user group additional/

$ chmod 777 my_courses/
$ ls -ld my_courses/
drwxrwxrwx user group my_courses/
```

Exercise 9: C Program with Execute Permission

Create program

```
$ cat > hello.c << EOF
#include <stdio.h>
int main() {
    printf("Hello World\n");
    return 0;
}
EOF

$ gcc hello.c -o hello

$ ls -l hello
-rwxr-xr-x user group hello
```

Execute permission already present (from gcc)

Run program

```
$ ./hello
Hello World
```

Remove execute permission

```
$ chmod u-x hello
$ ls -l hello
-rw-r--r-- user group hello

$ ./hello
bash: ./hello: Permission denied
```

Restore execute permission

```
$ chmod u+x hello
$ ./hello
Hello World
```

PART 11: REQUIRED COMMANDS AND OPTIONS SUMMARY

Commands to Know

File/Directory Organization

```
cd <dirname>
pwd
mkdir <dirname>
mkdir -p <path>
rmdir <dirname>
cp <file1> <file2>
cp -r <source> <dest>
cp -f <file> <dest>
mv <file1> <file2>
rm <file>
rm -i <file>
rm -r <directory>
rm -d <empty_dir>
```

Listing Contents

```
ls
ls -l
ls -a
ls -R
ls -t
ls -r
ls -d
cat <file>
head <file>
head -n N <file>
head -c N <file>
tail <file>
tail -n N <file>
tail -c N <file>
more <file>
```

Permissions

```
chmod u+r <file>
chmod u-w <file>
chmod g+x <file>
chmod o-rwx <file>
chmod a+x <file>
chmod 755 <file>
chmod 644 <file>
chmod 700 <file>
chmod 777 <file>
```

Comparison/Information

```
wc <file>
wc -l <file>
wc -w <file>
wc -c <file>
wc -m <file>
diff <file1> <file2>
diff -y <file1> <file2>
```

Wildcards

```
* (any sequence)
? (single character)
[...] (character range)
```

Redirection

```
< (input redirection)
> (output redirect - overwrite)
>> (output redirect - append)
| (pipe)
```

Manual Pages

```
man <command>
man <section> <command>
man -k <keyword>
```

Regular Expressions and grep: Complete Reference Guide

1. Basic Regular Expressions

Introduction to Regular Expressions

Regular expressions are patterns used to match character combinations in strings[78]. They are the same as those introduced in formal language theory but with different constructs. Regular expressions are used by `less`, `grep`, `sed`, `awk`, shells, and many text editors like `vi` and `emacs`[78].

Key Points:[78]

- Used by many Unix/Linux tools
- Searches are made line by line
- Newline character is not allowed in regular expressions
- Used for pattern matching and text processing

Basic Metacharacters

1. The Dot (.) - Match Any Character

The period `.` matches any single character except newline[78].

Pattern: `a.g`

Example file content:

```
arg
aeg
aig
abg
a3g
a g
```

Command:

```
$ grep 'a.g' file.txt
```

Output:

```
arg
aeg
aig
abg
a3g
a g
```

The pattern `a.g` matches any three-character sequence starting with 'a', followed by any character, and ending with 'g'[78].

2. Character Classes [...]

Character classes match any single character from the specified set[78].

Basic Syntax:

- `[Tt]` - matches upper- or lower-case T
- `[AEIOU]` - matches any upper-case vowel
- `[a-z]` - matches any lower-case letter
- `[a-zA-Z0-9]` - matches any alphanumeric character

Examples:

```
$ grep '[Tt]he' file.txt
```

Matches both "The" and "the"

```
$ grep '[A-Z][a-z ][a-z ].' file.txt
```

Matches capital letter followed by lowercase letter or space, followed by lowercase letter or space, followed by any character[78].

Character Ranges:

- `[a-z]` - lowercase letters
- `[A-Z]` - uppercase letters
- `[0-9]` - digits
- `[a-zA-Z]` - all letters
- `[a-zA-Z0-9]` - alphanumeric characters

3. Negated Character Classes [^...]

Use `^` after `[` to negate the character class[78].

Examples:

