

**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

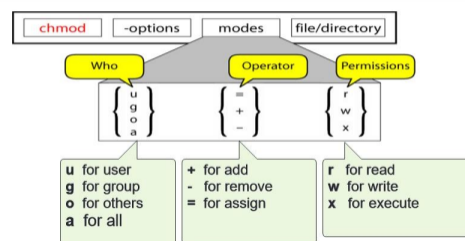
- User
- Group
- Other
- All

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

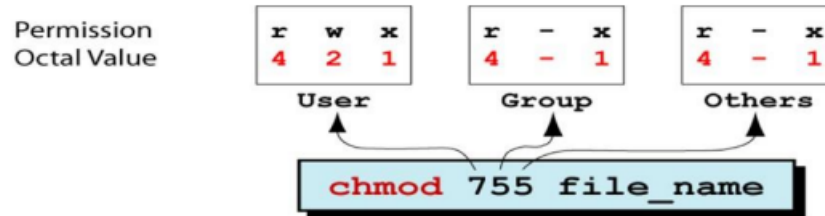
% chmod g-w file.txt
% chmod o-rw file.txt

% 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: 

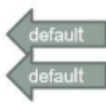
110	110	100	(664)
-----	-----	-----	-------
  - directory: 

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 ( <b>stdout</b> ) to a file, overwriting the file.
>> or 1>>	Redirects standard output ( <b>stdout</b> ) to a file, appending to the file.
2>	Redirects standard error ( <b>stderr</b> ) to a file, overwriting the file.
2>>	Redirects standard error ( <b>stderr</b> ) 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 ( <b>stdin</b> ).
<<	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.