

Unix and Network Programming
With Linux and C++

Nathan Warner



**Northern Illinois
University**

Computer Science
Northern Illinois University
February 16, 2023
United States

Contents

Commands

- **more, less, pg**: Display contents of file one page at a time
- **head**: Display beginning portion of file (Default: 10 lines)
- **tail**: Display end portion of file
- **wc**: Count file content (-l -w -c) (lines, words, characters)
- **diff**: Compare two files line by line
- **gzip, gunzip, zcat**: compress file content (.gz files)
- **sort**: Sort file contents (-r -n -t -k -f) (reverse, numeric, field delimiter, field1[,field2], ignore case)
- **quota -v**: Disk quota
- **lpr**: Send files to printer, -P to specify printer (lpcl, lpfl, etc)
- **lpq**: Show print queue
- **lprm**: Remove job from print queue

Permissions

Unix uses discretionary access control (DAC) model

- Each directory/file has owner
- Owner has discretion over access control details

With the exception of the super user

2.1 Changing Permissions

There are four categories regarding permissions

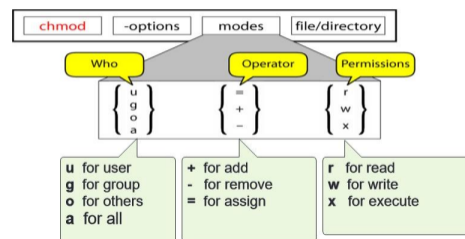
- User
- Group
- Other
- All

To change the permissions of a file, we use the `chmod` command

```
1  chmod -options mode file/directory
```

2.1.1 Changing Permissions: Symbolic mode

Changing Permissions: Symbolic Mode

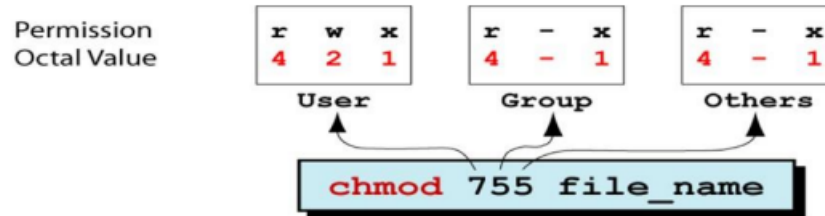


Examples: Symbolic Mode

```
% chmod u-w file.txt  
% chmod u+w file.txt  
% chmod u+x script.sh  
  
% chmod g-w file.txt  
% chmod o-rw file.txt  
  
% chmod ug=rwx play.cc  
% chmod a+wx other.html  
% chmod u+x,go=r script.sh
```

2.1.2 Changing permissions: Octal mode

Changing Permissions: Octal Mode



2.1.3 Exercise: Changing permissions

Suppose we want to change the permissions of "myfile". We want

- Read, write, and execute for user
- Read and execute for group
- Execute for other

```
1 chmod u=rwx, g=rx, o=x myfile
2 chmod 751 myfile
```

2.2 Special Permissions

3 additional permissions can be set on files and directories

- Set user ID (SUID)
- Set group ID (SGID)
- Sticky bit

2.2.1 Set user ID (SUID)

Concept 1: SUID is used for executable files, it makes executables run with permissions of file owner, rather than invoker

For example, the `passwd` command uses this permission. This allows user access to otherwise protected system files while changing password

2.2.2 Set group ID (SGID)

Concept 2:

- **For executables:** The logic for SGID is the same as SUID, but for group owner rather than file owner
- **For directories:** A file created in the directory will be owned by the group owner of the directory, not the group of the user who created the file

2.2.3 Sticky Bit

Concept 3:

- **For executables:** Executable is kept in memory even after it ended
- **For directories:** Files can only be deleted by the user that created it

2.2.4 Displaying special permissions

Concept 4: The `ls -l` command does not display special permission bits. However, since special permissions require execute, they mask the execute permission when displayed with `ls -l`

2.2.5 Setting special permissions (octal)

Setting Special Permissions

suid	sgid	stb	r	w	x	r	w	x	r	w	x
4	2	1	4	2	1	4	2	1	4	2	1
7			7			7			7		
Special			user			group			others		

Use the “chmod” command with octal mode:

- `chmod 7777 filename`

2.2.6 Setting special permissions (Symbolic)

Setting Special Permissions

- chmod with symbolic notation:

u+s	add SUID
u-s	remove SUID

g+s	add SGID
g-s	remove SGID

+s	add SUID and SGID
----	-------------------

+t	set sticky bit
----	----------------

2.3 User mask (umask)

File mode creation mask

- umask (user mask)
 - governs default permission for files and directories
 - sequence of 9 bits: 3 times 3 bits of rwx
 - default:

