CS 288 Intensive Programming in Linux

Professor Ding, Xiaoning

What is a regular expression?

- A regular expression (*regex*) describes a set of possible input strings.
- *Regular expressions* descend from a fundamental concept in computer science called *finite automata* theory
- *Regular expressions* are used by many tools in Unix, many languages, and many library functions.
  - **expr, [[, test, vi**, **grep, sed**, **emacs, …**
  - **C, awk**, **tcl**, **perl** and **Python**
  - **Compilers**
  - **Scanf (e.g., scanf("%[^\n]", str) to read a whole line)…**

expr: length of matching substring at beginning of a string

expr match STRING REGEX_STR

expr STRING : REGEX_STR

- **REGEX_STR** is a string describing the pattern (regular expression).
  - If included in single quotes, special characters must be escaped.
- expr prints out the number of characters matched
- Exit code is 0 if the string matches the pattern, and 1 otherwise.

```
$ stringZ=abcABC123ABCabc
#         |------|
#         12345678
$ expr match "$stringZ" 'abc[A-Z]*.2'    # 8
$ expr "$stringZ" : 'abc[A-Z]*.2'        # 8
```

expr : extracts substring at beginning of string

expr match STRING '\(REGEX_STR\)'

expr STRING : '\(REGEX_STR\)'

- **REGEX_STR** is a string describing the pattern (regular expression).
  - If included in single quotes, special characters must be escaped.

```
$ stringZ=abcABC123ABCabc
  #       ========
$ expr match "$stringZ" '\(.[b-c]*[A-Z]..[0-9]\)'
  abcABC1
$ expr "$stringZ" : '\(.[b-c]*[A-Z]..[0-9]\)'
  abcABC1
$ expr "$stringZ" : '\(.......\)'`
  abcABC1
```

expr : extracts substring at end of string

```
expr match STRING '.*\(REGEX_STR\)'
expr STRING : '.*\(REGEX_STR\)'
```

- **REGEX_STR** is a string describing the pattern (regular expression).
  - If included in single quotes, special characters must be escaped.

```
$ stringZ=abcABC123ABCabc
  #                ======
$ expr match "$stringZ" '.*\([A-C][A-C][A-C][a-c]*\)'
  ABCabc
$ expr "$stringZ" : '.*\(......\)'
  ABCabc
```

Using expr in if construct

```
$ if expr 1 "<" 2; then echo true; else echo false; fi
1
true
$ if expr filename : file; then echo true; else echo false; fi
4
true
$ if expr filename : File; then echo true; else echo false; fi
0
false
```

String matching using [[and =~

[[String =~ Pattern]]

- Pattern: extended regular expression (ERE) string
- The return value is 0 if the string matches the pattern, and 1 otherwise.

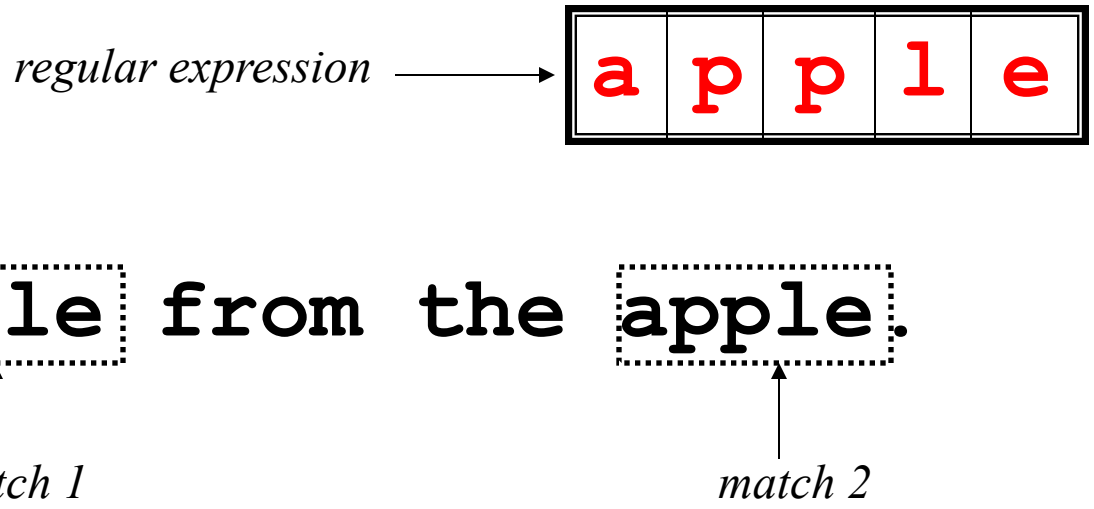
```
$ if [[ 'test' =~ 'es' ]]; then  echo true; else  echo false; fi
  true
$ if [[ 'test' =~ '^es' ]]; then  echo true; else  echo false; fi
  false
```

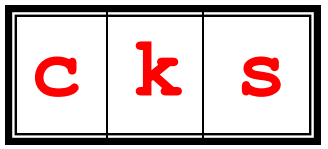
Regular expressions

- The simplest regular expressions are a string of literal characters to match.
- The string ***matches*** the regular expression if it contains the substring.

Regular expressions

- A regular expression can match a string in more than one place.



regular expression → 


```
UNIX Tools rocks.
```

match

```
UNIX Tools sucks.
```

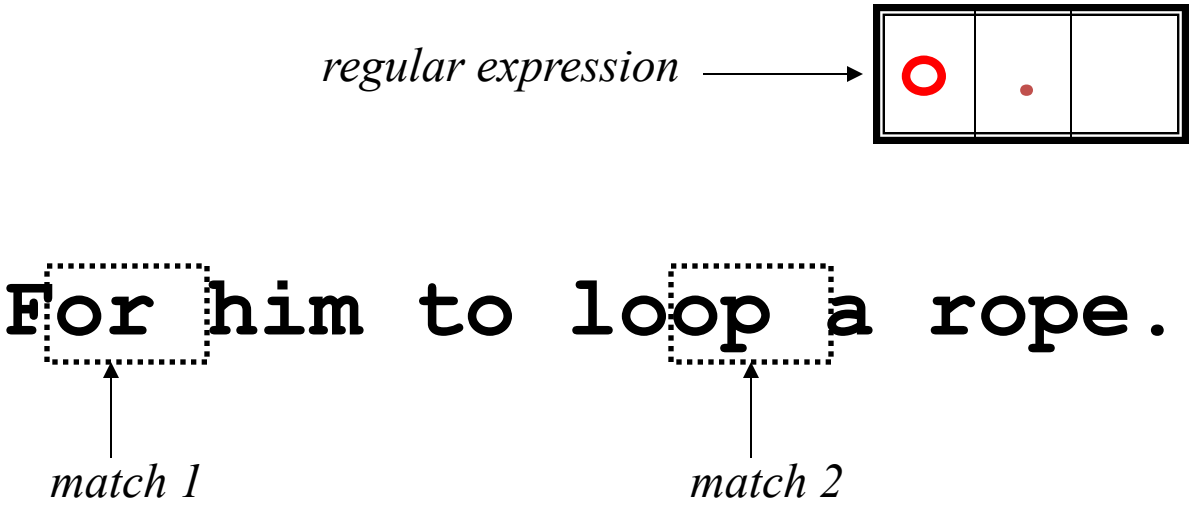
match

```
UNIX Tools is okay.
```

no match

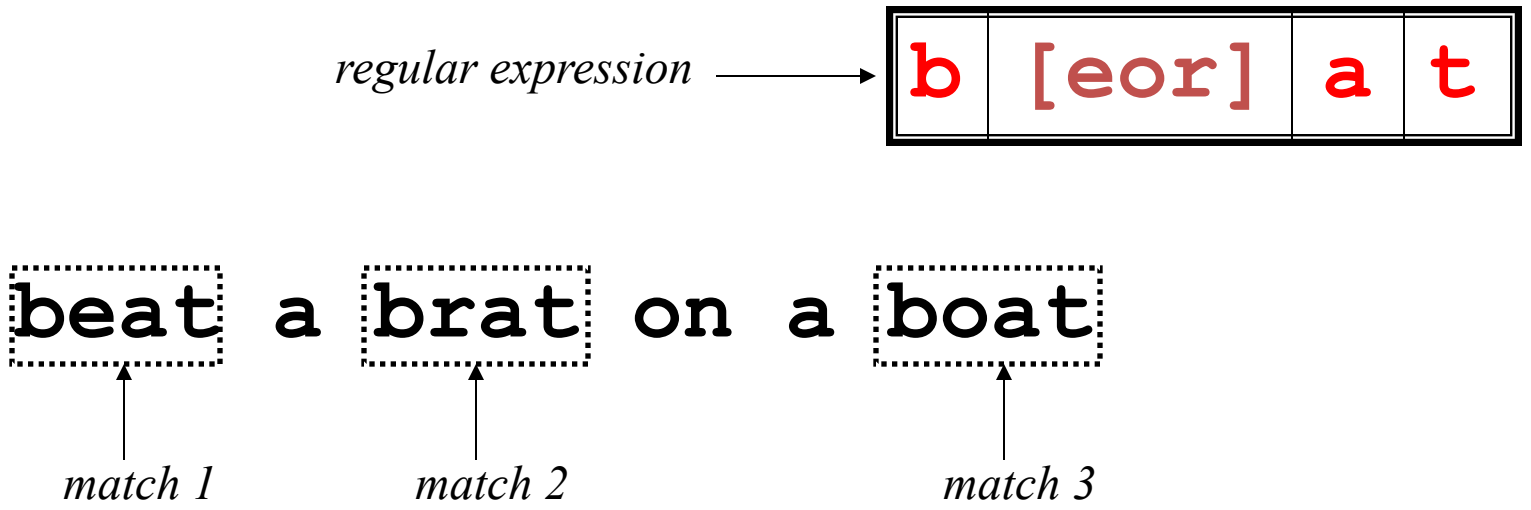
Regular expressions

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



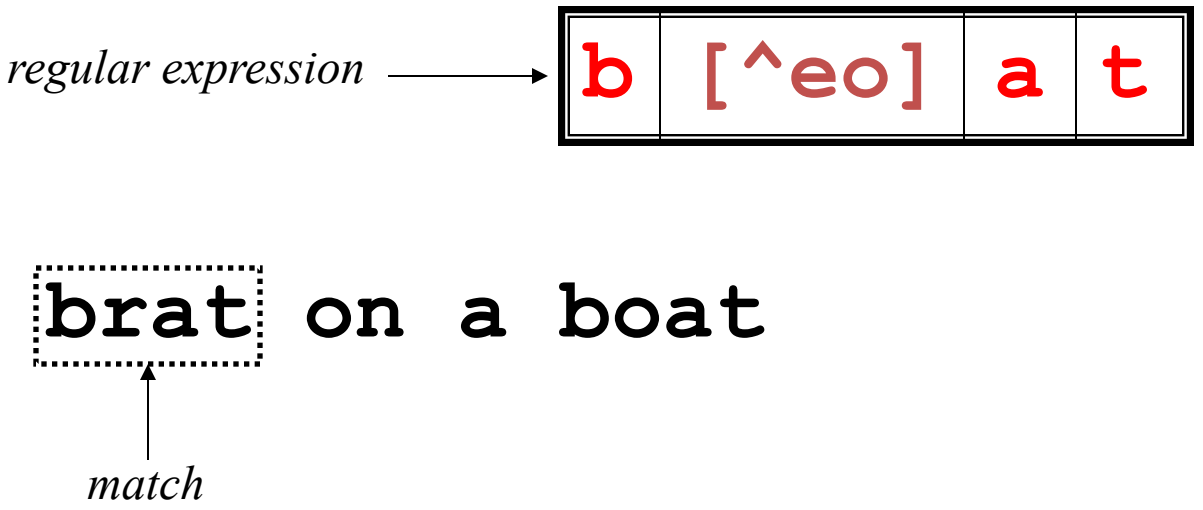
Character classes

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



Negated character classes

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



More about character classes

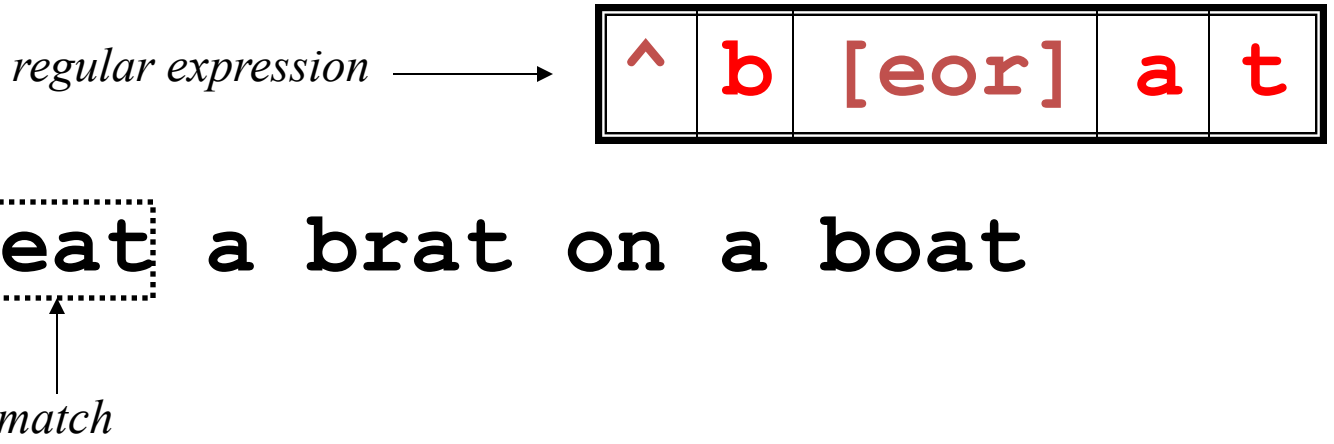
- `[aeiou]` will match any of the characters `a`, `e`, `i`, `o`, or `u`
- `[kK]orn` will match `korn` or `Korn`

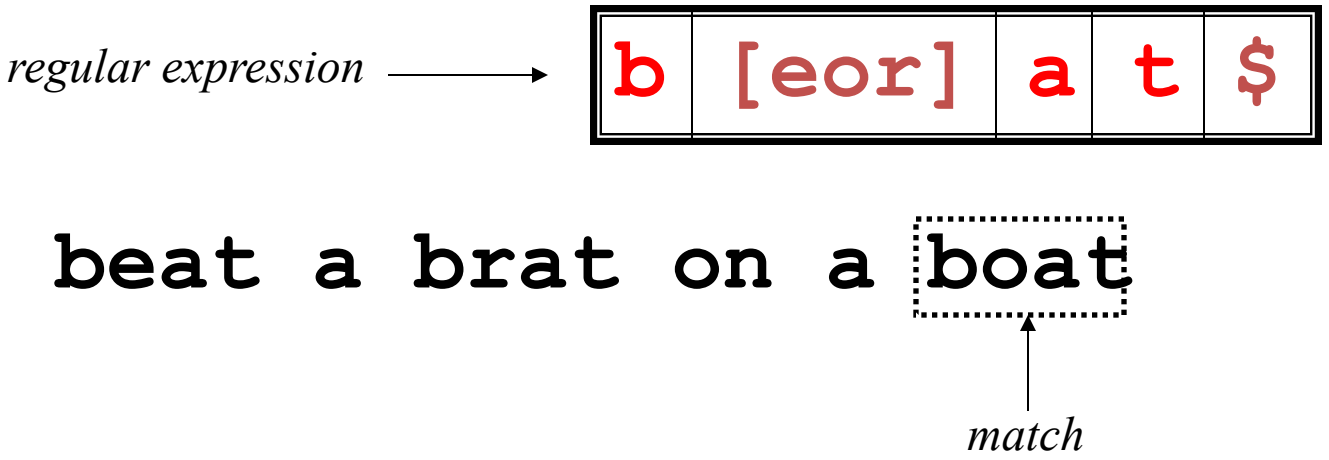
- Ranges can also be specified in character classes
  - `[1-9]` is the same as `[123456789]`
  - `[abcde]` is equivalent to `[a-e]`
  - You can also combine multiple ranges
    - `[abcde123456789]` is equivalent to `[a-e1-9]`
  - Note that the `-` character has a special meaning in a character class ***but only*** if it is used within a range,
    `[-123]` would match the characters `-`, `1`, `2`, or `3`

Named character classes

- Commonly used character classes can be referred to by name (*alpha*, *lower*, *upper*, *alnum*, *digit*, *punct*, *cntrl*)
- Syntax `[:name:]`
  - `[a-zA-Z]` `[[:alpha:]]`
  - `[a-zA-Z0-9]` `[[:alnum:]]`
  - `[45a-z]` `[45[:lower:]]`
- Important for portability across languages

- ***Anchors*** match at the beginning or end of a line (or both).
- `^` means beginning of the line, `$` means end of the line

