CS3157: Advanced Programming

Lecture #7

June 14

Shlomo Hershkop
shlomo@cs.columbia.edu

1

Announcements

- will be posting homework
- new lab tonight, will discuss it at end of class
- feedback?

2

Reading

- Chapter 15,16 c++, data abstraction
- Chapter 17,18 classes

3

Outline

- Intro CPP
  - Background stuff:
    - Language basics: identifiers, data types, operators, type conversions, branching and looping, program structure
  - data structures: arrays, structures
  - pointers and references differences
  - I/O: writing to the screen, reading from the keyboard, iostream library
  - classes: defining, scope, ctors and dtors
  - Bunch of code

- Reading: start on cpp, wrap up any old homeworks

4

differences between c++ and c

- history and background
- object-oriented programming with classes

- very brief history...
  - C was developed 69-73 at Bell labs.
  - C++ designed by Bjarne Stroustrop at AT&T Bell Labs in the early 1980's
  - originally developed as "C with classes"
  - Idea was to create reusable code
  - development period: 1985-1991
  - ANSI standard C++ released in 1991

5

Four main OOP concepts

- abstraction
  - creation of well-defined interface for an object, separate from its implementation
  - e.g., Vector in Java
  - e.g., key functionalities (init, add, delete, count, print) which can be called independently of knowing how an object is implemented
- encapsulation
  - keeping implementation details "private", i.e., inside the implementation
- hierarchy
  - an object is defined in terms of other objects
  - Composition => larger objects out of smaller ones
  - Inheritance => properties of smaller objects are "inherited" by larger objects
- polymorphism
  - use code "transparently" for all types of same class of object
  - i.e., "morph" one object into another object within same hierarchy

6

Basic differences

- Before we talk about OOP, lets discuss language differences:

1. Naming Conventions of files
2. Comments styles
3. Struct treated differently
4. I/O redesigned
5. Function abstraction enforced

7

Hello.cpp

```
#include <iostream>

using namespace std;
main() {
cout << "hello world\n";
cout << "hello" << " world" << endl;
printf( "hello yet again!\n" );
}
```

- compile using:

```
g++ hello.cpp -o hello
```

- like gcc (default output file is a.out)

8

Main difference between c and cpp

- C's power is driven by functions. You define a set of function which operate in a specific sequence to implement some algorithm
  - Top down
- CPP is an object oriented language
  - design parts of the system
  - put them together
  - bottom up approach

9

Compatible

- Cpp is backwards compatible with c
- Cpp is bottom up approach
- Cpp compilers will compile c code

10

Advantages

- There are a bunch of (claimed) advantages to using CPP over c

11

Advantages

- Can create new programs faster because we can reuse code
- Easier to create new data types
- Easier memory management
- Programs should be less bug-prone, as it uses a stricter syntax and type checking.
- `Data hiding', the usage of data by one program part while other program parts cannot access the data
- Will whiten your teeth 

12

Defining c++ functions

- a function's "signature" is its name plus number and type of arguments
- you can have multiple functions with same name, as long as the signatures are different
- example:

```
void foo( int a, char b );
void foo( int a, int b );
void foo( int a );
void foo( double f );
main() {
foo( 1,'x' );
foo( 1,2 );
foo( 3 );
foo( 5.79 );
}
```

- OVERLOADING – when function name is used by more than one function

13

C++ Function II

- Foo() or Foo(void) for void arguments
  - Different than c
- Foo(…) for unchecked parameters
  - Look up va_list and va_start
  - A cleaner approach is to pass in an array
- New Trick:
- Foo(int a, int b, int c=10)
  - Foo(4,5,2)
  - Foo(4,5)

14

Function III

- Inline functions
- Function overloading:
  - void foo(int a, char c)
  - void foo(char c)

  - Not allowed
    - void foo(int a)
    - int foo(int a)

