



1

Contents

- Overview of UNIX
 - Structures
 - File systems
 - absolute and relative pathname
 - security -~~rwx~~-x--x
 - Process:
 - has return value 0 (success) or non 0 (sth wrong)
 - communication: pipes who | sort who | grep Wang | wc -l
- Utilities/commands
 - Basic `mkdir`, `cat`, `cp`, `rm`, `mv`, `file`, `wc`, `chmod`
 - Advanced `grep/egrep`, `uniq`, `sort`, `diff/cmp`, `cut`, `find`
- Shell (common shell functionalities)
- Bourn (again) Shell
 - scripting language

2

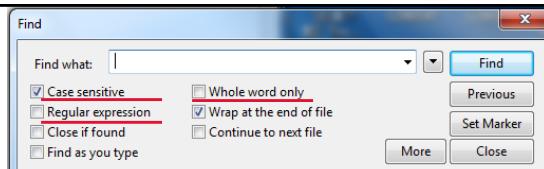
Utilities II – advanced utilities

Introduces utilities for power users, grouped into logical sets
We introduce about thirty useful utilities.

| Section | Utilities |
|------------------------------|--|
| Filtering files | egrep, fgrep, grep, uniq |
| Sorting files | sort |
| Comparing files | cmp, diff |
| Archiving files | tar, cpio, dump |
| Searching for files | find |
| Scheduling commands | at, cron, crontab |
| Programmable text processing | awk, perl |
| Hard and soft links | ln |
| Switching users | su |
| Checking for mail | biff |
| Transforming files | compress, crypt, gunzip, gzip, sed, tr, cut , ul, uncompress od |
| Looking at raw file contents | mount, umount |
| Mounting file systems | whoami |
| Identifying shells | nroff, spell, style, troff |
| Document preparation | time |
| Timing execution of commands | |

3

Searching for Regex: grep



\$ grep -w the inputFile.txt # -w: Whole word only

line2 So turn off **the** light,

line5 Beautiful mermaids will swim through **the** sea,

\$ grep -v -w the inputFile.txt # -v: reverse the filter.

line1 Well you know it's your bedtime,

line3 Say all your prayers and then,

line4 Oh you sleepy young heads dream of wonderful things,

line6 And you will be swimming there too.

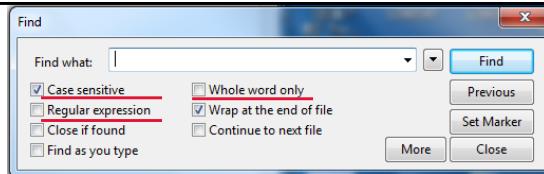
\$ grep -i -w the inputFile.txt # ignore case, default case sensitive

\$ grep -w Li EECS2031A # who have family name Li?

\$ grep -w Li EECS2031A | wc -l # how many ?

4

Searching for Regex: grep



```
$ cat info.txt      # -w: Whole word only
```

```
Name  Age
Judy   30
Sue    22
Joe    27
```

```
# want only the data, not the header?
```

```
$ grep -v Name info.txt  # -v: filter out the header
```

```
Judy   30
Sue    22
Joe    27
```

5

5

| Utility | Kind of pattern that may be searched for |
|---------|--|
| fgrep | fixed string only |
| grep | regular expression |
| egrep | extended regular expression |

Regular Expressions

Regular Expressions: Exact Matches

regular expression → cks \$ grep cks inputFile.txt

UNIX Tools rocks.

↑
match

UNIX Tools sucks

↑
match

UNIX Tools is okay.

no match



7

Regular Expressions: Matching Any Character

- The `.` regular expression can be used to match any character.

regular expression → O. \$ grep o. inputFile.txt

Force me to put on that

↑
match 1

↑
match 2

\$ grep -w o. inputFile.txt ?



8

Regular Expressions: Alternate Character Classes

- Character classes `[]` can be used to match any specific set of characters.

regular expression → `b [eor] a t`

\$ grep b[eor]at inputFile.txt

beat a brat on a boat

match 1 match 2 match 3

- `[aeiou]` will match any of the characters a, e, i, o, u
- `[kK]orn` will match `korn` or `Korn`



9

Regular Expressions: Negated Character Classes

- Character classes can be negated with the `[^]` syntax.

regular expression → `b [^eo] a t`

\$ grep b[^eo]at inputFile.txt

beat a brat on a boat

no match match no match

10

`scanf ("%[^\\n]s", str);`



10

Regular Expressions: Other Character Classes

- Other examples of character classes:

- [0123456789] will match any digit
- [abcde] will match a b c d e

- Ranges can also be specified in character classes

[0-9] is the same as [0123456789] \$ grep [0-9] inputFile.txt
[a-e] is equivalent to [abcde]

- You can also combine multiple ranges

[abcde123456789] is equivalent to [a-e1-9]
[a-zA-Z] all the letters

11

Regular Expressions: Anchors

- Anchors are used to match at the beginning or end of a line (or both).

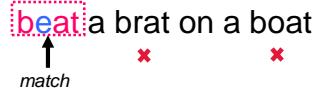
^ means beginning of the line

^ the begin with the

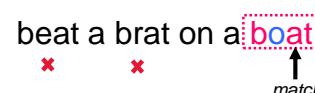
\$ means end of the line

the\$ end with the

regular expression → 

\$ grep ^b[eor]at inputFile.txt


regular expression → 

\$ grep b[eor]at\$ inputFile.txt




12

Regular Expressions: Anchors

- **Anchors** are used to match at the beginning or end of a line (or both).

 ^ means **beginning** of the line

 ^ the **begin** with the

 \$ means **end** of the line

 the \$ **end** with the

```
$grep cse EECS2031A
```

```
indigo 339 % grep cse EECS2031A
cse*****      ****       Yu Ying
cse*****      ****       Lee JunXu
eqao          cse*****   Tong Treacy
indigo 340 %
```

```
$grep ^cse EECS2031A
```

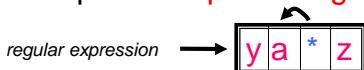
```
indigo 340 % grep ^cse EECS2031A
cse*****      ****       Yu Ying
cse*****      ****       Lee JunXu
indigo 341 %
```

13

Regular Expression: Repetitions

“Kleene Star”

- The ***** is used to define **zero or more** occurrences of the **single regular expression preceding it**.



```
$ grep ya*z inputFile.txt
```

I got mail, 

zero or more occurrences of 'a' (between y z)
yz yaz yaaz yaaaz

```
$ grep oa*o inputFile.txt
```

regular expression → 

For me to 

zero or more occurrences of 'a' (between o o)
oo oao oaaa ooaaac

14

14

Extended Regular Expressions: Repetition Shorthands

- The ***** (star) has already been seen to specify zero or more occurrences of the immediately preceding character
- The **?** (question mark) specifies an optional character, the single character that immediately precedes it
 - July?** will match **Jul** or **July** zero or one occurrence of y
 - Equivalent to **(Jul|July)**
 - abc?d** will match **abd** and **abcd** but will not match **abcccd**
 - x
- The **+** (plus) means one or more occurrence of the preceding character
 - abc+d** will match **abcd**, **abcccd**, or **abcccccccd** but will not match **abd**
 - x
 - one or more occurrence of c

15

15

Repetition recap

| Regex | Meaning |
|-------|-------------|
| a* | 0 or more a |
| a? | 0 or one a |
| a+ | 1 or more a |

- ab*c** matches **ac abc abbc abbbc abbbbc**
- ab?c** matches **ac abc**
- ab+c** matches **abc abbc abbbc abbbbc**



16

grep and egrep RE

| Pattern | Meaning | Example |
|---------|--------------------------------------|------------|
| c | Non-special, matches itself | 'tom' |
| \c | Turn off special meaning | '\$' |
| ^ | Start of line | 'ab' |
| \$ | End of line | 'ab\$' |
| . | Any single character | '.nodes' |
| [...] | Any single character in [] | '[tT]he' |
| [^...] | Any single character not in [] | '[^tT]he' |
| R* | Zero or more occurrences of R | 'e*' |
| R? | Zero or one occurrences of R (egrep) | 'e?' |
| R+ | One or more occurrences of R (egrep) | 'e+' |
| R1R2 | R1 followed by R2 | '[st][fe]' |
| R1 R2 | R1 or R2 (egrep) | 'the The' |

17

17

grep and egrep RE

| Pattern | Meaning | Example |
|---------|--------------------------------------|------------|
| c | Non-special, matches itself | 'tom' |
| \c | Turn off special meaning | '\$' |
| ^ | Start of line | 'ab' |
| \$ | End of line | 'ab\$' |
| . | Any single character | '.nodes' |
| [...] | Any single character in [] | '[tT]he' |
| [^...] | Any single character not in [] | '[^tT]he' |
| R* | Zero or more occurrences of R | 'e*' |
| R? | Zero or one occurrences of R (egrep) | 'e?' |
| R+ | One or more occurrences of R (egrep) | 'e+' |
| R1R2 | R1 followed by R2 | '[st][fe]' |
| R1 R2 | R1 or R2 (egrep) | 'the The' |

18

18

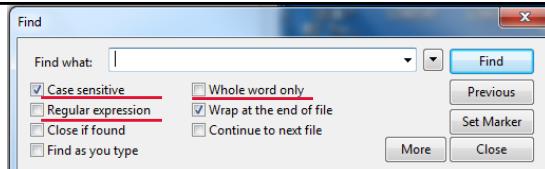
grep and egrep RE

| Pattern | Meaning | Example |
|---------|--------------------------------------|------------|
| c | Non-special, matches itself | 'tom' |
| \c | Turn off special meaning | '\$' |
| ^ | Start of line | 'ab' |
| \$ | End of line | 'ab\$' |
| . | Any single character | '.nodes' |
| [...] | Any single character in [] | '[tT]he' |
| [^...] | Any single character not in [] | '[^tT]he' |
| R* | Zero or more occurrences of R | 'e*' |
| R? | Zero or one occurrences of R (egrep) | 'e?' |
| R+ | One or more occurrences of R (egrep) | 'e+' |
| R1R2 | R1 followed by R2 | '[st][fe]' |
| R1 R2 | R1 or R2 (egrep) | 'the The' |

Don't get confused with UNIX metacharacter (filename wildcards) → ls file*.c *java
cp xFile?.c . one any

19

Searching for Regex: grep



```
$ grep ^[tT]he inputFile.txt      # begins with the or The
```

```
$ grep [0-9]x inputFile.txt      # contains digits followed by 'x'
```

```
$ grep ^[a-z] inputFile.txt      # begins with a lower case letter
```

```
$ grep .nd inputFile.txt        # contains one any character followed by nd
```

```
$ grep [ab]nd$ inputFile.txt     # ends with 'and' or 'bnd'
```

```
$ grep -w W[ao]ng EECS2031A    # who have family name Wang or Wong
```

```
$ grep -w W[ao]ng EECS2031A | wc -l          # how many ?
```

```
$ ls -l | grep webapp           # who submitted using web submission?
```

20

| Utility | Kind of pattern that may be searched for |
|---------|--|
| fgrep | fixed string only |
| grep | regular expression |
| egrep | extended regular expression |

- Since many of the [special characters](#) used in regexs (e.g., `*` `?` `|`) also have [special meaning to the shell](#), it's a good idea to get in the habit of [quoting](#) your regexs
 - This will protect any special characters from being operated on by the shell
 - If you habitually do it, you won't have to worry about when it is necessary

Need in
tsh

| | | | |
|---|---|---|------------------------|
| <code>\$egrep the lyrics</code> | <code>\$egrep 'the' lyrics</code> | <code>\$egrep "the" lyrics</code> | |
| <code>\$egrep ab? lyrics</code> | <code>\$egrep "ab?" lyrics</code> | <code>\$egrep 'ab?' lyrics</code> | Explained next chapter |
| <code>\$egrep ab*c lyrics</code> | <code>\$egrep 'ab*c" lyrics</code> | <code>\$egrep 'ab*c' lyrics</code> | |
| | | | |
| <code>\$egrep -w Chan Chen classlist</code> | | | |
| <code>\$egrep -w "Chan Chen" classlist</code> | | <code>\$egrep -w 'Chan Chen' classlist</code> | |

- `grep` may behave differently in different shells.
- So for this course
 - Work on [Bourne \(again\) shell](#) (issue `sh` or `bash`)
 - Use `grep -E` or `egrep`