000 000 010	(002)
-------------	-------

000 010 010	(022)
-------------	-------

 on turing/hopper
- in octal form its bits are removed from:
 - for a file:

110 110 110	(666)
-------------	-------
 - for a directory:

111 111 111	(777)
-------------	-------
- permission for new
 - file:

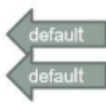
110 110 100	(664)
-------------	-------
 - directory:

111 111 101	(775)
-------------	-------

2.3.1 Examples

User Mask value examples

	Directory Default: 777	File Default: 666
000	777 (rwx rwx rwx)	666 (rw- rw- rw-)
111	666 (rw- rw- rw-)	666 (rw- rw- rw-)
222	555 (r-x r-x r-x)	444 (r-- r-- r--)
022	755 (rwx r-x r-x)	644 (rw- r-- r--)
002	775 (rwx rwx r-x)	664 (rw- rw- r--)
066	711 (rwx --x --x)	600 (rw- --- ---)
666	111 (--x --x --x)	000 (--- --- ---)
777	000 (--- --- ---)	000 (--- --- ---)



2.4 Permissions needed for file and directory actions

2.4.1 Files

- **Read:** View file contents (open, read)
- **Write:** Change file contents
- **Execute:** Run executable file

2.4.2 Directories

- **Read:** List directory contents (Only names)
- **Write:** Change directory contents (need execute aswell)
- **Execute:** Make it current directory, search for files in it
- **Renaming files:** Write and execute permissions on the directory
- **Deleting files:** Write and execute permissions on the directory

2.4.3 Both files and directories

- **Moving files:** Read permissions on file and write permissions on target directory

Network Utilitys

- Login to another computer
 - telnet, rlogin, rsh, ssh
- Copy files to another computer
 - scp
 - ftp, sftp

3.1 Login to another computer

- telnet rlogin, rsh no longer used
 - Transmit username/password without encryption
- ssh
 - Invokes shell on remote computer securely
 - **Used to:** Remote login and run command on remote computer

3.2 ssh

3.2.1 Syntax

```
1  ssh [user@]hostname [command]
```

This command logs in user to hostname, or if command is given, runs it on remote host

3.2.2 Common options

- **-l:** login-name
- **-X:** enable X11 forwarding

3.2.3 Examples

```
➡ % ssh turing.cs.niu.edu
➡ % ssh z123456@hopper.cs.niu.edu
➡ % ssh z123456@hopper.cs.niu.edu w
➡ % ssh -X turing.cs.niu.edu -l z123456
➡ % ssh -X ege@turing.cs.niu.edu thunar
```

3.3 Copy files to another computer

3.3.1 Currently in use

- ftp

3.3.2 Secure, encrypted, part of OpenSSH

- **sftp**: Secure file transfer
- **scp**: Secure copy to remote host

3.4 ftp

3.4.1 Syntax

```
1 ftp hostname
```

This will prompt for userid and password

3.4.2 Anonymous ftp

- **Userid**: ftp or anonymous
- **Password**: Your email address

3.4.3 Commands

- **help**
- **ls**
- **cd**
- **put, get**
 - copy a file from local to remote host, or vice versa
- **mput, mget**
 - put/get multiple files, can use wildcards
- **bye**

3.5 sftp (Secure file transfer)

3.5.1 Syntax

```
1 sftp user@hostname
```

- Will prompt for password
- Same commands as ftp

3.6 scp

3.6.1 Syntax

```
1 scp source target
```

- source and target use extended form of pathname

```
1 user@host:pathname
```

3.6.2 Common options

- **-r**: Recursively copy entire directories
- **-C**: Enables compression
- **-l**: Limit bandwidth, specified in Kbit/s

3.6.3 Examples

```
1 scp screenshot.png z123456@turing.cs.niu.edu:  
2 scp z123456@hopper.cs.niu.edu:assign1.cc .
```

Shell: Part 1

4.1 Basics

4.1.1 Customization

- variables, prompt, aliases

4.1.2 Command line behavior

- history
- sequence and substitution
- redirections and pipe

4.2 Predefined variables

- **HOME**: full pathname of home directory
- **PATH**: list of directories to search for commands
- **USER**: your user name, also UID for user id
- **SHELL**: full pathname of your login shell
- **PWD**: Current working directory
- **HOSTNAME**: current hostname of the system
- **HISTSIZE**: Number of commands to remember
- **PS1**: Primary prompty (also PS2, ...)
- **?**: Return status of most recently executed command
- **\$**: Process id of current process

