CS3157: Advanced Programming

Lecture #15

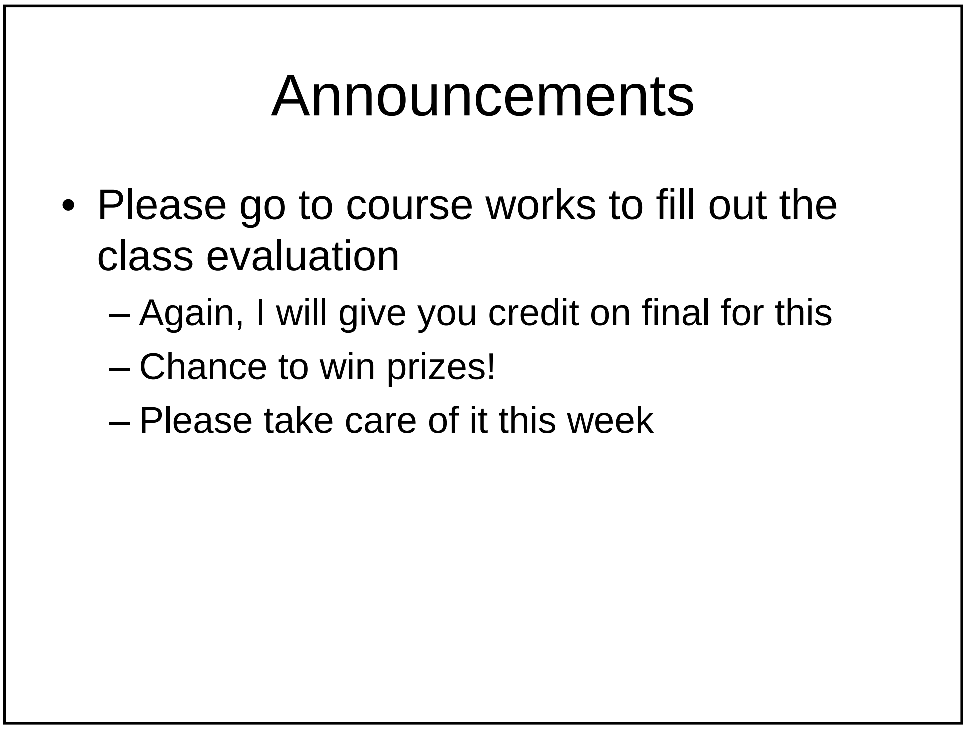
Apr 24

Shlomo Hershkop

shlomo@cs.columbia.edu

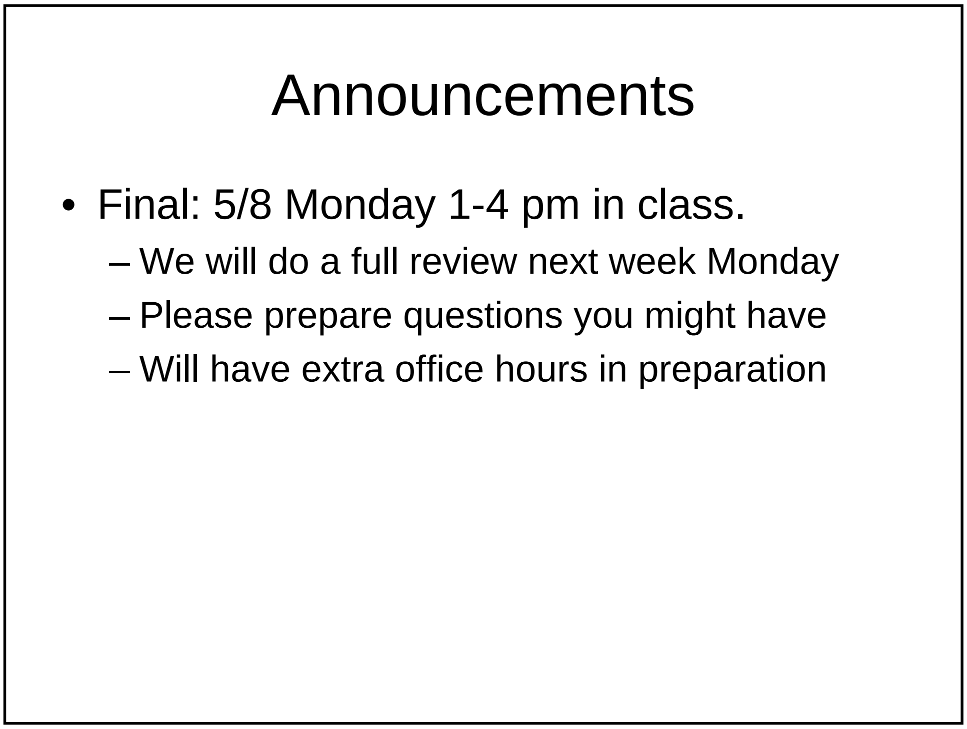
Outline

- C++ wrap up
- Shell commands
- Software engineering



Announcements

- Please go to course works to fill out the class evaluation
  - Again, I will give you credit on final for this
  - Chance to win prizes!
  - Please take care of it this week



Announcements

- Final: 5/8 Monday 1-4 pm in class.
  - We will do a full review next week Monday
  - Please prepare questions you might have
  - Will have extra office hours in preparation

Schedule:

- Will now wrap up cpp
- Next we will cover basic and not so basic unix utilities
- Might have time for some software engineering background
- Will meet for last lab this week
- Anyone want to see php next week?

Wrap up CPP

- Issues with last lab
  - Not sure what happened
  - I ran tests on some machines, but I don’t recall if I did all operators on all clic machine types ☺
  - Nature of the course, what lesson can you take away from it ?

Last Homework

- Very short
- Will be posted today

- Using the POWER of template programming you will be writing a fraction class for CPP and use a simple CGI front-end to make it work over the network