21

Exit code of grep/egrep

Matching found: 0 No matching: 1 No such file: 2

```
$ grep Wang EECS2031A
.....
.....
$ echo $?      # display its exit value.
0              # indicates success.

$ grep Leung EECS2031A
$ echo $?
1              # indicates failure (not matching).

$grep Wang EECS2038A
grep: EECS2038A: No such file or directory
$ echo $?
2              # indicates failure (not such a file).
```

Look for man
`man grep | grep -w "exit"`

Used in scripting



22

Utilities II – advanced utilities

Introduces utilities for power users, grouped into logical sets
We introduce about thirty useful utilities.

| Section | Utilities |
|------------------------------|---|
| Filtering files | egrep, fgrep, grep, uniq |
| Sorting files | sort |
| Extracting fields | cut |
| Comparing files | cmp, diff |
| Archiving files | tar, cpio, dump |
| Searching for files | find |
| Scheduling commands | at, cron, crontab |
| Programmable text processing | awk, perl |
| Hard and soft links | ln |
| Switching users | su |
| Checking for mail | biff |
| Transforming files | compress, crypt, gunzip, gzip, sed, tr, ul, uncompress |
| Looking at raw file contents | od |
| Mounting file systems | mount, umount |
| Identifying shells | whoami |
| Document preparation | nroff, spell, style, troff |
| Timing execution of commands | time |

23

find Utility

find pathList expression

- finds files starting at pathList
- finds files descending from there

```
find . -name "lab3a.c"
```

-
- allows you to perform certain actions on results
 - e.g., copying (**cp**), renaming (**mv**), deleting (**rm**) the files

“Find file lab3a.c and rename it to lab3a.bak”

```
find . -name "lab3a.c" -exec mv {} {}.bak \;
```

“Find all the Java class files and delete them”

```
find . -name "*.class" -exec rm {} \;
```



24

find Utility

- `-name pattern`
True if file's name matches *pattern*, which include shell
metacharacters * ? []
- `-mtime count`
True if the content of the file has been modified within *count* days
- `-atime count`
True if the file has been accessed within *count* days
- `-ctime count`
True if the contents of the file have been modified within *count* days or
any of its file attributes have been modified
- `-type -maxdepth`
- `-exec command`
True if the exit code = 0 from executing the command.
 - *command* must be terminated by `\;`
 - If `{}` is specified as a command line argument, it is replaced by the
file name currently matched

25

find examples

- `$ find . -mtime 14` # search for files/dir modified in the last 14
days in current and subdirectories
- `$ find / -name x.c` # search for file/dir named x.c in the entire file
system
- `$ find . -name '*.bak'` # "*.bak" search for all bak files in
current and subdirectories
- `$ find . -name 'a?.c'` # "a?.c" search for all file/dir named aX.c
 - a1.c
 - a2.c
 - a3.c
- `$ find . -name 'a?.c' | wc -l` # how many

26

find examples

- `$ find / -type f # search for files (only), in the entire file system`
- `$ find . -type f -mtime 14 # search for files (only) modified in the last 14 days, in current and subdirectories`
- `$ find . -type d -name 'lab*' # search for all directories named lab* in current and subdirectories`
- `$ find . -maxdepth 1 -type f -name 'lab*' # search for all file (only) named lab*, in current directory only (no sub-directories)`
- `$ find . -type f -name 'a?' | wc -l # how many files (no directory) with name a?, in current and subdirectories`

27

For your information

- `find examples -exec`
- `$ find . -name '*.bak' -exec rm {} \;`
remove all files that end with .bak
- `$ find . -name 'a?.c' -exec mv {} {}.bak \;`
find aX.c files and then rename them to aX.c.bak
`mv a1.c a1.c.bak`
`mv a2.c a2.c.bak`
`mv a3.c a3.c.bak`
- `$ find . -name '*.c' -exec cp {} {}.2019SU \;`
find all c files and then copy them to filename.c.2019SU
`cp a1.c a1.c.2019SU`
`cp lab3b.c lab3b.2019SU`
....
- `$ find . -name '*.c' -exec mv {} ..archive/2019SU \;`
find all c files and move them to directory ..archive/2019SU
- `$ find . -name '*.c' -exec chmod 770 {} \;`
find all c files and change mode to rwxrwx---

28



Utilities II – advanced utilities

Regular Expression

grep/egrep

```
grep -w -i ^[Tt]he file123
```

sort

```
sort -t : -k 4 -r -n/M file
```

default delimiter:
blank/tab

cut

```
cut -d " " -f 2,3 file
```

Default delimiter: tab

find

```
find . -name "* .c" -exec
```

```
cp {} {}.bak \;
```

Default: subdirectories
-maxDepth x to limit

29

Contents

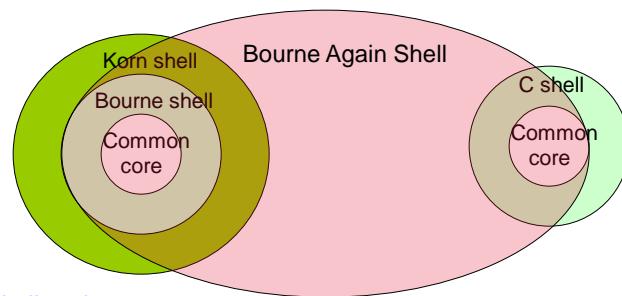
- Overview of UNIX
 - Structures
 - File systems
 - absolute and relative pathname
 - security $-rwx--x--x$
 - Process:
 - has return value 0 (success) or non 0 (sth wrong)
 - communication: pipes `who | sort who | grep Wang | wc -l`
- Utilities/commands
 - Basic, `mkdir`, `cat`, `more`, `cp`, `rm`, `mv`, `file`, `wc`, `chmod`
 - Advanced `grep/egrep`, `uniq`, `sort`, `diff/cmp`, `cut`, `find`,
- Shell (common shell functionalities)
- Bourn (again) Shell
 - scripting language

30

30

SHELL FUNCTIONALITY

- This part describes the **common core of functionality** that all four shells provide
 - E.g., pipe `who | sort`
 - E.g., filename wildcards `ls *.c` `ls a?.c`
- The relationship among the four shells:



YORK U
UNIVERSITÉ
UNIVERSITY

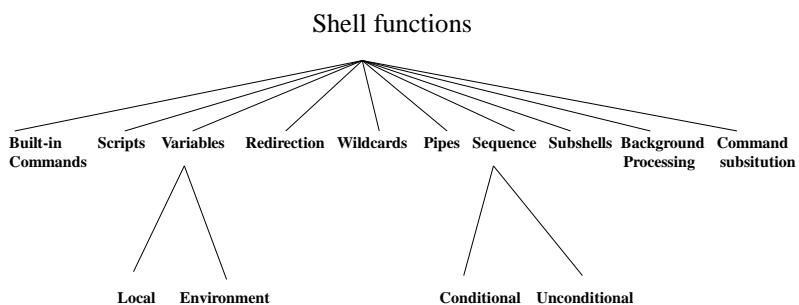
Login shell: tcsh

An enhanced but based on and completely compatible version of the C shell, *csh*

31

SHELL FUNCTIONALITY

A hierarchy diagram to illustrate the features shared by the four shells



YORK U
UNIVERSITÉ
UNIVERSITY

32

• METACHARACTERS

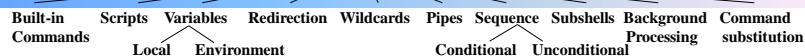
Some characters are processed specially by a shell and are known as [metacharacters](#).

All four shells share a core set of [common](#) metacharacters, whose meanings are as follow:

| Symbol | Meaning |
|---|---|
| > | Output redirection; writes standard output to a file. |
| >> | Output redirection; appends standard output to a file. |
| < | Input redirection; reads standard input from a file. |
| << | Input redirection; reads standard input from script up to tok. |
| * | Filename-substitution (wildcard); matches zero or more characters. |
| ? | Filename-substitution (wildcard); matches any single character. |
| [...] | Filename-substitution (wildcard); matches any character between the brackets. |
| CSE1020 lab tour. Don't get confused with Regex | |

33

Shell functions



| Symbol | Meaning |
|-------------|---|
| ` command ` | Command substitution; replaced by the output from command |
| \$ | Variable substitution. Expands the value of a variable. |
| & | Runs a command in the background. <code>jedit&</code> |
| | Pipe symbol; sends the output of one process to the input of another |
| ; | Used to sequence commands. <code>\$echo hello; wc lyrics</code> |
| | Conditional execution; executes a command if the previous one fails. |
| && | Conditional execution; executes a command if the previous one succeeds. |
| (...) | Groups commands. |
| # | All characters that follow up to a new line are ignored by the shell and program (i.e., used for a comment) |
| \ | Prevents special interpretation of the next character. |
| '' " " | quoting |

34

- When you enter a command, the shell scans it for metacharacters and (if any)processes them specially.

**When all metacharacters have been processed,
the command is finally executed.**

- To turn off the special meaning of a metacharacter, precede it by a `\` character. # Also " " ' ' (later)
 - Here's an example:

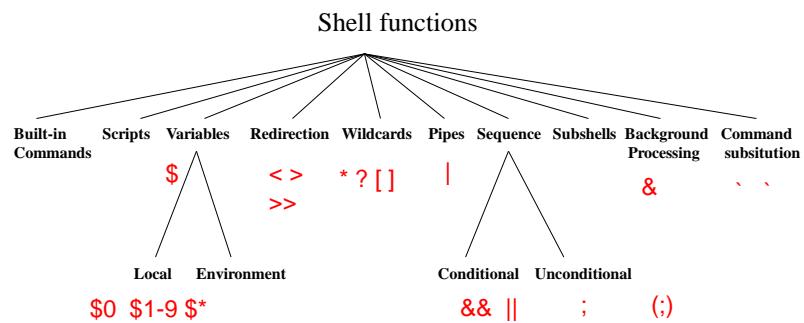
```
$ echo hi > file      # store output of echo in "file".
$ cat file             # look at the contents of "file".
hi

$ echo hi \> file2    # inhibit > metacharacter.
hi > file2            # > is treated like other characters.
$ cat file2           # look at the file again. Not written
ls: cannot access file2: No such file or directory such a file
```



35

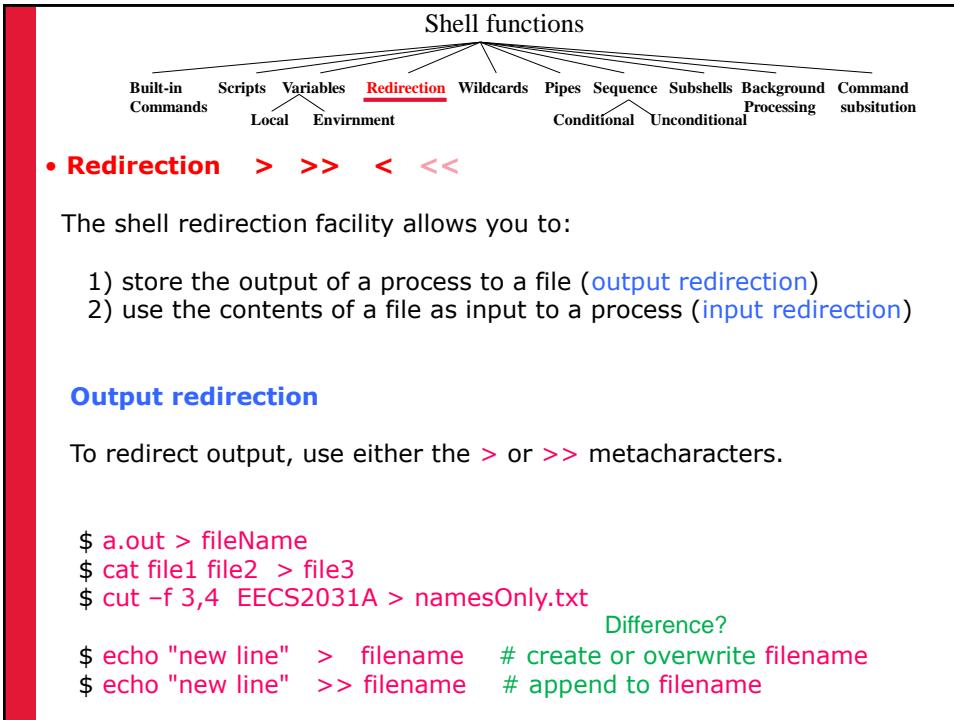
Functionalities and corresponding meta-characters