4.3 Customizing bash shell prompt

Can be set via the PS1 shell variable

4.3.1 Example:

```
1 PS1="$USER > "
```

4.3.2 Special PS1 shell variable settings

- `\w`: current working directory
- `\h`: hostname
- `\u`: username
- `\d`: date
- `\t`: time
- `\a`: ring the "bell"

4.4 Customization

- Via command line options: Rarely done
- Instead we use startup initialization file
 - `~/profile` (login session shell)
 - `~/bashrc` (invoked from command line)

Note: we also have `/etc/profile` and `/etc/bash.bashrc`

4.5 Command line behavior

- History
- Sequence
- Substitution
- I/O redirection and pipe

4.6 Shell history

- Record of previously entered commands
 - can be re-called, edited, and re-executed
- Size of history is set via shell variables
 - `HISTSIZE=500` (per session)
 - `HISTFILESIZE=100` (per user)

4.6.1 Syntax

```
1 history [-c] [count]
```

Where `-c` is used to search for specific text

Note:-

We use arrow keys to navigate, delete and backspace to remove, and tab to execute command

4.7 Command substitution

4.7.1 Backticks

The first method is using backticks

- Command surrounded by back quotes “ is run and replaced by its standard output
- Newlines in the output are replaced by spaces

4.7.2 Dollar sign parenthesis notation

Alternatively, we use the following syntax

```
1 $(command)
```

4.8 Here document

This uses <<. With this we can read input for current source

4.8.1 Syntax

```
1 command << LABEL
```

Reads following lines until line starting with "LABEL"

4.8.2 Example:

```
1 wc -l << DONE
2   > line one
3   > line two
4   > DONE
5 2
```

4.9 File Descriptor

- Positive integer for every open file
- Process tracks its open files with this number
 - 0 - standard input
 - 1 - standard output
 - 2 - standard error output
- Bash can use file descriptor to refer to a file

4.9.1 Table of redirection operators

Table 1: Common Redirection Syntaxes in Linux

Syntax	Description
> or 1>	Redirects standard output (stdout) to a file, overwriting the file.
>> or 1>>	Redirects standard output (stdout) to a file, appending to the file.
2>	Redirects standard error (stderr) to a file, overwriting the file.
2>>	Redirects standard error (stderr) to a file, appending to the file.
&>	Redirects both standard output and standard error to a file, overwriting the file.
>&	Redirects both standard output and standard error to a file, overwriting the file.
< or 0<	Redirects a file to standard input (stdin).
<<	Here document: redirects inline input to standard input.
<<<	Here string: redirects a single line of input to standard input.

4.9.2 Combining redirection

Before > was introduced, the way to redirect both stderr and stdout to a file looked something like

```
1  command > file 2>&1
```

This tells bash to redirect stdout from command to file, and then redirect stderr (2) to stdout (which is now pointing to file). Note that the order does matter, for example the following will not work

```
1  command 2>&1 > file
```

This would redirect stderr to wherever stdout is pointing, and then redirect stdout to file. Thus not bringing along stderr

4.10 The pipe

Bash introduced a concise way to redirect the output from a command as the input to some other command. For this, we use the pipe (`|`)

```
1  command1 | command2
```

Before the pipe syntax, we would have to do something like

```
1  command1 > file; command2 < file
```

4.11 Wildcards

Bash has special characters known as wildcards, these allow us to match filenames on the command line

- `*` (asterisk): Matches zero or more characters
- `?` (question mark): Matches exactly one character (any character)
- `[...]` (characters enclosed by brackets): Matches any of the enclosed characters
- `[a-z]` (range syntax): matches any characters in the specified range
- `{word1, word2,...}` (brace syntax): Similar to the brackets, but for words

4.11.1 Character classes

Character classes are defined by the bracket syntax, as we say in the previous list.

```
1  ls test[a-z] # Matches test followed by any character a-z
2  ls test[a-z]* # Matches test followed by any character a-z
   ↳ followed by any character
3  ls test[^a-z] # matches test followed by any character not found
   ↳ within the range a-z
4  ls test[!a-z] # matches test followed by any character not found
   ↳ within the range a-z
5  ls test[123] # matches test1, test2, or test3
```

4.11.2 Posix character classes

POSIX character classes are a special notation used in regular expressions and pattern matching that provides a locale-independent way to specify groups of characters.

```
1  ls [[:upper:]]*
```


The first set of brackets defines a bash character class, and within that we define a posix character class. We also have

- `[:alpha:]`: Matches any letter.
- `[:digit:]`: Matches any digit.
- `[:lower:]`: Matches any lowercase letter.
- `[:upper:]`: Matches any uppercase letter.
- `[:space:]`: Matches any whitespace character (including spaces, tabs, and form feeds).
- `[:alnum:]`: Matches any alphanumeric character (letters and digits).