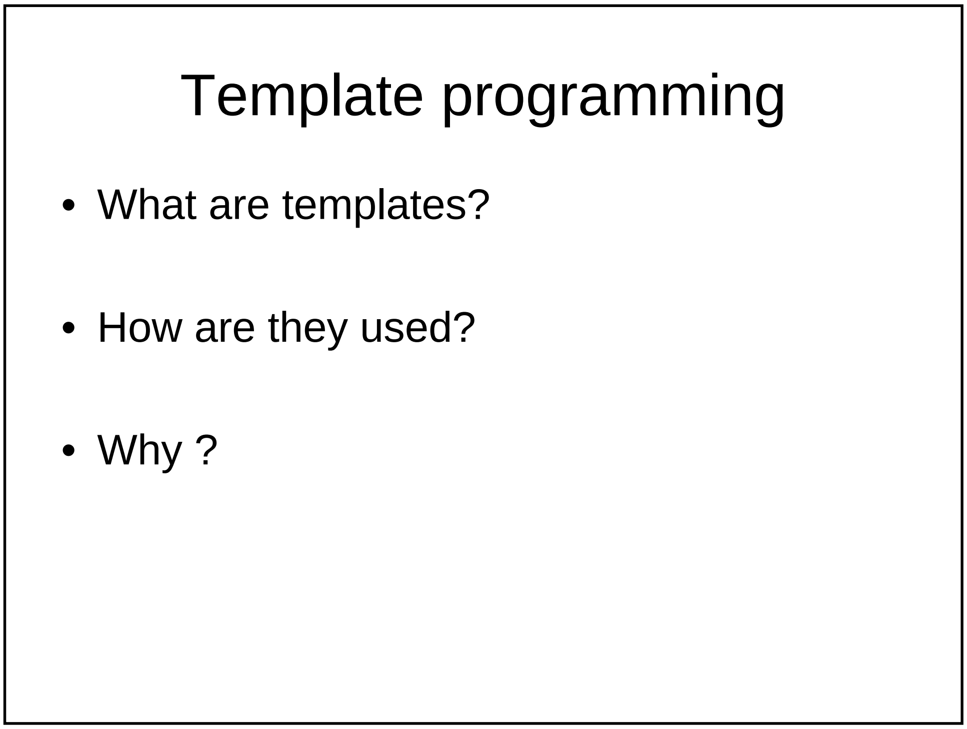
Fraction class

- When you want to add ½ + 1/3

- Convert to .5 + .3 = .8

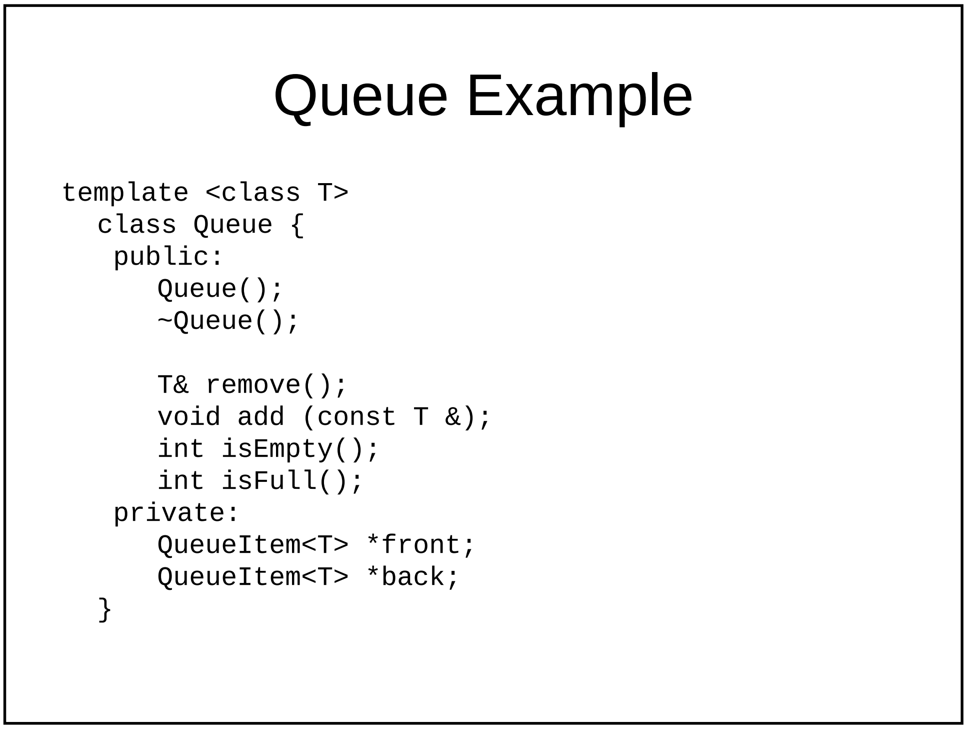
- Want to work with fraction natively
- Want to learn to use templates

- Also want to be able to operate on fractions and reduce fractions
  - Will need to code GCD



Template programming

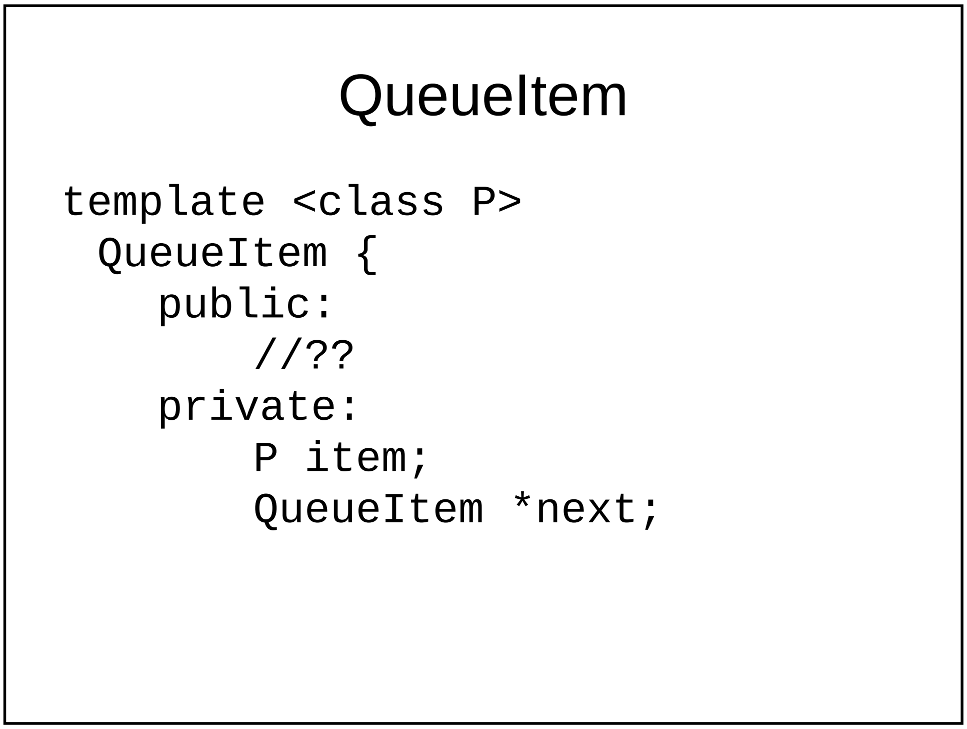
- What are templates?
- How are they used?
- Why ?



Queue Example

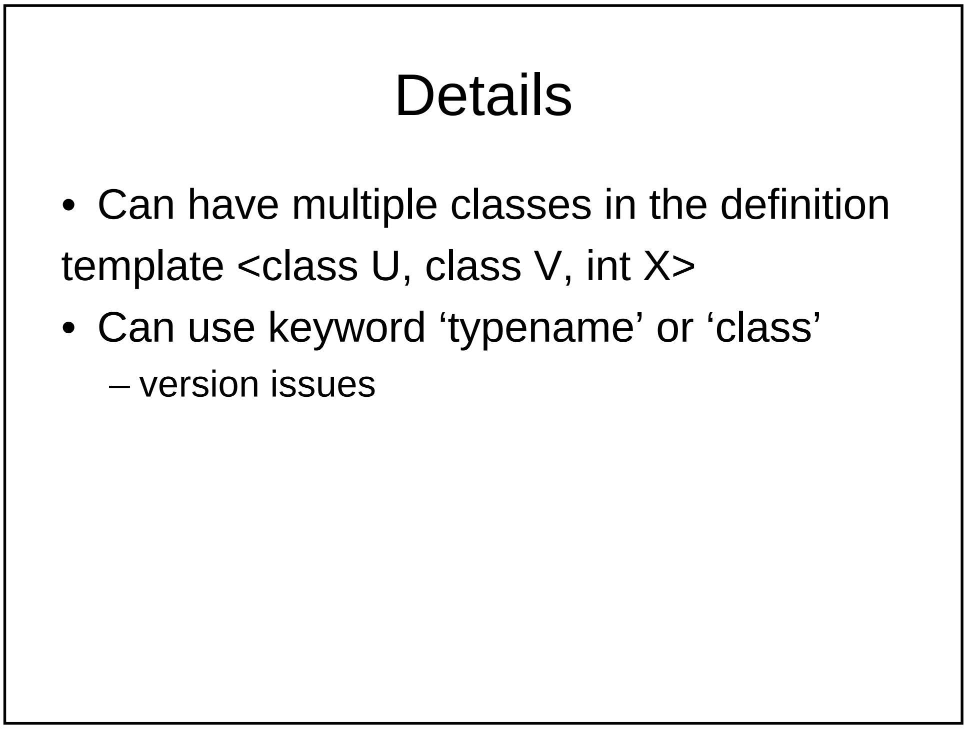
```
template <class T>
  class Queue {
   public:
      Queue();
      ~Queue();

      T& remove();
      void add (const T &);
      int isEmpty();
      int isFull();
   private:
      QueueItem<T> *front;
      QueueItem<T> *back;
  }
```