```
$ grep '[^ ]' file.txt
```

Matches any non-space character[78]

```
$ grep '[^aeiouAEIOU]' file.txt
```

Matches any character other than vowels[78]

```
$ grep '[^a-zA-Z]' file.txt
```

Matches any non-alphabetic character[78]

Complex Example:

```
$ grep '[^AEIOU][^a-zA-Z ].....[a-drt]' file.txt
```

- `[^AEIOU]` - not an uppercase vowel
- `[^a-zA-Z ]` - not a letter or space
- `.....` - exactly 5 characters
- `[a-drt]` - ends with a, b, c, d, r, or t[78]

4. The Asterisk (*) - Zero or More

The `*` quantifier matches zero or more of the preceding character or group[78].

Examples:

```
$ grep '.*' file.txt
```

Matches any string (including empty)[78]

```
$ grep '...*' file.txt
```

Matches any non-empty string[78]

```
$ grep '[a-z]*' file.txt
```

Matches any sequence of lowercase letters (including empty sequence)[78]

```
$ grep '[^a-zA-Z]*' file.txt
```

Matches any sequence of non-alphabetic characters[78]

Important Note: Longest possible matches are reported, starting as early as possible[78].

Example:

```
$ grep '[A-Z][a-zA-Z ]*[^ ]' file.txt
```

- `[A-Z]` - starts with uppercase letter
- `[a-zA-Z ]*` - followed by zero or more letters or spaces
- `[^ ]` - ends with non-space character[78]

Anchors

5. Start of Line (^) and End of Line ($)

Start of Line (^): Use `^` as the first symbol to match at the beginning of a line[78].

```
$ grep '^[A-Z][a-z]*' file.txt
```

Matches the first word of a line if it starts with a capital letter[78]

End of Line ($): Use `$` as the last symbol to match at the end of a line[78].

```
$ grep '[a-z]*$' file.txt
```

Matches the last word of a line if it ends with lowercase letters[78]

Combined Example:

```
$ grep '^[A-Za-z, ]*$' file.txt
```

Matches entire lines containing only letters, commas, and spaces[78]

Escaping Special Characters

To match special characters literally, escape them with backslash `\`[78]:

Special Characters to Escape:

- `\.` - literal dot
- `\[` - literal opening bracket
- `\]` - literal closing bracket
- `\*` - literal asterisk
- `\^` - literal caret
- `\$` - literal dollar sign
- `\\` - literal backslash
- `\/` - literal forward slash (used in substitution)

Note: Hyphen `-` need not be quoted[78]

Examples:

```
$ grep '[a-z]*\.' file.txt
```

Matches lowercase letters followed by a literal period[78]

```
$ grep '[a-z]*-[a-z-]*.*\.' file.txt
```

Matches pattern with hyphen and ending with literal period[78]

2. grep Command

Introduction to grep

grep stands for "Global Regular Expression Print"[78]. It locates lines that contain matches of regular expressions and may or may not highlight the match depending on terminal capabilities[78].

Basic Syntax:

```
grep [OPTIONS] [PATTERN] [FILE(S)]
```

Key Points:[78]

- The pattern is a regular expression
- Regular expressions may contain characters having special meanings to shell
- Preferable to quote the pattern with single quotes
- Quoting enables searching for patterns containing spaces

Basic grep Usage

Simple Pattern Matching

Test File (textfile.txt):[78]

```
Abstract

This tutorial focuses on algorithms for factoring large composite integers
and for computing discrete logarithms in large finite fields. In order to
make the exposition self-sufficient, I start with some common and popular
public-key algorithms for encryption, key exchange, and digital signatures.
These algorithms highlight the roles played by the apparent difficulty of
solving the factoring and discrete-logarithm problems, for designing
public-key protocols.

