COMP 206: Intro to Software Systems Review

Julian Lore

Last updated: April 22, 2017

Adapted from Joseph Vybihal's Winter 2017 COMP206 slides.

Contents

1 Software Systems

1.1 What is a Software System?

A **system** has several parts. By themselves, not special. They cooperate together to make something. Addition of several sub-systems (programs that depend on other programs). System is the complete application that interacts directly with user.
A software system consists of a system with components (based on software) that form a part of a computer system. I.e. has separate programs, config files, documentation. Single components from a software system are usually useless without each other.
In this class, our programs will send instructions to the operating system, which will deal with sending stuff to the actual hardware.

1.2 Examples of Software Systems

Email Email clients don't know how to send email, they just send data through a wire. Keeps sending it to the next person (connected in network), until finally ISP or something else knows what to do with the email. Each piece only knows so much, all have to be together to work.

Facebook Needs Internet, ISP, PC/phone/browser and then the server/application/database. With only one thing, wouldn't be able to use Facebook.

Object oriented programming/Java Requires JVM, computer, etc. Text file → byte code converter → JVM → Libs → OS → PC →GPU or Network→Internet

Internet The Internet is the quintessential software system! Has many sub-systems, such as:

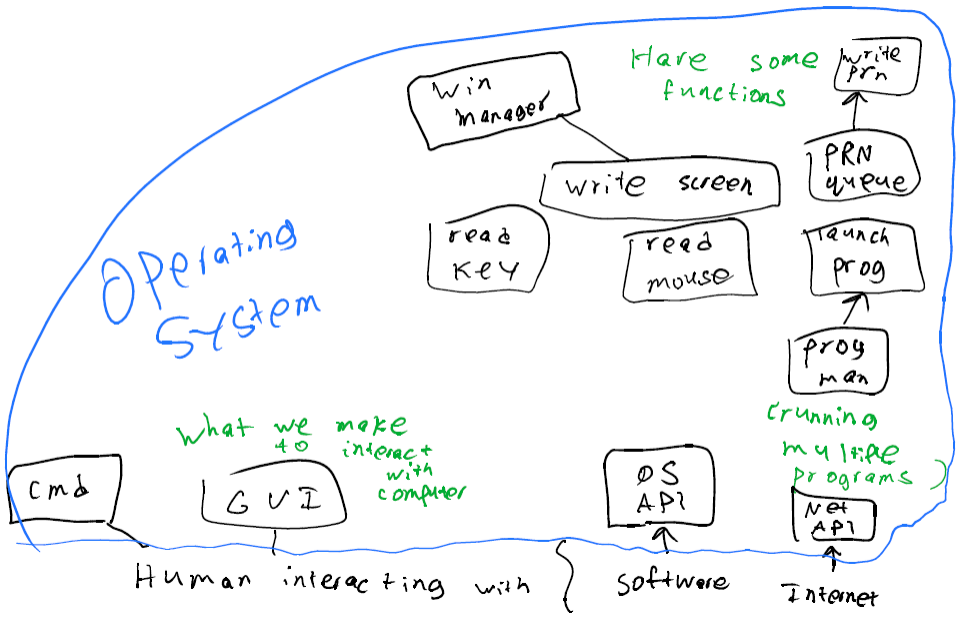
- Local PC
- Server
- Network
- Encryption
- etc.

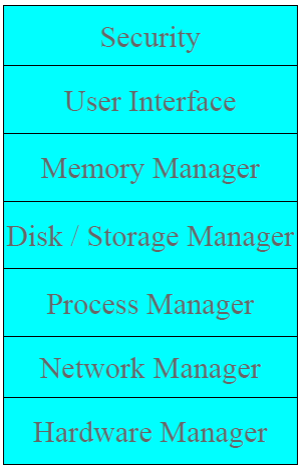
See 1.4 For more information.

1.3 Operating Systems

Drivers Small pieces of software, allow external devices to communicate with PC generically. So application does not have to deal separately with all different types of hardware.

What is an Operating System? Piece of software, allows users to use computer without knowing inner workings. Main use is to manage resources, execute programs properly. Middle man between low-level hardware & users/programs. Also provides libraries.





OS Architecture Uses ideas from systems and subsystems.

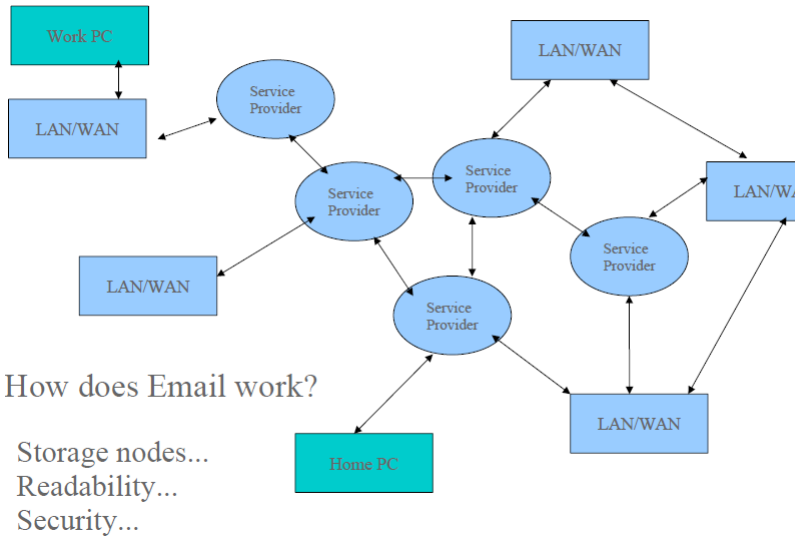
- Security
  - Passwords
  - Encryption
  - File permissions
- User Interface
  - GUI, CLI, Shell memory
- Memory Manager
  - Find memory for programs
  - Format RAM
  - Manage cache, buffers, spools
- Storage Manager
  - Secondary Storage
  - Floppy disks
  - Hard disks
  - Memory sticks
  - etc.
- Process Manager
  - Runs programs

  - Multi-processing/multi-CPU
  - Kills programs
  - Run-time errors

- Network Manager
  - LAN
  - Internet
  - etc.