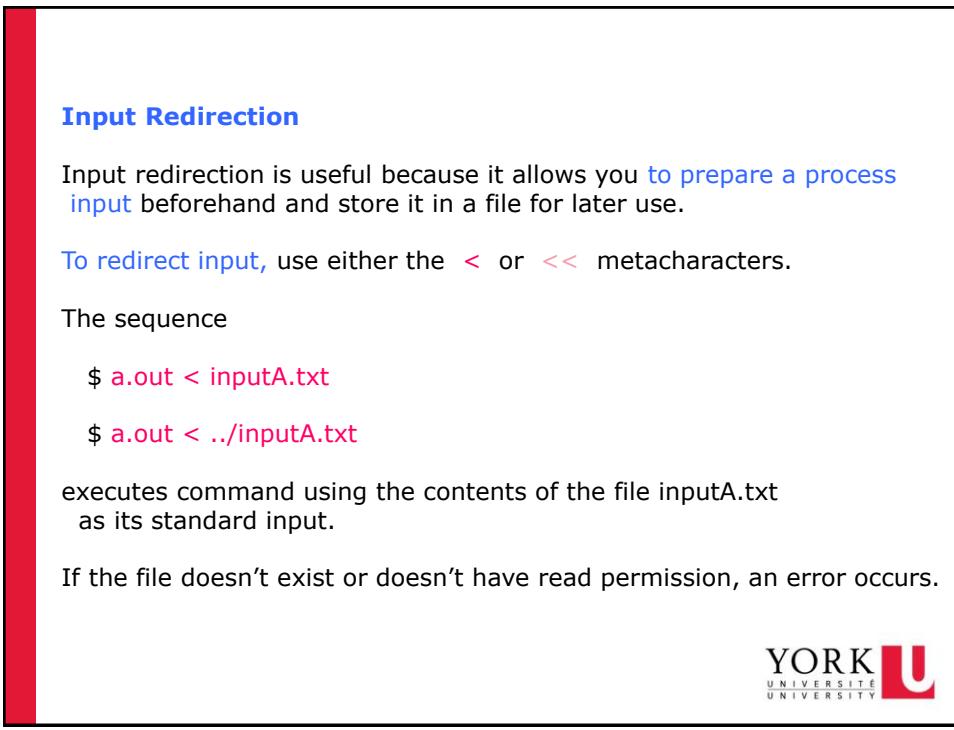
Now, let's go through (review) the functionalities, as well as their associated metacharacters



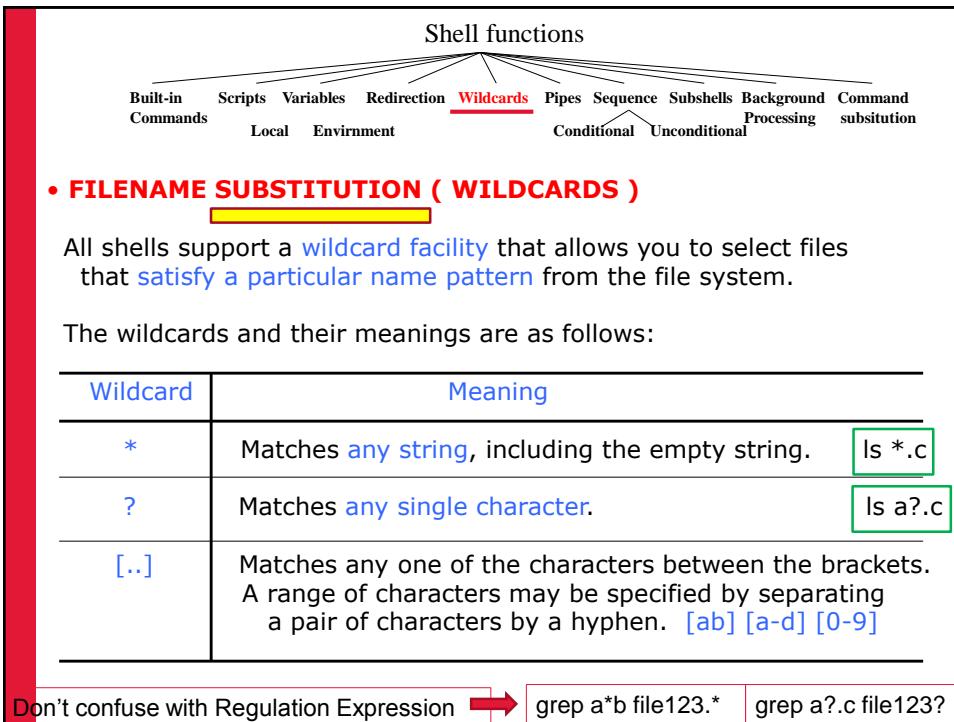
36



37



38



39

Used for filename wildcard, in **ls, cp, mv, rm, cat, more, chmod, grep,...** operating on multiple files.

Here are some examples of wildcards in action:

```
$ ls *.c      # list any text ending in ".c"
a.c          b.c          ax.c          aw2.c
```

```
$ ls ?.c      # list text for which one character is followed by ".c"
a.c          b.c
```

```
$ ls a*.c     # a followed by anything including empty before .c
a.c          ax.c         aw2.c
```

```
$ ls a?.c     # a followed by exactly one character before .c
ax.c
```

```
$ cp /eecs/dept/course/2018-19/S/2031/xFile? .
```

```
$ cp /eecs/dept/course/2018-19/S/2031/xFile* .
```

```
$ cp /eecs/dept/course/2018-19/S/2031/xFile[23] .
```



40

find examples revisit

- `$ find / -name x.c` # search for file x.c in the entire file system
- `$ find . -name '*.bak'` # "*.bak" search for all bak files in current and subdirectories
- `$ find . -name 'a?.c'` # "a?.c" search for all aX.c
 - a1.c
 - a2.c
 - a3.c
- `$ find . -name '*.c' -exec cp {} {}.2019SU \;`
find all c files and then copy it to filename.2019SU

41

grep Regex. Only place this course

| | Regular expression | Filename substitution (wildcard) |
|--|--------------------|----------------------------------|
| <code>a*</code> | 0 or more a | a followed by 0 or more anything |
| <code>a?</code> | 0 or one a | a followed by 1 anything |
| <code>a+</code> | 1 or more a | |
| <code>[abc]</code> <code>[a-c]</code> | a or b or c | a or b or c |

`$ grep a*b file12*.c`

Regex. 0 or more 'a'
followed by 'b'
Match
b ab aab aaab aaaab
....

Wildcard. C file whose
name begins with 'file12'
Match
file12.c file12A.c
file12AD.c file12ABEF.c
....

`$ grep a?b file12?.c`

Regex. 0 or 1 'a'
followed by 'b'
Match
b ab

Wildcard. Match
file12A.c



42
21

Shell functions

```

graph TD
    SF[Shell functions] --- BIC[Built-in Commands]
    SF --- Scripts
    SF --- V[Variables]
    SF --- R[Redirection]
    SF --- W[Wildcards]
    SF --- P[Pipes]
    SF --- S[Sequence]
    SF --- Subshells
    SF --- BP[Background Processing]
    SF --- CS[Command substitution]

    V --- Local
    V --- Environment

    S --- Conditional
    S --- Unconditional
  
```

COMMAND SUBSTITUTION used very very ... heavily in script!

A command surrounded by grave accents (`) - back quote - is executed, and its standard output is inserted in the command's place in the entire command line. Any new lines in the output are replaced by spaces.

For example:

```

$ echo the date today is `date` , right?
the date today is Sun Jul 20 08:57:44 EDT 2019, right?
$ _
$ echo there are `who | wc -l` users on the system
there are 31 users on the system
  
```




43

Shell functions

```

graph TD
    SF[Shell functions] --- BIC[Built-in Commands]
    SF --- Scripts
    SF --- V[Variables]
    SF --- R[Redirection]
    SF --- W[Wildcards]
    SF --- P[Pipes]
    SF --- S[Sequence]
    SF --- Subshells
    SF --- BP[Background Processing]
    SF --- CS[Command substitution]

    V --- Local
    V --- Environment

    S --- Conditional
    S --- Unconditional
  
```

COMMAND SUBSTITUTION used very very ... heavily in script!

A command surrounded by grave accents (`) - back quote - is executed, and its standard output is inserted in the command's place in the entire command line. Any new lines in the output are replaced by spaces.

For example:

```

$echo there are `cat EECS2031A | wc -l` students in the class
there are 135 students in the class

$echo has `cat EECS2031A | grep -w Wang | wc -l` students name Wang
has 4 students name Wang
  
```



44

Shell functions

```

graph TD
    SF[Shell functions] --> BIC[Built-in Commands]
    SF --> S[Scripts]
    SF --> V[Variables]
    SF --> R[Redirection]
    SF --> W[Wildcards]
    SF --> P[Pipes]
    SF --> SQ[Sequence]
    SF --> SB[Subshells]
    SF --> BP[Background Processing]
    SF --> CS[Command substitution]
    V --> LE[Local Environment]
  
```

COMMAND SUBSTITUTION used very very ... heavily in script!

A command surrounded by grave accents (`) - back quote - is executed, and its standard output is inserted in the command's place in the entire command line. Any new lines in the output are replaced by spaces.

Two more examples:

```
$ which mkdir # man which: show the full pathname of shell command
/bin/mkdir
$ file `which mkdir` # file /bin/mkdir
/bin/mkdir: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses
shared libs), for GNU/Linux 2.6.32,
BuildID[sha1]=8cec890564feb596de5a36b1a5321b05a089079f, stripped
```



```
$x=`wc -l classlist` # x get value 135 (talk later)
```

YORK 
UNIVERSITÉ
UNIVERSITY

45

Shell functions

```

graph TD
    SF[Shell functions] --> BIC[Built-in Commands]
    SF --> S[Scripts]
    SF --> V[Variables]
    SF --> R[Redirection]
    SF --> W[Wildcards]
    SF --> P[Pipes]
    SF --> SQ[Sequence]
    SF --> SB[Subshells]
    SF --> BP[Background Processing]
    SF --> CS[Command substitution]
    V --> LE[Local Environment]
  
```

- **VARIABLES and variable substitution \$**

- A shell supports two kinds of variables:
local and **environment** variables.

local:
user defined,
positional

Both kinds of variables hold data in a **string** format.

The child shell gets a copy of its parent shell's environment variables, but not its local variables.

Every shell has a set of predefined environment variables that are usually initialized by the startup files.

YORK 
UNIVERSITÉ
UNIVERSITY

46

Built-in local variables

-several common built-in local variables that have special meanings:

| Name | Meaning |
|----------|---|
| \$\$ | The process ID of the shell. |
| \$0 | The name of the shell script (if applicable). |
| \$1..\$9 | \$n refers to the n'th command line argument (if applicable). |
| \$* | A list of all the command-line arguments. |

\$ myscript paul ringo george john

↓ ↓ ↓ ↓ ↓
\$0 \$1 \$2 \$3 \$4
 └───┘
 \$*



47

variable substitution \$

```
$ x=5
$ echo value of x is $x    # value of x is 5
```

```
$ name=Graham
$ echo Hi, I am $name    # Hi, I am Graham
```

```
$ echo $?
```



48

QUOTING



There are often times when you want to **inhibit** the shell's **wildcard-substitution** * ? [], **variable-substitution** \$, and/or **command-substitution** ` mechanisms.

The shell's quoting system allows you to do just that.

- Here's the way that it works:

- 1) Single quotes (' ') inhibits both **wildcard substitution**, **variable substitution**, and **command substitution**.
- 2) Double quotes(" ") inhibits **wildcard substitution** **only**.

49

QUOTING

- The following example illustrates the difference between the two different kinds of quotes:

```
$ echo 3 * 4 = 12      # remember, * is a wildcard.  
3 a.c  b  b.c  c.c  4 = 12
```



```
$ echo "3 * 4 = 12"    # double quotes inhibit wildcards.  
3 * 4 = 12
```

```
$ echo '3 * 4 = 12'    # single quotes inhibit wildcards.  
3 * 4 = 12
```

another way?

```
$ echo 3 \* 4 = 12     # backslash inhibit a metacharacter  
3 * 4 = 12
```

50

```
$ name=Graham # assign value to name variable  
  
$ echo 3 * 4 = 12, my name is $name - today is `date`  
3 a.c b b.c c.c 4 = 12, my name is Graham - today is Sun Jul 21
```

51

```
$ name=Graham # assign value to name variable  
  
$ echo 3 * 4 = 12, my name is $name - today is `date`  
3 a.c b b.c c.c 4 = 12, my name is Graham - today is Sun Jul 21
```

- By using **single quotes (apostrophes)** around the text, we inhibit all **wildcarding** and **variable** and **command substitutions**:

```
$ echo '3 * 4 = 12, my name is $name - today is `date`'
```

?