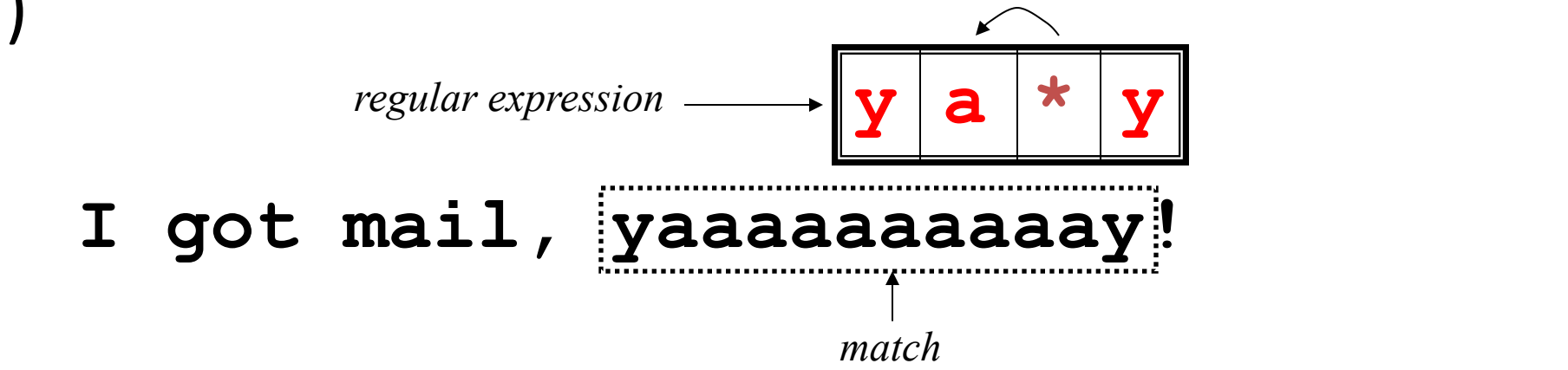


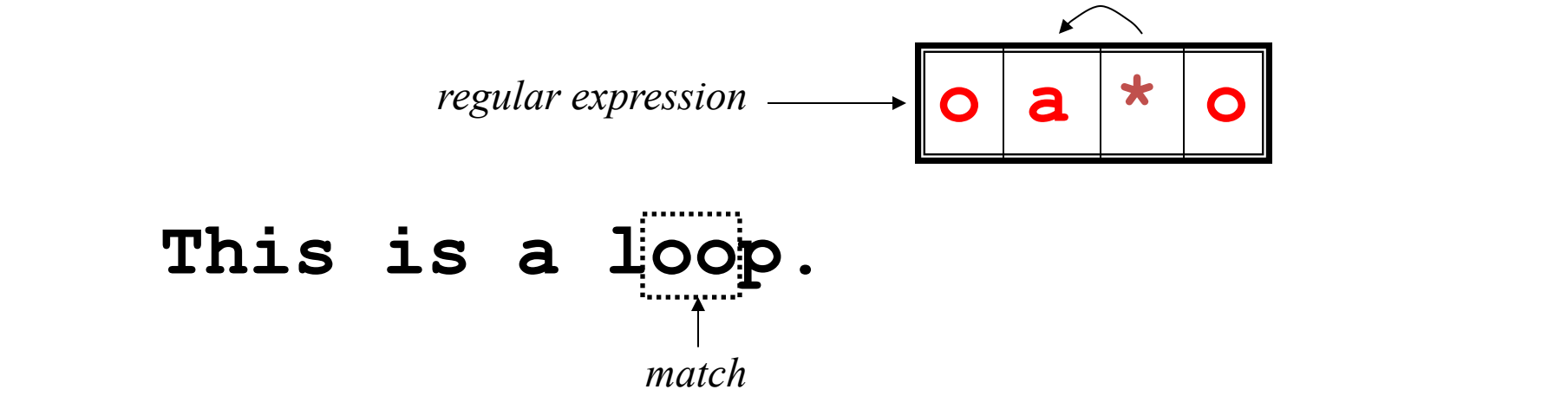


`^word$` `^$`

Repetition

The * defines **zero or more** occurrences of the *single* regular expression preceding it. (* is a normal character if no preceding expression)

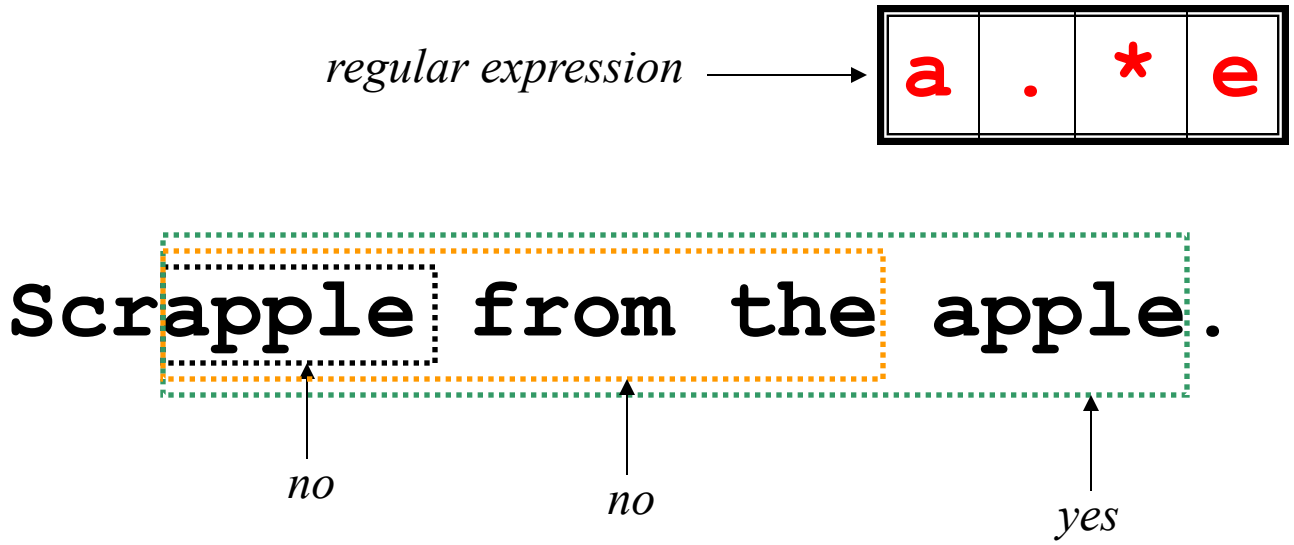




.*

Match length

- A match will be the longest string that satisfies the regular expression.



Repetition ranges

- Ranges can also be specified
  - `{ }` notation can specify a range of repetitions for the immediately preceding regex
  - `{n}` means exactly *n* occurrences
  - `{n,}` means at least *n* occurrences
  - `{n,m}` means at least *n* occurrences but no more than *m* occurrences
- Example:
  - `.{0,}` same as `.*`
  - `a{1,}` same as `aa*`

Subexpressions

- If you want to group part of an expression so that `*` or `{ }` applies to more than just the previous character, use `( )` notation
- Subexpresssions are treated like a single character
  - `a*` matches 0 or more occurrences of `a`
  - `abc*` matches `ab`, `abc`, `abcc`, `abccc`, …
  - `(abc)*` matches `abc`, `abcabc`, `abcabcabc`, …
  - `(abc){2,3}` matches `abcabc` or `abcabcabc`

backreferences

- Sometimes it is handy to be able to refer to a match that was made earlier in a regex
- This is done using *backreferences*
  - ***n*** is the backreference specifier, where *n* is a number (e.g., \1, \2, …)
  - Looks for *n*th subexpression, and repeats the corresponding match (e.g., repeat the match of 1st subexpression)

backreferences

```
$ cat ./test.txt
one is one
two is not one
one one two two
one two one two
```

```
'^([a-zA-Z]{1,}) .* \1$'
$ grep '^\([a-zA-Z]\{1,\}\) .* \1$' ./test.txt
one is one
```

```
'^([a-zA-Z]{1,}) .* ([a-zA-Z]{1,})$'
$ grep '^\([a-zA-Z]\{1,\}\) .* \([a-zA-Z]\{1,\}\)$' ./test.txt
one is one
two is not one
one one two two
one two one two
```

backreferences

```
$ cat ./test.txt
one is one
two is not one
one one two two
one two one two
```

```
grep '^([a-zA-Z]{1,}) ([a-zA-Z]{1,}) \1 \2$' ./test.txt
```

```
one two one two
```

```
$ grep "^\([a-zA-Z]\{1,\}\) \1 \([a-zA-Z]\{1,\}\) \2$" ./test.txt
one one two two
```

Escape special characters

- Special characters: . * ^ $

```
# search for text matching "*.jpg", e.g., "*ajpg", "*bjpg"
$ grep "*.jpg" mylist
# search for text matching "*.jpg".    "*ajpg", "*bjpg" don't match
$ grep "*\.jpg" mylist
$ grep "a*b*" grepme
$ grep "a\*b\*" grepme
```

Different tools, different RE standards

- BRE (Basic)
  - POSIX standard basic regular expressions.
  - Rules introduced earlier.
- ERE (Extended)
  - POSIX extended regular expression
- PCRE (Perl Compatible)
  - Perl-Compatible regular expressions, supported by many languages and tools, can recognize languages beyond regular expressions and therefore can match braces. etc.
- https://en.wikipedia.org/wiki/Regular_expression#Standards

A major syntax difference between BRE and ERE:

- **BRE**: `\(` and `\)`, `\{` and `\}`
- **ERE**: `(` and `)`, `{` and `}`

Examples: vim and grep

- Different tools support different regex standards
  - May not strictly follow the standards
- vim generally support PCRE (still, not strictly follow). In vim's manual:

```
9. Compare with Perl patterns *perl-patterns*

Vim's regexes are most similar to Perl's, in terms of
what you can do. The difference between them is mostly
just notation;  here's a summary of where they differ:
```

- **grep** (**g**lobal **r**egular **e**xpression **p**rint): an important tool to search strings or patterns from files.
  - Support all three standards.

ERE

A major syntax difference between BRE and ERE:

- **BRE**:
  - Subexpressions: `\(` and `\)`
  - Repetition: `\{` and `\}`
  - Normal parenthesis and brace symbols: `(`, `)` , `{`, and `}`
- **ERE**: `(` and `)`, `{` and `}`
  - Subexpressions: `(` and `)`
  - Repetition: `{` and `}`
  - Normal parenthesis and brace symbols: `\(`, `\)` , `\{`, and `\}`

ERE: alternation

- Regex also provides an alternation character `|` for matching one or another subexpression
  - `(T|Fl)an` will match ‘Tan’ or ‘Flan’
  - `^(From|Subject):` will match the From and Subject lines of a typical email message
    - It matches a beginning of line followed by either the characters ‘From’ or ‘Subject’ followed by a ‘:’
- Subexpressions are used to limit the scope of the alternation
  - `At(ten|nine)tion` then matches “Attention” or “Atninetion”, not “Atten” or “ninetion” as would happen without the parenthesis - `Atten|ninetion`

ERE: repetition shorthands

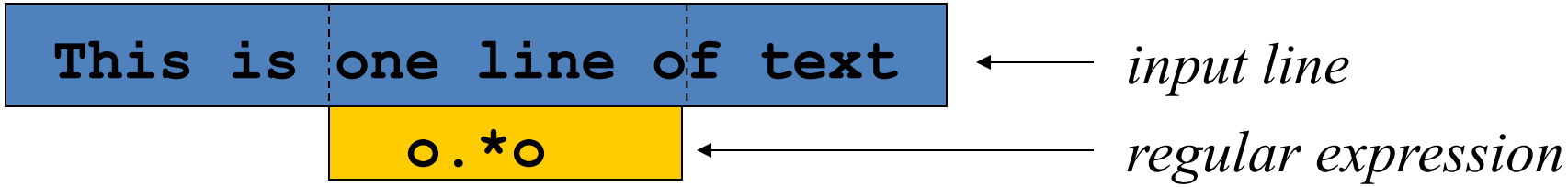
- The `*` (star) has already been seen to specify zero or more occurrences of the immediately preceding character
- `+` (plus) means “one or more”
  - `abc+d` will match ‘abcd’, ‘abccd’, or ‘abccccccd’ but will not match ‘abd’
  - Equivalent to `{1,}`

ERE: repetition shorthands cont

- The '`?`' (question mark) specifies an optional character, the single character that immediately precedes it
  - `July?` will match 'Jul' or 'July'
  - Equivalent to `{0,1}`
  - Also equivalent to `(Jul|July)`
- The `*`, `?`, and `+` are known as *quantifiers* because they specify the quantity of a match
- Quantifiers can also be used with subexpressions
  - `(a*c)+` will match 'c', 'ac', 'aac' or 'aacaacac' but will not match 'a' or a blank line

Practical regex examples with ERE

- Variable names in C
  - `[a-zA-Z_][a-zA-Z_0-9]*`
- Dollar amount with optional cents
  - `\$[0-9]+(\.[0-9][0-9])?`
- Time of day
  - `(1[012]|[1-9]):[0-5][0-9] (am|pm)`
- HTML headers <h1> <H1> <h2> …
  - `<[hH][1-4]>`



x	Ordinary characters match themselves (NEWLINES and metacharacters excluded)	*BRE, ERE*
xyz	Ordinary strings match themselves	
\m	Matches literal character *m*	
^	Start of line	
$	End of line	
.	Any single character	
[xy^$x]	Any of x, y, ^, $, or z	
[^xy^$z]	Any one character other than x, y, ^, $, or z	
[a-z]	Any single character in given range	
r*	zero or more occurrences of regex r	
r1r2	Matches r1 followed by r2	
\(r\)	Tagged regular expression, matches r	*BRE*
\n	Set to what matched the *n*th tagged expression (n = 1-9)	
\{n,m\}	Repetition	
r+	One or more occurrences of r	*ERE*
r?	Zero or one occurrences of r	
r1|r2	Either r1 or r2	
(r1|r2)r3	Either r1r3 or r2r3	
(r1|r2)*	Zero or more occurrences of r1|r2, e.g., r1, r1r1, r2r1, r1r1r2r1,…)	
{n,m}	Repetition	

Quick Reference

Regex usages

- *Regular expressions* are used by many tools in Unix
  - vi, ed, sed, and emacs
  - ***grep, egrep,*** fgrep

- **Regex support** is part of the standard library of many **programming/scripting languages**
  - ***C, Python***, Java, ….
  - Perl, tcl

grep and sed

grep: look for patterns from files or standard input

grep [-hilnv] regex [filenames]

-h Do not display filenames
-i Ignore case
-l List only filenames containing matching lines
-n Precede each matching line with its line number
-v Negate matches
-o Print only the matched (non-empty) parts
regex regular expression
filenames pathnames of files. If no files specified, grep checks stdin

grep is based on BRE by default, but support other standards

BRE: grep –G ERE: grep –E PCRE: grep -P

grep Examples

- `echo "mechism" | grep 'me'`
- `grep 'fo*' GrepMe`
- `grep –E 'fo+' GrepMe`
- `grep –E -n '[Tt]he' GrepMe`
- `grep –E 'NC+[0-9]*A?' GrepMe`

- **Find all lines with signed numbers**

```
$ grep -E '[-+][0-9]+\.?[0-9]*' *.c
bsearch. c: return -1;
compile. c: strchr("+1-2*3", t-> op)[1] - '0', dst,
convert. c: Integers in a base 2-16 (default 10)
convert. c: sscanf( argv[ i+1], "% d", &base);
strcmp. c: return -1;
strcmp. c: return +1;
```

Show only the matched part: grep -o

```
$ cat demo_file
THIS LINE IS THE 1ST UPPER CASE LINE IN THIS FILE.
this line is the 1st lower case line in this file.
This Line Has All Its First Character Of The Word With Upper Case.

Two lines above this line is empty.
And this is the last line.
```

```
$ grep -o "is.*line" demo_file
is line is the 1st lower case line
is line
is is the last line
```

Would be very useful when used to process irregular text. You will feel it when you do your homework.

Regex strings need quotes

- Regex strings may contain spaces
  - Spaces splits regex strings into multiple words/arguments.
- Regex strings may contain meta-characters
  - Special characters in regex strings may have special meaning to the shell
  - Shell converts (expansions) regex strings into something else.