QueueItem

```
template <class P>
 QueueItem {
    public:
        //??
    private:
        P item;
        QueueItem *next;
```



Details

- Can have multiple classes in the definition

template <class U, class V, int X>

- Can use keyword ‘typename’ or ‘class’
  – version issues

Unix Command Shell

- What is UNIX exactly ?
- What are Unix flavors ?
- What in the world is a command shell ??

Brief History

- Early on, OS were specialized to hardware
  - Upgrade = new OS
- 1965, Bell Labs and GE
  - Multics
    - System to support many users at the same time
    - Mainframe timesharing system
  - 1969 – Bell withdrew, but some researchers persisted on the idea of small operating system

More history

- So first ideas coded in Assembler and B
- Rewritten in C – wanted high level code
  - First concept of software pipes
  - Released in 1972
  - Released source through licensing agreements
  - Addition of TCP and specialization versions to different groups
  - Taught in university courses where it caught on
  - Brought to business by new graduates ☺ (early 80’s)
  - System V (1983)

Command shell

- Allows you to interact with the operating system
- Usually refer to non graphical one

- Windows NT/XP:
  - Start -> run -> cmd
- Windows 98
  - Start -> run -> command
- Unix
  - Log in (most of the time)
- Mac
  - terminal

Technical Details

- Shell is simply a program which takes your commands and interprets them
- Usually write your own in OS course
- Many different kinds of shells
  - Mainly to confuse you ☺
- Main advantage
  - Can use build in language to write simple but powerful scripts

Main shells (unix)

- Bourne Shell
  - sh
  - ksh
  - zsh
- C shell
  - csh
  - tcsh

shell

- sh is the “Bourne shell”, the first scripting language
- it is a program that interprets your command lines and runs other programs
- it can invoke Unix commands and also has its own set of commands

```
while ( 1 ) {
print prompt and wait for user to enter input;
read input from terminal;
parse into words;
substitute variables;
execute commands (execv or builtin);
}
```

- shell commands can be read:
  - – from a terminal == interactive
  - – from a file == shell script
- search path
  - – the place where the shell looks for the commands it runs
  - – should include standard directories:
    - /bin
    - /usr/bin
    - it should also include your current working directory (.)

- are you running the Bourne shell?

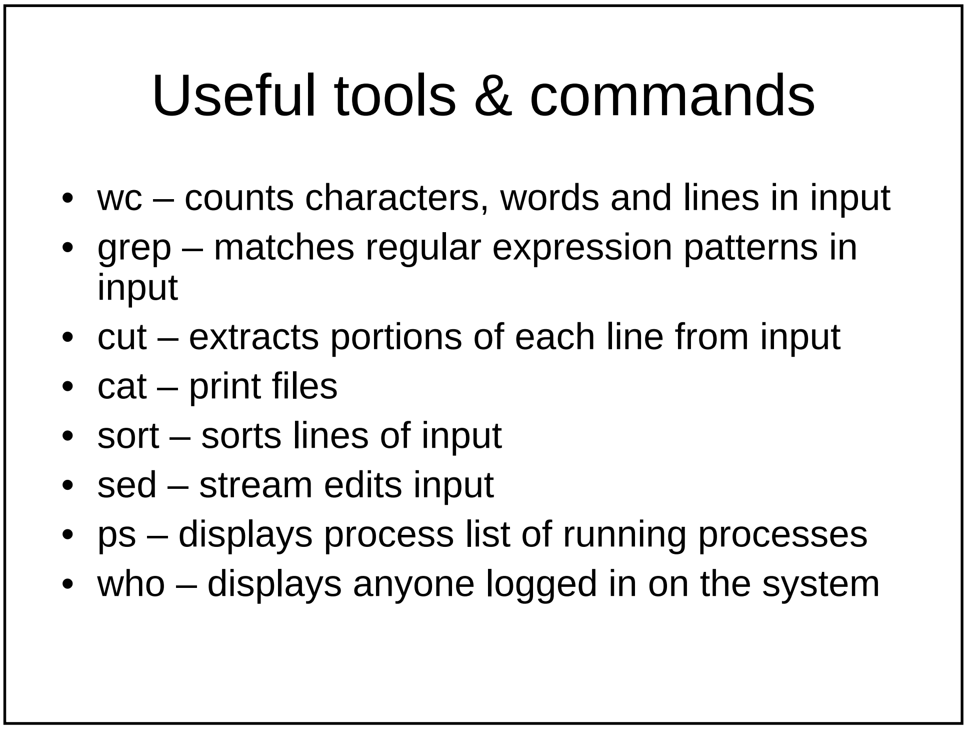
type:

```
$SHELL
```

- if the answer is /bin/sh, then you are
- if the answer is /bin/bash, then that’s close enough
- otherwise, you can start the Bourne shell by typing sh at the UNIX prompt
- enter Ctrl-D or exit to exit the Bourne shell and go back to whatever shell you were running before...

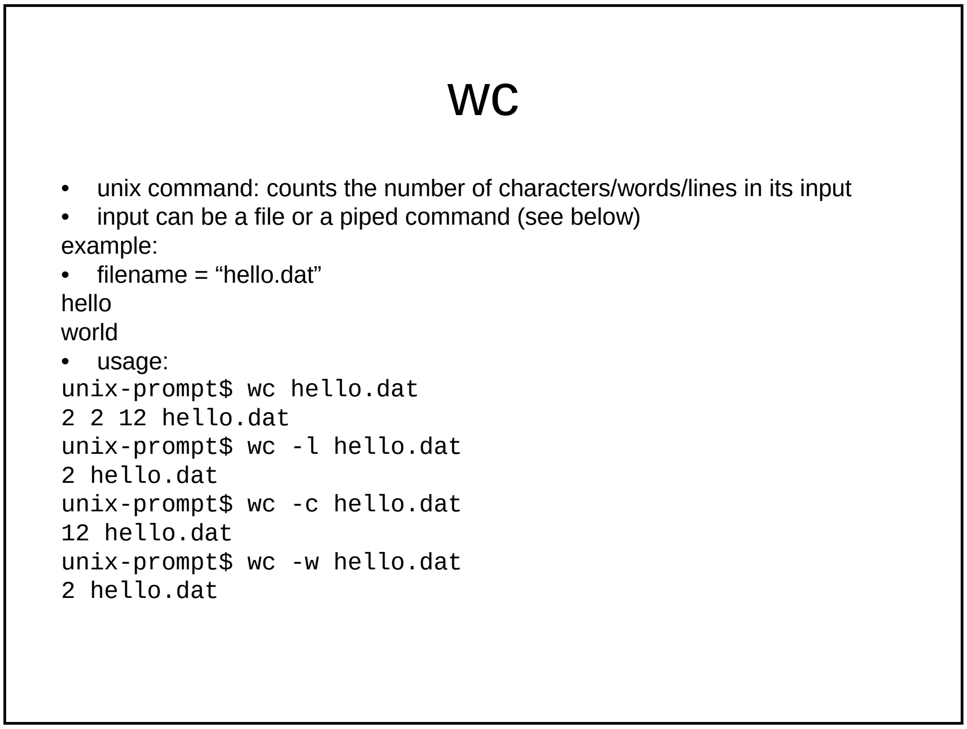
Power of Shells

- capable of both synchronous and asynchronous execution
  - synchronous: wait for completion
  - asychronous: in parallel with shell (runs in the background)
- allows control of stdin, stdout, stderr
- enables environment setting for processes (using inheritance between processes)
- sets default directory