52

```
$ name=Graham # assign value to name variable

$ echo 3 * 4 = 12, my name is $name - today is `date`
3 a.c b b.c c.c 4 = 12, my name is Graham - today is Sun Jul 21
```

- By using **single quotes (apostrophes)** around the text, we inhibit all **wildcarding** and **variable** and **command substitutions**:

```
$ echo '3 * 4 = 12, my name is $name - today is `date`'
3 * 4 = 12, my name is $name - today is 'date'
```

 inhibited

53

```
$ name=Graham # assign value to name variable

$ echo 3 * 4 = 12, my name is $name - today is `date`
3 a.c b b.c c.c 4 = 12, my name is Graham - today is Sun Jul 21
```

- By using **single quotes (apostrophes)** around the text, we inhibit all **wildcarding** and **variable** and **command substitutions**:

```
$ echo '3 * 4 = 12, my name is $name - today is `date`'
3 * 4 = 12, my name is $name - today is 'date'
```

 inhibited

- By using **double quotes around** the text, we inhibit **wildcarding**, but allow **variable** and **command substitutions**:

```
$ echo "3 * 4 = 12, my name is $name - today is `date`"
```

?

54

```
$ name=Graham # assign value to name variable

$ echo 3 * 4 = 12, my name is $name - today is `date`
3 a.c b b.c c.c 4 = 12, my name is Graham - today is Sun Jul 21
```

- By using **single quotes (apostrophes)** around the text, we inhibit all **wildcarding** and **variable** and **command substitutions**:

```
$ echo '3 * 4 = 12, my name is $name - today is `date`'
3 * 4 = 12, my name is $name - today is 'date'
```

\$ _____ inhibited

- By using **double quotes around** the text, we inhibit **wildcarding**, but **allow** **variable** and **command substitutions**:

```
$ echo "3 * 4 = 12, my name is $name - today is `date`"
3 * 4 = 12, my name is Graham - today is Sun Jul 21 23:25:26 EDT
```

\$ _____ inhibited \$ _____ interpreted



55

- Here's the way that it works:

- 1) Single quotes (' ') inhibits **wildcard substitution**, **variable substitution**, and **command substitution**.
- 2) Double quotes(" ") inhibits **wildcard substitution** only.

```
$ x=5
$ echo "value of x is $x"
```

" " does not inhibit variable substitution \$
" " does not inhibit command substitution

```
$ echo "there are `who | wc -l` people logged on"
there are 32 people logged on
```

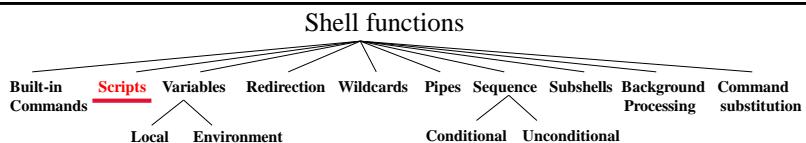
Both ' ' and " " inhibit wildcard substitution * ?

Needed
for Some
shell e.g.,
tosh

| | | |
|----------------------|------------------------|------------------------|
| \$ egrep the lyrics | \$ egrep 'the' lyrics | \$ egrep "the" lyrics |
| \$ egrep ab? lyrics | \$ egrep "ab?" lyrics | \$ egrep 'ab?' lyrics |
| \$ egrep ab*c lyrics | \$ egrep "ab*c" lyrics | \$ egrep 'ab*c' lyrics |

Better use quote on Regex to prevent (some) shell
from interpreting Regex repetition symbol * ? as
filename wildcards.

56



SHELL PROGRAMS: SCRIPTS

Any series of shell commands may be stored inside a regular text file for later execution.

A file that contains shell commands is called a *script*.

- batch file (.bat .cmd) in Windows

Before you can run a script, you must give it **execute** permission

`chmod u+x filename`

`echo hello world
date`

To run it, you need only to type its name.

Scripts are useful for storing commonly used sequences of commands, and they range in complexity from simple one-liners to fully blown programs.

57

• SHELL PROGRAMS: SCRIPTS

```
$ cat > script.sh
#!/bin/sh
# This is a sample sh script.
echo "Hello world"
echo The date today is `date`.
^D
```

create the bash script.

`` command substitution

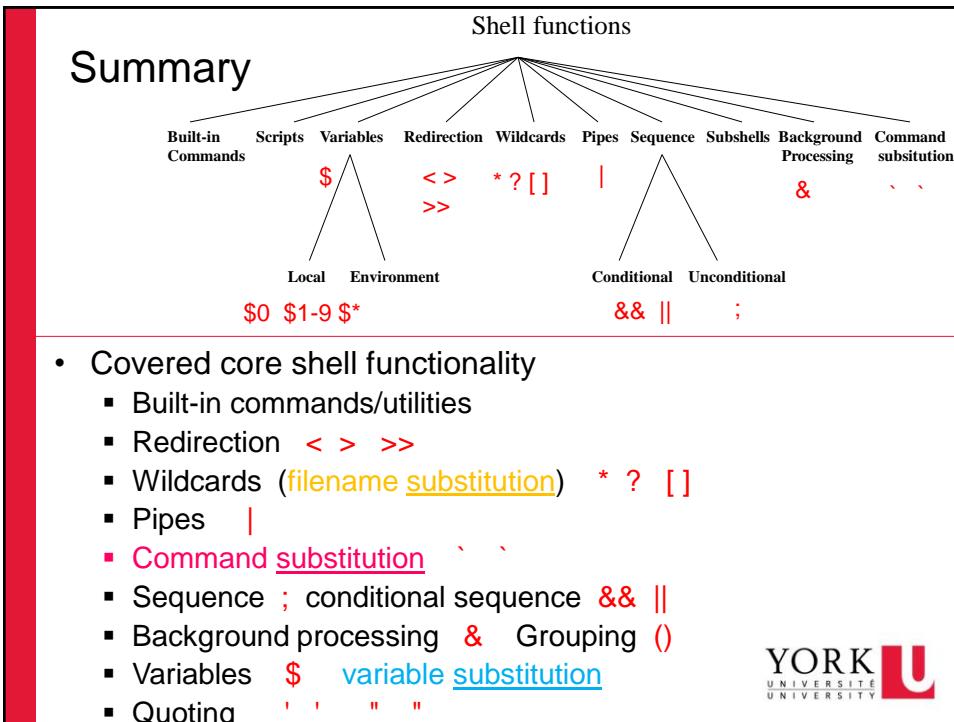
end of input.

```
$ chmod u+x script.sh      # make the scripts executable.
```

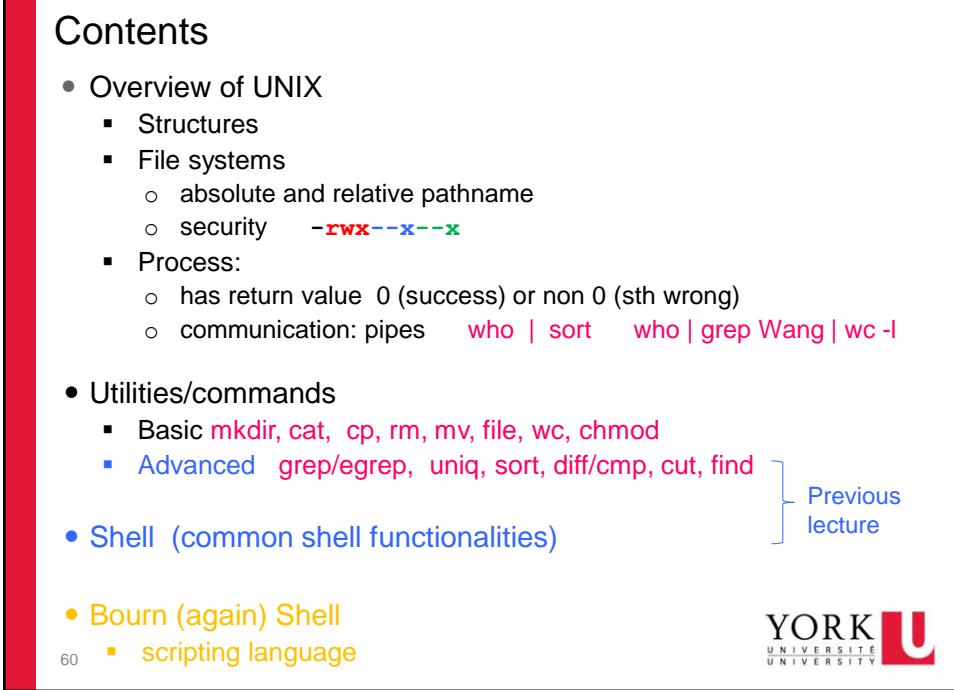
```
$ script.sh      # execute the shell script.
Hello world
The date today is Sun Jul 21 19:50:00 EDT 2019
```



58



59



60

The Bourne Shell and its script

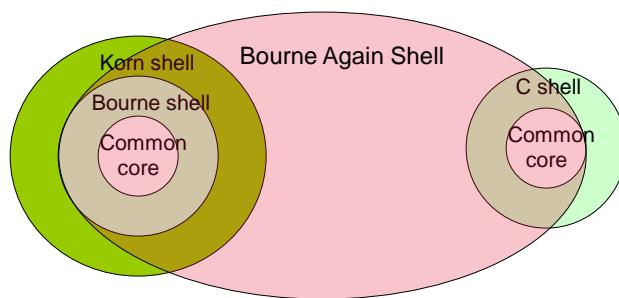
Ch5 Bourn shell
“UNIX for Programmers and Users”
Third Edition, Prentice-Hall, GRAHAM GLASS, KING ABLES

61



61

Bourne shell (sh) and Bourne Again Shell (bash)



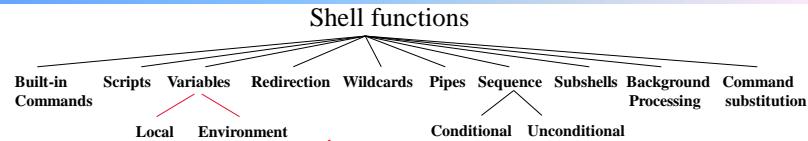
Login shell: tcsh

An enhanced but based on and completely compatible version of the C shell, csh



62

Ch. 4. The Bourne Shell



• Introduction

The Bourne shell supports all of the core-shell facilities describe earlier, plus the following new facilities:

- several ways to set and access variables
- a built-in programming language that supports conditional branching, looping etc
- advanced I/O redirection extensions to the existing redirection and command-sequence operations >&
- several new built-in commands

63

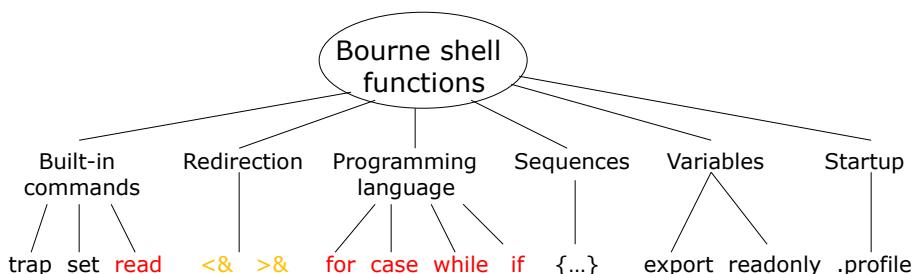


63

Ch. 4. The Bourne Shell

• Introduction

- These new facilities are described by this chapter and are illustrated by the following hierarchy diagram:



64



64

• SHELL PROGRAMS: SCRIPTS

```
$ cat myscript.sh  
#!/bin/sh  
# This is a sample sh script.  
echo "Hello world"  
echo The date today is `date`.
```

Variables, operators

read from user

branch if else

loop

functions

recursions

....

```
$ chmod u+x myscript.sh
```

```
$ myscript.sh      # execute the shell script.
```

hello world

The date today is Sun Jul 21 19:50:00 EDT 2019



65

CONTENTS

These utilities/commands constitutes basic components for a programming language

- variable (set / get)
- read from the user
- command line arguments
- arithmetic operation
- branching -- if else
- looping -- while / for loop
- functions, recursions
- ...