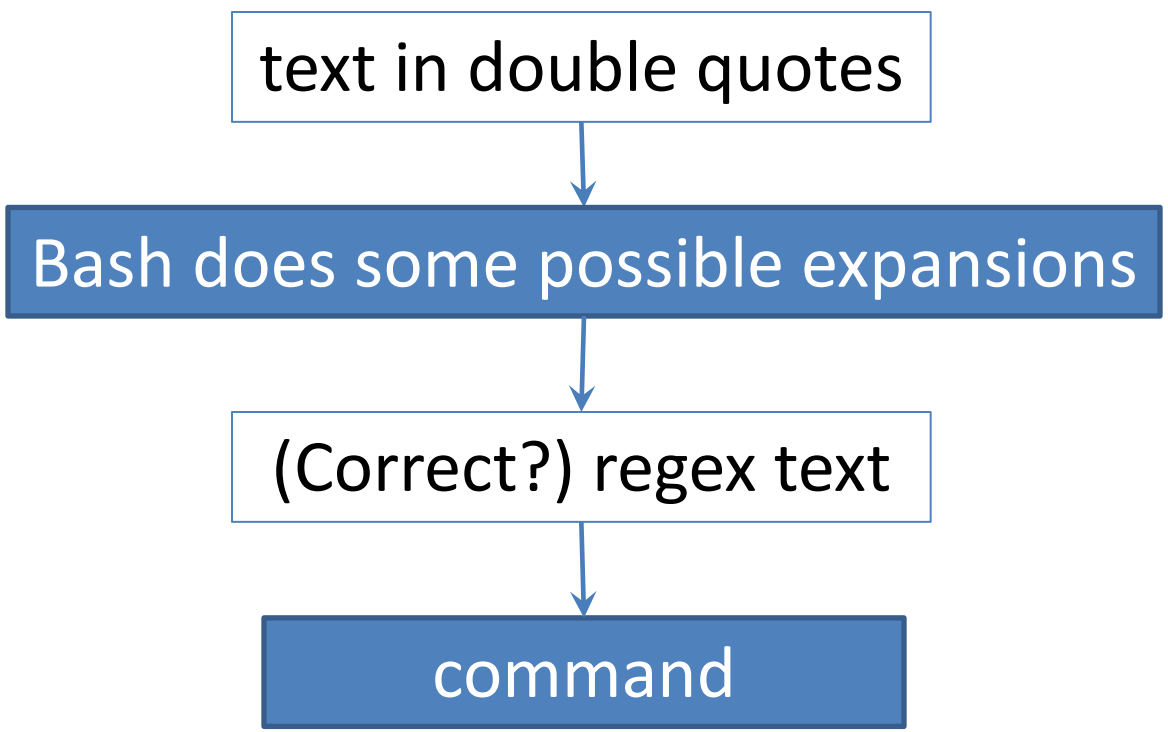
```
#filename expansion. Search the first jpg filename in other jpg files and mylist
#e.g. grep a.jpg b.jpg c.jpg mylist
$ grep *.jpg mylist
#~tom is replaced with user tom's home directory
$ expr $STR : ~tom
#search for abcd not $var
$ var='abcd'; grep $var grepme.c
```

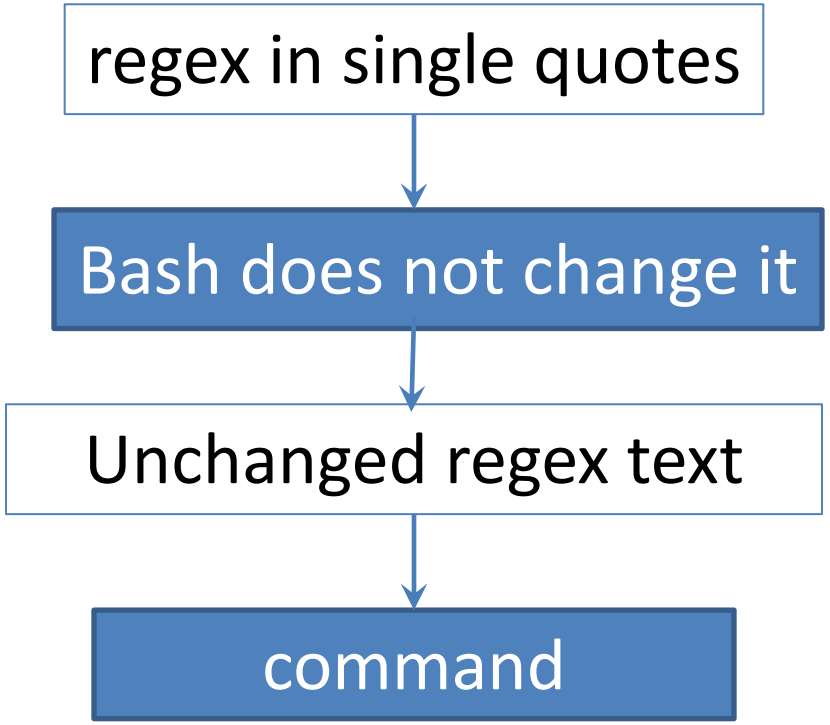
Single quotes vs. double quotes

- Bash rules on double quotes are complicated. Double quotes prevent some expansions but allows others.
  - What is performed in double quotes: Expansions beginning with $; command substitutions with back ticks `...`, Backslash escaping, filename expansion.
  - What is not: word splitting, brace expansion
- Single quotes remove the special meaning of every character (including \" between them.
  - The only character that cannot be safely enclose in single quotes is a single quote.
    - \ is not a escaping character for bash any more

```
# no filename expansion. search for text matching regex "*.jpg", e.g., "*ajpg", "*bjpg"
$ grep '*.jpg' mylist
$ grep "*.jpg" mylist
$ var='abcd'
$ grep '$var' grepme.c      #search for $var
$ grep "$var" grepme.c      #var name expansion. search for abcd
```

Single quotes vs. double quotes





no filename expansion. search for text matching regex "*.jpg", e.g., "*ajpg", "*bjpg"

```
$ grep '*.jpg' mylist
$ grep "*.jpg" mylist
$ var='abcd'
$ grep '$var' grepme.c      #search for $var
$ grep "$var" grepme.c      #var name expansion. search for abcd
```

Consider single quotes first

- Usually regex strings don't have special characters that need shell to interpret.
- Use double quotes cautiously (Use only when needed, e.g., when expansion results must be included)
- Split the generation of regex test into steps
  - Save the expansion results into variables.
  - Generate regex text by combining variables.
  - Optional: check the regex text.
  - Use the regex text.

```
filesize=`du -b myfile | cut -f 1`
regex="size: $filesize"
echo regex
grep "$regex" sometablefile
```

Sed: Stream-oriented, Non-Interactive, Text Editor

- Look for patterns one line at a time, like **grep**
- *process* lines in a file or from a pipe stream
- Non-interactive text editor
  - Editing commands come in as ***script*** in command line or script-file
  - Each command consists of up to two *addresses* and an *action*.
    - *address* can be a *regular expression* or *line number*.
- A Unix filter
  - All editing commands in a script are applied in order to each input line.
  - The original input file is unchanged. The results are sent to stdout (can be redirected).
  - Superset of previously mentioned tools, e.g., cat, grep, cut, …

Simple sed examples

```
$ cat > geekfile.txt
unix is great os. unix is opensource.
unix is free os. Do you choose unix?

$ sed 's/unix/linux/' geekfile.txt
linux is great os. unix is opensource.
linux is free os. Do you choose unix?

$ sed '2d' geekfile.txt
linux is great os. unix is opensource.
```

Sed advantages and drawbacks

- Regular expressions
- Fast
- Concise
- Not possible to go backward in the file

sed overview

sed *[-n] script [file…]* *or* ***sed*** *[-n] -e script [file…]*
sed *[-n] -f script-file [file…]*

- ***-f scriptfile*** - specify a filename containing editing commands.
- ***script -*** specify editing commands in a script in command line
- ***-e script*** - mostly used in command lines with multiple scripts (details later)
- For each line in the input file, *sed* does the following
  - reads the first command of the script and checks the *address* against the line.
    - If there is a match, the command is executed; otherwise, the command is ignored
  - repeats this action for every command in the script.
    - Commands are "**pipelined**“ --- later command processes the output of previous command
  - After all the commands, outputs the current line unless the ***-n*** option is set
- ***-n*** – quite, only print lines specified with print commands
  - If the first line of a scriptfile is “***#n***”, sed acts as though ***-n*** had been specified
- ***file***: input file. Input is the output piped from another command.

sed script overview: *[address[, address]][!]command [arguments]*

- An address can be either a line number or a regular expression, enclosed in slashes (***/pattern/***)
- A command with **one address** is applied when the address matches the line
  - One line if the address is a line number. Multiple lines if address is a regex.
- A command with **no address** is applied to the line unconditionally.
- A command with **two addresses** is applied to a range of lines between the two addresses, inclusively.
- **$** refers to the last line
- **!** negates an address
  - *address!command : command* is applied to all lines that do ***not*** match *address*

sed script: *[address[, address]][!]command [arguments]*

- *command* is a single letter.
- Some commonly used commands:

`p` - print	**`a`** - append
`=` - print line number	**`i`** - insert
`d` - delete	**`c`** - change
`y` - transform	**`q`** - quit
`s` - substitute	HhGg - pattern/hold spaces

Print: **`[address(es)]p`**

- The Print command (**`p`**) can be used to force the pattern space to be output, useful if the ***-n*** option has been specified
- Note: if the ***-n*** or ***#n*** option has not been specified, **`p`** will cause the line to be output twice!
- Examples:

  `1,5p` : display lines 1 through 5

  `/^$/,$p` : display lines from the first blank line through the last line

Print line number : [address(es)]=

- Find the lines, and print their line numbers
- Two commands finding the line numbers of the lines that contain a pattern

```
sed -n '/PATTERN/ =' file
cat -n file | grep 'PATTERN' | cut -f 1
```

- Two commands finding the number of lines in a file

```
sed -n '$=' file
wc -l file3.txt | cut -d' ' -f 1
```

Delete lines `[address(es)]d`

`d`	deletes the all lines
`6d`	deletes line 6
`/^$/d`	deletes all blank lines
`1,10d`	deletes lines 1 through 10
`1,/^$/d`	deletes from line 1 through the first blank line
`/^$/,$d`	deletes from the first blank line through the last line of the file
`/^$/,10d`	deletes from the first blank line through line 10
`/^ya*y/,/[0-9]$/d`	deletes from the first line that begins with yay, yaay, yaaay, etc. through the first line that ends with a digit

Transform: [address(es)]y/src/dst/

- similar to **tr**, character-to-character replacement
- Example: convert a hexadecimal number (e.g. 0x1aff) to upper case (0x1AFF)

```
$ echo '0x1aff' | sed '/0x[0-9a-fa-F]*/y/abcdef/ABCDEF/'
0x1AFF
$ echo 'Value 0x1aff' | sed '/0x[0-9a-fa-F]*/y/abcdef/ABCDEF/'
VAluE 0x1AFF
```

- Transform the whole line. Cannot be used to transform specific characters (or a word) in the line.

Substitute: *[address(es)]**s**/pattern/replacement/[flags]*

- *pattern* : **regex pattern**; *replacement* : replacement string for pattern
- *flags* are optional. / is needed even when there are no flags
  - **n** a number from 1 to 512 indicating which occurrence of *pattern* should be replaced (default is 1).
  - **g** global, replace all occurrences of *pattern*
  - **p** print resulted contents
- Some examples:
  - s/Unix/Linux/ : Substitute "Linux" for the first occurrence of "Unix"
  - s/Unix/Linux/2 : Substitutes "Linux" for the second occurrence of "Unix"
  - s/wood/plastic/p : Substitutes "plastic" for the first occurrence of "wood" and prints results
  - s/to/TWO/2g : replaces the second, third, etc occurrences of pattern “to” with “TWO”

Several special characters in the *replacement* string

- **&** - the entire string matching the regex
- *n* - substring matching the *n*th subexpression previously specified using “\(“ and “\)”
- **** - escape special characters, e.g., & and \

```
$ echo "the UNIX operating system …" | sed 's/.NI./wonderful &/'
"the wonderful UNIX operating system …"
$ cat test1
first:second
one:two
$ sed 's/\(.*\):\(.*\)/\2:\1/' test1
second:first
two:one
```

Several special characters in the *replacement* string

- **&** - the entire string matching the regex
- *n* – backreferencing, substring matching the *n*th subexpression previously specified using “\(“ and “\)”
- **** - escape special characters, e.g., & and \

```
$ script='s/\([[:alpha:]]\)\([^ \n]*\)/\2\1ay/g'
$ echo "unix is fun" | sed "$script"
nixuay siay unfay

#A "pig latin" translator: for each English word, move the
  first letter to the end and suffixes an "ay"
```

Append, insert, and change

- append/insert a new line
  - works for single lines only, not ranges
- Change a line or lines
  - When applied to a range, the entire range is replaced, not each line
  - *Exception*: If “Change” is executed with other commands in **{ }** that act on a range of lines, **each line** will be replaced with *text*
- Syntax for these commands is a little strange. They **must** be specified on multiple lines
  - line continuation with backslash \
  - *text* must begin on the next line. Leading whitespaces will be discarded unless whitespaces are "escaped“ with \

append after	*[address]a*\ *text*
Insert before	*[address]i*\ *text*
change	*[address(es)]c*\ *text*

```
$ cat ./file1.txt
<Insert Text Here>
$ sed '$ a\
Something else' ./file1.txt | tee ./file2.txt
<Insert Text Here>
Something else
$ sed '/<Insert Text Here>/i\
First Line\
Second Line\
\    Third line' ./file2.txt | tee ./file3.txt
First Line
Second Line
    Third line
<Insert Text Here>
Something else
```

```
$ sed '2,4c\
2~4 lines replaced' ./file3.txt
First Line
2~4 lines replaced
Something else
$ sed '2,4{
s/Line/line/
c\
New Line
}
' ./file3.txt
First Line
New Line
New Line
New Line
Something else
```

Multiple commands in a script

- Braces **{ }** apply multiple commands to the line
- **[address(es)]{cmd1;cmd2;cmd3}**
  - Used for simple commands.
- **[address(es)]{**

  command1

  command2

  command3

  }
  - *For commands using multiple lines*
  - *opening brace* must be the last character on a line.
  - *closing brace* must be on a line by itself.
  - no spaces should follow the braces

Using !

- If address(es) are followed by an exclamation (**!**), the associated command is applied to all lines that don't match the address(es)
  - e.g., **`1,5!d`** would delete all lines except 1 through 5

```
$ cat file.txt
The brown cow
The black cow

# replace "horse" for "cow" on the lines without "black"
$ sed '/black/!s/cow/horse/' file.txt
The brown horse
The black cow
```

Quit

- Quit causes **sed** to stop reading new input lines and stop sending them to standard output
- It takes at most a single line address
  - Once a line matching the address is reached, the script will be terminated
  - This can be used to save time when you only want to process some portion of the beginning of a file
- Example: to print the first 100 lines of a file (like *head*) use:
  - **`sed '100q' filename`**
  - sed will, by default, send the first 100 lines of *filename* to standard output and then quit processing

Multiple scripts

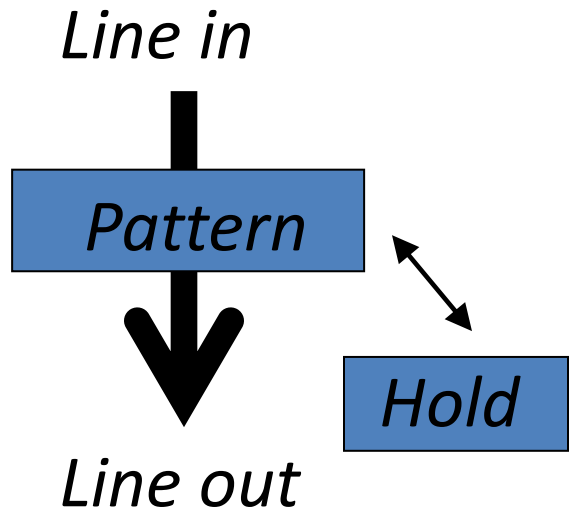
- Using new lines in a script file (the '-f' option), one script a line
- On the command line, each sed script may be
  - a newline, or
  - an argument with '-e' option, or
  - a substring separated using semicolon ";"

```
$ seq 6 | sed '1d
    3d
    5d'
    2
    4
    6
$ seq 6 | sed -e 1d -e 3d -e 5d
    2
    4
    6
$ seq 6 | sed '1d;3d;5d'
    2
    4
    6
```

Pattern space and hold space

- ***Pattern space***: Workspace or temporary buffer where a single line of input is held while the editing commands are applied
- ***hold buffer***: a long-term storage catching some information, storing it, and reusing it when it is needed. Empty initially.

H h	Append/copy contents in pattern space to hold space
G g	Append/copy contents in hold space to pattern space.
x	exchanges contents in pattern space and holding area.



Pattern space and hold space examples

```
#add empty line
$ seq 2 | sed 'G'
1

2

#copy back and forth
$ seq 2 | sed 'h;g'
1
2
#print in reverse order
$ seq 3 | sed -n '1!G;h;$p'
3
2
1
```

- First line: *h* places it into the hold space.
- 2nd line onwards:
  - *1!G* appends the hold space content with the pattern space. (Remember 2nd line is in pattern space, and 1st line is in hold space. Thus, now 1st and 2nd line got reversed.)
  - *h* copies all this to the hold space.
  - Repeat the above steps till last line.
- The last line:
  - 1!G appends the hold space content with the pattern space.
  - h copies all this to hold space (no actual effects)
  - $p prints the pattern space.

Regular Expression Support in python

Using Regular Expressions in Python

Python “re”: https://docs.python.org/2/library/re.html

- import the regexp module with “import re”.
- Call re.search(regex, subject) to apply a regex pattern to a subject string.
  - re.search() returns None if the matching attempt fails,
  - it returns a Match object otherwise.
  - The Match object stores details about the part of the string matched by the regular expression pattern.
  - Since None evaluates to False, you can easily use re.search() in an if statement.