Useful tools & commands

- wc – counts characters, words and lines in input
- grep – matches regular expression patterns in input
- cut – extracts portions of each line from input
- cat – print files
- sort – sorts lines of input
- sed – stream edits input
- ps – displays process list of running processes
- who – displays anyone logged in on the system



wc

- unix command: counts the number of characters/words/lines in its input
- input can be a file or a piped command (see below)

example:

- filename = “hello.dat”

hello
world

- usage:

```
unix-prompt$ wc hello.dat
2 2 12 hello.dat
unix-prompt$ wc -l hello.dat
2 hello.dat
unix-prompt$ wc -c hello.dat
12 hello.dat
unix-prompt$ wc -w hello.dat
2 hello.dat
```

Global Regular Expression Parser GREP

- one of the most useful tools in unix
- three standard versions:
  - plain old grep
  - extended grep: egrep
  - fast grep: fgrep
- used to search through files for ... regular expressions!
- prints only lines that match given pattern
- a kind of filter
- BUT it’s line oriented

- input can be one or more files or can be piped into grep
- examples:

```
grep "ˆ[aeiou]" myfile
ls -1 | grep t
```

- useful options:
- -i ignore case
- -w match pattern as a word
- -l return only the filename if there’s a match
- -v reverse the normal action (i.e., return what doesn’t match)

- examples:

```
grep -i "ˆ[aeiou]" myfile
grep -v "ˆ[aeiou]" myfile
grep -iv "ˆ[aeiou]" myfile
```

- how do you list all lines containing a digit?
- how do you list all lines containing a 5?
- how do you list all lines containing a 0?
- how do you list all lines containing 50?
- how do you list all lines containing a 5 and an 0?

cut

- unix command: extracts portions of each line from input
- input can be a file or a piped command
- Can cut file according to deliminators (fields) and characters
- syntax: cut <-c|f> <-d>
- note that c and +f+ start with 1; default delimiter is TAB

cat

- Concatenate files and print to standard out

- Easy way to pipe the contents of a file to another command

sort

- unix command: sorts lines of input

- input can be a file or a piped command (see below)

- three modes: sort, check (sort -c), merge (sort -m)

- syntax: sort <-t> <-n> <-r> <-o> POS1 -POS2+
- note that POS starts with 0; default delimiter is whitespace

sed

- stream editor
- does not change the file it “edits”

- commands are implicitly global
- input can be a file or can be piped into sed

- example: substitute all A for B:
- sed ’s/A/B/’ myfile
- cat myfile | sed ’s/A/B/’

- use the -e option to specify more than one command at a time:
- sed -e ’s/A/B/’ -e ’s/C/D/’ myfile

- pipe output to a file in order to save it:
- sed -e ’s/A/B/’ -e ’s/C/D/’ myfile >mynewfile

sed

- sed can specify an address of the line(s) to affect
- if no address is specified, then all lines are affected
- if there is one address, then any line matching the address is affected
- if there are two (comma separated) addresses, then all lines between the two addresses
- are affected
- if an exclamation mark (!) follows the address, then all lines that DON’T match the
- address are affected
- addresses are used in conjunction with commands

- examples (using the delete (d) command):

```
sed ’$d’ myfile
sed ’/ˆ$/d’ myfile
sed ’1,/under/d’ myfile
sed ’/over/,/under/d’ myfile
```

- order of commands is important
- input is line oriented
- all editing commands are applied to each line, one at a time
- then next line is read and editing commands are applied to that linei
- etc

- for example:

```
sed -e ’s/pig/cow/’ -e ’s/cow/horse’ myfile
```

- What does this do?

- Regular expression like grep
- Except forward slash
- delimiter is slash (/)
- backslash (escape) it if it appears in the command, e.g.:

```
sed ’s/\/usr\/bin\//\/usr\/etc/’
 myfile
```

- meta-character ampersand (&) represents the extent of the pattern matched
- example:

```
sed ’s/[0-9]/#&/’ myfile
```

- what does this do?

- you can also save portions of the matched pattern:

```
sed ’s/\([0-9]\)/#\1/’ myfile
sed ’s/\([0-9]\)\([0-9]\)/#\1-\2/’ myfile
```

- transformation command: y
- example:

```
sed ’y/ABC/abc’ myfile
```

- print command: p

- example:

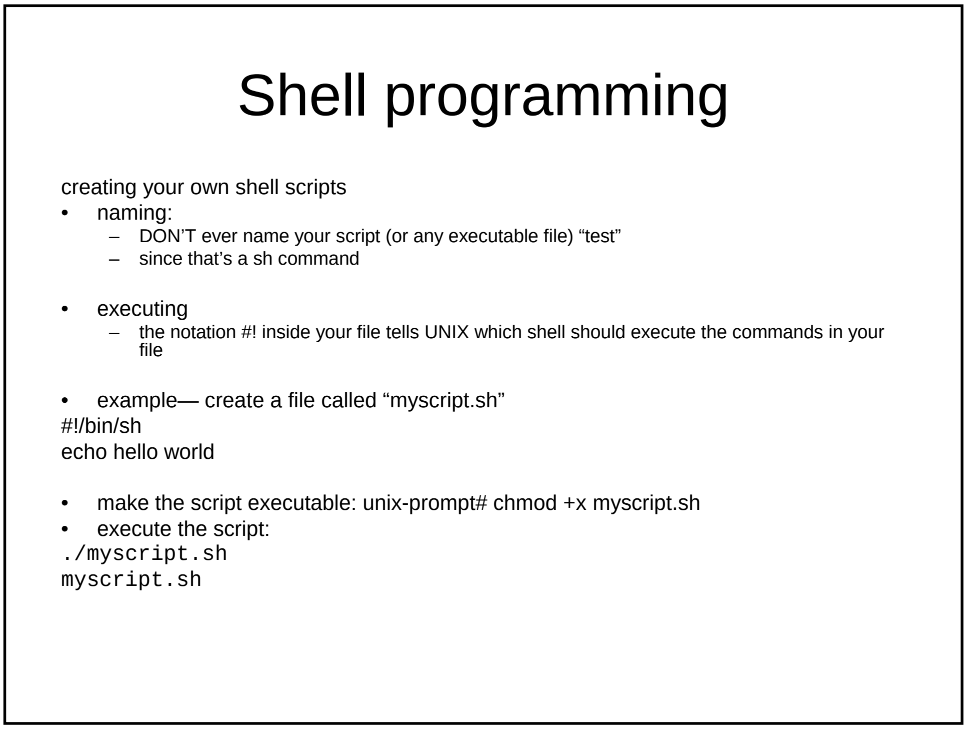
```
sed ’/begin/,/end/p’ myfile
sed -n ’/begin/,/end/p’ myfile
```

- what do the following sed commands do?

```
sed ’s/xx/yy’ myfile
sed ’/BSD/d’ myfile
sed ’/ˆBEGIN/,/ˆEND/p@’ myfile
```

- how do you change the content of all your html files to lowercase?