Shell Scripts

5.1 Local / Global Variables

Concept 5: In the Bash shell, variables defined within functions are not automatically local to those functions. By default, **variables in Bash functions are global**, meaning if you define or modify a variable within a function without explicitly declaring it as local, its value is accessible and can affect the script outside the function.

5.1.1 Local variables

To make a variable local to a function, we prefix the variable with the keyword `local`

```
1 function fn() {  
2     local x=12  
3 }
```

5.1.2 Return from function

The `return` keyword ends the execution of the function, optional return value sets return status

5.1.3 Exit command

The `exit` command is similar to `return` but it will exit the program

$$f(x, y) = 3x + y.$$

find

$$f(< 2, 1 > + h < 0, 1 >).$$

5.2 Functions

5.2.1 Syntax

Functions are defined in the following way

```
1 function name() {  
2     body ...  
3 }  
4 name # Call to function
```

5.2.2 Parameters

In Bash scripting, unlike many programming languages where you define function parameters within parentheses next to the function name, you do not specify parameters in the function definition itself. Instead, you simply pass the arguments to the function when you call it, and within the function, you access these arguments using 1,2, etc., without declaring them in the function definition.

```
1  #!/bin/bash
2
3  # Define a function without specifying parameters in the
   ↪ definition
4  function add {
5      # Access parameters using 1,2, etc.
6      local sum=$(( $1 + $2 ))
7      echo "The sum is: $sum"
8  }
9
10 # Call the function with arguments
11 add 5 7
```

5.3 Echo

The `echo` command is a simple way to write to standard output

5.3.1 Syntax

```
1  echo [-ne] arg[s]
```

- **-n** Suppresses trailing newlines
- **-e** Enables escapes
 - `\t` tab
 - `\b` backspace
 - `\a` alert
 - `\n` newline

5.4 Printf

`printf` is a command similar to `echo`, but it allows us to format the output

5.4.1 Syntax

```
1 printf format [arg[s]]
```

5.4.2 Examples

```
1 printf "\%-10s" "Hello world" #Left justified with field width
   ↪ of 10
```

5.5 The set command

The set command is a shell builtin that allows for debugging by tracing the execution

5.5.1 Options

- **-v** print shell input lines as they are read
- **-x** displays expanded commands and its arguments

5.5.2 Turning options on or off

```
1 set -xv
2 set +xv
```

5.5.3 Setting args withing the shebang

The options described above can also be set directly in the shebang

```
1 #!/usr/bin/bash -xv # Turns on the options
```

5.6 The until loop

The until loop is essentially the opposite of the while loop, where the while loop executes the body as long as the test is true, the until loop will execute its body while the test is false ("until" the body is true)

5.6.1 Syntax

```
1  until test-command; do
2      body ...
3  done
```

5.6.2 Example

```
1  a=1
2
3  until [[ $a -gt 5 ]]; do
4      echo $a
5      a=$(( $a + 1 ))
6  done
```

5.7 Simple looping over args

```
1  for parm; do
2      echo $parm
3  done
```

Note:-

The name is not important

Awk

6.1 What is awk?

Awk is a scripting language used for manipulating data and generating reports. The awk command programming language requires no compiling and allows the user to use variables, numeric functions, string functions, and logical operators.

Awk is a utility that enables a programmer to write tiny but effective programs in the form of statements that define text patterns that are to be searched for in each line of a document and the action that is to be taken when a match is found within a line. Awk is mostly used for pattern scanning and processing. It searches one or more files to see if they contain lines that matches with the specified patterns and then perform the associated actions.