Python's Regular Expression Syntax

- Basically PCRE dialect with distinct implementation.
  - re "module provides regular expression matching operations similar to those found in Perl".
- In addition to ERE
  - "\d" matches any digit; "\D" any non-digit
  - "\s" matches any whitespace character; "\S" any non-whitespace character
  - "\w" matches any alphanumeric character; "\W" any non-alphanumeric character
  - "\b" matches a word boundary; "\B" matches a character that is not a word boundary

Search and Match

- The two basic functions are **re.search** and **re.match**
  - Search looks for a pattern anywhere in a string
  - Match looks for a match staring at the beginning
- Both return *None* (logical false) if the pattern isn't found and a "match object" instance if it is

```
>>> import re
>>> pat = "a*b"
>>> re.search(pat,"fooaaabcde")
<_sre.SRE_Match object at 0x809c0>
>>> re.match(pat,"fooaaabcde")
```

What's a match object?

- an instance of the match class with the details of the match result

```
>>> r1 = re.search("a*b","fooaaabcde")
>>> r1.group()  # group returns string matched
'aaab'
>>> r1.start()  # index of the match start
3
>>> r1.end()    # index of the match end
7
>>> r1.span()   # tuple of (start, end)
(3, 7)
```

What got matched?

Here’s a pattern to match simple email addresses

\w+@(\w+\.)+(com|org|net|edu)

```
>>> pat1 = "\w+@(\w+\.)+(com|org|net|edu)"
>>> r1 = re.match(pat,"dingxn@njit.edu")
>>> r1.group()
'dingxn@njit.edu’
```

re.findall and re.finditer

- To get all matches from a string, call re.findall(regex, subject).
  - return an array of all non-overlapping matches in the string.
  - "Non-overlapping" means that the string is searched through from left to right, and the next match attempt starts beyond the previous match.

```
>>> re.findall("\d+”,"12 dogs,11 cats, 1 egg")
['12', '11', ’1’]
```

- More efficient is re.finditer(regex, subject).
  - Returns an iterator that enables you to loop over the regex matches in the subject string:

```
for m in re.finditer(regex, subject):
```

The for-loop variable m is a Match object with the details of the current match.

Compiling regular expressions to a pattern object

- If you plan to use a re pattern more than once, compile it to a Pattern object
  - Python produces a special data structure that speeds up matching
- Pattern objects have methods that are similar to the re functions

```
>>> p1 = re.compile("\w+@\w+\.+com|org|net|edu")
>>> p1.match("steve@apple.com").group(0)
'steve@apple.com'
>>> p1.search(”Email steve@apple.com
today.").group(0)
'steve@apple.com’
>>> p1.findall("Email steve@apple.com and
bill@msft.com now.")
['steve@apple.com', 'bill@msft.com’]
```

More re functions

- re.split() is like split but can use patterns

```
>>> re.split("\W+", “This... is a test, of split().”)
['This', 'is', 'a', 'test', 'of', 'split’, ‘’]
```

- re.sub substitutes one string for a pattern

```
>>> re.sub('(blue|white|red)', 'black', 'blue socks and red shoes')
'black socks and black shoes'
```

Regular Expression Support in C

POSIX RE facilities (C API)

Function	Purpose
regcomp()	Compiles a regex into a form that can be later used by regexec
regexec()	Matches string (input data) against the precompiled regex created by regcomp()
regerror()	Returns error string, given an error code generated by regcomp or regexec
regfree()	Frees memory allocated by regcomp()

regcomp()

```
#include <regex.h>
int regcomp(regex_t *preg, const char *pattern,
  int cflags);
```

- ***preg***: pointer to structure that will hold compiled RE
- ***pattern***: RE string
- ***cflags*** set options for the pattern
  - REG_EXTENDED: Extended EREs, not basic.
  - REG_NOSUB: Don't provide copies of substring matches; instead, just report if it matched or not. Almost always use this when filtering
  - REG_ICASE: Case insensitive setting
  - REG_NEWLINE: Detect lines for anchor characters to be effective

Returns nonzero if error - error code for regerror()

regexec()

```
#include <regex.h>
int regexec(const regex_t *preg, const char
  *string, size_t nmatch, regmatch_t pmatch[],
  int eflags);
```

- ***preg***: Compiled regex created by regcomp()
- ***string***: the string (data) to match against RE preg
- ***nmatch, pmatch***: used to report matches
- ***eflags***: Usually used when parsing a partial string and you do not want a beginning of line or end of line match
  - REG_NOTBOL: The first character of the string pointed to by string is not the beginning of the line. Therefore, anchor (^) will not match the beginning of string.
  - REG_NOTEOL: The last character of the string pointed to by string is not the end of the line. Therefore, anchor will not match the end of string.
- For filtering nmatch, pmatch, eflags aren't usually useful
- Returns 0 if match, REG_NOMATCH if no match, else error

pmatch and regmatch_t

- pmatch is filled in by regexec() with substring match addresses.
  - The match offsets for the ith subexpression (starting at the ith open parenthesis) are stored in pmatch[i].
  - The entire regular expression's match addresses are stored in pmatch[0].

- a pmatch entry is invalid, if its rm_so field is -1.

```
typedef struct{
     regoff_t rm_so;
     regoff_t rm_eo;
} regmatch_t;
```

  - ***rm_so*** that is not -1 indicates the start offset of the next largest substring match.
  - ***rm_eo*** indicates the end offset of the match (the offset of the first character after the matching text.

regfree and regerror

```
void regfree(regex_t *preg);
```

- Frees memory allocated by regcomp() to prevent memory leak
- ***preg***: Compiled regex created by regcomp()

```
size_t regerror(int errcode, const regex_t *preg,
        char *errbuf, size_t errbuf_size);
```

- ***errbuf***: a pointer to a character string buffer
- ***errbuf_size***: the size of the string buffer
- regerror() ***returns*** the size of the errbuf required to contain the null-terminated error message string.

If both errbuf and errbuf_size are nonzero, errbuf is filled in with the first errbuf_size - 1 characters of the error message and a terminating null byte ('\0').

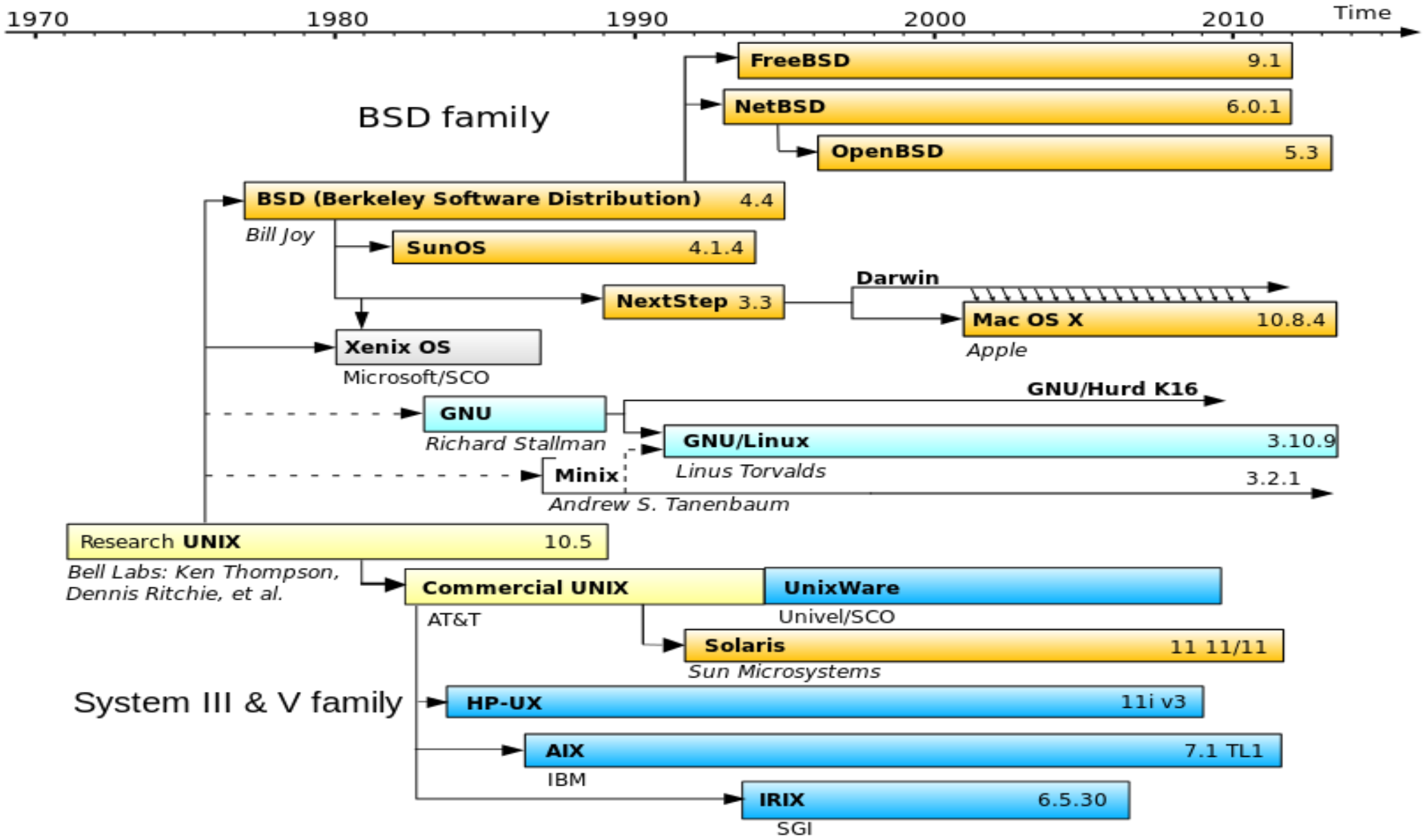
```c
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)  {
   regex_t    preg;
   char       *string = "a very simple simple simple string";
   char       *pattern = "\\(sim[a-z]le\\) \\1";
   char       buff[100];
   int        rc, i;
   size_t     nmatch = 2;
   regmatch_t pmatch[2];

   if (0 != (rc = regcomp(&preg, pattern, 0))) {
      regerror(rc, &preg, buf, 100);
      printf("regcomp() failed: %d(%s)\n", rc, buf);
      exit(EXIT_FAILURE);
   }
```

```
    if (0 != (rc = regexec(&preg, string, nmatch, pmatch, 0))) {
        regerror(rc, &preg, buf, 100);
        printf("Failed to match '%s' with '%s`: %d(%s).\n",
        string, pattern, rc, buf);
        exit(EXIT_FAILURE);
    }
    for(i = 0; i < nmatch; i++) {
      if(pmatch[i].rm_so == -1) break;
      printf("a match \"%.*s\" is found at position %d to %d.\n",
              pmatch[i].rm_eo - pmatch[i].rm_so,
              &string[pmatch[i].rm_so],
              pmatch[i].rm_so, pmatch[i].rm_eo - 1);
    }
    regfree(&preg);
    return 0;
}

    a match "simple simple" is found at position 7 to 19.
    a match "simple" is found at position 7 to 12.
```

CS 288 Intensive Programming in Linux

Professor Ding, Xiaoning

Linux was a Unix-based OS



Brief Intro of Linux

- Created in 1991 by Linus Torvalds
- PC-based operating system
- Based on the existing UNIX operating system
- Released in 1994 as Version 1
- Initially was developed for 80x86 processors (IA32 or i386 architecture processors)
- Today it support various processor
  - AMD, ARM, Power PC, etc.

Linux is popular but not well taught

"You use Linux every day but you don't know it. It's such a fundamental part of our lives. It runs air traffic control, it runs your bank, and it runs nuclear submarines. Your life, money, and death is in Linux's hands …”

--- Jim Zemlin, executive director of the Linux Foundation

- Which courses have taught you a lot of Linux knowledge?

Linux distributions (distro) and Linux OS kernel

- Each distro is a package including the OS and different applications
  - Linux OS kernel.
  - X Window System and GUI interfaces.
  - Different applications
    - vi, Libre Office, Tex, photo editor, etc.
  - Documentations
- Different distros provide different applications
  - Major distros: Ubuntu, Linux Mint, Fedora, MX Linux, …
  - Many many different distros: http://en.wikipedia.org/wiki/List_of_Linux_distributions
  - Distribution suggested in the class: Ubuntu 18.04.3 LTS or above
    - **Ubuntu terminal and Wubi are NOT suggested to use in the class.**

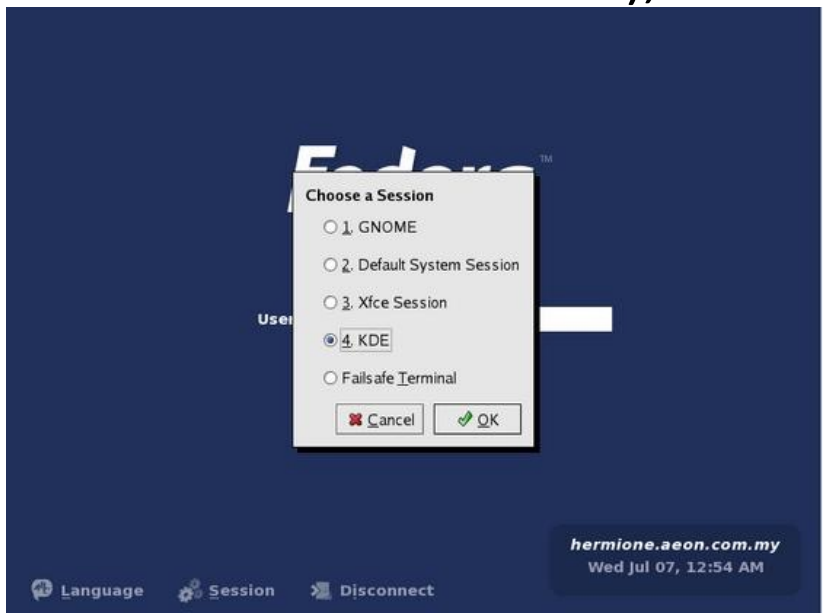
Linux command line interface (CLI)

- CLI is provided by a Linux **shell**
  - The first thing you see after you log into the server.
  - Interact with you and run your commands.

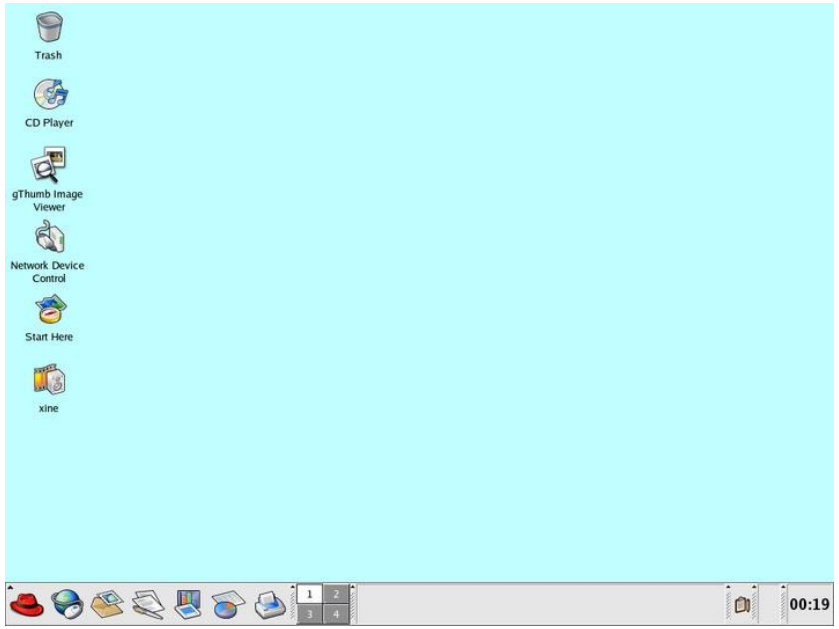
- A variety of Linux shells are available
  - bash, Bourne Shell (sh), C Shell, tcsh, scsh
  - We use **bash** in the course

Linux graphic user interface

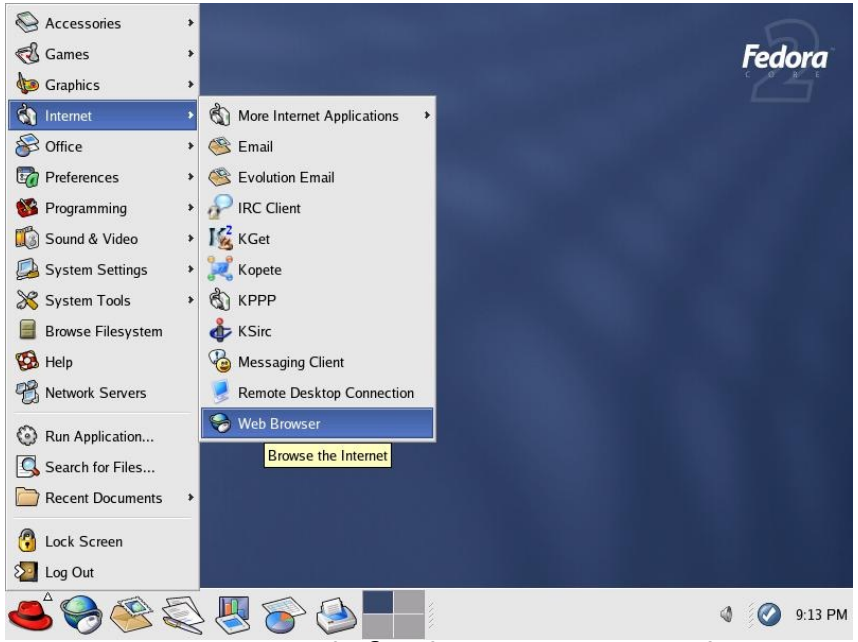
- X Window System or X provides standard mechanisms for displaying device-independent, bit-mapped graphics
- How the actual interface looks or feels depends on the GUI interface
  - KDE (K-desktop Environment), GNOME (GNU Network Object Model Environment), etc.