- how do you change all the html commands to lowercase?



Shell programming

creating your own shell scripts

- naming:
  - DON'T ever name your script (or any executable file) “test”
  - since that's a sh command

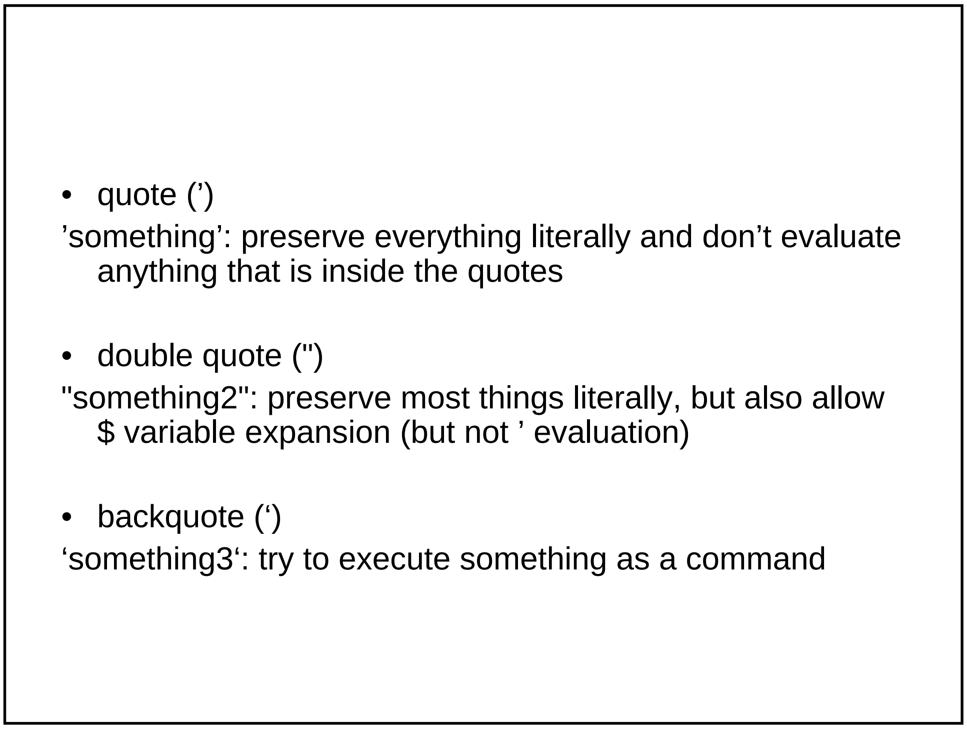
- executing
  - the notation #! inside your file tells UNIX which shell should execute the commands in your file

- example— create a file called “myscript.sh”

```
#!/bin/sh
echo hello world
```

- make the script executable: unix-prompt# chmod +x myscript.sh
- execute the script:

```
./myscript.sh
myscript.sh
```



- quote (’)

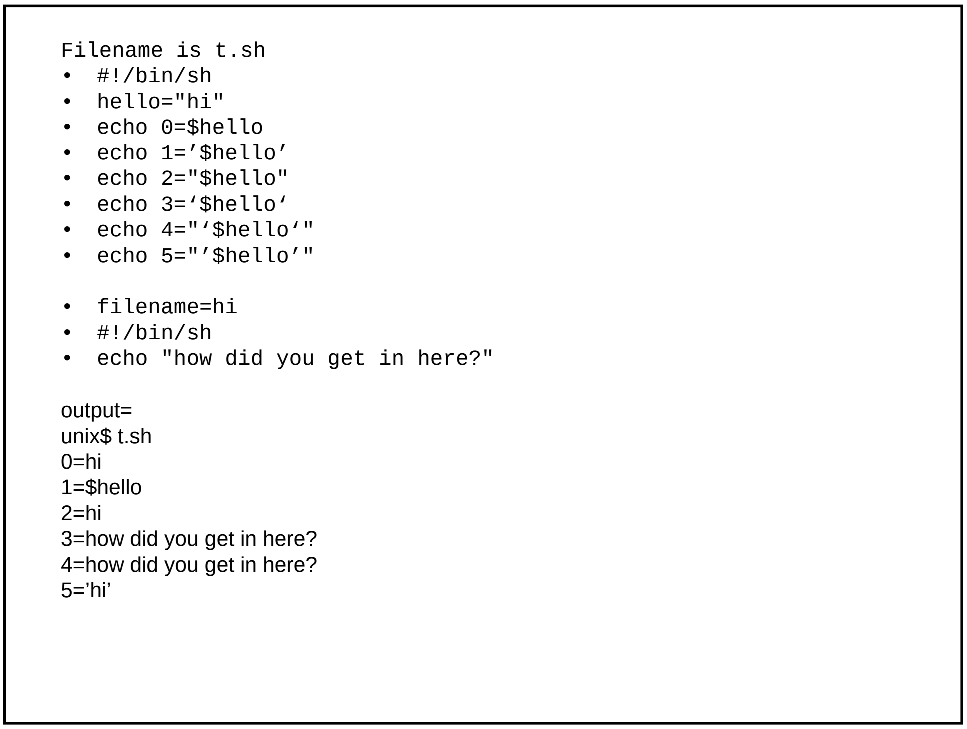
’something’: preserve everything literally and don’t evaluate anything that is inside the quotes

- double quote (")

"something2": preserve most things literally, but also allow $ variable expansion (but not ’ evaluation)

- backquote (‘)

‘something3‘: try to execute something as a command

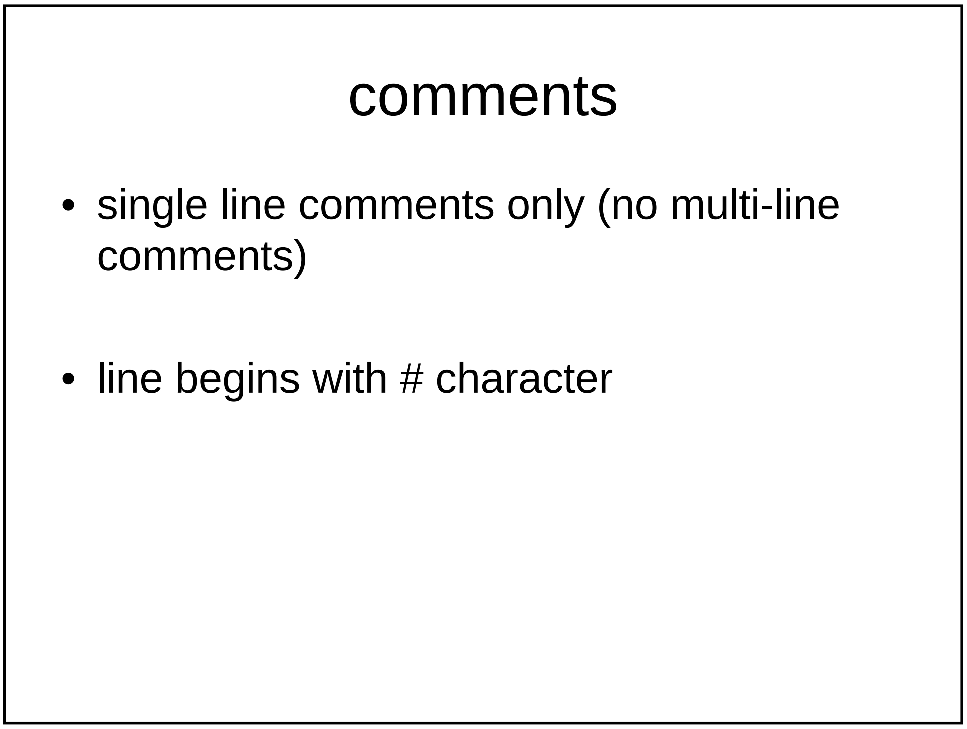


```
Filename is t.sh
•  #!/bin/sh
•  hello="hi"
•  echo 0=$hello
•  echo 1=’$hello’
•  echo 2="$hello"
•  echo 3=‘$hello‘
•  echo 4="‘$hello‘"
•  echo 5="’$hello’"

•  filename=hi
•  #!/bin/sh
•  echo "how did you get in here?"
```