66

66

CONTENTS

These utilities/commands constitutes basic components for a programming language

- variable (set / get)
- **read from the user**
- **command line arguments**
- **arithmetic operation**
- **branching -- if else**
- **looping -- while / for loop**
- **functions, recursions**
- ...



67

67

Ch. 4. The Bourne Shell

• VARIABLES

The Bourne shell can perform the following variable-related operations:

- simple assignment and access
- testing a variable for existence
- reading a variable from standard input
- making a variable read only
- exporting a local variable to the environment

- Creating/Assigning a Variable

The Bourne-shell syntax for assigning a value to a variable is:

{name=value}+



68

68

Ch. 4. The Bourne Shell

• VARIABLES

```
$ firstName=Graham  
$ lastName=Glass  
$ age=29
```

No space!

assign variables.

```
$ echo "Hi, I'm $firstName $lastName, am $age years old"  
Hi, I'm Graham Glass, am 29 years old
```

Variable substitution!

```
$ name=Graham Glass      # syntax error.  
Glass: not found
```

```
$ name="Graham Glass"    # use quotes to built strings.  
$ echo $name              # now it works.  
Graham Glass  
$
```

No need to declare! If assigned does not exist, create!

69

69

Ch. 4. The Bourne Shell

• Accessing a Variable

- The Bourne shell supports the following access methods:

| Syntax | Action |
|---------------|---|
| \$name | Replaced by the value of name. |
| \${name} | Replaced by the value of name.  |
| \${name-word} | Replaced by the value of name if set, and word otherwise. |
| \${name+word} | Replaced by the word if name is set, and nothing otherwise. |
| \${name=word} | Assigns word to the variable name if name is not already set and then is replaced by the value of name |
| \${name?word} | Replaced by name if name is set. If name is not set, word is displayed to the standard error channel and the shell is exited. If word is omitted, then a standard error message is displayed instead. |

70

Ch. 4. The Bourne Shell

- Example

```
$ verb=sing          # assign a variable.  
$ echo I like $verbing    # there's no variable "verbing".  
I like  
$ echo I like ${verb}ing    # now it works.  
I like singing  
$ -  
  
$ echo I like $verb ing  
$ I like sing ing
```

71



71

CONTENTS

- These utilities/commands constitutes basic components for a programming language

- **variable (set / get)**
- read from the user
- command line arguments
- arithmetic operation
- branching -- if else
- looping -- while / for loop
- functions
- ...

72



72

Ch. 4. The Bourne Shell

- Reading a Variable from Standard Input

The read command allows you to read variables from standard input and works like this:

Shell Command: `read {variable}+`

`read` reads one line from standard input and then assigns successive words from the line to the specified variables.

Any words that are left over are assigned to the last named variable.

```
$ read x          $ read x  
Hello           5  
$ echo $x        $ echo $x      No need to declare x  
Hello           5  
73
```



73

Ch. 4. The Bourne Shell

- If you specify just one variable,
the entire line is stored in the variable.

Here's an example script that prompts a user for his or her full name:

```
$ cat readName.sh          No need to declare name  
echo -n "Please enter your name: "  
read name                  # read just one variable.  
echo your input is $name    # display the variable.
```

```
$ readName.sh  
Please enter your name: Graham Walker Glass  
your input is Graham Walker Glass  
$ -
```

the whole line is read

74

74

Ch. 4. The Bourne Shell

- Here's other example script

```
$ cat readNames.sh
echo -n "Please enter your name: "
read first last
echo your first name is $first
echo your last name is $last
```

Don't need to declare `first`, `last`

`# read two variables.`

`# display the variables.`

```
$ readNames.sh
Please enter your name: Graham Walker Glass
your first name is Graham
your last name is Walker Glass      # the whole rest line is read.
$
```

75

```
$ read a
1 2 3 4 5 6 7
```

```
$ read a b
1 2 3 4 5 6 7
$ echo $a
1
$ echo $b
2 3 4 5 6 7
```

```
$ read a b c
1 2
$ echo $a
1
$ echo $b
2
$ echo $c
```

Ch. 4. The Bourne Shell

- Here's other example script

```
$ cat mygrep.sh
echo -n "Please enter file to search: "
read file          # read two variables.
echo -n "Please enter search key: "
read pattern
grep -w $pattern $file      # display the variables.
echo last exit value is $?
```

```
$ mygrep.sh
Please enter file to search: EECS2031A
Please enter searech key: Yang
ryang**      *****
yanguh*      *****
Yang Rui
Yang Yuhui
the last exit value was 0
$
```

76

CONTENTS

- These utilities/commands constitutes basic components for a programming language
 - variable (set / get)
 - read from the user: read var
 - command line arguments
 - arithmetic operation
 - branching -- if else
 - looping -- while / for loop
 - functions
 - ...



77

77

-Recall: several common (core) built-in local variables that have special meanings:

| Name | Meaning |
|----------|---|
| \$\$ | The process ID of the shell. |
| \$0 | The name of the shell script (if applicable). |
| \$1..\$9 | \$n refers to the n'th command line argument (if applicable). |
| \$* | A list of all the command-line arguments. |

\$ myscript we are the genius

\$0 \$1 \$2 \$3 \$4
 ^-----
 |
 \$*



78

39

Ch. 4. The Bourne Shell

- Predefined Local Variables

In addition to the core predefined local variables (\$\$, \$0, \$1..9, \$*)
the Bourne shell defines the following local variables:

| Name | Value |
|------|---|
| \$@ | an individually quoted list of all of the positional parameters |
| \$# | the number of positional parameters (command arguments) |
| \$? | the exit value of the last command |
| \$! | the process ID of this last background command |
| ... | ... |
| | |



79

79

Ch. 4. The Bourne Shell

- Predefined Local Variables

In addition to the core predefined local variables (\$\$, \$0, \$1..9, \$*)
the Bourne shell defines the following local variables:

| Name | Value |
|------|---|
| \$@ | an individually quoted list of all of the positional parameters |
| \$# | the number of positional parameters (command arguments) |
| \$? | the exit value of the last command |
| \$! | the process ID of this last background command |

\$ myscript we are the genius
 ↓ ↓ ↓ ↓ ↓ \$# = 4
 \$0 \$1 \$2 \$3 \$4
 ↓ ↓
 \$* \$@

80

Ch. 4. The Bourne Shell



- Here's a small shell script that illustrates the first three variables.

```
$ cat mygrepArg.sh
echo "there are $# command line arguments: $@"
egrep -w $2 $1
echo the last exit value was $?          # display exit code.

$ mygrepArg.sh EECS2031A Leung
there are 2 command line arguments: EECS2031A Leung
the last exit value was 1      # match not found

$ mygrepArg.sh EECS2031A Yang
there are 2 command line arguments: EECS2031A Yang
ryang**      *****
yanguh*      *****
the last exit value was 0      # match find
```

81 (We can remove the output using > \dev\null)

81

CONTENTS

- These utilities/commands constitutes basic components for a programming language

- variable (set / get)
- read from the user: read var
- command line arguments
- arithmetic operation
- branching -- if else
- looping -- while / for loop
- functions
- ...

82

Ch. 4. The Bourne Shell

- ARITHMETIC

- Although the Bourne shell doesn't directly support arithmetic, it may be performed by using the **expr** utility, which works like this:

Utility : **expr expression** **\$expr 2 + 4**

expr evaluates **expression** and sends the result to standard output.

All of the components of expression must be separated by blanks,

The result of **expression** may be assigned to a shell variable by the appropriate use of **command substitution**.

x= `expr 2 + 4`

83

Space!

83

Ch. 4. The Bourne Shell

- ARITHMETIC

- **expression** may be constructed by applying the following binary operators to integer operands, grouped in decreasing order of precedence:

| OPERATOR | RESPECTIVE MEANING |
|---------------|-------------------------------------|
| * / % | multiplication, division, remainder |
| + - | addition, subtraction |
| => >= < <= != | comparison operators |
| & | logical and |
| | logical or |

84



84

Ch. 4. The Bourne Shell

- The following example illustrates some of the functions of `expr` and makes plentiful use of command substitution:

```
$ x=1          # initial value of x.  
$ x=`expr $x + 1` # increment x.  
$ echo $x  
2  
  
$ echo x=`expr 2 + 15 / 5`      # / is conducted before +.  
x=5  
Space!
```

```
Bourn again shell (bash):  
x=$((x+1))      x=$(( 2 +15/5 ))
```

```
Bourn again shell (bash):  
((x=x+1))  
((x++))  ((x+=1))
```

85

Ch. 4. The Bourne Shell

- An example script illustrate `expr` and position parameter

```
$ cat add.sh  
sum=`expr $1 + $2`          # add two parameters.  
echo "sum is: $sum"          # display the variables.
```

```
$ add.sh 5 7  
sum is: 12
```

- Here's the `bash` version

```
$ cat add.sh  
sum=$(( $1 + $2 ))          # add two parameters.  
echo "sum is: $sum"          # display the variables.
```

```
$ add.sh 5 7  
sum is: 12
```

86

86

CONTENTS

These utilities/commands constitutes basic components for a programming language

- variable (set / get)
- read from the user: read var
- command line arguments
- arithmetic operation
- branching -- if else
- looping -- while / for loop
- functions
- ...



87

87

Ch. 4. The Bourne Shell

• CONDITIONAL EXPRESSIONS

- The control structures often branch based on the value of a logical expression—that is, an expression that evaluates to true or false.

Utility: test expression

test returns a zero exit code if **expression** evaluates to **true**; otherwise, it returns a nonzero exit status.

```
$ test 3 -eq 3 ; echo $?    0    true
$ test 3 -eq 33 ; echo $?    1    false
```

The exit status is typically used by shell control structures for branching purposes. \$ if test \$x -eq 3

Some **Bourne shells** supports **test** as a built-in command, in which case they support the second form of evaluation as well.

```
$ [ 3 -eq 3 ]      $ if [ 3 -eq 3 ]
```

88

88

Ch. 4. The Bourne Shell

- The brackets of the second form must be surrounded by spaces in order for it to work. $\$ [3 -eq 4] \Leftrightarrow \$ test 3 -eq 4$

Space!

A `test` expression may take the following forms:

| Form | Meaning |
|--------------------------|--|
| <code>-b filename</code> | True if filename exists as a block special file. |
| <code>-c filename</code> | True if filename exists as a character special file. |
| <code>-d filename</code> | True if filename exists as a directory. $[-d classlist]$ |
| <code>-f filename</code> | True if filename exists as an ordinary file $test -f classlist$ |
| <code>-g filename</code> | True if filename exists as a “set group ID” file. |
| <code>-h filename</code> | True if filename exists as a symbolic link. |
| <code>-k filename</code> | True if filename exists and has its sticky bit set. |
| <code>-p filename</code> | True if filename exists as a named pipe. |
| <code>-u filename</code> | True if filename exists as a “set user ID” file. |
| <code>-s filename</code> | True if filename contains at least 1 char (none empty) $test -s emptyF$ |

89

89

Ch. 4. The Bourne Shell

| Form | Meaning |
|--|--|
| <code>-r filename</code> | True if filename exists as a readable file. |
| <code>-w filename</code> | True if filename exists as a writeable file. |
| <code>-x filename</code> | True if filename exists as an executable file. |
| | |
| <code>-n string</code> | True if string contains at least one character. |
| <code>-z string</code> | True if string contains no characters. empty string |
| <code>str1 = str2</code> | True if str1 is equal to str2. |
| <code>str1 != str2</code> | True if str1 is not equal to str2. |
| | |
| <code>int1 -eq int2</code> | True if integer int1 is equal to integer int2. $test 4 -eq 5$ |
| <code>int1 -ne int2</code> | True if integer int1 is not equal to integer int2. $[4 -ne 5]$ |
| <code>int1 -gt int2</code> | True if integer int1 is greater than integer int2. |
| <code>int1 -ge int2</code> | True if integer int1 is greater than or equal to integer int2. |
| <code>int1 -lt int2</code> | True if integer int1 is less than integer int2. |
| <code>int1 -le int2</code> | True if integer int1 is less than or equal to integer int2. |
| | |
| <code>! expr</code> | True if expr is False |
| <code>expr1 -a expr2</code> | True if expr1 and expr2 are both true |
| ⁹⁰ <code>expr1 -o expr2</code> | True if expr1 or expr2 are true |