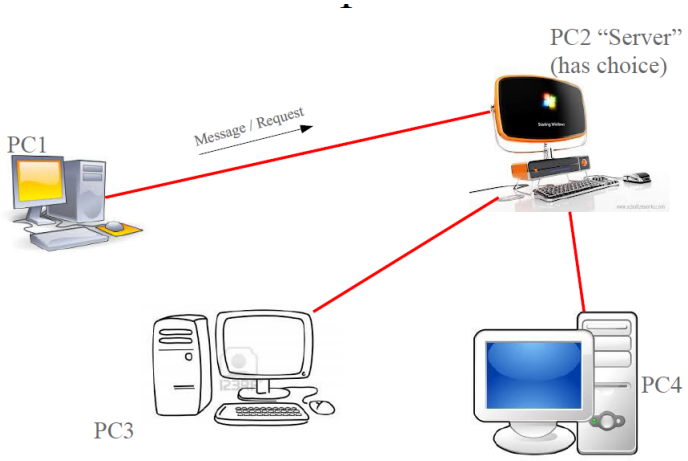
- Hardware Manager
  - Drivers
  - Assembler & registers
  - etc.

1.4 Internet



All interconnected computers around the world.

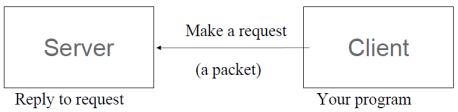
What is a Network? Wire, permits communication between computers. **LAN:** Local Area Network, computers in same room/building. **WAN:** Wide Area Network, computers in different buildings/cities, more than 1 server interconnecting big network nodes.



Client/Server

- Client sends request packet to server
- Server tries to find requested thing
- Server sends reply packet with data/program
- Client copies into memory and executes

Programs run on client, but programs stored on server, data is spread out. **MasterComputer/Main frame** is the opposite, programs stored & run on server, client just sees it. Additional terms: **Handshaking** consists of agreeing on packet format & passwords. **Comm-error**, packet lost, resend request or time-out.

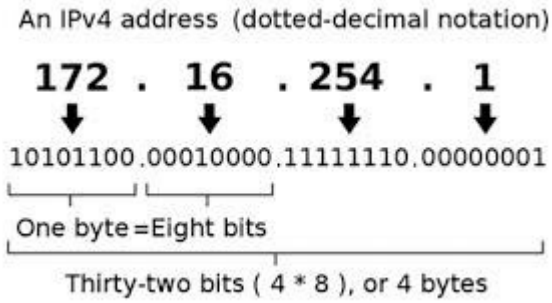


Service Providers Special server, knows locations of other servers on the Internet. Uses this knowledge to deliver package to right destination. Internet = WAN of service providers that are connected to independent LAN & WAN networks.

Web Subset of Internet with only interconnected ISP servers. Interacts solely with browsers.

Public_HTML Servers connected to Internet have a directory/folder called public_html, which is public. By default, only this is accessible from Browser. The default operation for the Internet is to display folders and files at that Internet address (like a file-browser), like this. Need an index.html or other stuff to change default behavior and display a “web page".

Communicating with Other Computers Each computer needs a unique ID, today we



use our **IP Address** as our unique ID number.

2 Unix

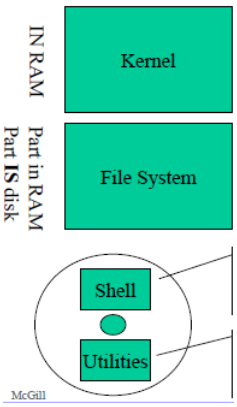
2.1 About Unix

History Failed OS by AT&T Bell Laboratories called Multics, 1^{st} OS with 2 windows & 2 programs at the same time. Ken Thompson was working on project, writing game called *Space Travel*. Ported game to PDP-7 when project was canceled. Wrote Unix to make it easier to port. 3 types: System V UNIX (based off original), BSD UNIX (based on Berkeley Software Distribution) and UNIX-like (behave like UNIX, includes **Linux**, which we'll be using).

General Info Unix is:

- Optimized/simple
- Password-based security
- Driven by command line
- Client-server

Unix is client/server! Companies buy one huge machine (server) and many small terminals (client). Can SSH into a SOCS machine at school.

OS Components Unix is a modular OS. 

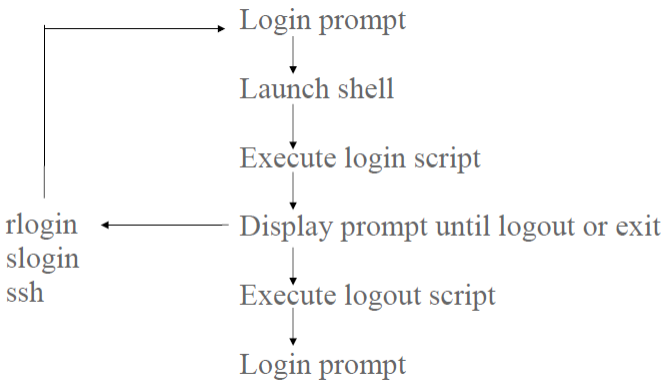
Several components:

- Kernel
  - In RAM
  - Handles logging in
  - Task switching
  - Manages everything
  - Basic interface
  - Drivers, run-time stack
- File System
  - How is disk drive formatted
  - FAT (File Allocation Table)
  - Data structure making files "real"
  - Everything is a "Program"
  - R/w to disk & peripherals
- Shell
  - More advanced UI
  - Global memory
  - Commands to interact with shell
- Utilities
  - Extra commands and programs
  - Drivers

Users Need an account on a Unix machine to use it. Consists of a user name and password. User gets a **home directory** (∼). Accounts are members of at least one group. Groups used for permission purposes. Every Unix machine has a **root** account, with ALL permissions.