output=

```
unix$ t.sh
0=hi
1=$hello
2=hi
3=how did you get in here?
4=how did you get in here?
5=’hi’
```



comments

- single line comments only (no multi-line comments)
- line begins with # character

Simple commands

- sequence of words

- first word defines command
- can be combined with &&, ||, ;
  - to execute commands sequentially:
    cmd1; cmd2;
  - to execute a command in the background :
    cmd1&
  - to execute two commands asynchronously:
    cmd1&
    cmd2&
  - to execute cmd2 if cmd1 has zero exit status:
    cmd1 && cmd2
  - to execute cmd2 only if cmd1 has non-zero exit status:
    cmd1 || cmd2

- set exit status using exit command (e.g., exit 0 or exit 1)

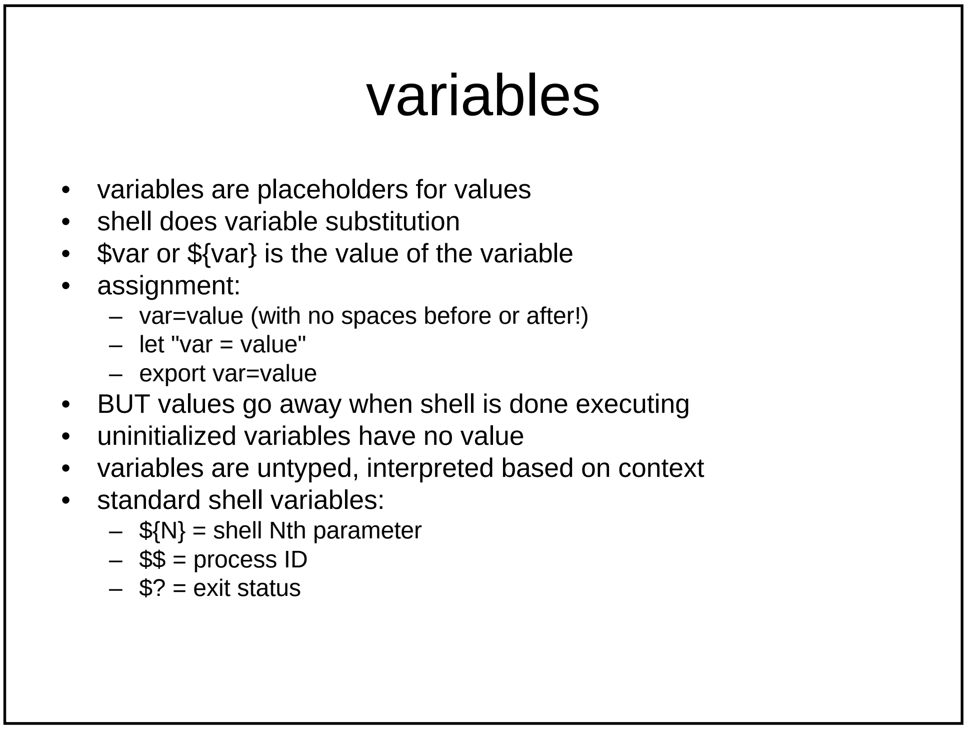
pipes

- sequence of commands
- connected with |

- each command reads previous command’s output and takes it as input

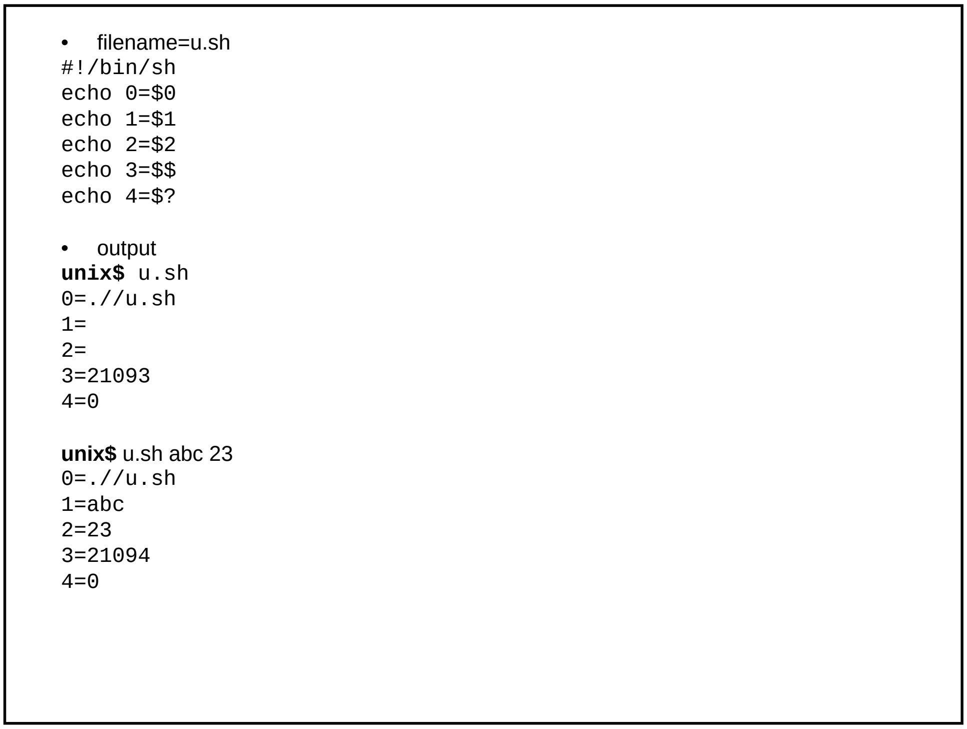
- example:

```
echo "hello world" | wc -w
2
```



variables

- variables are placeholders for values
- shell does variable substitution
- $var or ${var} is the value of the variable
- assignment:
  – var=value (with no spaces before or after!)
  – let "var = value"
  – export var=value
- BUT values go away when shell is done executing
- uninitialized variables have no value
- variables are untyped, interpreted based on context
- standard shell variables:
  – ${N} = shell Nth parameter
  – $$ = process ID
  – $? = exit status



- filename=u.sh

```
#!/bin/sh
echo 0=$0
echo 1=$1
echo 2=$2
echo 3=$$
echo 4=$?
```