Choosing KDE at the Login Screen



The KDE Desktop



default GNOME Desktop

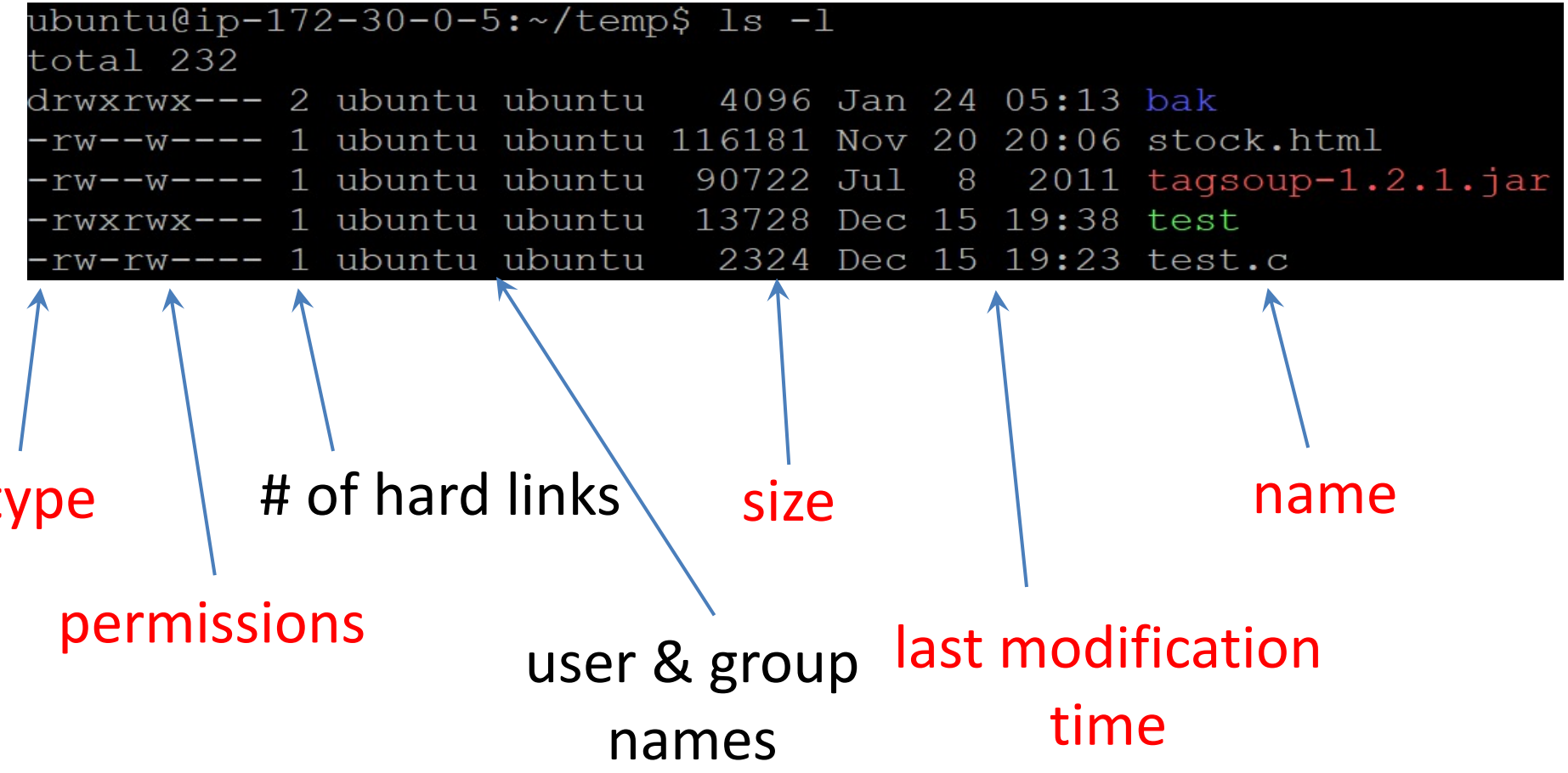
You must know how to use ***help*** command

- help command displays information about shell built-in commands.
- It is also useful when writing a shell script.
- "`help`" or "`help topic`"
  - "`help`" prints the list
  - "`help topic`" prints the help page of the topic.
- what is "`help help`"?
  - Help page of the "help" command.

You must know how to use ***man*** command

- e.g., `man chmod`
- man [section #] topic
  - 1 Executable programs or shell commands (e.g., `man 1 read`)
  - 2 System calls (functions provided by the kernel) (e.g., `man 2 read`)
  - 3 Library calls (functions within program libraries) (e.g., `man 3 read`)
  - 4 Special files (usually found in /dev)
  - 5 File formats and conventions eg /etc/passwd
  - …
- Most manuals can be found online
  - Try to google “man 2 read”

File and directory information

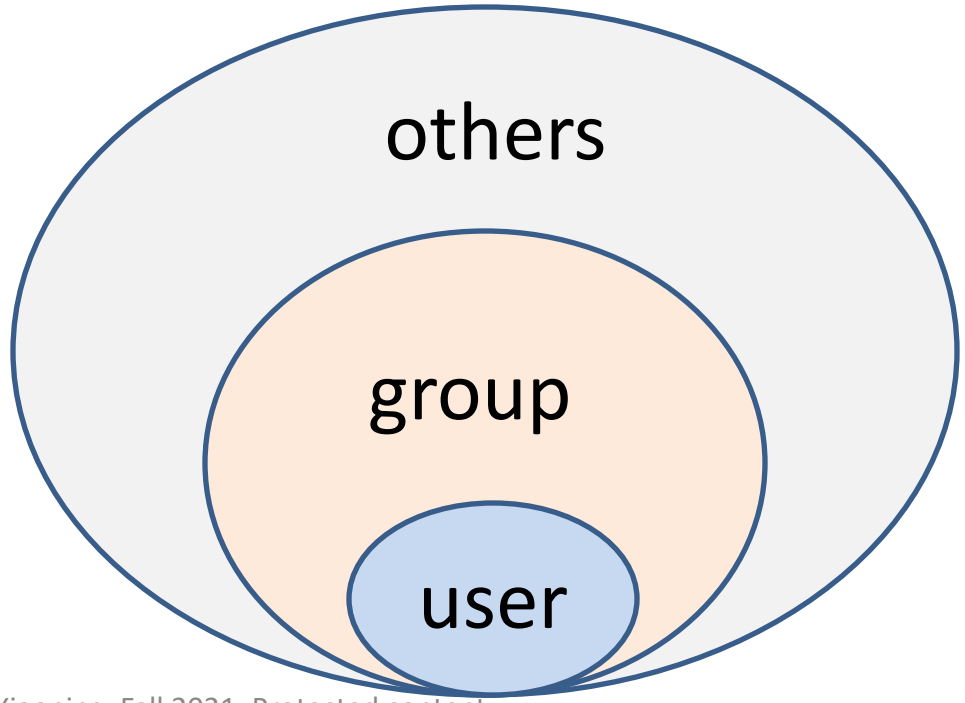


File system permissions in Linux

Permission type	**When used with files**	**When used with directories**
Read	Read a file or copy a file	List the contents of a directory
Write	Write to the file, including deleting the file	Create files
Execute	Execute programs and shell scripts, which are text files containing Linux commands	Modify the file permissions

Linux Permissions

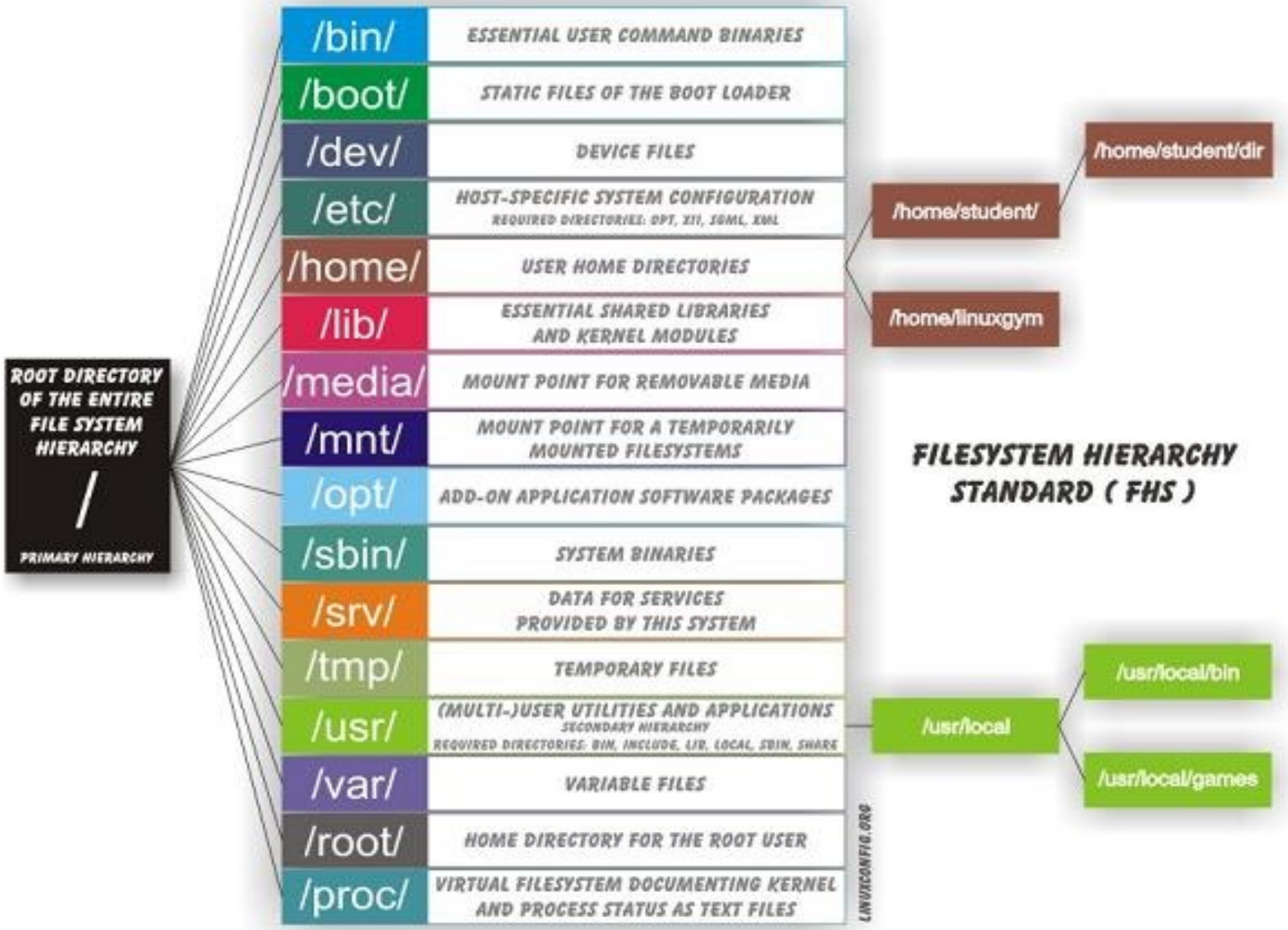
- Permissions are set for user, group, and others
- Each permission is set with a single digit from 0 to 7 based on the combination of permissions
  - read = 4
  - write = 2
  - execute = 1



Using chmod to Set Permissions

Command	**Permissions**		
	Owner	**Group**	**Other**
`chmod 755 myfile`	rwx 111	r-x 101	r-x 101
`chmod 540 myfile`	r-x 101	r— 100	--- 000
`chmod 744 myfile`	rwx	r--	r--

Linux directory hierarchy



File pathname

- Pathname: location and name of the file.
- Remember root is **/**, current directory (current working directory, pwd) is **.**, parent directory is **..**, home directory is **~**.
- Absolution pathnames start from root.

  $ cd /home/CS288/tom
- Relative pathnames are usually relative to the current directory.

  $ cd ./hw/hw1

  $ cd ../hw2

Some commands

- ls – list files

```
$ ls /home/john
```

- cd – change directory

```
$ cd /tmp
```

- pwd – current working directory
- mkdir – create a directory

```
$ mkdir /home/alice/proj1
```

- echo- display a line of text

```
$ echo "hello world"
```

- cat - concatenate files and print on the standard output

```
$ cat a.txt b.txt
```

  - With no file, or when file is -, read standard input.

Some commands

- rmdir – remove a directory
  ```
  $ rmdir /home/tom/tmp
  ```
- rm – remove a file/files
  - remove directories and their contents recursively: `$ rm -r`
- mv – move or rename a file
  ```
  $ mv /etc/ftpaccess /var/ftp/ftpaccess
  ```
- cp – copy a file
  ```
  $ cp var/ftp/ftpaccess /home/amy/
  ```
- date -- current time
- time --- run program and report execution time
  ```
  $ time ls /usr/bin
  ```
- du – file size/space consumption
  ```
  $ du -b /home/amy/hw1
  ```
- locate, whereis, which – find a file
  ```
  $ locate ftpaccess
  $ whereis bash
  ```

Linux utility conventions

- POSIX recommends some conventions for command line arguments.
  - Programs implement them. getopt() makes it easy.
  - man and help describes command synopses following them (e.g., help cd).
  - https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html

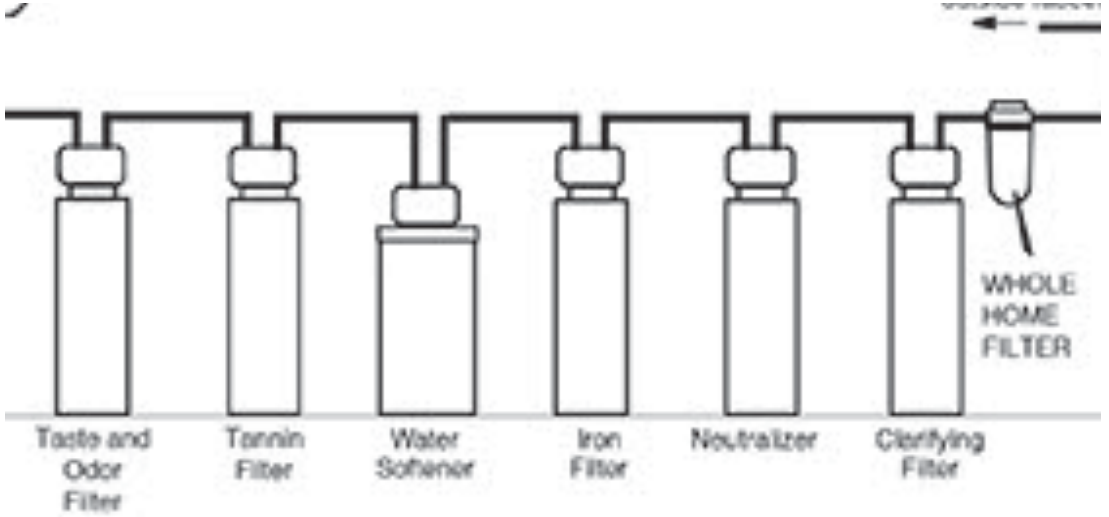
utility_name [-a] [-b] [-c option_arg] [-d|-e] [-f [option_arg]] [operand...]