Passwords Since all Unix security is based on passwords, important to have a good password. Breach consists of someone else logging in with your password. Shouldn't use dictionary words because there are dictionary attacks. Other attacks include getting to know the user and brute force. Want a mix of upper and lower case, numbers, punctuation.

2.2 Sessions



One session goes from login, loops the prompt many times until you logout. rlogin, slogin, ssh used to create another session within a login.

Environment Session Memory Has stuff like user name, home, shell, etc.
When you start the shell, OS sets up environment, a collection of variables which can be accessed from any application launched from that environment. env & set show you current environment variables. setenv and set are not used in bash to change an environment variable in Bash, just write var=value. Echo can give you specific env variables, like echo $var.

3 Bash/Command-line

Command-line Prompt Basically a while look that keeps reading the input you give it until you logout or exit.

Syntax Program -switches arguments
Program is any command or executable. Switches are parameters that modify program's execution, arguments are fed to program.

3.1 Good Commands to Know

Command	Switches/Args	Examples	Usage
ls	-l (long output) -a (all, hidden) [file/dir]	ls ; ls -l ; ls text.txt	lists files and directories
whoami			Tells you user logged in as
who			Who is logged on server
exit			exits shell
logout			logs out of session, can logout of a nested session
finger	username@host	finger jlore	info on user
ssh	username@host	ssh jlore@mimi.cs.mcgill.ca	ssh into this user
mkdir	directoryname	mkdir test	make directory
cp	filename destination file	cp a.txt b.txt ; cp a.txt /jl	copy a file
cd	[directory] [..]	cd test ; cd .. (goes up 1 dir) ; cd (goes home)	change dir
cat	file	cat test.txt	reads text, concatenates files, can read 2 at once
rm	file	rm test.txt	removes a file
sort	-(reverse alphabetical)		sorts text alphabetically
pwd			prints working directory
rmdir	directory	rm test	removes EMPTY directory
chgrp	group file	chgrp friends test.txt	changes group of file

chmod	u(user)g(group)a(all) +(adds perm)-(removes)=(replaces) rwx 421 file (0=no perm 1=exec) (2=write 4=read)	chmod u=r test.txt (changes user to read) chmod 000 test.txt	change permissions
chown	owner file	chown jlore test.txt	changes owner of file
mv	file1 file2	mv test.txt /jlore/test.txt	moves file1 into file2
echo	text/string	echo hi	echoes string to std-out
head	[-number] file	head -2 test	display first n or first few (if no number) lines of file
tail	[-number] file	tail -2 test	display last n or last few lines of file
more	file	more test	page through file, enter advances a page
less	file	less test	navigate through paginated file, better than more
man	command/program	man ls	get manual page for a program
date	[options]		gets current date & time
du	[options] [dir/file]	du source/	shows disk space usage
hostname			Gives name of machine
uname	[option]		Prints system info

script	file	script record.txt	Records everything appearing on screen (IN/OUT) to file until exit or ctrl-D
which	command	which ls	Shows path where command is located
kill	[options] [-SIGNAL] [pid#]	kill -9 1234	Kills a process id, -9 is sigkill, aggressive/forceful kill
ps	[options] -a (all processes/users) -e (environment) -g (group leaders too) -l (long) -u (shows user, also more info like % resources) -x (includes things not run from terms) -f (full)	ps -a	Shows status of active processes
top			Monitors resource usage of active processes
tar	-c (create new) -r (update) -x (extract) -f (archive name) -v (verbose) -z (compress using gzip) (files/dir)	tar -cvf log.tar *.log ; tar -zcvf log.tgz *.log ; tar -xvf log.tar /tmp/log	Archive manipulation using tar, see 3.2
diff	[op] file1 file2	diff text text2	compares 2 files
file	[op] file	file text	What “kind" of file?
find	[op] [path] expressions	find test	Finds files matching pattern

ln	-s (soft link) source target	ln fav.txt read	Links source to target. Default is hard link, gives another name to a file. Soft/symbolic link is an indirect pointer, does not affect target file.
paste	[op] file1 file2	paste text1 text2	combines 2 files, one after other
touch	[op] [date] file	touch text	Create empty file or update access time
wc	[op] file(s)	wc text	word count
write	userid	write jlore	Sends text message to someone ^D to end, mesg -y/n to turn on/off
wall			write to all
grep	[op] -i (ignore case) -c (return only count of matches) -v (invert, display ones that don't match) -n (adds line number of match) -l (filenames of matches) string files	grep hi text	Search for occurrences of String, supports regex
sed	[op] -i (interactive, update current file) files	sed -i '1d 'text (deletes first line) ; sed -i '1iHello 'text (inserts Hello at 1st line)	stream editor
awk	[op] files	awk 'NR==1'text (reads first line)	scan for patterns in file

clear			clears screen/prompt
expr	math	expr 1 + 2	Does integer math

Common switches (mainly cp, mv, rm)