90

| Argument | Test is true if ... | |
|------------------------|---|---|
| <code>-d file</code> | <code>file</code> is a directory | <code>test -d classlist</code> |
| <code>-f file</code> | <code>file</code> is an ordinary file | <code>test -f classlist</code> |
| <code>-r file</code> | <code>file</code> is readable | |
| <code>-s file</code> | <code>file</code> size is greater than zero | <code>test -s classlist</code> |
| <code>-w file</code> | <code>file</code> is writable | |
| <code>-x file</code> | <code>file</code> is executable | |
| <code>! -d file</code> | <code>file</code> is not a directory | |
| <code>! -f file</code> | <code>file</code> is not an ordinary file | |
| <code>! -r file</code> | <code>file</code> is not readable | |
| <code>! -s file</code> | <code>file</code> size is not greater than zero | |
| <code>! -w file</code> | <code>file</code> is not writable | |
| <code>! -x file</code> | <code>file</code> is not executable | |
| <code>n1 -eq n2</code> | integer <code>n1</code> equals integer <code>n2</code> | <code>test 4 -eq 5</code> <code>((4 == 5))</code> |
| <code>n1 -ge n2</code> | integer <code>n1</code> is greater than or equal to integer <code>n2</code> | <code>[4 -ge 5]</code> <code>((4 >= 5))</code> |
| <code>n1 -gt n2</code> | integer <code>n1</code> is greater than integer <code>n2</code> | <code>[4 -gt 5]</code> <code>((4 > 5))</code> |
| <code>n1 -le n2</code> | integer <code>n1</code> is less than or equal to integer <code>n2</code> | <code>[4 -le 5]</code> <code>((4 <= 5))</code> |
| <code>n1 -ne n2</code> | integer <code>n1</code> is not equal to integer <code>n2</code> | <code>[4 -ne 5]</code> <code>((4 != 5))</code> |
| <code>n1 -lt n2</code> | integer <code>n1</code> is less than integer <code>n2</code> | <code>[4 -lt 5]</code> <code>((4 < 5))</code> |
| <code>s1 = s2</code> | string <code>s1</code> equals string <code>s2</code> | |
| <code>s1 != s2</code> | string <code>s1</code> is not equal to string <code>s2</code> | |

Memorize?

Bash:

91

Ch. 4. The Bourne Shell

• CONTROL STRUCTURES

- The Bourne shell supports a wide range of control structures that make it suitable as a high-level programming tool.

Shell programs are usually stored in scripts and are commonly used to automate maintenance and installation tasks.

Branch: `if then elif else fi case..in..esac`

Loop: `while do done for do done until do done`

92

Ch. 4. The Bourne Shell

- **if...then...fi**

The **if** command supports **nested conditional branches** and has the following syntax:

```
if list1  
then  
    list2  
elif list3      optional,  
              the elif part may be repeated several times.  
then  
    list4  
else          optional,  
    list5  
fi
```

93



93

Ch. 4. The Bourne Shell

- Here's an example of a script that uses an **if** control structure:

```
$ cat numberScript.sh  
echo -n "enter a number: "  
read num  
if test $num -lt 0      # if [ $num -lt 0 ]      if (( $num < 0 ))  
then  
    echo "negative"  
elif [ $num -eq 0 ]     # elif test $num -eq 0    elif (( $num == 0 ))  
then  
    echo "zero"  
else  
    echo positive  
fi
```

Bash:

```
$ numberScript.sh      # run the script.  
enter a number: 1  
positive  
$ numberScript.sh      # run the script again.  
enter a number: -1  
negative  
$
```

94



94

Ch. 4. The Bourne Shell

- Another example of a script that uses an *if* control structure:

```
$ cat week.sh # version 1
echo -n "enter a date: "
read dat
if test $dat = "Fri"          # if [ $dat = Fri ] compare string is easy
then
    echo "Thank God it is Friday"
elif [ $dat = Sat ]
then
    echo "You should not be here working, go home!!! "
elif [ $dat = "Sun" ]
then
    echo "You should not be here working, go home!!! "
else
    echo "Not weekend yet. Get to work"
fi

$ week.sh
enter a date: Wed
Not weekend yet. Get to work
$
```

Comparing strings is easier
than comparing integers



95

Ch. 4. The Bourne Shell

- Another example of a script that uses an *if* control structure:

```
$ cat week.sh # version 2
echo -n "enter a date: "
read dat
if test $dat = "Fri"          # if [ $dat = Fri ] compare string easy
then
    echo "Thank God it is Friday"
elif [ $dat = Sat ] || [ $dat = "Sun" ] ←
then
    echo "You should not be here working, go home!!! "
else
    echo "Not weekend yet. Get to work"
fi

$ week.sh
enter a date: Wed
Not weekend yet. Get to work
$ week.sh
enter a date: Fri
Thank God it is Friday
$
```

Comparing strings is easier



96

Ch. 4. The Bourne Shell

- Another example of a script that uses an *if* control structure:

```
$ cat week.sh # version 3
echo -n "enter a date: "
read dat
if test $dat = "Fri"      # if [ $dat = Fri ] compare string easy
then
    echo "Thank God it is Friday"
elif [ $dat = Sat -o $dat = "Sun" ]      ← -o or
then
    echo "You should not be here working, go home!!! "
else
    echo "Not weekend yet. Get to work"
fi
```

Comparing strings is easier

```
$ week.sh
enter a date: Wed
Not weekend yet. Get to work
$ week.sh
enter a date: Fri
Thank God it is Friday
$
```



97

Ch. 4. The Bourne Shell

- Here's another example of a script that uses an *if* control structure:
- If a file empty, echo, else ls it

```
$ cat if.sh
echo -n "enter a file name: "
read name
if [ ! -s $name ]      # if test ! -s $name
then
    echo "File $name is empty"
    exit 1
else
    ls -l $name
fi
```

| Argument | Test is true if ... |
|----------------|---------------------------------------|
| -d file | file is a directory |
| -f file | file is an ordinary file |
| -r file | file is readable |
| -s file | file size is greater than zero |
| -w file | file is writable |
| -x file | file is executable |

-s filename

True if filename contains at least 1 char (none empty)

98

Ch. 4. The Bourne Shell

- Here's another example of a script that uses an *if* control structure

```
$ cat if2.sh
```

```
echo -n "enter a file name: "
read name
if [ -d $name ]    # if test -d $name
then
    echo $name is a directory
elif [ -x $name ]
then
    echo "File $name is executable"
else
    echo "File $name is not executable"
    chmod u+x $name
    echo "File $name is executable now"
fi
```

99

| Argument | Test is true if ... |
|----------|--------------------------------|
| -d file | file is a directory |
| -f file | file is an ordinary file |
| -r file | file is readable |
| -s file | file size is greater than zero |
| -w file | file is writable |
| -x file | file is executable |

99

Ch. 4. The Bourne Shell

- Here's an example of a script that uses an *if* control structure:
improved version of *grep*

```
$ cat mygrepArgNotFound.sh      # list the script.
# arg1 file to search
# arg2 search pattern
```

```
grep $2 $1
```

How to check if found or not programmatically



A: Exit Code of grep execution!

```
$ mygrepArgNotFound.sh Leung EECS2031A
pattern Leung not found in file EECS2031A, try another!
```

Ch. 4. The Bourne Shell

- Here's an example of a script that uses an *if* control structure:
improved version of `grep`

```
$ cat mygrepArgNotFound.sh      # list the script.  
# arg1 file to search  
# arg2 search pattern  
  
grep $2 $1  
if [ $? -ne 0 ]      # not 0 --- not successful [ $? gt 0 ]  
then  
    echo pattern $2 not found in file $1, try another!  
else  
    echo pattern $2 found in file $1!  
fi  
  
$ mygrepArgNotFound.sh Leung EECS2031A  
pattern Leung not found in file EECS2031A, try another!
```

Do it

101

Ch. 4. The Bourne Shell

- **while...do...done**

- The `while` command executes one series of commands as long as another series of commands succeeds.

Here's its syntax:

```
while list1  
do  
    list2  
done
```

The `while` command executes the commands in `list1` and ends if the last command in `list1` fails;
otherwise, the commands in `list2` are executed and the process is repeated.

102



102

Ch. 4. The Bourne Shell

- If `list2` is empty, the `do` keyword should be omitted.
A `break` command causes [the loop to end immediately](#),
and a `continue` command causes [the loop to immediately jump to the next iteration](#).
- Here's an example of a script that uses a `while` control structure to generate hello several times:

```
$ cat repeat.sh
```

```
count=0
while [ $count -lt 5 ]
do
    echo Hello $count
    count=`expr $count + 1`
done
```

103 Bash: $((\text{count} < 5))$

$\text{count}=\$((\text{count}+1))$ or $((\text{count}=\text{count}+1))$ or $((\text{count}++))$

```
sh-4.1$ repeat.sh
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
sh-4.1$
```

103

Ch. 4. The Bourne Shell

Write a script `matrix.sh` to generate a matrix

```
$ matrix.sh 7
      $1
1-1  1-2  1-3  1-4  1-5  1-6  1-7
2-1  2-2  2-3  2-4  2-5  2-6  2-7
3-1  3-2  3-3  3-4  3-5  3-6  3-7
4-1  4-2  4-3  4-4  4-5  4-6  4-7
5-1  5-2  5-3  5-4  5-5  5-6  5-7
6-1  6-2  6-3  6-4  6-5  6-6  6-7
7-1  7-2  7-3  7-4  7-5  7-6  7-7
$
$ matrix.sh 4
1-1  1-2  1-3  1-4
2-1  2-2  2-3  2-4
3-1  3-2  3-3  3-4
4-1  4-2  4-3  4-4
```

104

```
C, Java, C++:
a = atoi(argv[1])
x = 1
while (x < a)
{
    y = 1
    while (y < a)
    {
        printf ("%d-%d",x,y)
        y = y+1
    }
    x = x + 1
}
```

104

Ch. 4. The Bourne Shell

\$ cat matrix.sh

or, if \$# -ne 1

```

if [ -z $1 ]; then    # if $1 is empty string -z
    echo "Usage: matrix number"
    exit
fi
x=1
while [ $x -le $1 ]      # outer loop
do
    y=1
    while [ $y -le $1 ]      # inner loop
    do
        echo -n "$x-$y      "
        y=`expr $y + 1`          # y++ update inner-loop count
    done
    echo
    x=`expr $x + 1`          # x++ update outer-loop count
done

```

```

x = 1
while (x < 7){
    y = 1
    while (y < 7){
        printf( "%d-%d", x,y)
        y = y+1
    }
    x = x + 1
}

```

Indent not mandatory
y++ update inner-loop count

x++ update outer-loop count

105

Ch. 4. The Bourne Shell

\$ cat matrix.sh

or, if \$# -ne 1

```

if [ -z $1 ]; then    # if $1 is empty string -z
    echo "Usage: matrix number"
    exit
fi
x=1
while (( $x <= $1 ))      # outer loop
do
    y=1
    while (( $y <= $1 ))      # inner loop
    do
        echo -n "$x-$y      "
        y=$((y + 1))          # y++ update inner-loop count
    done
    echo
    x=$((x+1))              # x++ update outer-loop count
done

```

```

x = 1
while (x < 7){
    y = 1
    while (y < 7){
        printf( "%d-%d", x,y)
        y = y+1
    }
    x = x + 1
}

```

Indent not mandatory
y++ update inner-loop count

x++ update outer-loop count

Bourn again shell version

106

Example: Loop + branch

loopBranch.sh

```
#!/bin/sh
echo -n "enter a number or 'quit': "
read num

while [ $num != "quit" ]
do
    if [ $num -lt 0 ]
    then
        echo "a negative number"
    elif [ $num -eq 0 ]
    then
        echo "this is zero"
    else
        echo "a positive number"
    fi

    # read again
    echo -n "enter a number: "
    read num
done
```

Bourn again shell

```
if (($num < 0))
```

```
elif (($num ==0))
```



107

SUMMARY

- These utilities/commands constitutes basic components for a programming language

| | |
|---------------------------------|---|
| • variable (set / get) | x=4 x=hello echo \$x |
| • read from the user | read x |
| • command line arguments | \$0-9, \$#, \$* |
| • arithmetic operation | x=`expr 3 + 4` x=\$((3 + 4)) |
| • branching -- if then else fi | if test \$x -lt 4 then if [\$x -lt 4] then |
| • looping -- while / for loop | while .. do .. done |
| • parameter shifting | shift shift 3 |
| • enhanced I/O redirection | > /dev/null 2>&1 |
| 108 • functions, recursions ... | 1> /dev/null 2> /dev/null |

Now see 2 examples

108

Contents

- Overview of UNIX
 - Structures
 - File systems
 - absolute and relative pathname
 - security `-rwx--x--x`
 - Process:
 - has return value 0 (success) or non 0 (sth wrong)
 - communication: pipes `who | sort | grep Wang | wc -l`
- Utilities/commands
 - Basic `mkdir`, `cat`, `cp`, `rm`, `mv`, `file`, `wc`, `chmod`
 - Advanced `grep/egrep`, `uniq`, `sort`, `diff/cmp`, `cut`, `find`
- Shell (common shell functionalities)
 - Bourn (again) Shell
 - 109 ▪ scripting language

} Previous
lecture



109

This concludes the course.