- Arguments are **options** if they begin with a hyphen delimiter (‘-’).
  - Option names are single alphanumeric characters
  - Multiple options may follow a hyphen delimiter in a single token if the options do not take arguments. Thus, ‘-abc’ is equivalent to ‘-a -b -c’.
  - Certain options require an **option argument**.
- Options typically precede other non-option arguments, named **operands**.
- Arguments separated by '**|**' are **mutually-exclusive**.
- Arguments in '**[**' and '**]**' are **optional.**

A class of Unix tools called ***filters***

- Utilities that read from standard input, transform file contents, and write to standard output
  - Standard input: keyboard, i.e., where your program reads from using scanf()
  - standard output: screen, i.e., where your program prints to using printf().
- Usually work on text files in a line-by-line manner.

```
filter < abc > xyz
```



Redirecting (>, >>, <, |)

- Using > and >> to change standard output to a file
  - e.g., `cmd > filename`
  - *> overwrite, >> append*
- Using < to change standard input to a command.
  - `cmd < file.txt`
- Using | to change standard input and standard output to another program
  - `cmd1 | cmd 2`
- Difference from wring multiple commands on the same line.
  - `cmd1; cmd 2; cmd 3`

cat: the simplest filter

- The cat command copies its input to output unchanged (*identity filter*). When supplied a list of file names, it con**cat**enates them onto stdout.
- Some options:
  - **`-n`** **n**umber output lines (starting from 1)

```
cat file*
ls / cat -n
```

head

- Display the first few lines of a specified file
- Syntax: *head [-n] [filename...]*
  - *-n* - number of lines to display, default is 10
  - *filename...* - list of filenames to display
- When more than one filename is specified, the start of each files listing displays the filename as follows:

```
==>filename<==
```

tail

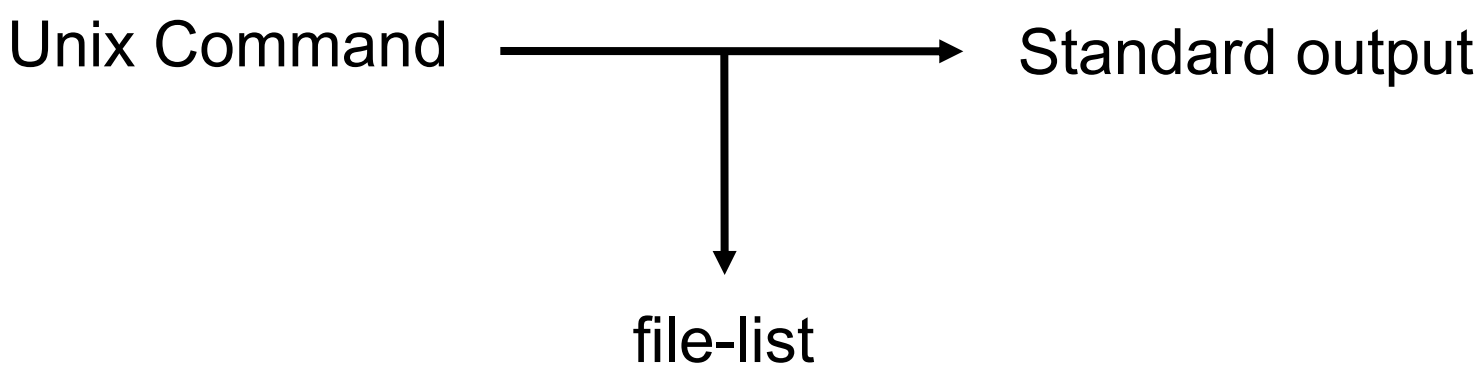
- Displays the last part of a file
- Syntax: *tail -n +|-number [filename]*

  or: *tail +|-number [filename]*
  - *+number* - begins copying at distance *number* from beginning of file
  - *-number* - begins from end of file
  - *if number isn't given, defaults to 10*

head and tail examples

```
head /etc/passwd
head /etc/*.conf
tail +20 /etc/passwd
ls /bin/ | tail -3
head -100 /etc/passwd | tail -5
```

tee



- Copy standard input to standard output and one or more files
  - Captures intermediate results from a filter in the pipeline

tee con’t

- Syntax: *tee [-a] file-list*
  - *-a* - append to output file rather than overwrite, default is to overwrite (replace) the output file
  - *file-list* - one or more file names for capturing output
- Examples

```
ls | head –10 | tee first_10 | tail –5

who | tee user_list | wc
```

Text files as delimited data

Tab separated

```
John      99
Anne      75
Andrew    50
Tim       95
Arun      33
Sowmya    76
```

pipe-separated

```
COMP1011|2252424|Abbot, Andrew John |3727|1|M
COMP2011|2211222|Abdurjh, Saeed |3640|2|M
COMP1011|2250631|Accent, Aac-Ek-Murhg |3640|1|M
COMP1021|2250127|Addison, Blair |3971|1|F
COMP4012|2190705|Allen, David Peter |3645|4|M
COMP4910|2190705|Allen, David Pater |3645|4|M
```

colon-separated

```
root:ZHolHAHZw8As2:0:0:root:/root:/bin/ksh
jas:nJz3ru5a/44Ko:100:100:John Shepherd:/home/jas:/bin/ksh
cs1021:iZ3sO90O5eZY6:101:101:COMP1021:/home/cs1021:/bin/bash
cs2041:rX9KwSSPqkLyA:102:102:COMP2041:/home/cs2041:/bin/csh
cs3311:mLRiCIvmtI9O2:103:103:COMP3311:/home/cs3311:/bin/sh
```

cut: select columns

- The cut command prints selected parts of input lines.
  - can select columns (assumes tab-separated input)
  - can select a range of character positions
- Some options:
  - **-f** *listOfCols*: print only the specified columns (tab-separated) on output
  - **-c** *listOfPos*: print only chars in the specified positions
  - **-d** *c*: use character *c* as the column separator
- Lists are specified as ranges (e.g. 1-5) or comma-separated (e.g. 2,4,5).

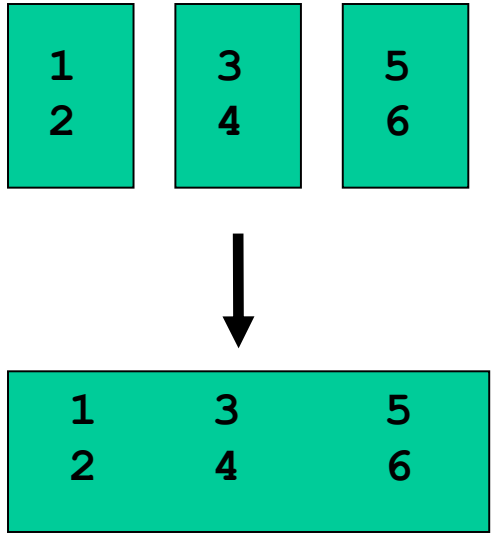
cut examples

```
cut -d':' -f 1 /etc/passwd
cut -d':' -f 1-3  /etc/passwd
cut -d':' -f 1,4  /etc/passwd
cut -d':' -f 4-  /etc/passwd
cut -d':' -f 1-3  data
cut -c 1-4 /etc/passwd
```

Unfortunately, there's no way to refer to "last column" without counting the columns.

paste: join columns

- The paste command displays several text files "in parallel" on output.
- If the inputs are files **a**, **b**, **c**
  - the first line of output is composed of the first lines of **a**, **b**, **c**
  - the second line of output is composed of the second lines of **a**, **b**, **c**



- Lines from each file are separated by a tab character.
- If files are different lengths, output has all lines from longest file, with empty strings for missing lines.

paste example

```
cut -d':' -f 1 /etc/passwd > data1
cut -d':' -f 2 /etc/passwd > data2
cut -d':' -f 3 /etc/passwd > data3
paste data1 data3 data2 > newdata
```

sort: sort lines of a file

- The sort command copies input to output but ensures that the output is arranged in ascending order of lines.
- Very power:
  - can sort based on dictionary order (default) or numeric order.
  - Can specify the key used in the sort in a very flexible way.
    - By default, key is the whole line.

sort: options

- Syntax: *sort [options] [-o filename] [filename(s)]*

-n Numeric order, sort by arithmetic value

-d Dictionary order

-f Ignore case (fold into lower case)

-t Specify delimiter

-k Key

-r Sort in reverse order

-o filename - write output to filename

- Lots of more options…

sort: specifying fields

- Delimiter : **`-t`***d*
- Key: **`-k [start_position][,stop_position]`**
  - start_position and stop_position start from 1 and are inclusive
    - If stop_position is omitted, stop_position is the line's end by default.
  - Formation of each position: f[.c][options]
    - f is a field number, c is a character position in the field
    - options: one or more single-letter ordering options (dictionary? Numeric? Reverse?), which override global ordering options
  - There can be multiple keys.
  - **`-k 3.1,4 -k 1,3 -k4n`**

sort examples

```
sort -t':' -k3nr /etc/passwd
sort -o mydata mydata
sort -t':' -k7 -k3nr /etc/passwd
```

uniq: list UNIQue items

- Remove or report *adjacent* duplicated lines
  - How can you remove duplicated lines that are not adjacent?
- Syntax: *uniq [-cdu] [input-file] [output-file]*

`-d` Write only the duplicated lines

`-u` Write only those lines which are not duplicated

`-c` Supersede the -u and -d options and generate an output report with each line preceded by an occurrence count

- The default output is the union (combination) of *-d* and *-u*

wc: Counting results

- The word count utility, **wc**, counts the number of lines, characters or words
- Options:

  `-l` Count lines

  `-w` Count words

  `-c` Count characters
- Default: count lines, words and chars

wc and uniq Examples

- `who | sort | uniq -d`
- `wc my_essay`
- `who | wc`
- `sort file | uniq | wc -l`
- `sort file | uniq -d | wc -l`
- `sort file | uniq -u | wc -l`

tr: translate or delete characters

- tr [OPTION]... SET1 [SET2]

text transformations, e.g., uppercase to lowercase, squeezing repeating characters, deleting specific characters and basic find and replace.

```
echo "url_that_I_have" | tr "_" "-"
url-that-I-have
```

-c first complement SET1

-d delete characters in SET1, do not translate

-s squeeze-repeats

tr: translate or delete characters

```
echo abcdefghijklmnop | tr -c 'a' 'z'
azzzzzzzzzzzzzzz

echo 'Phone: 01234 567890' | tr -cd '[:digit:]'
01234567890

echo 'clean this up' | tr -d 'up'
clean this

echo 'too many spaces here' | tr -s '[:space:]'
too many spaces here

echo Bad\ File\ nAme.txt | tr -d '\\' | tr ' ' '_' | tr
  '[:upper:]' '[:lower:]'
bad_file_name
```

Tools you need to write programs in Linux

- Editor: vi/vim
- Compiler: gcc
- Debugger: gdb

The following three slides give you brief intro.

Though we may not into details in the class, reading through the online materials will greatly help you on finishing the assignments later.

Vi editor (additional materials in canvas)

- Why vi, fast and easy
- Basic modes- edit and command,
  - ‘esc’ for command mode
  - ‘i, a’ for edit mode (insert or append mode)
- Other commands using colon- :q,:w,:q!,:e
  - :q for quit, :w for write, :q! quit without save
  - :e open another file for editing, :wq write and quit
- Searching using ‘/’
  - In command mode use ‘/’ then write the word you want to search
  - ‘n’ for forward search, ‘N’ for backward search
- Search and replace
  - :%s/this/that - will search string “this” and replace with “that”
  - :%s/this/that/gc --- search and replace interactively
- Advanced vi – **vim**(vi improve) and **gvim**(gnu vim)

Write and run a c program in Linux (additional materials in canvas)

- Write your c code:

```
vi  myprog.c
```

- Compile it using gcc

```
gcc myprog.c -o myprog
```

- run it

```
./myprog
```

Debug a program in Linux using gdb (Additional materials in canvas)

- Must compile the program in a special way (to include debugging information)

```
gcc  -g myprog.c -o myprog
```

or

```
gcc  -ggdb myprog.c -o myprog
```

- Run the program in gdb (the debugger)

```
gdb ./myprog
```

CS 288 Intensive Programming in Linux

Professor Ding, Xiaoning

The shell of Linux

- Different linux shells: Bourne shell (sh), C shell (csh), Korn shell (ksh), TC shell (tcsh), Bourne Again shell (bash).
- Bash: the most popular Linux shell
  - It is a command line interface. We used it to type in and run commends.
  - It is a scripting language. It interprets and runs scripts. We will write bash scripts.
- Shell scripting uses the shell's facilities and existing software tools as building blocks to automate a lot of tasks.
  - Shell facilities: if, for loops, arrays, some built-in commands in shell (e.g., echo).
  - Existing software tools: grep, tr, uniq, …., any other executable files (binary and scripts).
  - Do not need to type in a lot of commands repeatedly.
  - Do not need to build programs from scratch (e.g., instructions).

The first bash program

- Create a script and save the script
  - The first line (Shebang) tells Linux to use the bash interpreter to run this script.
  - Note # also starts the comments. But first line is special.
- make the file executable using `chmod`.
- Run the script.
- Revise the script if it does not run correctly.

```
$vi hello.sh

 #!/bin/bash
 echo hello

$chmod 700 hello.sh
$./hello.sh
  hello

$vi hello.sh

 #!/bin/bash
 echo Hello
```

Including multiple commands in a script

```
$ mkdir trash
$ mv * trash
```

Using commands to create a directory and copy all files into that directory before removing them.

```
$ vi trash.sh

  #!/bin/bash
  mkdir trash
  mv * trash

$ ./trash.sh
```

Instead of having to type all the commands interactively on the shell, write a script

Bash scripts view all the data as texts/strings.

- When a text contains space, tab, newline, the text must be enclosed in either single or double quotes.

```
$ cat my file1.txt    #print out files "my" and "file1.txt"?
$ cat "my file1.txt" #print out file "my file1.txt"
```

- When a text contains special characters, to be safe, the text must be enclosed in either single or double quotes.
  - The parts with special characters may be translated and replaced (see "expansions"), and new text may contain space/tab/newline.

Variables

- Variable values are always stored as strings
  - Introduce later: How to convert variables to numbers for calculations?
- No need to declare a variable
  - assigning a value to a variable creates it.
- Value extracted using $
  - Use { } when necessary

```
$ cat variable.sh
#!/bin/bash
STR="Hello World!"
echo $STR
STR2=Hello
echo $STR2
echo ${STR}2
$ ./variable.sh
Hello World!
Hello
Hello World!2
```

Single and double quotes

When assigning character data containing spaces or special characters, the data must be enclosed in either single or double quotes.

Using double quotes to show a string of characters will allow any variables in the quotes to be resolved.

```
#!/bin/bash
var="test string"
newvar="Value of var is $var"
echo $newvar
```

Output: Value of var is test string

Using single quotes to show a string will not allow variable resolution.

```
#!/bin/bash
var='test string'
newvar='Value of var is $var'
echo $newvar
```

Output: Value of var is $var

Single and double quotes

- Quotes marking the beginning and end of a string are not saved in variables

```
#!/bin/bash
var="test string"
#get the first character, will introduce later
echo ${var:0:1} # echo prints letter t not quote
var="\"test string\"" #escape quotes to include them
echo ${var:0:1} # echo prints double quote
```

- Apply quotes properly when the string in a variable is retrieved and there exits space character(s) in the string.
  - Without quotes, space characters break one string into multiple strings.

```
#!/bin/bash
var="my file.txt"  #a space character in file name
cat $var           #cannot find the file
                   #cat: my: No such file or directory
                   #cat: file.txt: No such file or directory
cat "$var"         #print file content correctly
```

Scope of a variable

By default, all variables are global, even if declared inside a function.

- Can be accessed from anywhere in the script regardless of the scope.
- *Inaccessible* from outside of the script
- *Inaccessible* in other scripts run by the script defining the variable

```
$ cat a.sh
#!/bin/bash
a=hello
echo $a

$ ./a.sh
hello
$ echo $a

$
```

nothing is printed out

What if we want to make *b.sh* print out *"hello"*

```
$ cat a.sh
#!/bin/bash
a=hello
./b.sh

$ cat ./b.sh
#!/bin/bash
echo $a

$ ./a.sh

$
```

nothing is printed out

Environment variables and export command

```
$ cat a.sh
#!/bin/bash
export a=hello
./b.sh

$ cat ./b.sh
#!/bin/bash
echo $a

$ ./a.sh
hello
$
```

The ***export*** command makes a variable an **environment variable**, so it will be accessible from "children" scripts.

If a “child” script modifies an environment variable, it will NOT modify the parent’s original value.

```
$ cat ./a.sh
#!/bin/bash
export a=hello
./b.sh
echo $a

$ cat ./b.sh
#!/bin/bash
a=bye

$ ./a.sh
hello
$
```

Some common environment variables

- Created by the system for saving some system settings
- Can be found with the ***env*** command.
- Accessible in command line interface and any shell scripts.

```
$ echo $SHELL
/bin/bash
$ echo $PATH
/usr/X11R6/bin:/usr/local/bin:/bin:/usr/bin
$ cat a.sh
#!/bin/bash
echo $HOME
$ ./a.sh
/home/fall2020/tom
```

Some common environment variables

- ?: **exit status of previous command**
- LOGNAME, USER: contains the user name
- RANDOM: random number generator
- SECONDS: seconds from the beginning of the execution
- PS1: sequence of characters shown before the prompt

\t hour	\d date	\w current directory
\W last part of the current directory	\u user name	\$ prompt character

Example:

```
$ PS1='hi \u *$'
hi userid*$ _
```

Read command

The read command allows you to prompt for input and store it in a variable.

```
#!/bin/bash
echo -n "Enter pathname of file to backup: "
read file_pathname
cp $file_pathname /home/tom/backup/
```

The script reads a pathname into variable ***file_pathname***, and copies the corresponding file into the backup directory.