. -i: interactive
prompt and wait for confirmation

. -r or -R: recursive
visits directory recursively, visits files and subdirs

. -f: force
don't prompt for confirmation, overrides -i

Killing ^Z (ctrl-Z) forceful kill
^C (ctrl-C) gentle kill

3.2 Files & Directories

Hidden Files/Folders : Start with ., i.e. .config
Can be seen with ls -a

Wild cards

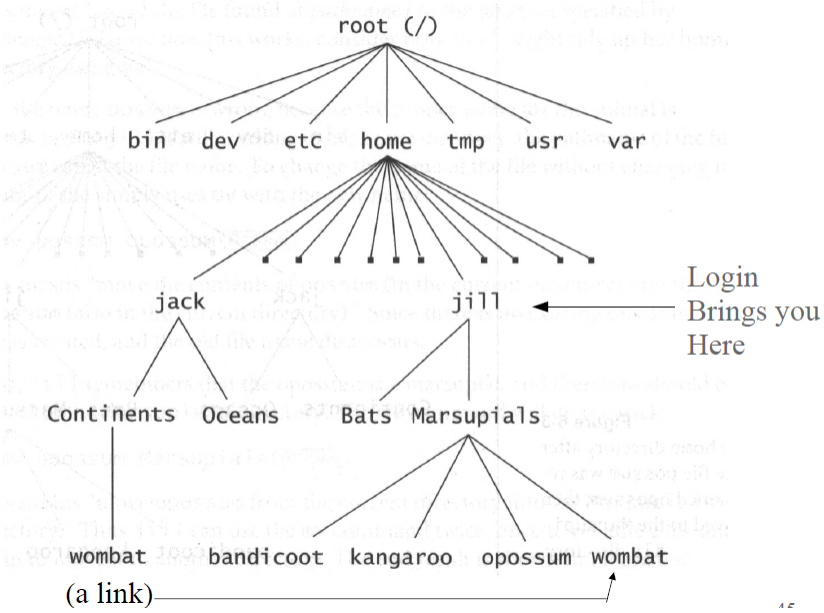
*	anything
?	any single char
[abc]	a or b or c

Directories Directories are folders. Use same permission system as files. Root denoted by /
Common Unix system has following directories:

- /etc : config files, pws
- /bin : OS executables
- /usr : Application installations
- /opt : Another application installation dir

- /dev : Device files for hardware
- /var : Files that vary a lot, like logs

Typical Directory Structure



Absolute vs Relative Paths

. Relative paths: Path from your current directory

. Absolute paths: Path from root

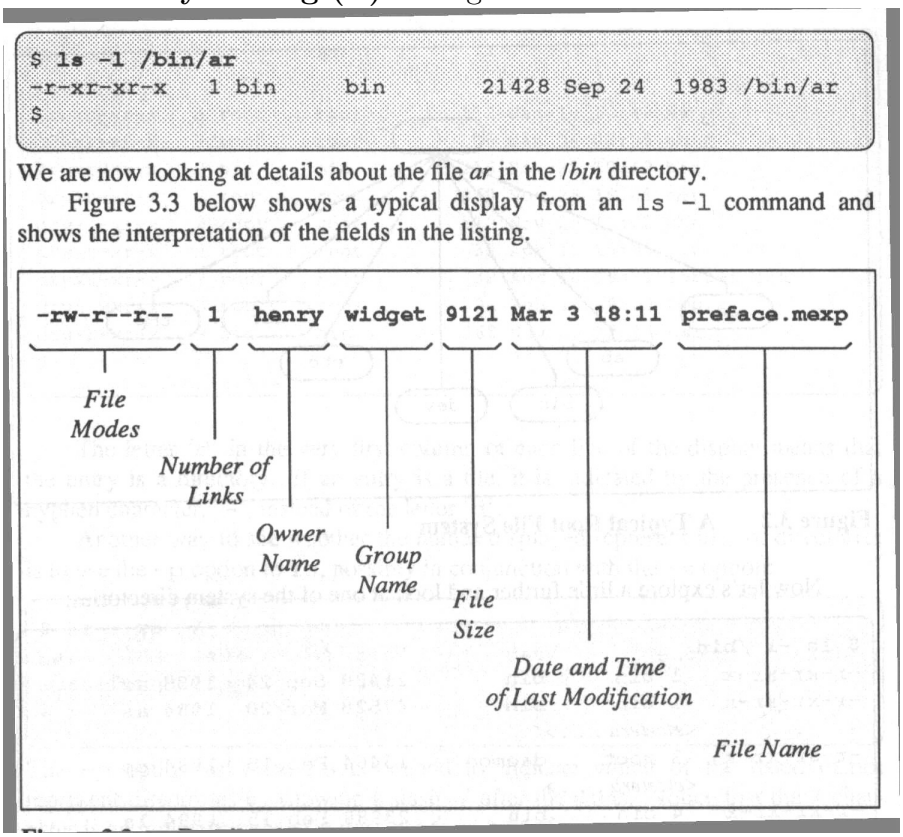
- ./folder from current directory, same as folder
- ../folder parent directory one up

File Descriptors Created by OS when file opened. Reference to that file. Unix has 3 special file descriptors that are always opened.

- STDIN 0 : keys typed by user gathered here
- STDOUT 1 : normal application output sent here
- STDERR 2 : where error output is sent

Permissions Three levels, user, group and other. 3 types of rights, read, write and execute. Can give any combination of these rights to the 3 levels. Permissions usually shown by string of 10 characters, first is if directory or not, then next bunches of 3 are rwx for 3 levels.

Directory Listing (ls) Long format:



Overlapping If all/other have rwx, but everyone else has nothing, owner and group cannot rwx, unless the Unix system interprets other as all.

Archives TAR, GZIP & GUNZIP. Archive is a collection of files combined into one file, often compressed. tar combines, gzip compresses.