6.2 Awk operations

- (a) Scans a file line by line
- (b) Splits each input line into fields
- (c) Compares input line/fields to pattern
- (d) Performs action(s) on matched lines

6.3 Syntax

```
1  awk options 'selection_criteria { action }' input-file >
```

6.4 Options

- **-f**: Reads the AWK program source from the file program-file, instead of from the first command line argument
- **-F**: Change the field separator

6.4.1 Example

```
1  # File that contains awk commands "filename: actions"
2
3  { print $1 }
4
5  # File that contains data "filename: data"
6
7  0.999,0.18179,0.2711768644
8  1.0323,0.195853,0.47739746
9  1.0656,0.217119,0.6253140641
10
11 # In command line
12 aws -F ',' -f actions data
13
14 # Output
15 0.999
16 1.0323
17 1.0656
```

6.5 The print action

One of the simplest actions we can preform with awk is the print action

6.5.1 Example

```
1  awk '{ print }' <filename>
2  awk '{ print $1}' <filename>
3  awk '{ print $1,$3}' <filename>
```

Awk will scan the file line by line, split the input line into fields, with space as the field separator, and then perform the action print. With a bare print statement, each line will be printed. We can of course be more specific with our statement. We can access each field with \$0-\$n, where \$0 is the entire line, and \$n is a specific field number

6.6 Regex

We can as you might expect use regular expressions with awk, let's take a look

6.6.1 Whole lines

```
1  awk '/foo/ { print }' <filename>
```

This will print all lines that contain the word 'foo'

6.6.2 Field matching

```
1 man date | awk '$0~/day/'
2 man date | awk '$0!~/day/'
```

Here we use `awk` on the manual for the `date` command, and we output any line that contains the word `day`. In the second line, we do the same thing, but output any line that does not contain the word `day`.

6.6.3 Range matching

```
1 man date | awk '/DESCRIPTION/,/EXAMPLES/'
```

6.7 Other Awk variables

- **NR**: Number of the current record
- **NF**: Number of fields in the current record
- **FS**: Field separator

6.8 BEGIN and END blocks

- **BEGIN{...}** Executes whatever is inside the brackets before starting to view the input file

```
1 awk 'BEGIN { FS=":" } { print $1 }' /etc/passwd
```

This command sets the field separator to `:` before processing any lines of the input file `/etc/passwd`. It then prints the first field of each line, which typically contains the usernames in a Unix/Linux system.

- **END{...}** Executes whatever is inside the brackets after `awk` is finished reading the input file

```
1 awk 'BEGIN { count=0 } { count++ } END { print "Number of
↪ lines: ", count }' /etc/passwd
```

Before processing the file, it initializes a counter `count` to 0. For each line in `/etc/passwd`, it increments `count`. After processing all lines, it prints the total count, effectively giving the number of lines in the file.

6.9 Typical awk script



Note:-

The symbol for comments are the same as bash, the hashtag #

6.10 If statements in awk commands

6.10.1 Syntax

awk supports conditional statements, allowing you to perform actions based on specific conditions. The syntax for an if statement in awk is similar to that in other programming languages. Here's the basic structure:

```
1  if (condition) {  
2      // actions to perform if condition is true  
3  } else {  
4      // actions to perform if condition is false  
5  }
```

Note:-

Awk does not have the concept of elseif

6.10.2 Example

```
1  echo -e "3\n-1\n0\n5" | awk '{ if ($1 > 0) print $1 " is  
↪ positive"; else print $1 " is not positive" }'
```

6.10.3 Checking for existence in array

Similar to python, we can do the following

```
1  if (item in array) {
2      ...
3  } else {
4      ...
5  }
```

6.11 For loops in awk commands

6.11.1 Syntax

```
1  for (initialization; condition; increment) {
2      // Code block to execute
3  }
```

6.11.2 Example

```
1  awk 'BEGIN {
2      for (i = 1; i <= 5; i++) {
3          print "Number is", i
4      }
5  }'
```

6.11.3 Iterating Over an Array

```
1  for (index in array) {
2      // Code block using array[index]
3  }
```

6.12 The while loop in awk commands

6.12.1 Syntax

```
1 while (condition) {  
2     // Code block to execute  
3 }
```

6.12.2 Example

```
1 awk '{  
2     while ($0 !~ /stop/) {  
3         print "Processing:", $0  
4         if (getline <= 0) break  
5     }  
6 }' input.txt
```

6.13 Awk variables

In awk, we can define variables with an assignment statement. We can define integers, strings, and arrays. Variables only come into existence when first used. Variables are initialized to either 0 or ""

6.14 Awk arrays

Awk allows the creation of one dimensional arrays, the index (and the elements) can be either a number or a string. Unlike arrays in *c*, we need not declare the size or type of the array. The array elements are created when first used and initialized either 0 or ""