- output

```
unix$ u.sh
0=.//u.sh
1=
2=
3=21093
4=0

unix$ u.sh abc 23
0=.//u.sh
1=abc
2=23
3=21094
4=0
```

- shell variables are generally not visible to programs
- environment variables are a list of name/value pairs passed to sub-processes
- all environment variables are also shell variables, but not vice versa

- show with env or echo $var

- standard environment variables include:
  - HOME = home directory
  - PATH = list of directories to search
  - TERM = type of terminal (vt100, ...)
  - TZ = timezone (e.g., US/Eastern)

Loops

- similar to C/Java constructs, but with commands
- until test-commands; do consequent-commands; done
- while test-commands; do consequent-commands; done
- for name [in words ...]; do commands; done

- also on separate lines
- break and continue control loop

- while

```
i=0
while [ $i -lt 10 ]; do
echo "i=$i"
((i=$i+1)) # same as let "i=$i+1"
done
```

- for

```
for counter in ‘ls *.c‘; do
echo $counter
done
```

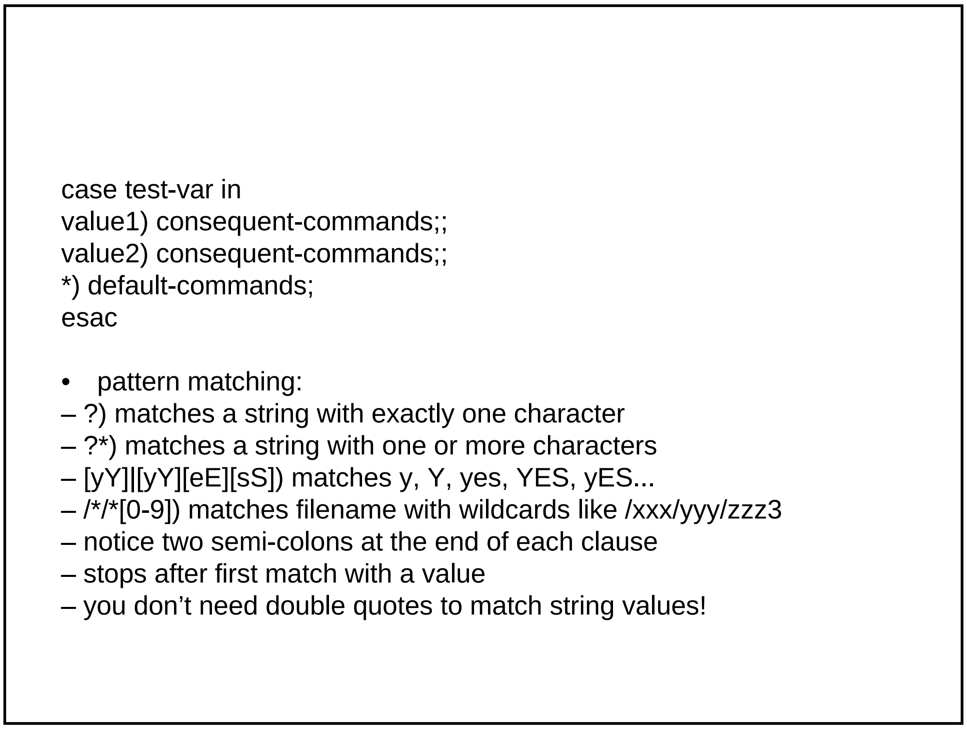
if

if test-commands; then
    consequent-commands;
[elif more-test-commands; then
    more-consequents;]
[else alternate-consequents;]
fi

- colon (:) is a null command

- example

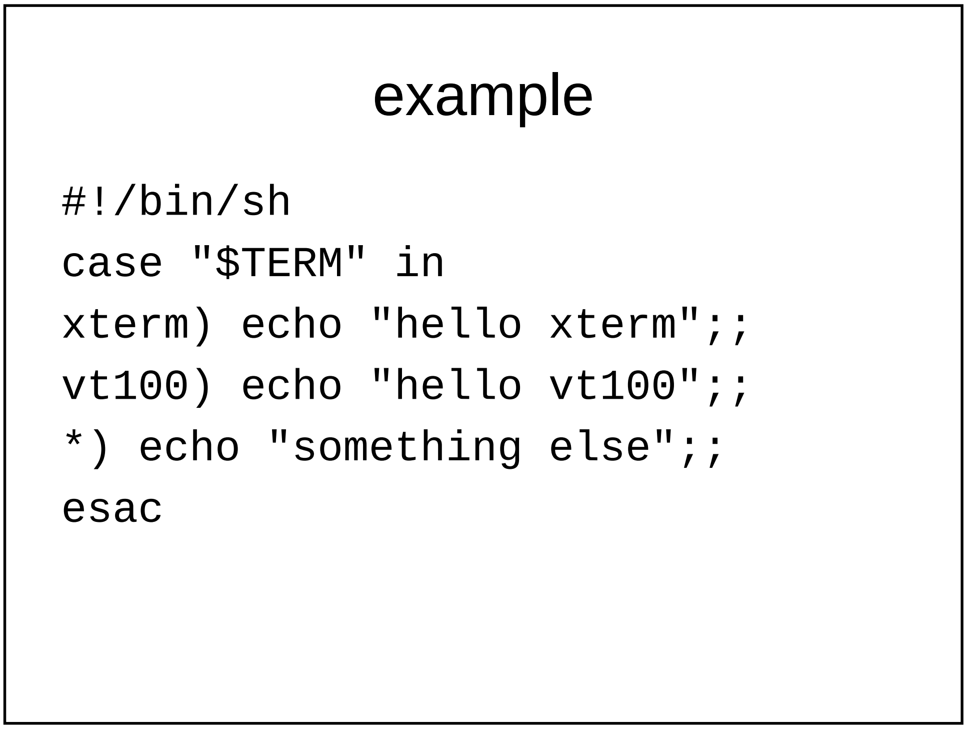
```
#!/bin/sh
if expr $TERM = "xterm"; then
echo "hello xterm";
else
echo "something else";
fi
```



case test-var in
value1) consequent-commands;;
value2) consequent-commands;;
*) default-commands;
esac

- pattern matching:

– ?) matches a string with exactly one character
– ?*) matches a string with one or more characters
– [yY]|[yY][eE][sS]) matches y, Y, yes, YES, yES...
– /*/*[0-9]) matches filename with wildcards like /xxx/yyy/zzz3
– notice two semi-colons at the end of each clause
– stops after first match with a value
– you don’t need double quotes to match string values!



example

```
#!/bin/sh
case "$TERM" in
xterm) echo "hello xterm";;
vt100) echo "hello vt100";;
*) echo "something else";;
esac
```

- biggest difference from traditional programming languages
- shell substitutes and executes
- order:
  - brace expansion
  - tilde expansion
  - parameter and variable expansion
  - command substitution
  - arithmetic expansion
  - word splitting
  - filename expansion

Command subing

- replace $(command) or ‘command‘ by stdout of executing command
- can be used to execute content of variables:

```
unix$ x=ls
unix$ $x
myfile.c
a.out
unix$ echo $x
ls
unix$ echo ‘ls‘
myfile.c
a.out
unix$ echo ‘x‘
sh: x: command not found
unix$ echo ‘$x‘
myfile.c
a.out
unix$ echo $(ls)
myfile.c
a.out
unix$ echo $(x)
sh: x: command not found
unix$ echo $($x)
myfile.c
a.out
```