3.3 Redirection

Output > redirects STDOUT to a text file
>> appends STDOUT to a text file.

Input < takes all input from a file
To redirect something specific, can prefix by file descriptor, i.e. prog 2>errors, redirects errors from prog to errors file.

Piping/Chaining

. Piping commands: redirects STDOUT to another program. i.e. (ls | more) paginates ls output

. Doing multiple commands in SUCCESSION: (ls ; echo hi) does ls then echoes hi

. Doing multiple commands at ONCE: (ls & echo hi) does ls and echo at the same time

3.4 Quotes

- ` nested execute symbol ` , executes what's in between first → string
- 'as is '
- "pre-process "

Escape Characters

- \ ← escape character
- \n new line
- \t tab
- \a bell (noise)

3.5 Editors

Command line text editors allow for creation/editing of files at CLI.

- vi, one of the original ones on Unix, hard to learn. Available on every Unix machine.
- pico, simple, based on pine mail client, available on most
- emacs, popular, powerful, heavyweight

There are also graphical text editors.

Vi Different modes:

- Edit (i, a, o, O, etc.)
  - Edit text
  - Can press any char

  - Some vis let you use arrows

- Escape Mode (ESC)
  - Stops edit
  - Can use arrow keys
  - Can use special one letter commands (i,a,h,j,k,l,etc.)
- Command Mode (:)
  - Can save, load, quit, etc

ESC:

- dd deletes a whole line
- x deletes current char
- r replaces current char by next char types
- / to search

Command:

- w writes
- q quites
- wq both
- q! quit without saving
- e filename to edit file

3.6 Regular Expressions

Some commands/editors allow you to search text patters, known as regex. See grep, sed and awk in 3.1 for info about those commands.

Examples

- grep -i '^[aeiouy]'text , want first letter to be a vowel
- '[aeiouy]$ ', want last letter to be vowel
- '[aeiouy]{2,} '{min,max}, want it at least 2 times, 2 vowels next to each other
- '^.e ', Start with anything then an e
- '^e|a ', start with e or a
- '^[a-e] ', start with anything from a to e on unicode table

4 Bash Scripts

Scripts are collections of commands grouped in a file to execute in a sequence. Not compiled, interpreted. Run from top to bottom, can alter flow with if statements and loops. Can also create functions (before called). # indicates a comment. Scripts sensitive to spaces. Good uses for scripts:

- Backups
- Startups
- Scheduled
- Maintenance
- Programmer

Boot & Login Scripts Modify OS environment. Boot made by root for all users, login scripts created by users for themselves. At login, shell looks for default login script.

Login Scripts Used for:

- Configure UI (prompt, color)
- TERM communication method, how server speaks to computer
- Routines

In Bash, to change your prompt:
PS1=“Something here"
Aliasing commands
alias lsa='ls -a'

PATH Set of directories a shell searches for executables, separated by (:)

Command-line Scripts Created by users to automate command-line things. Everything is a text file, text files can be executed, just add x permission using chmod. Launch a script in current directory using ./myScript

4.1 Shell Scripting

Start off your script with the sha-bang #!, to show that script is directly executable and specify shell language/path. #!/bin/bash

Variables 3 kinds in a shell script

- Environment Variable, used to customize OS, used by shell
- User-created, created by script itself
- Positional Parameters, store what was used to start script

Positional Variables

- $# number of args on command line
- $- options supplied to shell
- $? exit val of last command executed
- $$ process number of current process
- $! process number of last command done in background
- $n argument on command line, n=1-9
- shift shifts all arguments by 1, lose $0, but get $10
  - $0 name of shell/program
  - $* all args on command line as 1 string

- $@ all args, separately quoted with spaces

Declaring variables: Just write x=10

Some Default Variables

- $HOME
- $SHELL
- $TERM
- $USER
- $PWD

Reading Use the *read* command to read a string from STDIN. i.e. read name $\implies$ $name now stores whatever string the user typed.

Capturing Complex Output If you want to parse multiple args from a command: set ` date ` will store output in $n ($1,$2, ...). Will erase data already there.

Arithmetic Use, either expr, bc or tell Bash it's math. To make Bash treat Strings as a number, do:
$((1+1)) or $[1+1]
For fractions, use bc. echo "scale=#ofdecimals;3/4 "| bc

Conditionals Use the test command to evaluate an expression or case. Bash does not require the test command though. Can evaluate at file, string or integer level.

File Tests

- -r file : exists + readable
- -w file : exists + writable
- -x file : exists + executable
- -f file : exists + regular
- -d name : exists + directory

- -h or -L file : exists + link
- etc.

String Tests

- -z string : string length zero (Not null, but 0)
- -n string : length non-zero
- string1 = (or ==) string2 : strings identical
- string1 != string2 : not identical
- string : not null

Integer Tests

- n1 -eq n2 : integers equal
- n1 -ne n2 : not equal
- n1 -gt n2 : $n1 > n2$
- n1 -ge n2 : $n1 \geq n2$
- n1 -lt n2 : $n1 < n2$
- n1 -le n2 : $n1 \leq n2$

if Statement

```
if _condition_
then
        stuff
elif _condition_
then
        stuff
else
        stuff
fi
```

case Statement

```
case what_to_check in

condition1) action1;;
condition2) action2;;
*) else_action;;
esac
```

for Loop Iterates.

```
for var in list
do
        stuff
done
```

while Statement Continues until statement false.

```
while condition
do
        stuff
        continue (back to beginning)
        break (end loop)
done
```

```
function() {
stuff
}
```

Functions must be declared before called.

Etc. Good practice to exit scripts with exit codes, 0 for no errors, 1-255 for errors.