15

Other additions

- C++ includes many compiler side additions to help the programmer (yes that is you) to write better code
- Other technical changes (will be pointing them out as we pass them)

16

Void pointers

- C allows you to assign and convert void pointers without casting
- C++ needs a cast

```
void * V;
..
Foo *f = (Foo)V;
```

17

enums

- Are treated a little differently in c++
- enum day {Sunday, Monday , .. }
- day X = 1; //only works in c
- day X = Sunday;

18

main()

- In C main is the first thing to run
- C++ allows things to run before main, through global variables
  - What is the implications ?
- Variable which are declared outside of main, have global scope (will cover limits).
- Can have function calls here

19

File conventions

- No one convention
  - .C
  - .cc
  - .cp
  - .cpp ←I prefer this
  - .cxx
  - .c++

20

Keywords c++

- asm
- catch
- class
- friend
- delete
- inline
- new
- operator
- private
- protected
- public
- this
- throw
- template
- try
- virtual

21

C++ vs. Java

- advantages of C++ over Java:
  - C++ is very powerful
  - C++ is very fast
  - C++ is much more efficient in terms of memory
  - compiled directly for specific machines (instead of bytecode layer, which could also be seen as a portability advantage of Java over C++...)
- disadvantages of C++ over Java:
  - Java protects you from making mistakes that C/C++ don't, as you've learned now from working with C
  - C++ has many concepts and possibilities so it has a steep learning curve
  - extensive use of operator overloading, function overloading and virtual functions can very quickly make C++ programs very complicated
  - shortcuts offered in C++ can often make it completely unreadable, just like in C

22

Identifiers

- i.e., valid names for variables, methods, classes, etc
- just like C:
  - names consist of letters, digits and underscores
  - names cannot begin with a digit
  - names cannot be a C++ keyword

- literals are just like in C with a few extras:
  - numbers, e.g.: 5, 5u, 5L, 0x5, true
  - characters, e.g., ’A’
  - strings, e.g., "you" which is stored in 4 bytes as ’y’, ’o’, ’u’, ’\0’

23

data types

- simple native data types: bool, int, double, char, wchar_t

- bool is like boolean in Java

- wchar_t is “wide char” for representing data from character sets with more than 255 characters

- modifiers: short, long, signed, unsigned, e.g., short int

- floating point types: float, double, long double

- enum and typedef just like C

24

Operators

- same as C, with some additions
- if you recognize it from C, then it's pretty safe to assume it is doing the same thing in C++

25

Type conversions

- all integer math is done using int datatypes, so all types (bool, char, short, enum) are promoted to int before any arithmetic operations are performed on them
- mixed expressions of integer / floating types promote the lower type to the higher type according to the following hierarchy:

```
int < unsigned < long < unsigned long
< float < double < long double
```

26

Conversions II

- you can do explicit conversions like in C using cast
  - (int)something
- you can also do explicit conversions using C++ operators:
  - static_cast
    - safe and portable; e.g. c = static_cast<char>(i);
  - reinterpret_cast
    - system dependent, not good to use
  - const_cast
    - lets you change a const into a modifiable variable
  - dynamic_cast
    - used at run-time for casting objects from one class to another (within inheritance hierarchy); this is sort of like Java but can get really messy and is really a more advanced topic...

27

Branching and Looping

- if, if/else just like C and Java
- while and for and do/while just like C and Java
- break and continue just like C and Java
- switch just like C and Java
- goto just like C (but don't use it!!!)