- Final written exam:
 - Aug 07, Wednesday, 2:00-5:00pm. ACW 206
 - Approximately 50-60% C and 40-50% Unix
 - For C, 10-20% before midterm, 80-90% after midterm,
 - For Unix, more on stuff before today + Little shell script.
 - Final Exam page on course website (shortly)



110

110

What we have done so far

- Type, operators and expressions (Ch 2) :
 - Types and sizes
 - Basic types, their size and constant values (literals)
 - ✓ char: `x >= 'a' && x <= 'z'; x >= '0' && x <= '9'` avoid `x>=48 && x<=57`
 - ✓ int: 122, 0122, 0x12F convert between Decimal, Bin, Oct, Hex
 - Arrays (one dimension) and strings (Ch1.6,1.9)
 - ✓ "hello" has size 6 byte
 - Expressions
 - Basic operators (arithmetic, relational and logical)
 - ✓ `y=x++;` `y=++x;`
 - ✓ `!0` !-3 if (`x = 2`)
 - Type conversion and promotion `9/2*2.0 2.0*9/2` `int i= 3.4`
 - Other operators (bitwise, bit shifting , compound assignment, conditional)
 - ✓ Bit: `|, &, ~, ^, <<, >>`
 - ✓ Compound: `x += 10;` `x >>= 10;` `x += y + 3`
 - Precedence of operators `flag | 1 << 3`
- ¹¹¹ Functions and Program Structure (Ch 4)

111

What we have done so far

Ch 4 + others

- C program structure, functions
 - Multiple files
 - Communication by global variables
 - “Call by value” `increment()` `swap()`
- Categories, scope and life time, initialization of variables (and functions)
 - global and local variables
 - `static`
- C Preprocessing
 - `#include`, `#define`
- Recursion
- Other C material before pointer
 - Common library functions [Appendix of K&R]
 - 2D array, string manipulations
 - `sscanf`, `sprintf`,
 - `fgets`, `fputs`

112

What we have done before midterm

K&R Ch 5 Pointers

- Basics: Declaration and assignment
 - Pointer to Pointer
 - Pointer and functions (pass pointer by value)
 - Pointer arithmetic $+-\text{ }++\text{ }--$
 - Pointers and arrays (5.3)
 - Stored consecutively
 - Pointer to array elements $p + i = \&a[i]$ $*(p+i) = a[i]$
 - Array name contains address of 1st element $a = \&a[0]$
 - Pointer arithmetic on array (extension) $p1-p2$ $p1 < > != p2$
 - Array as function argument – “decay”
 - Pass sub_array



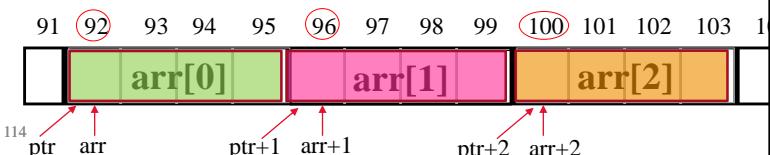
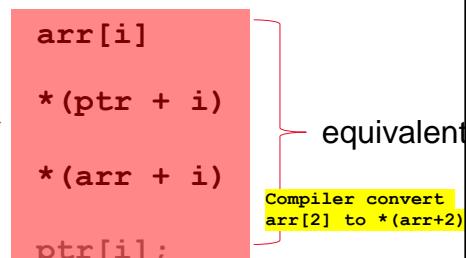
113

Summary

arr can be used as a pointer

```
int arr[3]; int * p;  
ptr = arr; /* ptr = &arr[0] */
```

`arr+i == &arr[i]`



114

Pointers K&R Ch 5

- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- Pointer and functions (pass pointer by value) (5.2)
- Pointer arithmetic +- ++ -- (5.4)
- Pointers and arrays (5.3)
 - Stored consecutively
 - Pointer to array elements $p + i = \&a[i] \quad *(p+i) = a[i]$
 - Array name contains address of 1st element $a = \&a[0]$
 - Pointer arithmetic on array (extension) $p1-p2 \quad p1 <= p2$
 - Array as function argument – “decay”
 - Pass sub_array
- Array of pointers (5.6-5.9)
- Command line arguments (5.10)
- Memory allocation (extra)
- Pointer to structures (6.4)
- Pointer to functions

After midterm



115

Array of pointers to scalar types

```
main(){
    int a=4, b=10, c=20;
    pa=&a, pb=&b;

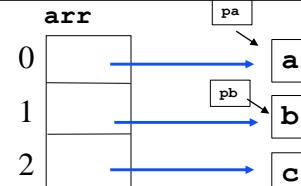
    int * arr[3]; // an array of 3 (uninitialized) int pointers
    arr[0]= pa;   arr[1]= pb;   arr[2]= &c;

    printf("%d\n", *arr[0]); // 4      arr[i] is pointer
    printf("%d\n", *arr[1]); // 10     *(arr[i]) is an int
    printf("%d\n", *(arr[2])); // 20   **(arr+2)

    *arr[1] = 100; // alias of b. Set b to 100

    for (i=0; i<3, i++)
        printf("%d ", *arr[i]); // 4 100 20
}
```

Pointee level



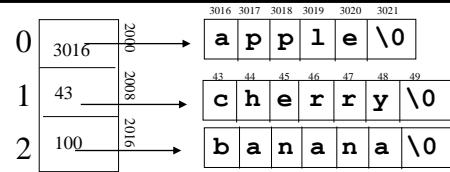
116

Array of pointers to strings

```
#include<stdio.h>
main(){
    char * words[]={ "apple", "cherry", "banana" };

    printf("%s\n", words[0]); // apple   *words
    printf("%s\n", words[1]); // cherry  *(words+1)
    printf("%s\n", words[2]); // banana  *(words+2)

    for (i=0; i<3, i++)
        printf("%d ", strlen(words[i]) );
} // 5 6 6           *(words+i)
Recall: int a=10;    char arr[]="apple";
         int pA = &a;    char * pArr = arr;
         printf("%d %d", a, *pA);      // pointee level
         printf("%s %s", arr, pArr); // pointer level
```



117

117

Array of pointers to scalar types

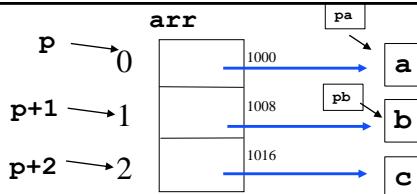
```
main(){
    int a,b,c, *pa, *pb;          p+2-->2
    a=4; b=10;c=20;              p+1-->1
    pa=&a, pb=&b;                p-->0

    int * arr[3];
    arr[0]= pa;    arr[1]= pb;  arr[2]= &c;

    int ** p = arr; // p = &arr[0] == 1000

    printf("%d\n", **p); // 4    *arr[0] "pointee level"
    printf("%d\n", **(p+1)); // 10  *arr[1]
    printf("%d\n", *(*(p+2))); // 20  *arr[2]

    for (i=0; i<3, i++)
        printf("%d\n", **(p+i));
}
```



Recall:

$p + i == \&arr[i]$
 $*(p+i) == arr[i]$

118

Array of pointers to strings

| | | | | | |
|-----------|-----------------|--------------|---------------------|----------------------|--|
| words → 0 | 3016 → 2000 → a | p → 3016 → a | p+1 → 43 → 3017 → p | p+2 → 100 → 3018 → p | 3016 → 3017 → 3018 → 3019 → 3020 → 3021 → \0 |
| | | | 44 → 3017 → p | 45 → 3018 → p | |
| | | | 46 → 3019 → p | 47 → 3020 → p | |
| | | | 48 → 3021 → \0 | 49 → \0 | |

```

main() {
    char * words[] = {"apple", "cherry", "banana"};
    char ** p = words; // p = &words[0] == 2000

    printf("%p %s\n", p, *p); // 2000 apple words[0]
    printf("%p %s\n", p+1, *(p+1)); // 2008 cherry words[1]
    printf("%p %s\n", p+2, *(p+2)); // 2016 banana words[2]

    for (i=0; i<3, i++)
        printf("%d ", strlen(*(p+i))); // 5 6 6
    }
    Recall: p + i == &words[i]
    * (p+i) == words[i]
119     printf("%c\n", *(*p+1)+5)); // words[1][5] y
    printf("%c\n", **p); // words[0][0] a

```

Hardest today

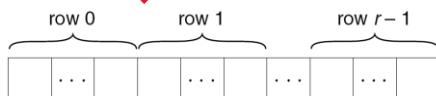
119

```

char planets[][] = {"Mercury", "Venus", "Earth",
                    "Mars", "Jupiter", "Saturn",
                    "Uranus", "Neptune", "Pluto"};

```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|----|----|----|----|----|
| M | e | r | c | u | r | y | \0 |
| V | e | n | u | s | \0 | \0 | \0 |
| E | a | r | t | \0 | \0 | \0 | \0 |
| M | a | r | \0 | \0 | \0 | \0 | \0 |
| J | u | p | i | \0 | \0 | \0 | \0 |
| S | a | t | u | \0 | \0 | \0 | \0 |
| U | r | a | n | \0 | \0 | \0 | \0 |
| N | e | p | t | \0 | \0 | \0 | \0 |
| P | l | u | t | \0 | \0 | \0 | \0 |

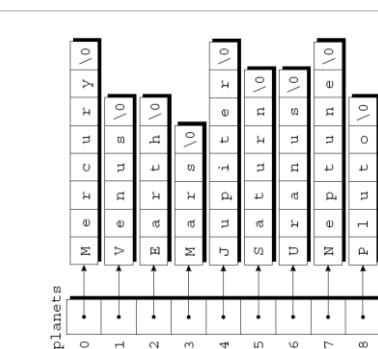


```

char tmp[8];
tmp = planets[0] ???
planets[0] = planets[1] ???
planets[1] = tmp; ???
strcpy(tmp, planets[0]);
strcpy(planets[0], planets[1]);
strcpy(planets[1], tmp);
120

```

O(n)



How to swap? →

```

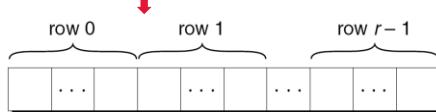
char *planets[] =
{"Mercury", "Venus", "Earth",
 "Mars", "Jupiter", "Saturn",
 "Uranus", "Neptune", "Pluto"};

```

120

```
char planets[][][8] = {"Mercury", "Venus", "Earth",
    "Mars", "Jupiter", "Saturn",
    "Uranus", "Neptune", "Pluto"};
```

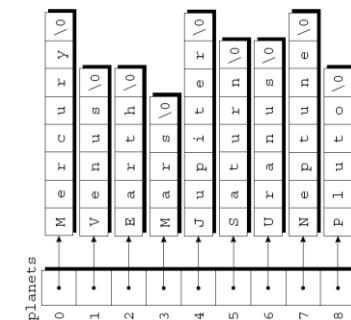
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | H | c | x | e | u | r | y |
| 1 | V | e | n | v | s | t | o |
| 2 | M | a | r | s | b | u | u |
| 3 | J | u | p | i | t | r | u |
| 4 | S | a | t | u | s | a | u |
| 5 | U | r | u | n | u | u | u |
| 6 | P | l | u | o | u | u | u |
| 7 | N | e | p | t | u | u | u |
| 8 | P | l | u | t | o | u | u |