5 C

5.1 About C

History Successor for B and BCPL. Creation parallel to development of early Unix OSes (1969-1973). Good because it's portable and can access hardware, lower level language. C++, successor to C. Java, C# and JavaScript based on C.

Comparison to Java Same as Java for:

- If
- For loop
- While/do-while loop
- Methods (called functions)
- Types : int, float, double, char
- Variable creation
- Mathematical and logical expressions

Mathematical operators (+,*, etc.) exact same as Java. Type casting sometimes implicit, like int → float, but can cast specifically like in Java.
Similar to Java for:

- Arrays
- References (pointers)
- Main method (main function)
- Scope

Different from Java for:

- Strings
- No objects (but there are modules)
- Libraries
- Pre-processor, compiling in different languages, Eng/Fr, etc.

5.2 C Program Structure

```
#include <stdio.h> // C library, always need stdio.h
int main (int argc, char *argv[]){
//main function, *argv[] are command-line parameters
        printf("Hello World \n"); //prints
        return 0; //return if there's an error or not to calling program
}
```

int argc consists of number of arguments at command-line (including argv[0], prog name)
char *argv[], each cell has one command-line parameter.

5.3 Format of Types for Strings/printf,etc.

To input a char or String or int or something else indirectly into printf, have to declare the spacing of it with %.

$$\%\overbrace{\#}^{\text{int, optional. How much space?}}\underbrace{c}_{\text{type, c=char, s=string, d=int, f=float}}$$

More control codes:

- %d or %i : signed integer
- %x : unsigned hexadecimal
- %u : unsigned decimal
- %E : double of form m.ddExx

Ex.

```
printf("user %s hello \n", name);
```

Prints user bob hello, if bob stored in name. %10s represents 10 white spaces, first x amount with variable. If you write less numbers than the length of the String, it won't truncate, will just use length, like %s.
For floats, %5.2f, 5 is total amount of numbers, .2 is number after decimal.
x++ increments after operation (x++-3=x-3+1), ++x increments before (++x-3=x+1-3)

5.4 Libraries & Functions

Functions If you want to write the code for a function after calling it, have to use a function prototype, that is, declare the function at the beginning, but write the code later

(telling compiler to look for the function). Names of variables in prototype don't have to match, but return type has to match.

```
void add (int,int);
.
.
.
void main(void){
int z = sum (5,10);
}
.
.
.
void add (int a, int b){
int x=a+b
}
```

You **CANNOT** overload functions in C, counts as conflicting signatures (function name, return type and parameters).

What are libraries? Toolbox for common routines that are often optimized and speed up development time. They allow you to create reusable code to share with others, but can make code size large. Include at beginning of code. Can also include things like headers or c code.
Made up of .h, the header file with function prototypes & typedefs and .o, compiled from .c without a main

stdio.h

function	**example(s)**	**use**
printf("stuff")	printf("Hello World"); printf("%d ", 25);	Prints to screen
scanf(a)	scanf("%d %d ", &age1, &age2);	scans variables from user
int getchar(void)		gets on char from STDIN
int putchar(int)		displays one char to STD-OUT
gets(array)		reads string into array

fgets(char array, int size, *stream)		reads string of certain size from file into array
int getc(any stream)/getchar(STDIN only);		read one char
int puts(*string)		place char
remove (*file)		
rename(*file1, *file2)		
fopen (filename, mode [rt,wt,at])	fopen (text.txt, "wt ");	opens file ptr
fprintf(ptr, description, vars)		
fscanf(ptr, description, &vars)		
feof(ptr)		0, 1 if end of file
fclose(ptr)		closes file read/write, adds EOF char(write)
fflush(ptr)		empties buffer without writing

scanf anomaly: Does not process carriage returns properly, viewed as string or char. When you read numbers, carriage return is left in input buffer! You should scan numbers after all strings, unless you have intermediate garbage (char) scans to scan leftover carriage returns.
scanf limiting vs gets, need specific amount of args/words, get gives you the whole line.
sprintf & sscanf do same thing as printf & scanf, except they return how many they were able to read/print

stdlib.h

- NULL 0
- EXIT_FAILURE 1
- EXIT_SUCCESS 0
- rand(void), 0 to RAND_MAX

- system(string), sends command to system, command-line (launches a new shell), returns int, exit code (can check if BASH had an error)
- atof(string), ascii to float
- atoi(string), ascii to integer
- abs(int), abs val
- exit(int), 1 or 0

math.h

- sqrt(double);
- power(base,exponent);
- abs(int);
- fabs(double);
- floor(double);
- ceil(double);
- sin, cos, tan, asin, etc.

ctype.h

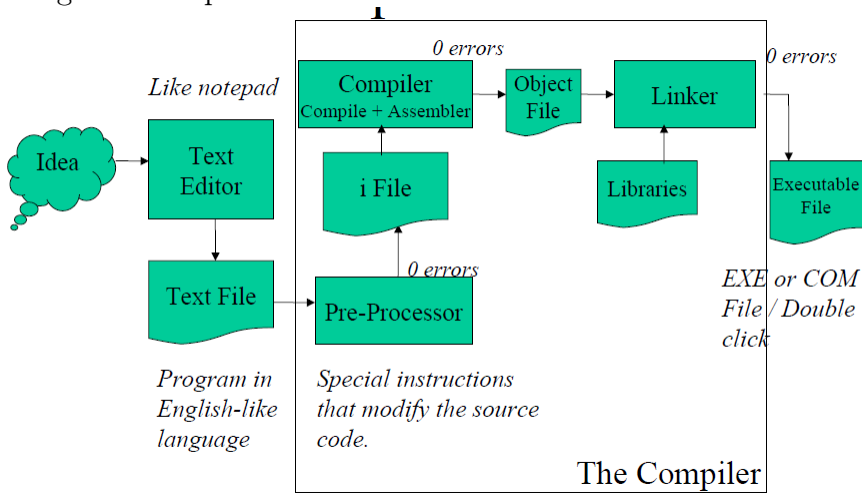
- toupper(int);
- tolower(int);
- isalpha(int);
- isalphanum(int);
- isdigit(int);

string.h

- strlen(string)
- char *strcpy(char *dest, char *src), copies string
- char *strcat(char *dest, char *src), concatenates string
- strcmp(char *s1, char *s2), compares contents of 2 strings (instead of addresses), 0 if same

5.5 Compiling

Use gcc to compile. Default executable name is a.out.