Filename expansion

- any word containing *?([is considered a pattern
- * matches any string
- ? matches any single character
- [...] matches any of the enclosed characters

```
unix$ ls
myfile.c
a.out
a.b
unix$ ls a*
a.out
a.b
unix$ ls a?
ls: No match.
unix$ ls a.*
a.out
a.b
unix$ ls a.?
a.b
unix$ ls a.???
a.out
unix$ ls [am].b
a.b
```

redirection

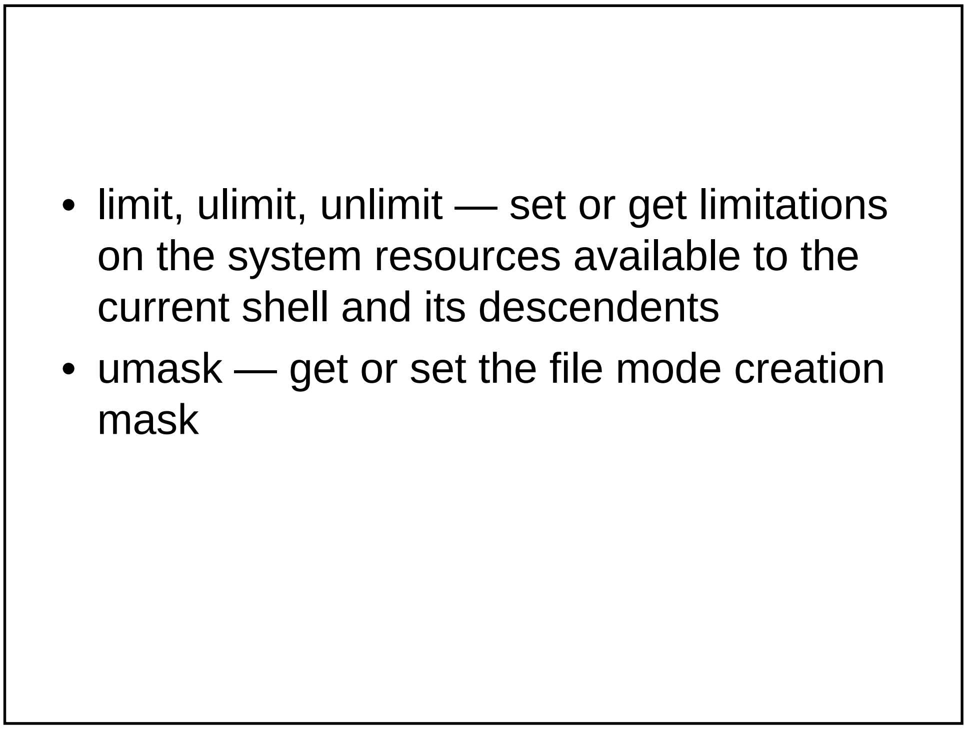
- stdin, stdout and stderr may be redirected
- < redirects stdin (0) to come from a file
- > redirects stdout (1) to go to file
- >> appends stdout to the end of a file
- &> redirects stderr (2)
- >& redirects stdout and stderr, e.g.: 2>&1 sends stderr to the same place that stdout is going
- << gets input from a here document, i.e., the input is what you type, rather than reading from a file

Built in commands

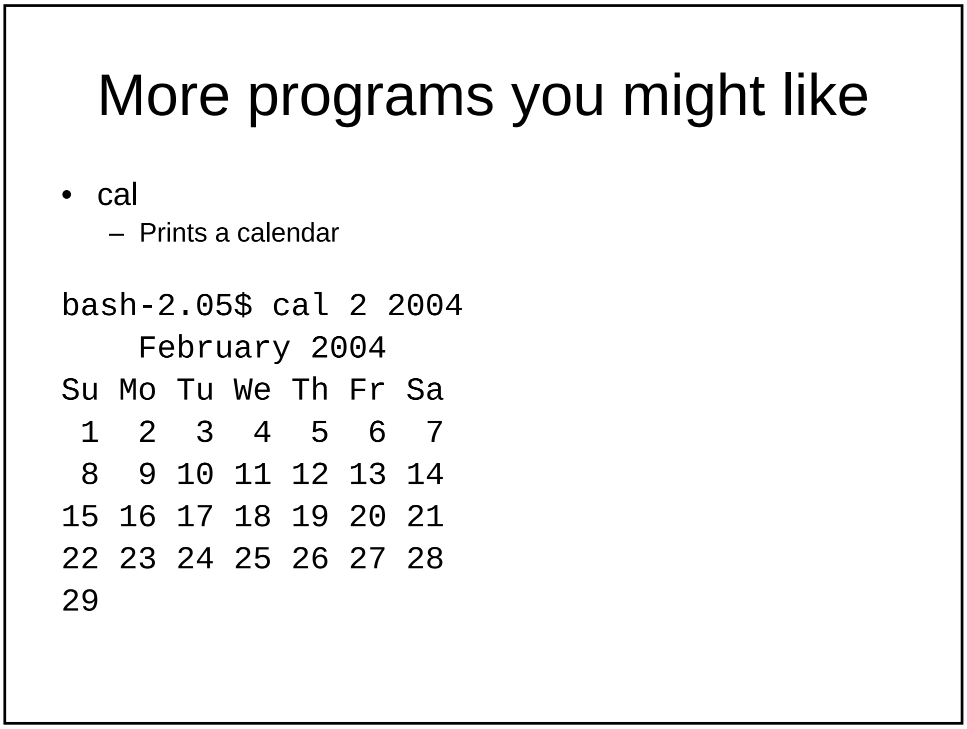
- alias, unalias — create or remove a pseudonym or shorthand for a command or series of commands
- jobs, fg, bg, stop, notify — control process execution
- command — execute a simple command
- cd, chdir, pushd, popd, dirs — change working directory
- echo — display a line of text
- history, fc — process command history list
- set, unset, setenv, unsetenv, export — shell built-in functions to determine the characteristics for environmental variables of the current shell and its descendents

- getopts — parse utility options
- hash, rehash, unhash, hashstat — evaluate the internal hash table of the contents of directories
- kill — send a signal to a process

- pwd — print name of current/working directory
- shift — shell built-in function to traverse either a shell's argument list or a list of field-separated words
- readonly — shell built-in function to protect the value of the given variable from reassignment
- source — execute a file as a shell script
- suspend — shell built-in function to halt the current shell
- test — check file types and compare values
- times — shell built-in function to report time usages of the current shell
- trap, onintr — shell built-in functions to respond to (hardware) signals
- type — write a description of command type
- typeset, whence — shell built-in functions to set/get attributes and values for shell variables and functions



- limit, ulimit, unlimit — set or get limitations on the system resources available to the current shell and its descendents
- umask — get or set the file mode creation mask



More programs you might like

- cal
  - Prints a calendar

```
bash-2.05$ cal 2 2004
    February 2004
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29
```

Usage stuff

- df

```
bash-2.05$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda3             197M  157M   31M  84% /
/dev/hda7             296M   65k  280M   1% /tmp
/dev/hda5             2.4G  2.0G  385M  84% /usr
```

- du

```
bash-2.05$ du -ch code2
48k     code2/ai1
56k     code2
56k     total
```

- quota

Next time

- Lab Wednesday
  - Please come on time, will be wrapping up all labs and answering any lab questions you have
  - Will have extra credit lab on unix programming
  - Will give you homework hints/help
- Monday – review and practice final class