28

Program structure

- just like in C
- program is a collection of functions and declarations
- language is block-structured
- declarations are made at the beginning of a block; allocated on entry to the block and freed when exiting the block
- parameters are call-by-value unless otherwise specified

29

arrays

- similar to C
- dynamic memory allocation handled using new and delete instead of malloc (and family) and free
- examples:

```
int a[5];
char b[3] = { 'a', 'b', 'c' };
double c[4][5];
int *p = new int(5); // space allocated and *p set to 5
int **q = new int[10]; // space allocated and q = &q[0]
int *r = new int; // space allocated but not initialized
```

30

Structures

- struct keyword like in C (but you don't need typedef) (last class)
- use dot operator or -> to access members (fields) of a struct or struct *
- C++ allows **functions** to be members, whereas C only allows data members (i.e., fields)
- example

```
struct point {
public:
void print() const { cout << "(" << x "," << y << ")"; }
void set( double u, double v ) { x=u; y=v; }
private:
double x, y;
}
```

31

Pointers and References

- pointers are like C:
  - int *p means "pointer to int"
  - p = &i means p gets the address of object i. references are not like C!! they are basically aliases – alternative names – for the values stored at the indicated memory locations, e.g.:

```
int n;
int &nn = n;
double a[10];
double &last = a[9];
```

- The difference between them:

```
int a = 5; // declare and define a
int *p = &a; // p points to a
int &refa = a; // alias (reference) for a
*p = 7; // *p points to a, so a is assigned 7
refa = *p + 1; // a is assigned value of *p=7 plus 1
```

32

I/O Screen

```
// hello world in C++
#include <iostream>
using namespace std;
int main() {
cout << "hello world" << endl;
}
```

- comment characters are // or /* ... */, just like Java
- using namespace is sort of like importing a package in Java; it is used in conjunction with the header declaration
- you could also say #include <iostream.h> and leave out the using namespace std; line; this is an older style of C++ but it still works
- cout << is like System.out.print in Java or like printf() in C
- endl outputs a newline; saying cout << "\n"; does the same thing
  - Advantage is its system dependant

33

iostream.h

- it's preferred not to use C's stdio (though you can), because it's not "type safe" (i.e., compiler can't tell if you're passing data of the wrong type, as you know from getting run-time errors...)
- stdio functions are not extensible
- note << is left-shift operator, which iostream "overloads"
- you can string multiple <<'s together, e.g.:
- cout << "hello" << " world" << "\n";
- cout is like stdout
- cerr is like stderr

34

I/O keyboard

- read from the keyboard using cin >>, which is like scanf() in C

- example:

```
#include <iostream>
using namespace std;
int main() {
int i;
cout << "enter a number: ";
cin >> i;
cout << "you entered " << i <<"\n";
}
```

35

C++ iostream

- two bit-shift operators:
  - << meaning “put to” output stream (“left shift”)
  - >> meaning “get from” input stream (“right shift”)

- three standard streams:
  - cout is standard out
  - cin is standard in
  - cerr is standard error

- the iostream library is “type safe”, so you don’t have to use formatting statements:
  variables are input/output based on their datatype

36

ostream and istream

- ostream
  - cout is an ostream, << is an operator
  - use cout.put(char c) to write a single char
  - use cout.write(const char *p, int n) to write n chars
  - use cout.flush() to flush the stream
- istream
  - cin is an istream, >> is an operator
  - use cin.get(char &c) to read a single char
  - use cin.get(char *s, int n, char c='\n') to read a line (inputs into string s at most n-1 characters, up to the specified delimiter c or an EOF; a terminating 0 is placed at the end of the input string s)
  - also cin.getline(char *s, int n, char c='\n')
  - use cin.read(char *s, int n) to read a string

37

Formatted output

- in <iomanip> header file, the following are defined:

- scientific – prints using scientific notation
- left – fills characters to right of value
- right – fills characers to left of value
- internal – fills characters between sign and value
- setfill(int) – sets fill character
- setw(int) – sets field width
- setprecision(int) – sets floating point precision

38

Example

- cout << setprecision(3) << 2.34563;

39

Declaring Class

- Almost like struct, the default privacy specification is private whereas with struct, the default privacy specification is public
- example

```
class point {
double x, y; // implicitly private
public:
void print();
void set( double u, double v );
}
```

- classes can be nested (like java)
- static is like in Java, with some weird subtleties

40

Using

```
point x;
x.set(3,4);
x.print();

point *pptr = &x;

pptr->set(3,2);
pptr->print();
```

41

Classes: function overloading and overriding

- overloading:
  - when you use the same name for functions with different signatures
  - functions in derived class supercede any functions in base class with the same name
- overriding:
  - when you change the behavior of base-class function in a derived class
  - DON’T OVERRIDE BASE-CLASS FUNCTIONS!!