C Files

- Source files: file.c (program). file.h (header, shared)
- Pre-processed: file.i
- Object & assembler: file.o, file.s
- Executables: file, a.out

GCC Syntax GCC is the GNU C Compiler.
gcc -o executablename sourcefilenames
Variations include:

- gcc -E main.c, also make main.i

- gcc -S main.c, also make assembler code
- gcc -c main.c, also make object code
- gcc main.c, make a.out
- Extra switches:
  - -o filename, specify name of output
  - -v verbose
  - -w suppress warnings
  - -W extra warnings
  - -Wall all warning messages
  - -O1 optimize for size and speed
  - -O2 optimize even more

C Types

DESCRIPTION	RESERVED WORD	BITS	RANGE
Integer	short	8	- 128 to + 127
	int	16	+/- 32,768
	long	32	+/- 2,147,483,648
Floating Point	float	32	+/- 3.4 x 10^{38} with 7 significant digits
	double	64	+/- 1.7 x 10^{308} with 15 significant digits
Boolean	short, int, long		0 is false, other true
Character	char	8	0 to 256
String	char *	32	address in memory (special case of pointer)
Pointers	TYPE*	32	address in memory

Notice that booleans don't really exist (they're just ints) and Strings are char pointers. Strings end with a null.

Booleans 0 = false, anything else is true. Greatly affects if statements, as your condition can solely be math and it'll check if it's 0 or not. So you won't have a compilation error for something like x=5, but very different from x==5.

Strings Strings are just character arrays terminated by a null. Can declare some like: char *x = "Bob ";
Cannot scanf into a *, as * constant/fixed to what they were when declared. When scanning into a char array, don't use &.

Arrays int x[10];
Can also declare literally and also leave extra commas to denote extra spaces.
int x[]={1,2,3,4,,,,,,}
In C, you can go past the indexes of an array.
char x[10]; → [char][char]...[char]
char *y[10]; → [ptr][ptr]...[ptr]

Declaring a Variable
SCOPE MODIFIER TYPE VAR_NAME;
SCOPE is static, extern or not used. MODIFIED is unsigned(no signed bit), short($\frac{1}{2}$ bits), long(2× bits) or not used. TYPE, a built in type.
Variables are **NOT** defaulted to 0.
Can also chain assignments, int a,b,c,d=4;
Can also declare constants
int const a = 1; or const int a = 2;
Variables declared at either top of file (global variable) or in a function (local variable). Global variables are positionally global, accessible to everything below it. Only one copy of a global variable exists. Global variables should be avoided if not needed, hard to debug and not considered clean.
Variable preference: block vars → local vars → global → extern → error/compiler error

5.6 Pointers

Special unsigned integer storing an address. Can reference anything really. Mainly used to point to a variable or location in RAM. Note that a pointer is not the same thing as a reference, a reference is a Java concept.

Pointer Operations

- & unary/monadic operator, gives address of variable
- * indirection/dereference, gives contents of object pointed to, declare pointers using *

	content	address of
int a	a	&a
int *p	*p	p

```
int a,b,*p;
```
p=&a

p will now point at a's address.

```
int a, b;
int *p;


a = 5;
b = 10;
p = &a; // p is pointing to a
*p = 6; // Value of a is now 6
p = &b; // p is pointing to b
*p = 11 // Value of b is now 11;
```

Giving type to your pointer means it can only point to that type. void * can point to anything.

```
char* x,y,z; // all vars are char*
char x, *y, z; // only y is char*
```

Pointers are not constant, can change (since they're variables), unless you declare explicitly, like a String (then it's static/literals).

```
char *string = "bob"; // Literal / static
char array[10]; // Variables / dynamic
```

void* Pointers void pointers can point to anything, anywhere. They are the bit-size of addresses.

```
int x=5,y;
void *p;
```

```
p=&x;
y=*p; // Warning
y=*(int *)p; // Cast
```

Arrays When you create an array in C, you're allocating a block of memory and creating a pointer to the first element of the block (like using malloc).

```
SCOPE MODIFIER char* VARNAME = "characters"
```

Going to different indices in a String/pointer, add the offset.

```
*(str+2); // 3rd char of the String
```

If we don't terminate a String by a null, it will keep going until it hits a null (when trying to read it).

5.7 Parameter Passing

- Primitive types are passed by value. You get a local copy of the variable for the function.
- Arrays are passed by reference, affects the whole program if you change them in a function (same for pointers).
- Passing a parameter with the unary (&) operator makes it pass as a pointer, which is how scanf works.
- You cannot make a function that swaps primitives **unless** you use pointers to the primitives.