Two exponential-time integer-factoring algorithms are first covered:
trial division and Pollard's rho method. This is followed by two
sub-exponential algorithms based upon Fermat's factoring method. Dixon's
method uses random squares, but illustrates the basic concepts of the
relation-collection and the linear-algebra stages. Next, I introduce the
Quadratic Sieve Method (QSM) which brings the benefits of using small
candidates for smoothness testing and of sieving.

As the third module, I formally define the discrete-logarithm problem (DLP)
and its variants. As a representative of the square-root methods for solving
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
index calculus method (ICM) as a general paradigm for solving the DLP.
Various stages of the basic ICM are explained both for prime fields and
for extension fields of characteristic two.

```

Basic Search:

```
$ grep method textfile.txt
```

Output:

~~~
trial division and Pollard's rho method. This is followed by two
sub-exponential algorithms based upon Fermat's factoring method. Dixon's
method uses random squares, but illustrates the basic concepts of the
and its variants. As a representative of the square-root methods for solving
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
index calculus method (ICM) as a general paradigm for solving the DLP.
```[78]

**Quoted Pattern with Space:**
```bash
$ grep 'method ' textfile.txt
~~~

Output:

~~~
method uses random squares, but illustrates the basic concepts of the
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
index calculus method (ICM) as a general paradigm for solving the DLP.
```[78]

**Regular Expression with Character Class:**
```bash
$ grep 'method[ \.]' textfile.txt
~~~

Output:

~~~
trial division and Pollard's rho method. This is followed by two
sub-exponential algorithms based upon Fermat's factoring method. Dixon's
method uses random squares, but illustrates the basic concepts of the
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
index calculus method (ICM) as a general paradigm for solving the DLP.
```[78]

**Complex Pattern:**
```bash
$ grep '[a-z]*-[a-z-]*.*\.' textfile.txt
~~~

Output:

~~~
public-key algorithms for encryption, key exchange, and digital signatures.
public-key protocols.
sub-exponential algorithms based upon Fermat's factoring method. Dixon's
relation-collection and the linear-algebra stages. Next, I introduce the
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
```[78]

## 3. grep Options

### 3.1 Multiple Pattern Search (-e)

The `-e` option allows specifying multiple patterns[78].

**Usage:**
```bash
grep -e pattern1 -e pattern2 file
~~~

Example:

```
$ grep -e 'method' -e 'algorithm' textfile.txt
```

Output:

~~~
This tutorial focuses on algorithms for factoring large composite integers
public-key algorithms for encryption, key exchange, and digital signatures.
These algorithms highlight the roles played by the apparent difficulty of
Two exponential-time integer-factoring algorithms are first covered:
trial division and Pollard's rho method. This is followed by two
sub-exponential algorithms based upon Fermat's factoring method. Dixon's
method uses random squares, but illustrates the basic concepts of the
and its variants. As a representative of the square-root methods for solving
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
index calculus method (ICM) as a general paradigm for solving the DLP.
```[78]

**Patterns Starting with Hyphen:**
Without `-e`, patterns starting with `-` are interpreted as options:

```bash
$ grep '-key' textfile.txt
~~~

Output:

~~~
grep: invalid option -- 'k'
Usage: grep [OPTION]... PATTERNS [FILE]...
Try 'grep --help' for more information.
```[78]

**Correct way:**
```bash
$ grep -e '-key' textfile.txt
~~~

Output:

~~~
public-key algorithms for encryption, key exchange, and digital signatures.
public-key protocols.
```[78]

### 3.2 Inverted Search (-v)

The `-v` option prints lines that do NOT contain matches[78].

**Example - Lines without uppercase letters:**
```bash
$ grep -v '[A-Z]' textfile.txt
~~~

Output:

~~~
public-key algorithms for encryption, key exchange, and digital signatures.
solving the factoring and discrete-logarithm problems, for designing
public-key protocols.
method uses random squares, but illustrates the basic concepts of the
candidates for smoothness testing and of sieving.
for extension fields of characteristic two.
```[78]

### 3.3 Case-Insensitive Search (-i)

The `-i` option performs case-insensitive matching[78].

**Normal case-sensitive search:**
```bash
$ grep 'method' textfile.txt
~~~

Output:

~~~
trial division and Pollard's rho method. This is followed by two
sub-exponential algorithms based upon Fermat's factoring method. Dixon's
method uses random squares, but illustrates the basic concepts of the
and its variants. As a representative of the square-root methods for solving
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
index calculus method (ICM) as a general paradigm for solving the DLP.
```[78]

**Case-insensitive search:**
```bash
$ grep -i 'method' textfile.txt
~~~

Output:

~~~
trial division and Pollard's rho method. This is followed by two
sub-exponential algorithms based upon Fermat's factoring method. Dixon's
method uses random squares, but illustrates the basic concepts of the
Quadratic Sieve Method (QSM) which brings the benefits of using small
and its variants. As a representative of the square-root methods for solving
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
index calculus method (ICM) as a general paradigm for solving the DLP.
```[78]

Notice the additional match: "Quadratic Sieve Method (QSM)" due to case-insensitive matching.

### 3.4 Word-Based Search (-w)

The `-w` option matches whole words only[78].

**Example - Lines containing uppercase letters:**
```bash
$ grep '[A-Z]' textfile.txt
~~~

Output:

~~~
Abstract
This tutorial focuses on algorithms for factoring large composite integers
and for computing discrete logarithms in large finite fields. In order to
make the exposition self-sufficient, I start with some common and popular
These algorithms highlight the roles played by the apparent difficulty of
Two exponential-time integer-factoring algorithms are first covered:
trial division and Pollard's rho method. This is followed by two
sub-exponential algorithms based upon Fermat's factoring method. Dixon's
relation-collection and the linear-algebra stages. Next, I introduce the
Quadratic Sieve Method (QSM) which brings the benefits of using small
As the third module, I formally define the discrete-logarithm problem (DLP)
and its variants. As a representative of the square-root methods for solving
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
index calculus method (ICM) as a general paradigm for solving the DLP.
Various stages of the basic ICM are explained both for prime fields and
```[78]

**Lines containing single-letter uppercase words:**
```bash
$ grep -w '[A-Z]' textfile.txt
~~~

Output:

~~~
make the exposition self-sufficient, I start with some common and popular
relation-collection and the linear-algebra stages. Next, I introduce the
As the third module, I formally define the discrete-logarithm problem (DLP)
the DLP, the baby-step-giant-step method is explained. Next, I introduce the
```[78]

### 3.5 Line Numbers and Counting

#### Print Line Numbers (-n)

The `-n` option prints line numbers before matched lines[78].

**Example:**
```bash
$ grep '[^a-zA-Z]$' textfile.txt
~~~

Output:

~~~
public-key algorithms for encryption, key exchange, and digital signatures.
public-key protocols.
Two exponential-time integer-factoring algorithms are first covered:
candidates for smoothness testing and of sieving.
As the third module, I formally define the discrete-logarithm problem (DLP)
index calculus method (ICM) as a general paradigm for solving the DLP.
for extension fields of characteristic two.
```[78]

**With line numbers:**
```bash
$ grep -n '[^a-zA-Z]$' textfile.txt
~~~

Output:

~~~
6:public-key algorithms for encryption, key exchange, and digital signatures.
9:public-key protocols.
11:Two exponential-time integer-factoring algorithms are first covered:
17:candidates for smoothness testing and of sieving.
19:As the third module, I formally define the discrete-logarithm problem (DLP)
22:index calculus method (ICM) as a general paradigm for solving the DLP.
24:for extension fields of characteristic two.
```[78]

#### Count Matches Only (-c)

The `-c` option prints only the count of matching lines[78].

**Example:**
```bash
$ grep -c '[^a-zA-Z]$' textfile.txt
~~~

Output:

~~~
7
```[78]

### 3.6 Recursive Search (-r, -R)

The `-r` or `-R` options search recursively in subdirectories[78].

**Example:**
```bash
$ grep -r 'nodep' .
~~~

Output:

~~~
./libstaque/static/defs.h:typedef node *nodep;
./libstaque/static/stack.h:typedef nodep stack;
./libstaque/static/queue.h:
nodep front;
./libstaque/static/queue.h:
nodep back;
./libstaque/shared/defs.h:typedef node *nodep;
./libstaque/shared/stack.h:typedef nodep stack;
./libstaque/shared/queue.h:
nodep front;
./libstaque/shared/queue.h:
nodep back;
```[78]

### 3.7 List Filenames Only (-l)

The `-l` option prints only the names of files that contain matches[78].

**Example:**
```bash
$ grep -r -l 'nodep' .
~~~

Output:

~~~
./libstaque/static/defs.h
./libstaque/static/stack.h
./libstaque/static/queue.h
./libstaque/shared/defs.h
./libstaque/shared/stack.h
./libstaque/shared/queue.h
```[78]

### 3.8 Show Only Matches (-o)

The `-o` option shows only the matched part of the line (not in PDF, commonly used).

**Example:**
```bash
$ echo "The quick brown fox jumps" | grep -o 'qu[a-z]*'
~~~

Output:

```
quick
```

3.9 Show Context Lines (-A, -B, -C)

Show lines after (-A), before (-B), or around (-C) matches (not in PDF, commonly used).

Examples:

```
$ grep -A 2 -B 1 'pattern' file.txt    # 1 line before, 2 lines after
$ grep -C 3 'pattern' file.txt         # 3 lines before and after
```

3.10 Quiet Mode (-q)

The `-q` option suppresses output, useful in scripts for testing existence.

Example:

```
$ grep -q 'pattern' file.txt
$ echo $?    # 0 if found, 1 if not found
```

3.11 Suppress Filename Display (-h)

When searching multiple files, `-h` suppresses filename display.

Example:

```
$ grep -h 'pattern' file1.txt file2.txt
```

3.12 Show Filename Only (-H)

Force showing filename even for single file.

Example:

```
$ grep -H 'pattern' file.txt
```

4. Pattern Matching with grep

4.1 Extended Regular Expressions (-E)

Use `-E` for extended regular expressions (egrep equivalent).

Additional Metacharacters:

- `+` - one or more
- `?` - zero or one

- `|` - alternation (OR)
- `()` - grouping
- `{n}` - exactly n times
- `{n,}` - n or more times
- `{n,m}` - between n and m times

Examples:

```
# One or more digits
$ grep -E '[0-9]+' file.txt

# Optional 's' at end
$ grep -E 'algorithms?' file.txt

# Either 'method' or 'algorithm'
$ grep -E 'method|algorithm' file.txt

# Grouping with alternation
$ grep -E '(public|private)-key' file.txt

# Exactly 3 digits
$ grep -E '[0-9]{3}' file.txt

# 2 to 4 letters
$ grep -E '[a-z]{2,4}' file.txt

# 5 or more characters
$ grep -E '.{5,}' file.txt
```

4.2 Quantifiers with Braces {n,m}

Exact Count :

```
$ grep -E '[a-z]{5}' file.txt        # exactly 5 lowercase letters
$ grep -E '[0-9]{3}' file.txt        # exactly 3 digits
```

Minimum Count {n,}:

```
$ grep -E '[a-z]{5,}' file.txt       # 5 or more lowercase letters
$ grep -E '[0-9]{3,}' file.txt       # 3 or more digits
```

Range {n,m}:

```
$ grep -E '[a-z]{3,7}' file.txt      # 3 to 7 lowercase letters
$ grep -E '[0-9]{2,4}' file.txt      # 2 to 4 digits
```

4.3 Advanced Pattern Examples

Email Pattern:

```
$ grep -E '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}' file.txt
```

IP Address Pattern:

```
$ grep -E '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}' file.txt
```

Phone Number Patterns:

```
$ grep -E '[0-9]{3}-[0-9]{3}-[0-9]{4}' file.txt     # 123-456-7890
$ grep -E '\([0-9]{3}\)[0-9]{3}-[0-9]{4}' file.txt  # (123)456-7890
```

URL Pattern:

```
$ grep -E 'https?://[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}' file.txt
```

C Function Declaration:

```
$ grep -E '^[a-zA-Z_][a-zA-Z0-9_]*[ ]+[a-zA-Z_][a-zA-Z0-9_]*\(' file.c
```

4.4 Multiple Patterns with File Input (-f)

Store patterns in a file and use `-f` option.

patterns.txt:

```
method
algorithm
^[A-Z]
[0-9]+
```

Command:

```
$ grep -f patterns.txt textfile.txt
```

4.5 Perl-Compatible Regular Expressions (-P)

Use `-P` for Perl-compatible regular expressions (if available).

Examples:

```
$ grep -P '\d+' file.txt              # digits (same as [0-9]+)
$ grep -P '\w+' file.txt              # word characters
$ grep -P '\s+' file.txt              # whitespace characters
$ grep -P '(?=.*method)(?=.*test)' file.txt  # lookahead assertions
```

4.6 Practical Examples from PDF Exercises

Finding printf (not fprintf or sprintf)

```
$ grep -E '\bprintf\b' program.c
```

or

```
$ grep -w 'printf' program.c
```

C Block Analysis

Lines opening blocks but not closing:

```
$ grep '{.*[^}]$' program.c
```

or

```
$ grep '{' program.c | grep -v '}'
```

Lines closing blocks but not opening:

```
$ grep '}' program.c | grep -v '{'
```

Lines both opening and closing blocks:

```
$ grep '{.*}' program.c
```

Number Range Matching (500-5000)

Lines not containing 876:

```
$ grep -v '876' foonums.txt
```

Lines not containing unlucky numbers (500-5000):

```
$ grep -vE '[5-9][0-9]{2,3}|[1-4][0-9]{3}|5000' foonums.txt
```

Lines containing unlucky numbers but not 876:

```
$ grep -E '[5-9][0-9]{2,3}|[1-4][0-9]{3}|5000' foonums.txt | grep -v '876'
```

File Permission Analysis

Files with owner execute permission:

```
$ ls -l | grep '^...x'
```

Non-directory files with owner execute permission:

```
$ ls -l | grep '^-..x'
```

Files >= 1MB (assuming 6+ digits in size column):

```
$ ls -l | grep -E ' [0-9]{7,} '
```

/etc/passwd Analysis

Users with 4-digit UIDs:

```
$ grep -E ':[0-9]{4}:' /etc/passwd
```

Users with bash shell (excluding rbash):