Expansions: a few ways to operate texts

- Bash may perform a few types of expansions to commands before executing them.
- Replace special expressions with texts
  - variable expansion
  - brace expansion
  - tilde expansion
  - command substitution
  - arithmetic expansion
  - filename expansion

Variable expansion

${var} : string saved in var

${#var} gives the string length

${var:position} extracts sub-string from $string at $position

${var:position:length} extracts a sub-string of $length from $position

```
$ st=0123456789
$ echo ${#st}
     10
$ echo ${st:6}
     6789
$ echo ${st:6:2}
     67
```

Brace expansion and tilde expansion

Brace expansion expands a sequence expression or a comma separated list of items inside curly braces "{}"

- Brace expansion is performed before any other expansions, and any characters special to other expansions are preserved in the result.
- "${" for variable expansion is not considered eligible for brace expansion

```
$ echo  a{d,c,b}e
   ade ace abe
$ echo  a{0..3}b
   a0b a1b a2b a3b
$ mkdir home_{tom,berry, jim}
$ ls
home_berry  home_jim  home_tom
```

Tilt expansion replaces an unquoted tilde character "~" at the beginning of a word with pathname of home directory

~ : home directory of current user ($HOME)
~/foo : foo subdirectory under the home
~fred/foo : the subdirectory foo of the home directory of the user fred

Command substitution: saving the output of a command into a variable

```
$ LIST=`ls`
$ echo $LIST
hello.sh read.sh

$ PS1="`pwd`>"
/home/userid/work> _
```

command substitution **using backquotes** : `command` (use backquote "`" , not single quote "'").

Command substitution using $ and (): $(command)

```
$ LIST=$(ls)
$ echo $LIST
hello.sh read.sh

$ rm $( find / -name "*.tmp" )

$ cat > backup.sh
#!/bin/bash
BCKUP=/home/$USER/backup-$(date +%F).tgz
tar -czf $BCKUP $HOME
```

Evaluating arithmetic expressions

Translate a string into a numerical expression

- Command substitute and expr: `expr expression` or $(expr expression)
  - e.g., `z=`expr $z + 3``
  - Read manual of command expr
- double parentheses: $((expression))

Arithmetic expansion

```
$ echo "$((123+20))"
143
$ echo "$((123*$VALORE))"
$ echo "$((123*VALORE))"
```

- The let statement: let var=expression

```
$ X=2; let X=10+X*7
$ echo $X
24
```

Arithmetic operators: +, -, /, *, %

```
$ cat arithmetic.sh
#!/bin/bash
echo -n "Enter the first number: "; read x
echo -n "Enter the second number: "; read y
add=$(($x + $y)); sub=$(($x - $y))
mul=$(($x * $y)); div=$(($x / $y))
mod=$(($x % $y));
echo "Sum: $add"
echo "Difference: $sub"
echo "Product: $mul"
echo "Quotient: $div"
echo "Remainder: $mod"
```

filename expansion

Bash scans each word for the characters '*', '?', and '['. If one of these characters appears, then the word is regarded as a pattern, and **replaced with an alphabetically sorted list of filenames matching the pattern**.

- `*` Matches any string, including the null string.
- `?` Matches any single character.
- `[...]` Matches any one of the enclosed characters.

```
$ ls *.pdf
$ ls fig?.pdf
$ ls fig[0-9].pdf
$ ls fig_[abc].pdf
```

```
$ mkdir home{1..3}
$ mkdir home{1,2}{a..c}
$ echo home*
$ echo home*
home1 home1a home1b home1c home2 home2a home2b home2c home3
$ echo home[12345]
home1 home2 home3
$ echo home?[bc]
home1b home1c home2b home2c
```

Spaces and word Splitting

The shell scans the results of variable expansion, command substitution, and arithmetic expansion for word splitting.

- Results from filename expansion are not spitted
- Usually happens when the results are used in command lines, not in assignments
- double quotes prevent world splitting

```
$ echo "Hello      World"
"Hello      World"
$ a="Hello      World"
$ echo ${a}
Hello World
$ echo ${a#}
16
$ echo "${a}"
Hello      World
$b=$a
$ echo ${b#}
16
```

rule of thumb: double-quote every expansion except filename expansion

Conditional statements

```
if COMMANDS
then
    statements
elif COMMANDS
then
    statements
else
    statements
fi
```

```
if COMMANDS; then
    statements
elif COMMANDS; then
    statements
else
    statements
fi
```

```
if COMMANDS; then statements; elif COMMANDS; then statements; else statements; fi
```

- elif (else if) and else sections are optional
- Conditions are exit code ($?) of COMMAND

Conditional statements

```
if [ expression ]; then
    statements
elif [ expression ]; then
    statements
else
    statements
fi
```

- [is a command usually used in if
  - [is another implementation of the traditional test command.
  - [or test is a standard POSIX utility.
  - Implemented in all POSIX shells.
- An expression can compare numbers, strings, check files, combine multiple conditions…
- Put spaces before and after each expression, and around the operators in each expression.

Comparing numbers

-eq compare if two numbers are equal
-ge compare if one number is greater than or equal to a number
-le compare if one number is less than or equal to a number
-ne compare if two numbers are not equal
-gt compare if one number is greater than another number
-lt compare if one number is less than another number

- Examples:

[n1 -eq n2] true if n1 same as n2, else false
[n1 -ge n2] true if n1greater then or equal to n2, else false
[n1 -le n2] true if n1 less then or equal to n2, else false
[n1 -ne n2] true if n1 is not same as n2, else false
[n1 -gt n2] true if n1 greater then n2, else false
[n1 -lt n2] true if n1 less then n2, else false

Examples

```
$ cat number.sh
#!/bin/bash
echo -n "Enter a number 1<x<10:"
read num
if [ $num -lt 10 ]; then
  if [ $num -gt 1 ]; then
    echo "$num*$num=$(($num*$num))"
  else
    echo "Wrong number!"
  fi
else
  echo "Wrong number!"
fi
```

Comparing strings

= compare if two strings are equal

!= compare if two strings are not equal

-n evaluate if string length is greater than zero

-z evaluate if string length is equal to zero

```
$ cat user.sh
#!/bin/bash
echo -n "Enter your login
  name: "
read name
if [ "$name" = "$USER" ];
then
  echo "Hello, $name."
else
  echo "You are not $USER"
fi
```

- Examples:

[s1 = s2] true if s1 same as s2, else false

[s1 != s2] true if s1 not same as s2, else false

[s1] true if s1 is not empty, else false

[-n s1] true if s1 has a length greater then 0, else false

[-z s2] true if s2 has a length of 0, otherwise false

Checking files/directories

-e check if file/path name exists
-d check if path given is a directory
-f check if path given is a file
-r check if read permission is set for file or directory
-s check if a file has a length greater than 0
-w check if write permission is set for a file or directory
-x check if execute permission is set for a file or directory

```
#!/bin/bash
read fname
if [ -f $fname ]; then
  cp $fname .
  echo "Done."
else
  if [ -e $fname ]; then
    echo "Not a file."
  else
    echo "Not exist."
  fi
  exit 1
fi
```

- Examples:

[-d fname] (true if fname is a directory, otherwise false)
[-f fname] (true if fname is a file, otherwise false)
[-e fname] (true if fname exists, otherwise false)
[-s fname] (true if fname length is greater then 0, else false)
[-r fname] (true if fname has the read permission, else false)
[-w fname] (true if fname has the write permission, else false)
[-x fname] (true if fname has the execute permission, else false)

Exercise

Write a shell script which:

- Allows user to type in a file name (e.g., ./myfile.txt)
- checks if the file exists
- if the file exists, make a copy of the file under the same directory. Append a ".bak" to the file name of the copy (e.g., ./myfile.txt.bak).
- If the file does not exist, print out "file does not exist."

Logically operators: AND (-a, &&), OR (-o, ||), NOT (!)

```
#!/bin/bash
echo -n "Enter a number 1<x<10:"
read num
if [ $num -lt 10 ]; then
  if [ $num -gt 1 ]; then
    echo "$num*$num=$(($num*$num))"
  else
    echo "Wrong number!"
  fi
else
  echo "Wrong number!"
fi
```

```
#!/bin/bash
echo -n "Enter a number 1<x<10:"
read num
if [ $num -lt 10 -a $num -gt 1 ]; then
    echo "$num*$num=$(($num*$num))"
else
  echo "Wrong number!"
fi
```

Pay attention to the forms when combining conditions

```
if [ condition1 ] && [ condition2 ]
if [ condition1 -a condition2 ]
if [ condition1 ] || [ condition2 ]
if [ condition1 -o condition2 ]
```

```
#!/bin/bash
echo -n "Enter a number 1<x<10:"
read num
if [ $num -gt 1 ] && [ num -lt 10 ];
  then
  echo "$num*$num=$(($num*$num))"
else
  echo "Wrong number!"
fi
```

Case statement

```
case var in
val1)
     statements;;
val2)
     statements;;
*)
     statements;;
esac
```

- Execute statements based on specific values.
- each set of statements must be ended by a pair of semicolons;
- a *) is used to accept any value not matched with list of values

```
$ cat case.sh
#!/bin/bash
echo -n "Enter a number 1 < x < 10: "
read x
case $x in
          1) echo "Value of x is 1.";;
          2) echo "Value of x is 2.";;
          3) echo "Value of x is 3.";;
          4) echo "Value of x is 4.";;
          5) echo "Value of x is 5.";;
          6) echo "Value of x is 6.";;
          7) echo "Value of x is 7.";;
          8) echo "Value of x is 8.";;
          9) echo "Value of x is 9.";;
          0 | 10) echo "wrong number.";;
          *) echo "Unrecognized value.";;
esac
```

for loop

```
for VARIABLE in PARAM1 PARAM2 PARAM3
do
  statements
done
```

- for loop executes for each param in the list.
- The VARIABLE is initialized with a param value which can be accessed in inside the for loop scope
- Param can be any number, string etc.

```
#!/bin/bash
let sum=0
for num in 1 2 3 4 5
do
   let "sum = $sum + $num"
done
echo $sum
```

```
#!/bin/bash
for x in paper pencil pen
do
    echo "The value of variable x is: $x"
    sleep 1
done

for x in paper "a pencil"  "two pens"
do
    echo "The value of variable x is: $x"
    sleep 1
done
```

Example: Changes all filenames to lowercase

```
#!/bin/bash
# for all files in a directory.
for filename in `ls ./*`
do
  # filename in lowercase.
  n=`echo $filename | tr A-Z a-z`
  # Rename only files not already lowercase.
  if [ "$filename" != "$n" ]; then
              mv $filename $n
      fi
done
exit 0
```

Using range in a for loop

```
#!/bin/bash
for value in {1..5}
do
    echo $value
done
for value in {10..0..2}
do
    echo $value
done
```

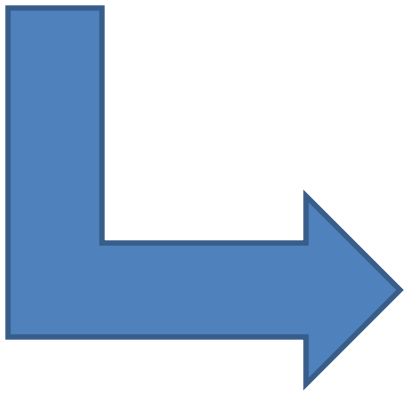
Range: *{start..end}*, or *{start..end..step}*

- *Start* and *end* determine the direction (counts up/down)
- *Step* determines the increment (no need to be negative when counting down).
- Brace expansion is performed before any other expansions, and any characters special to other expansions are preserved in the result.
  - "${" for variable expansion is not considered eligible for brace expansion
  - Use **seq** instead.

seq FIRST INCREMENT LAST

```
#!/bin/bash
begin=1
end=5
for value in {${begin}..${end}}
do
    echo $value
done
```

Invalid



```
#!/bin/bash
begin=1
end=5
for value in `seq ${begin} ${end}`
do
    echo $value
done
```

Using range in a for loop

```
#!/bin/bash
for value in {1..5}
do
    echo $value
done
for value in {10..0..2}
do
    echo $value
done
```

Range: *{start..end}*, or *{start..end..step}*

- *Start* and *end* determine the direction (counts up/down)
- *Step* determines the increment (no need to be negative when counting down).
- Brace expansion is performed before any other expansions, and any characters special to other expansions are preserved in the result.
- "${" for variable expansion is not considered eligible for brace expansion

for loop in C style

First, the arithmetic expression EXPR1 is evaluated.

EXPR2 is then evaluated repeatedly until it evaluates to 0.

Each time EXPR2 is evaluates to a non-zero value, statements are executed and EXPR3 is evaluated.

```
for (( EXPR1; EXPR2; EXPR3 ));
do
    statements
done
```

```
$ cat ./mysum.sh
#!/bin/bash
echo -n "Enter a number: "
read x
sum=0
for ((i=1;i<=x;i=i+1)) ; do
    sum=$(($sum+$i))
done
echo "Sum of 1...$x is: $sum"
```

While structure

Execute a set of commands while a specified condition is true.

- The loop terminates as soon as the condition becomes false.
- If condition never becomes false, loop will never exit.

```
while [ some_test ]
do
  statements
done
```

```
$ cat while.sh
#!/bin/bash
echo -n "Enter a number: "
read x
sum=0; i=1
while [ $i -le $x ]; do
  let "sum = $sum + $i"
  let "i = $i + 1"
done
echo "sum of 1...$x is: $sum"
```

Menu

```
#!/bin/bash
clear ; loop=y
while [ "$loop" = y ] ;
do
  echo "Menu";  echo "===="
  echo "D: print the date"
  echo "W: print the users who are currently log on."
  echo "P: print the working directory"
  echo "Q: quit.";   echo
  read choice
  case $choice in
    D | d) date ;;
    W | w) who ;;
    P | p) pwd ;;
    Q | q) loop=n ;;
    *) echo "Illegal choice." ;;
  esac
  echo
done
```

Until structure: loops until the condition is true

```
until [ some_test ]
do
   statements
done
```

```
$ cat countdown.sh
#!/bin/bash
echo "Enter a number: "
read x
echo "Count down"
until [ "$x" -le 0 ]; do
    echo $x
    x=$(($x -1))
    sleep 1
done
```

Continue: skip the remaining part in current iteration and jump to the next iteration

```
$ cat continue.sh
#!/bin/bash
echo "Print numbers 1 to 20 (but not 3 and 11)"
a=0
while [ $a -le 19 ]; do
    a=$(($a+1))
    if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]; then
     continue
    fi
    echo -n "$a "
done
```

Break terminates the loop

```
$ cat break.sh
#!/bin/bash
echo "Print numbers 1 through 20, but nothing after 12"
a=0
while [ $a -le 19 ];  do
      a=$(($a+1))
      if [ "$a" -gt 12 ]; then
            break
      fi
      echo -n "$a "
done
echo
```

Using arrays

- Bash does not offer lists, tuples, etc. **Just arrays**.
- bash has two types of arrays: one-dimensional indexed arrays and associative arrays
- An array is a variable containing multiple values.
- No maximum limit to the size of an array.
- No requirement that member variables be indexed or assigned contiguously

Index arrays

- Arrays are **zero-based**: the first element is indexed with the number 0.
- Creating an array
  - First way:

```
#3 elements
pet=("a dog" "a cat" fish)
#2 elements
pet=([2]=fish [0]="a dog")
```

  - Second way:

```
pet[0]="a dog"
pet[1]="a cat"
pet[2]=fish
```

- Third way:

```
#brace expansion
pet=(a{1..3})
#(a1 a2 a3)
#filename expansion
files=(./*)
```

Using index arrays

- To extract a value: ${arrayname[i]}
  $ echo ${pet[0]}
  a dog
- extract all the elements:
  ${arrayname[*]}, ${arrayname[@]}
- extract the count of the elements: ${#arrayname[@]}
- Extract all the indices that have been assigned: ${!arrayname[@]}
- extracts sub-array at $position: ${arrayname[@]:position}
- extracts $length elements from $position:
  ${arrayname[@]:position:length}
- Search and replace an element: ${arrayname[@]:OldText:NewText}
- Add new elements: arrayname+=(new_ele1 new_ele2)
- Delete an element: unset arrayname[index]

```
pet=("a dog" "a cat" fish)
echo $pet      # a dog
echo $pet[1]   # a dog[1]
echo ${pet[1]}  # a cat
```

Associative arrays

- The index can be any arbitrary string.
- Creation: must be declared with ***typeset -A*** or ***declare –A***
- Individual element can be accessed using the index string.
- features of indexed arrays are available to associative arrays.

```
#!/bin/bash
declare -A shade
shade[apple]="dark red"
shade[banana]="bright yellow"
#add a new element
shade+=([grape]=purple)
for i in apple banana grape
do
  echo ${shade[$i]}
done
for i in ${!shade[@]}; do
  echo $i ${shade[$i]}
done
#remove an element
unset shade[apple]
```

Example: Picking a random poker card (random suit & random rank)

```
#!/bin/bash
Suits="Clubs Diamonds Hearts Spades"
Ranks="2 3 4 5 6 7 8 9 10 Jack Queen King Ace"

# Read into array variable.
suit=($Suits)
rank=($Ranks)

# Count how many elements.
num_suits=${#suit[*]}
num_ranks=${#rank[*]}
echo -n "${rank[$(($RANDOM%num_ranks))]} of "
echo ${suit[$(($RANDOM%num_suits))]}
```