Note:-

When we use strings as indices, we are essentially creating a map

6.14.1 Syntax

```
1 arrayName[index] = value
```

6.14.2 Examples

```
1 arr[1] = "some value"  
2 arr[2] = 25  
3 arr["foo"] = "bar"
```

6.14.3 Example: Process sales data

Suppose we have the data file

1	clothing	3141
1	computers	9161
1	textbooks	21321
2	clothing	3252
2	computers	12321
2	textbooks	154622
	supplies	2242

We wish to output a summary of department sales, then we could write an awk program

```
1 # p.awk (awk -f p.awk data)
2 {
3     deptsales[$2] += $3
4 }
5 END {
6     for (x in deptsales) {
7         print x, deptsales[x]
8     }
9 }
```

6.15 Builtin Functions

6.15.1 Arithmetic

- sqrt
- rand

6.15.2 String

- index
- length
- split
- substr
- sprintf
- tolower
- toupper

6.15.3 Misc

- system
- systime

6.16 The split function

6.16.1 Syntax

```
1 split(string, array, fieldsep)
```

The split function divides **string** into pieces separated by **fieldsep** and stores the pieces in **array**. If **fieldsep** is omitted, the value of FS is used.

6.16.2 Example

```
1 split("26:Miller:Comedian", fields, ":")
```

Now we have a fields array with the contents of the string

6.17 The gsub function

The gsub function in AWK is a powerful tool for performing global search and replace operations on strings. It searches for all occurrences of a pattern in a string and replaces them with a specified replacement text.

6.17.1 Syntax

```
1 gsub(regex, replacement, target)
```

Stream Editor (sed)

7.1 What is sed?

Sed is a non-interactive stream editor. We use sed to

- Automatically perform edits on files
- Simplify doing the same edit on multiple files
- Write conversion programs
- Do editing operations from shell script

7.2 Syntax

```
1 sed -e 'address command' input_file # (inline script)
2 sed -f script.sed input_file # (script file)
```

7.3 How does sed work?

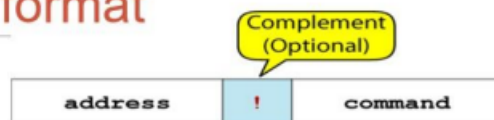
Sed reads files line by line, each line of input is copied into a temporary buffer called the **pattern space**, then the editing instructions are applied to line in the pattern space. Next, the line is sent to output (unless -n was used). Last, the line is removed from the pattern space. Sed does this for all lines until the end of the file

Note:-

Input file is unchanged unless -i option is used

7.4 Instruction format

sed instruction format



The address determines which lines in the input file are to be processed by the commands, if no address is given, then the command is applied to each input line

7.4.1 Address types

- Single-line address
- Set-of-lines address
- Range address

7.4.2 Single-line address

A Single-line address specifies only one line in the input file

Note:-

The dollar sign denotes the last line of input file

```
1 sed -n -e "3 p" infile # Show only line 3
2 sed -n -e "$ p" infile # Show last line
3 sed -e "10 s/endif/fi/" infile # Substitute "endif" with "fi" on
  ↪ line 10
```

7.4.3 Set-of-lines address

With this, we can use regex to match lines

- Written between two slashes
- Process only lines that match
- May match several lines
- Lines don't have to be consecutive

```
1 sed -i -e "/Key/ s/more/other" infile
2 sed -n -e "/r..t/ p" input-file
```

7.4.4 Range address

Defines a set of consecutive lines. Format: startAddr,endAddr (inclusive)

Examples:

- 10,50
- 10,/funny/

7.4.5 Address complement

Address with an exclamation point (!). Command applies to lines that **don't** match the address

```
1 sed -n -e '/Obsolete/!p' infile # (print lines that do not  
  ↪ contain "Obsolete")
```

7.5 Sed commands

7.5.1 Modify

- insert
- append
- change
- delete
- substitute

7.5.2 Input/Output

- Next, print
- Read, write

7.5.3 Other

- quit

7.6 Commands i,a,c

- **i** adds lines before the address
- **a** adds lines after the address
- **c** replaces an entire matched line with new text

7.6.1 Syntax

```
1 [address] i\  
2 text
```

7.7 Delete command

Deletes the entire pattern space

Note:-

Commands following the delete command are ignored since the deleted text is no longer in the pattern space

7.7.1 Syntax

```
1 [address] d
```

7.8 Substitute command (s)

7.8.1 Syntax

```
1 [address] s/search/replacement/[flag]
```