- because compiler can invoke wrong version by mistake
- but init() is okay to override
- (more explanation in ch 12...)

42

Access specifiers

- In class declaration can have:
- Public
  - Anyone can access
- Private
  - Only class members and friends can access

43

Access specifiers

- public
  - public members
  - can be accessed from any function
- private members
  - can only be accessed by class's own members
  - and by "friends" (see ahead)
- Protected
  - Class members, derived, and friends.

- "access violations" when you don't obey the rules...
- can be listed in any order
- can be repeated

44

Class scope

- ::
- example:

```
::i // refers to external scope
point::x // refers to class scope
std::count // refers to namespace scope
```

- given previous definition of point, we could do:

```
point p;
p.print();
p.point::print(); // redundant but legal
```

45

Defining functions

```
void point::print(){
cout << "(" << x "," << y << ")";
}

void point::set( double u, double v )
{ x=u; y=v; }
```

46

Constructors and destructors

- constructors are called ctors in C++; they take the same name as the class in which they are defined, like in Java
- destructors are called dtors in C++; they take the same name as the class in which they are defined, preceded by a tilde (~); sort of like finalize in Java
- ctors can be overloaded and can take arguments
- dtors can not
- default constructor has no arguments
- constructor with one argument is a conversion constructor that converts its argument datatype to an object of the class being constructed
- constructor initializer is a special type of constructor that is used to initialize the values of data members of a class

47

```
class point {
double x,y;
public:
point() { x=0;y=0; } // default
point( double u ) {x =u; y=0; }
// conversion
point( double u, double v )
  { x =u; y =v;}
.
.
.
}
```

48

usage

point p;

49

Unix Command Shell

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

50

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

51

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)

52

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

53

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

54

Main shells (unix)

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

55

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);
}
```

56

- 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 (.)

57

- are you running the Bourne shell?

type:

```
echo $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...

58

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

59

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

60

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
```

61

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

62

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

- examples:

```
grep "^[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)

63

- examples:

```
grep -i "^[aeiou]" myfile
grep -v "^[aeiou]" myfile
grep -iv "^[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?

64

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

65

cat

- Concatenate files and print to standard out
- Easy way to pipe the contents of a file to another command

66

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

67

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

68

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 '$d' myfile
sed '/^$/d' myfile
sed '1,/under/d' myfile
sed '/over/,/under/d' myfile
```

69

- 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?

70

- 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
```

71

- 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
```

72

- transformation command: y
- example:

```
sed 'y/ABC/abc' myfile
```

73

- print command: p

- example:

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

74

- what do the following sed commands do?

```
sed 's/xx/yy' myfile
sed '/BSD/d' myfile
sed '/^BEGIN/,/^END/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?

75

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
```

76

- 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

77

```
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’

78

comments

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

79

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)

80

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

81

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

82

- 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
```

83

- 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)

84

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

85

- 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
```

86

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
```

87

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!

88

example

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

89

- 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

90

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
```

91

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
```

92

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

93

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

94

- 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

95

- 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

96

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
```

97

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

98

What is this ?

```
#!/bin/sh

foo()
{
  if [ "$1" -gt "1" ]; then
    i=`expr $1 - 1`
    j=`foo  $i`
    k=`expr $1 \* $j`
    echo $k
  else
    echo 1
  fi
}


while :
do
  echo "Enter a number:"
  read x
  foo $x
done
```

99

- start the lab early

100