${arrayname[*]} and ${arrayname[@]}

- ${arrayname[*]} and ${arrayname[@]} are all the words in all the elements (as if elements are merged and divided into words)
  - `pet=("a dog" "a cat" fish)`
  - ${pet[*]} and ${pet[@]} get the contents in all elements and put them together: `a dog a cat fish`
- "${arrayname[*]}" : a single string containing all the words from all the elements (all words in the same pair of quotes)
  - "${pet[*]} " gets the contents in all elements, puts them together and inside double quotes: `"a dog a cat fish"`
- "${arrayname[@]}" : a string for each element (each element has a pair of quotes)
  - For each element, "${pet[@]} " gets its content and puts it inside double quotes : `"a dog" "a cat" "fish"`

Using array in a loop

```
$ cat arrayele.sh
#!/bin/bash
array=(one two three four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in ${array[*]}
do
    echo $item
done
$ ./arrayele.sh
one
two
three
four
```

```
$ cat arrayele.sh
#!/bin/bash
array=(one "two three" four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in ${array[*]}
do
    echo $item
done
$ ./arrayele.sh
one
two
three
four
```

Using array in a loop

```
$ cat arrayele.sh
#!/bin/bash
array=(one two three four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in ${array[@]}
do
    echo $item
done
$ ./arrayele.sh
one
two
three
four
```

```
$ cat arrayele.sh
#!/bin/bash
array=(one "two three" four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in ${array[@]}
do
    echo $item
done
$ ./arrayele.sh
one
two
three
four
```

Using array in a loop

```
$ cat arrayele.sh
#!/bin/bash
array=(one two three four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in "${array[*]}"
do
    echo $item
done
$ ./arrayele.sh
One two three four
```

```
$ cat arrayele.sh
#!/bin/bash
array=(one "two three" four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in "${array[*]}"
do
    echo $item
done
$ ./arrayele.sh
one two three four
```

Using array in a loop

```
$ cat arrayele.sh
#!/bin/bash
array=(one two three four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in "${array[@]}"
do
    echo $item
done
$ ./arrayele.sh
one
two
three
four
```

```
$ cat arrayele.sh
#!/bin/bash
array=(one "two three" four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in "${array[@]}"
do
    echo $item
done
$ ./arrayele.sh
one
two three
four
```

Example: Changes all filenames to lowercase

```
#!/bin/bash
#filename expansion into an array
files=(*)
for filename in "${files[@]}"
do
   # filename in lowercase.
   n=`echo $filename | tr A-Z a-z`
   # Rename only files not already lowercase.
   if [ "$filename" != "$n" ]; then
                mv $filename $n
        fi
done
exit 0
```

Shell parameters

- Positional parameters are assigned from arguments when a script is invoked.
- N-th positional parameter is ${N} or $N when N is single digit.
  - $1 : first command line argument
  - $0 : the name of the script
- Other special parameters
  - $# the number of parameters passed
  - $* all positional parameters except $0
  - $@ all positional parameters except $0

```
$ cat sparameters.sh
#!/bin/bash
echo "$#; $0; $1; $2; $*; $@"
$ sparameters.sh arg1 "arg #2"
2; ./sparameters.sh; arg1; arg #2; arg1 arg #2; arg1 arg #2
```

Example: Trash

```
$ cat trash.sh
#!/bin/bash
if [ $# -eq 1 ]; then
  if [ ! -d "$HOME/trash" ]; then
          mkdir "$HOME/trash"
  fi
  mv $1 "$HOME/trash"
else
  echo "Use: $0 filename"
  exit 1
fi
```

Difference between $* and $@

```
$ cat args.sh
#!/bin/bash
echo "Arg list as a single string"
index=1;
for arg in "$*" ; do
        echo "Arg $index = $arg"
        let "index+=1"
done

echo; index=1;
echo "Arg list as separate strings"
for arg in "$@" ; do
        echo "Arg $index = $arg"
        let "index+=1"
done
```

```
$ ./args.sh arg1 "arg2 arg3" arg4
Arg list as a single string
Arg 1 = arg1 arg2 arg3 arg4

Arg list as separate strings
Arg 1 = arg1
Arg 2 = arg2 arg3
Arg 3 = arg4
```

Indirection with !

What does the following script print out?

```
#!/bin/bash
for ((i=0;i<=$#;i++)); do
    echo $i
done
```

What is $i?
Is it $0, $1, …?

```
#!/bin/bash
for
((i=0;i<=$#;i++)); do
    echo ${!i}
done
```

Iterate arguments

When the list part in a for loop is left off, var is set to each argument ($1, $2, $3,…)

```
$ cat for1.sh
#!/bin/bash
for x
do
     echo "The value of variable x is: $x"
     sleep 1
done
$ for1.sh arg1 arg2
The value of variable x is: arg1
The value of variable x is: arg2
```

Functions

- Functions are like mini-scripts. They can
  - accept parameters ($1, $2, …)
  - create variables only known within the function
  - return values to the calling shell (not caller).
- A function is called by its name

```
function name
{
   commands;
   return x;
}

function name()
{
   commands;
   return;
}
```

```
$ cat function.sh
#!/bin/bash
function check()
{
if [ -e "/home/$1" ]; then
   return 0
else
   return 1
fi
}
echo "Enter a file name:"
read x
if check $x
then
   echo "$x exists !"
else
   echo "$x not exists!"
fi.
```

Variables created in a function and local variables

- In contrast to C, a Bash variable declared inside a function is local ONLY IF declared as such.

```
local var_name
```

- If not declared as local, variables are global by default.
- Before a function is called, all variables declared within the function are invisible outside the body of the function, not just those explicitly declared as local.

```
$ cat ./var_in_func.sh
#!/bin/bash
func ()
{
  local loc_var=23   # Declared as local variable.
  echo "\"loc_var\" in function = $loc_var"
  global_var=999
  echo "\"global_var\" in function = $global_var"
}

func
# $loc_var not visible globally.
echo "\"loc_var\" outside function = $loc_var"
# $global_var is visible globally.
echo "\"global_var\" outside function = $global_var"
$ ./var_in_func.sh
loc_var outside function =
global_var outside function = 999
```

```
$ cat ./var_in_func.sh
#!/bin/bash
func ()
{
global_var=37
}

# $global_var is not visible here. "func" not called,
echo "global_var = $global_var"
func
# $global_var has been set by function call.
echo "global_var = $global_var"
$ ./var_in_func.sh
global_var =
global_var = 37
```

Return a value from Bash functions

Using a global variable

```
#!/bin/bash
function F1()
{
  retval='Like programming'
}

retval='Hate programming'
echo $retval
F1
echo $retval
```

Using function command

```
#!/bin/bash
function F2()
{
  local retval='BASH Func'
  echo "$retval"
}

getval=$(F2)
echo $getval
```

Return a value from Bash functions using $?

```
#!/bin/bash -x

function factorial()
{
   if (( $1 < 2 ))
   then
     return 1
   else
     factorial $(( $1 - 1 ))
     result=$(( $1 * $? ))
     return ${result}
   fi
}

factorial $1
echo $?
```

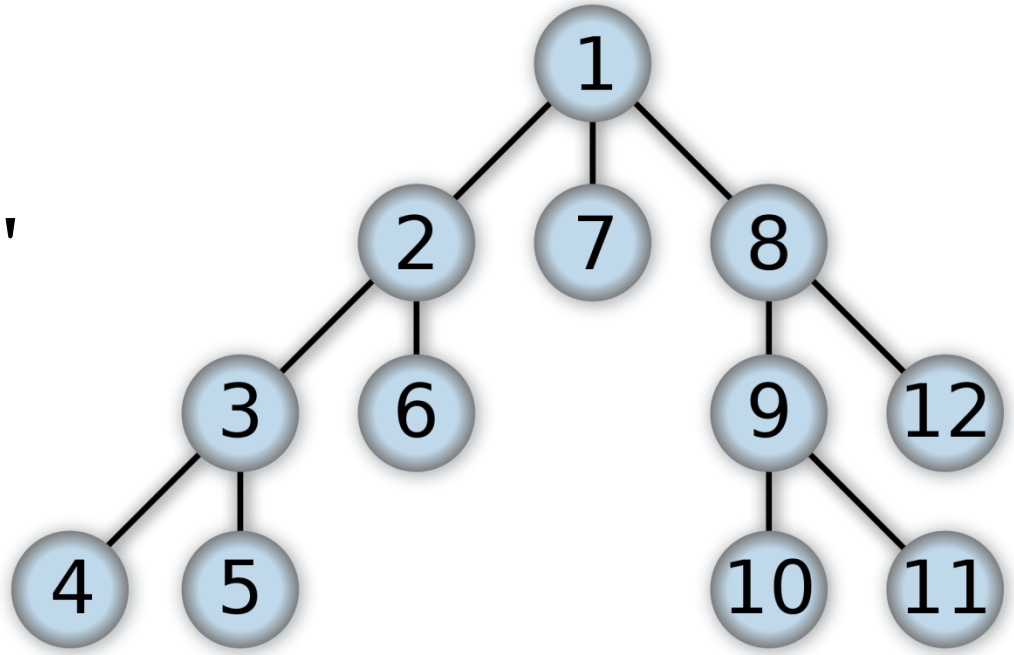
- Problem: $? must be an integer in the 0 - 255 range
- The code on the left works for 1, 2, …, 5, but not 6.

Example: factorial of a number

```
#!/bin/bash

function factorial()
{
   if (( $1 < 2 ))
   then
     echo 1
   else
     echo $(( $1 * $(factorial $(( $1 - 1 ))) ))
   fi
}

factorial $1
```

Example: traverse a directory (**depth-first**)

```
#!/bin/bash
traverse() {
echo $1
entries=("$1"/*)
for entry in "${entries[@]}"
do
    traverse "$entry"
done
}

traverse "$1"
```



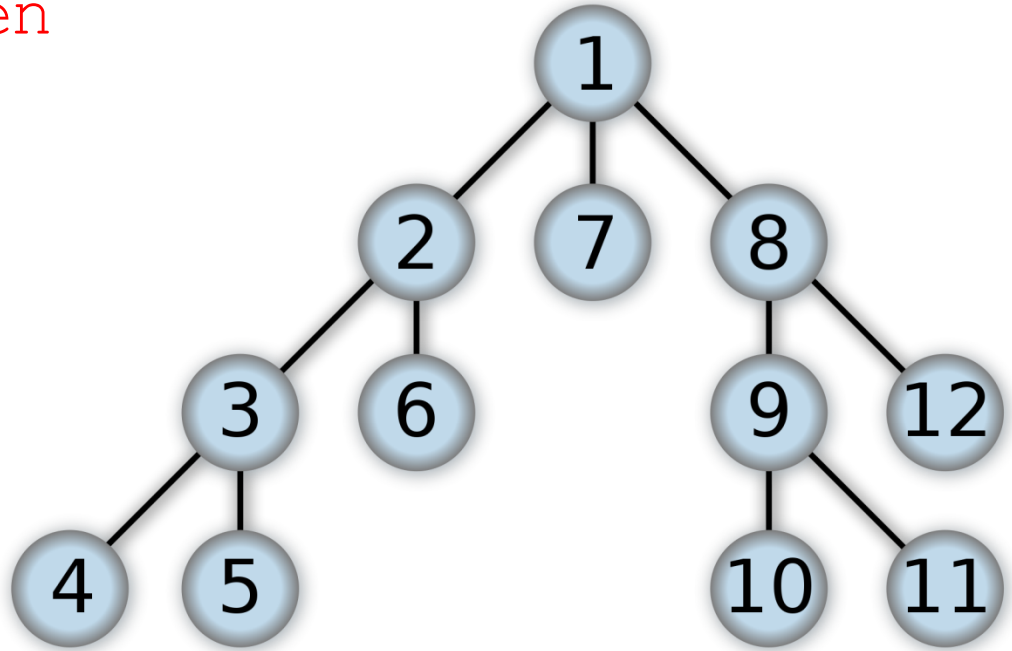
Can this code traverse correctly?

Example: traverse a directory (depth-first)

```
#!/bin/bash
traverse() {
echo $1
if [ ! -d "$1" ]; then
    return
fi
if [ `ls "$1" | wc -l` -eq 0 ]; then
   return
fi
local entries=("$1"/*)
local entry
for entry in "${entries[@]}"
do
    traverse "$entry"
done
}

traverse "$1"
```

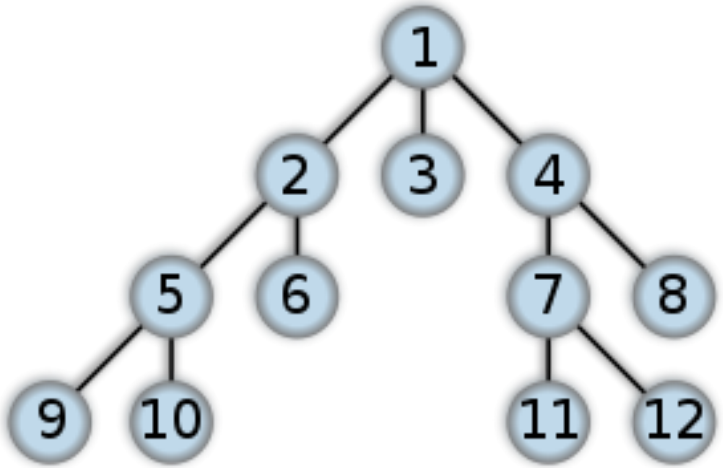
What if we remove the quotes?

What if we remove "local"?



Example: traverse a directory (**breadth-first**)

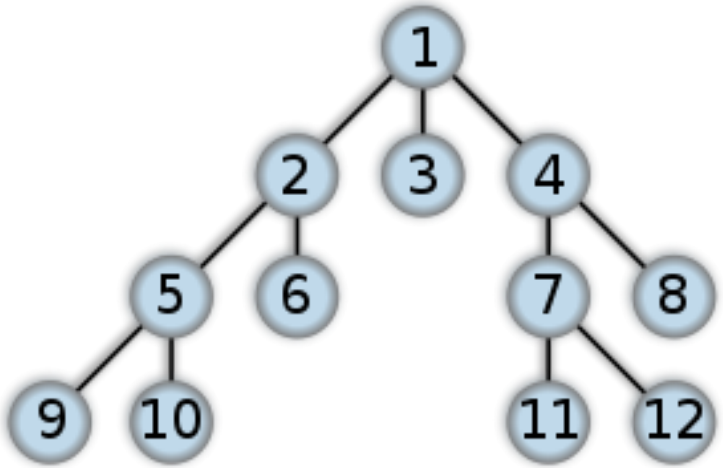
```
#!/bin/bash
function traverse(){
  if [ ${#queue[@]} -eq 0 ]; then return; fi
  echo ${queue[0]}
  if [ -d "${queue[0]}" ] && [ `ls "${queue[0]}" | wc -l` -ne 0 ]
  then
    entries=("${queue[0]}"/*)
    #merge two arrays
    queue=("${queue[@]}" "${entries[@]}")
  fi
  queue=("${queue[@]:1}") #remove elem #0
  traverse
}

queue[0]="$1"
traverse
```



Example: traverse a directory (**breadth-first**)

recursion ➔ loop

```
#!/bin/bash
queue[0]="$1"
while [ ${#queue[@]} -ne 0 ]; do
   echo ${queue[0]}
   if [ -d "${queue[0]}" ] && [ `ls "${queue[0]}" | wc -l` -ne 0 ]
   then
      entries=("${queue[0]}"/*)
      #merge two arrays
      queue=("${queue[@]}" "${entries[@]}")
   fi
   queue=("${queue[@]:1}") #remove elem #0
done
```



Debugging

Two debug options on the first script line:

#!/bin/bash -v or #!/bin/bash -x

-v : displays each line of the script as typed before execution

-x : displays each line of the script with variable substitution and before execution

```
$ cat for3.sh
#!/bin/bash -x
echo -n "Enter a number: "; read x
sum=0
for ((i=0;i<=x;i=i+1)); do
  sum=$(($sum + $i))
done
echo "the sum of 1...$x is: $sum"
```

```
$ ./for3.sh
+ echo -n 'Enter a number: '
Enter a number: + read x
2
+ sum=0
+ (( i=0 ))
+ (( i<=x ))
+ sum=0
+ (( i=i+1 ))
+ (( i<=x ))
+ sum=1
+ (( i=i+1 ))
+ (( i<=x ))
+ sum=3
+ (( i=i+1 ))
+ (( i<=x ))
+ echo 'the sum of 1...2 is: 3'
the sum of 1...2 is: 3
```

Programming or scripting?

- Programming languages are faster
  - source code is compiled into an executable. One time translation effort, and a lot of optimization during compilation.
  - script is not compiled into an executable. An interpreter reads, interprets, and executes the statements in a script. A lot of format conversions. Some inconvenience (e.g., lack of types and formats).
- Programming languages are usually more flexible and powerful: more facilities and various libraries.
- Scripts: fast development, easy to change/improve.
  - do not need to build programs from scratch (e.g., instructions).
- Common practice --- combining both: compiled parts for speed (e.g., building blocks), script parts for flexibility.