↓ How to swap?

```
char tmp[8];
tmp = planets[0] ???
planets[0] = planets[1] ???
planets[1] = tmp; ???

for(i=0;i<8;i++){ //copy char one by one
    char tmp = planets[0][i];
    planets[1][i] = planets[0][i];
    planets[0][i] = tmp;
} O(n)
```



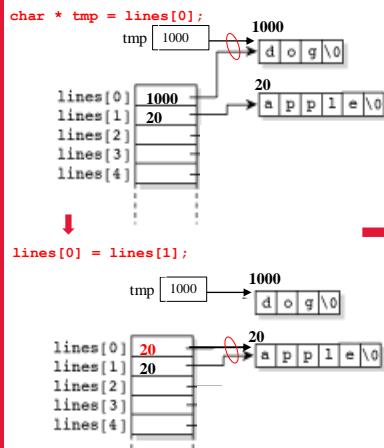
How to swap? →

```
char *planets[] =
{"Mercury", "Venus", "Earth",
 "Mars", "Jupiter", "Saturn",
 "Uranus", "Neptune", "Pluto"};
```

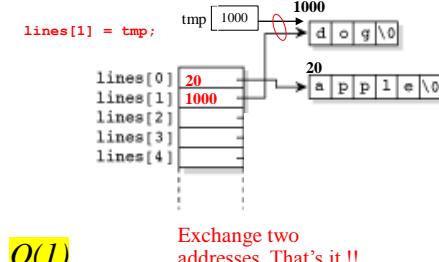
121

Efficient manipulation of strings

```
char *lines[]={"dog", "apple", "zoo", "program", "merry"};
// [0] vs [1]
char * tmp = lines[0]; // tmp gets 1000, pointing to "dog"
lines[0] = lines[1]; // [0] gets 20, pointing to "apple"
lines[1] = tmp; // [1] gets 1000, pointing to "dog"
```



Exchange pointers
(addresses)
Not real data ☺



Exchange two
addresses. That's it !!

O(1)

122

- Pointers (Ch5)
 - Basics: Declaration and assignment (5.1)
 - Pointer to Pointer (5.6)
 - Pointer and functions (pass pointer by value) (5.2)
 - Pointer arithmetic + - ++ -- (5.4)
 - Pointers and arrays (5.3)
 - Stored consecutively
 - Pointer to array elements $p + i = \&a[i]$ $*(p+i) = a[i]$
 - Array name contains address of 1st element $a = \&a[0]$
 - Pointer arithmetic on array (extension) $p1-p2$ $p1 <= p2$
 - Array as function argument – “decay”
 - Pass sub_array
 - Array of pointers (5.6-5.9)
 - Command line arguments (5.10)
 - Memory allocation (extra)
- Structures (Ch6)
 - Pointer to structures (6.4)
 - Self-referential structures (extra)



123

Insert into the list example1

```
struct node * head;
```

```
public void insert(int d1, double d2)
    Node newN = new Node(d1, d2);
    newN.nextLink = first;
    first = newN;
```

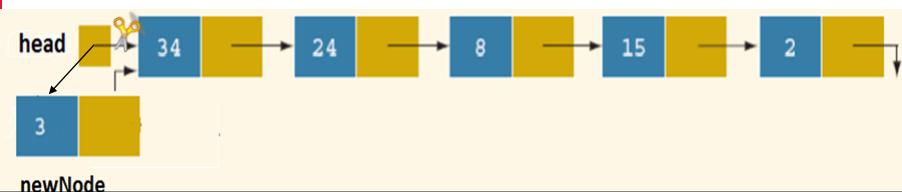
```
void insert_beginning(int dat)
{
    struct node newNode;

    newNode.data = dat;
    newNode.next = head;

    head = &newNode;
}
```



newNode
is in stack!



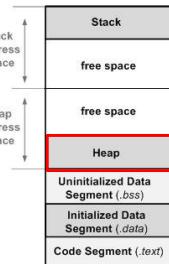
124

Stack vs. heap

- Local (**stack**) memory, automatic
 - Allocated on function call, and deallocated automatically when function exits
- Dynamic **heap** memory
 - The heap is an area of memory available to allocate areas ("blocks") of memory for the program.
 - Not deallocated when function exits.



What we need!



- Request a heap memory:
 - malloc() / calloc() / realloc() in C
 - new in C++ and Java
 - Student s = new Student();
- Deallocate from heap memory:
 - free() in C,
 - delete in C++
 - garbage collection in Java

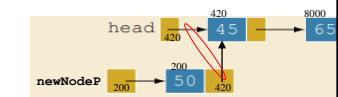
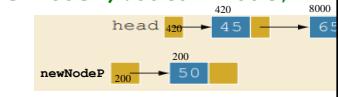
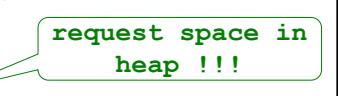
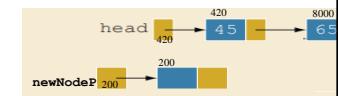
125

125

Insert into the list example1

```
struct node * head;  
  
void insert_beginning(int dat)  
{  
    struct node * newNodeP;  
    newNodeP = malloc(sizeof(struct node));  
  
    newNodeP -> data = dat; // (*newNodeP).data = dat;  
  
    newNodeP -> next = head; // (*newNodeP).next = head  
  
    head = newNodeP;  
}
```

126



126

- Labtest 2 review

127



127

```

4 int maxIndex(char *);           INSERTION-SORT(A) // A is a string
5 void insertionSort(char *);      0. n ← index of the last element in A
6 void setValue(char *, char);    1. for i ← 1 to n
7
8 int main()                      2.   key ← A[i]
9 {                                3.   j ← i - 1
10    int result; char c; int i; int count=0; 4.   while j ≥ 0 AND A[j] appears later in the ASCII table than key
11    char arr[50];                  5.     A[j+1] ← A[j]
12    fgets(arr,50,stdin);          6.     j ← j-1
13    while ( strcmp(arr,"quit\n") ) 7.   call sub-routine setValue() to set A[j+1] ← key
14    {
15      arr[strlen(arr)-1] = '\0'; // remove the trailing \n
16
17      int i = maxIndex(arr);
18      printf("%d %d %c\n", i, *(arr+i), *(arr+i));
19
20      insertionSort(arr);
21      printf("%s\n\n", arr);
22
23      //read again
24      fgets(arr,50,stdin);
25
26    }
27    return 0;
28 }
29
30 //                                           ↓
31 int maxIndex(char *c){
32
33
34   int maxI = 0;
35   int len = strlen(c)-1;
36   int i;
37   for(i=1; i<len; i++){
38     if (*c+i) > *(c+maxI)){
39       maxI = i;
40     }
41   }
42   return maxI;
43 }
```

```

45 /* insertion sort */
46 void insertionSort(char * c)
47 {
48
49   int n = strlen(c) -1;
50   int i;
51   for (i=1; i<=n; i++){
52     char key = *(c+i);
53     int j = i-1;
54     while(j >=0 && *(c+j)>key){
55       *(c+j+1) = *(c+j);
56       j--;
57     }
58     //*(c+j+1) = key;
59     setValue(c+j+1, key);
60   }
61
62 }
63
64 void setValue(char *ptr, char c){
65   *ptr =c;
66 }
```

128

```

    fgets(arr,50,stdin);
    while ( strcmp(arr,"quit\n") )
    {
        arr[strlen(arr)-1] = '\0'; // remove the trailing \n
        int minIndex, maxIndex;

        findIndexes(arr, &minIndex, &maxIndex);
        printf("minIndex:%d '%c'  maxIndex:%d '%c'\n",
               minIndex, *(arr+minIndex),
               maxIndex, *(arr+maxIndex));
        insertionSort(arr);
        printf("%s\n\n", arr);

        //read again
        fgets(arr,50,stdin);

    }
    return 0;
}

// void findIndexes(char *c, int *a, int *b){

int maxI, minI;
maxI = minI = 0;
int n = strlen(c)-1;
int i;
for(i=1; i<= n; i++){
    if (*(c+i) > *(c+maxI)){
        maxI = i;
    }
    else if (*(c+i) < *(c+minI))
        minI = i;
}
*a = minI;
*b = maxI;
}

```

129

```

71 void exchange(char arr[][50], int n)
72 {
73     char tmp[30];
74     int i;
75     for(i=0; i<n-1; i+=2){
76         strcpy(tmp, arr[i]);
77         strcpy(arr[i], arr[i+1]);
78         strcpy(arr[i+1], tmp);
79     }
80 }
81 void reverse(char *p[], int n)
82 {
83     char *tmp; // a pointer
84
85     int i; int last=n-1;
86     for(i=0; i<n/2; i++){
87
88         // swap pointers only
89         tmp = p[i]; /*(p+i); */
90         p[i] = p [last-i];
91         p[last-i] = tmp;
92     }
93 }
94
95 void printArray(char **p, int n)
96 {
97     int count;
98
99     for (count = 0; count < n; count++)
100         printf("[%d] ---> %s", count, *(p+count));
101
102 }
103
104

```

char tmp[30];

for(...){ // Works but not efficient!

```

strcpy( tmp, p[i]);
strcpy( *p[i], p[last-i] );
strcpy( p[last-i], tmp );
}

```

char * tmp; // not initialized!

for(...){

```

strcpy( tmp, p[i] );
strcpy( p[i], p[last-i] );
strcpy( p[last-i], tmp );
}

```

130

Whenever you need to set a pointer's pointee

e.g.,

- `*ptr = var`
- `*ptrArr[2] = var // pointer arrays`
- `scanf("%s", ptr)`
- `strcpy(ptr, "hello")`
- `fgets(ptr, .)`
-



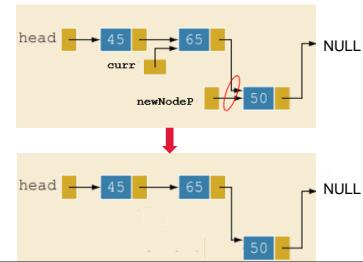
Ask yourself: have I done one of the following

1. `ptr = &var. /* direct */
a[20]; ptr=a;`
2. `ptr = ptr2 /* indirect, assume ptr2 is good */`
3. `ptr = (...)malloc(...) /* previous lecture */`

131

131

```
40
41 int len(){
42     int count = 0;
43     struct node * curr;
44     curr = head;
45     while (curr != NULL){
46         count++;
47         curr = curr -> next;
48     }
49     return count;
50 }
51 }
52
53 /* Insert a new data element with key d into the end of the list. */
54 void insert(int d) // at the end
55 {
56     struct node * newP = malloc (sizeof(struct node));
57     newP -> data = d;
58
59     /* special case: list is empty, need to ch
60     if (head == NULL){/* the list is empty */
61
62         head = newP;
63     }
64     else{ // general case, insert at the end.
65         struct node * current = head;
66
67         /* traverse to the end node */
68         while (current -> next != NULL)
69         {
70             current = current -> next;
71         }
72         /* now at the end node */
73
74         current-> next = newP;
75     }
76 }
```



132

UNIX

- Overview of UNIX
 - Structures
 - File systems
 - absolute and relative pathname
 - security `-rwx--x--x`
 - Process:
 - has return value 0 (success) or non 0 (sth wrong)
 - communication: pipes `who | sort who | grep Wang | wc -l`
- Utilities/commands
 - Basic `mkdir, cat, cp, rm, mv, file, wc, chmod`
 - Advanced `grep/egrep, uniq, sort, diff/cmp, cut, find`
- Shell (common shell functionalities)
 - Bourn (again) Shell
 - scripting language

133

Lab8 is helpful



133

That's all for the course

- I know you might have suffered more or less from the course.
Learning C & UNIX is not an easy task, and this course has never been an easy course.
- I tried my best to make the course doable.
- Hope you find this course useful, though.
- Thank you for all your supports!



134

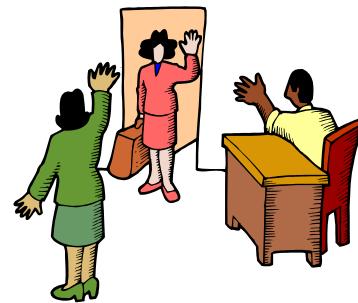


Time to say good bye

- Good luck with your exams and future studies
- Enjoy the rest of Summer!



so long ...



YORK U
UNIVERSITÉ
UNIVERSITY

135

135