```
$ grep ':/bin/bash$' /etc/passwd
```

Summary of All grep Options

Option	Description	Example
`-e pattern`	Specify pattern (multiple allowed)	`grep -e 'foo' -e 'bar' file`
`-v`	Invert match (show non-matching lines)	`grep -v 'pattern' file`
`-i`	Case-insensitive matching	`grep -i 'Pattern' file`
`-w`	Match whole words only	`grep -w 'cat' file`
`-n`	Show line numbers	`grep -n 'pattern' file`

Option	Description	Example
`-c`	Count matching lines only	`grep -c 'pattern' file`
`-l`	List filenames with matches	`grep -l 'pattern' *.txt`
`-r, -R`	Recursive search	`grep -r 'pattern' directory/`
`-o`	Show only matched parts	`grep -o 'pattern' file`
`-A n`	Show n lines after match	`grep -A 3 'pattern' file`
`-B n`	Show n lines before match	`grep -B 3 'pattern' file`
`-C n`	Show n lines around match	`grep -C 3 'pattern' file`
`-q`	Quiet mode (no output)	`grep -q 'pattern' file`
`-h`	Suppress filename in output	`grep -h 'pattern' *.txt`
`-H`	Show filename even for single file	`grep -H 'pattern' file`
`-E`	Extended regex (egrep)	`grep -E 'pattern|other' file`
`-F`	Fixed strings (fgrep)	`grep -F 'literal.string' file`
`-P`	Perl-compatible regex	`grep -P '\d+' file`
`-f file`	Read patterns from file	`grep -f patterns.txt file`

Complete Regular Expression Reference

Basic Regular Expression Elements

Element	Meaning	Example	Matches
`.`	Any character	`a.c`	abc, axc, a3c
`*`	Zero or more of preceding	`ab*c`	ac, abc, abbc
`^`	Start of line	`^The`	The (at line start)
`$`	End of line	`end$`	end (at line end)
`[abc]`	Any of a, b, or c	`[aeiou]`	vowels
`[a-z]`	Range a through z	`[0-9]`	digits
`[^abc]`	Not a, b, or c	`[^0-9]`	non-digits
`\`	Escape character	`\.`	literal dot

Extended Regular Expression Elements (with -E)

Element	Meaning	Example	Matches
`+`	One or more	`ab+c`	abc, abbc, abbbc
`?`	Zero or one	`ab?c`	ac, abc
`\\|`	Alternation (OR)	`cat\\|dog`	cat or dog
`()`	Grouping	`(ab)+`	ab, abab, ababab
`{n}`	Exactly n times	`a{3}`	aaa
`{n,}`	n or more times	`a{2,}`	aa, aaa, aaaa...
`{n,m}`	Between n and m	`a{2,4}`	aa, aaa, aaaa

This comprehensive guide covers all aspects of regular expressions and grep usage as found in the PDF, plus commonly used extensions and practical examples for real-world usage.

---

1. bash.pdf↩↩↩

2. http://web.stanford.edu/class/archive/cs/cs107/cs107.1262/resources/valgrind (http://web.stanford.edu/class/archive/cs/cs107/cs107.1262/resources/valgrind)↩

3. https://bytes.usc.edu/cs104/wiki/valgrind/ (https://bytes.usc.edu/cs104/wiki/valgrind/)↩

4. https://stackoverflow.com/questions/76698927/why-is-valgrind-ignoring-my-error-exitcode-option (https://stackoverflow.com/questions/76698927/why-is-valgrind-ignoring-my-error-exitcode-option)↩

5. https://stackoverflow.com/questions/11242795/how-to-get-the-full-call-stack-from-valgrind (https://stackoverflow.com/questions/11242795/how-to-get-the-full-call-stack-from-valgrind)↩

6. https://docs.oracle.com/en/operating-systems/oracle-linux/6/porting/ch02s05s02.html (https://docs.oracle.com/en/operating-systems/oracle-linux/6/porting/ch02s05s02.html)↩

7. https://stackoverflow.com/questions/5134891/how-do-i-use-valgrind-to-find-memory-leaks (https://stackoverflow.com/questions/5134891/how-do-i-use-valgrind-to-find-memory-leaks)↩↩

8. https://plus.tuni.fi/graderT/static/compcs300-compcs300-october-2024/lectures/trees/valgrind/tools.html (https://plus.tuni.fi/graderT/static/compcs300-compcs300-october-2024/lectures/trees/valgrind/tools.html)↩

9. https://stackoverflow.com/questions/40810319/valgrind-warning-unknown-option-track-origins-yes (https://stackoverflow.com/questions/40810319/valgrind-warning-unknown-option-track-origins-yes)↩

10. bash_1.pdf↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩↩

11. https://gist.github.com/gaul/5306774 (https://gist.github.com/gaul/5306774)↩

12. https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/guide_valgrind.html (https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/guide_valgrind.html)↩