5.8 Switch Statement

Check variable for equality against a list, check for each switch case.

```
switch(expr){
    case const-expr:
        printf("hi");
        break; // optional
```

```
    case 2:
        printf("It's a 2!");
        break;

    default: //optional
        printf("None");
}
```

5.9 Reading/Writing to a File

Files/text, displayed in 2D but actually 1D structure, like an array. The Disk actually consists of the FAT (File Allocation Table), a bunch of name of files and pointers. To read/write to a file, need to get a pointer to disk.
Modes for opening files (fopen):

- r: opens file in read mode
- w: write mode (if none, creates. If existing, deletes)
- a: append mode (if none, creates)
- r+w: read and write (not destructive)

```
#include <stdio.h>
int main(int argc, char* argv[]){
        FILE* out = fopen("test.txt", "wt"); // wt for writing text
        fprintf(out, "Hello World\n"); // Write Hello World to file
        fclose(out); // Close the file opened, end with EOF char

        FILE* in = fopen("test.txt", "rt"); // rt for reading text
        int numbers; // Int for number of chars in file
        while(fgetc(in)!=EOF){ // Get char until EOF
                numbers++; // Keep incrementing as long as characters are val
        }
        char file[numbers]; // Char array for numbers amount of chars
        rewind(in); // Rewind to the beginning of the file
        for(int i=0; i<numbers; i++){ // Loop through file
```

```
            file[i]=fgetc(in); // Get each char one by one
        }
        fclose(in); // Done with file now
        printf("%s", file); // Print the resulting String loaded
}
```

Named Space Location where data is stored, these locations have names.

5.10 Pre-processor

The pre-processor takes care of things like directives. Takes the source.c file and makes it into a source.i file, which is then fed into the compiler.

```
#include<stdio.h> // <> -> in library folder
#include "path/textfile" // ""->path
```

What include does is basically insert source file at that location.

```
#define NAME EXPRESSION //format
#define MACRO EXPRESSION //format
```

EXPRESSION consists of any legal C expression. MACRO follows: NAME(PARAMETERS), with params being a comma separated list. Convention is to name something defined in all caps so we know it was defined using #define. Notice that FILE is written all in caps, was defined in fileio.h.

```
#define LIMIT 10
int array[LIMIT]; // Allows something like this -> array[10]
#define BOOLEAN int
#define TRUE 1
#define FALSE 0
// Macros
#define INC(x) x++
#define MAX(A,B) (A<B)?B:A
// becomes B if true: A if false
```

Define vs const const int a = 5; can be typechecked(safer) by compiler, but uses extra memory.

Ifdef Check if something is/isn't defined. Must end by #endif. Good to protect against defining same thing twice. Can also include stuff like function prototypes/code. Can put these in the middle of your code. Useful for writing a multilingual program. ifdef the language everywhere and print corresponding language's output. If the program defines that language in the beginning, you'll get the program in the language.

```
#ifndef AGE // If age isn't already defined (possibly from include)
#define AGE 20
#else // optional
#endif
```

Using the Pre-processor for Collaboration Multiple programmers on one team, all working on the same project.

Single Source File Programming Have all the .c files in the same directory, include them all in main.c, single source because only one .i file results from this. Everything except local variables are global (no named spaces), all compiled in same place.

Header Files C code is separated into source (.c) files and header (.h) files. They contain:

- Pre-processor commands
- Global variable declarations (extern)
- Function prototypes

Extern Modifier To use a variable from another file/source, write extern type varname; at the top of source code (can put in header file to know which vars are supposed to be global). The extern modifier lets a named-space access data in another named-space (compiler will look in other source files, can only have one other declaration of this variable in all yoru source files). Assumed on functions.

```
// Math.h would look something like:
extern double PI;
int factorial(int n);
```

Modular Programming Everyone has their own (hidden) named space. Make a .c source file, a .h header file, which includes function prototypes for functions that you allow others to use in their source. .h files included in sources that need functions from those parts (main should have .h for each file).

```
gcc -c f1.c # Gives resultant f1.o from preproc
```

.o files are semi private, have to reverse binary/assembler to see. To compile the final project:

```
gcc main.c f1.o f2.o
```

Lots of .o files to include. Can make a BASH script ot do this. Another way of fixing stuff is using a **makefile**.

5.11 GNU Tools

Make An automated build utility. Determines which pieces need to be recompiled and issues corresponding commands. Make can be used with any language. Can use variables like in BASH.
Advantages: Allows us to define multiple ways to compile, has scripting ability. Greatly speeds up compile time with selective compilation.

```
myProg: main.o f1.o f2.0 # If these exist, exec line below
    gcc -o myProg main.o f1.o f2.o
    # Indents mandatory, can put multiple commands
main.o: main.c f1.h f2.h
    gcc -c main.c
misctag: #i.e. make misctag, make clean to remove compiled files
```

Makefile reads from bottom to top. Also checks date. If source code is newer than compiled file, will recompile that section, i.e. main.c newer than main.o, recompile main.o. Make has an implicit rule where it will update .o file from corresponding .c file (i.e. you don't need to write the .c file in the conditions if it has the same name), but it's better not to use the implicit rule.

Terminology

- **Unit**: collection of files compiled into a single .o file. (.c and .h)
- **Project**: collection of units linked into a single executable file. (collection of .o files)
- **make**: GNU program that compiles projects

- **makefile**: text file containing script

In order to use make, have makefile in the source directory and run "make".

Repository Database that stores all versions of files you've been working on. Lets you revert to previous versions and branch out and experiment with different ways of developing files. Repositories store any kind of files. Also lets you merge a source file if 2 people edited it at the same time and can password lock source files.
Repository Tools:

- RCS (root based)
- CVS (root based)
- SVN
- GIT (peer based)

Root-Based Code Repositories Code is shared online, users have access to it.

Peer-Based Code Repositories Everyone has their personal part, which is staged (git add <files>) which gets committed to the local repository (git commit) and pushed to the remote repository (git push)

Using Git root(1.1)→trunk(1.2)→branch(1.2.1)→tip(1.2.2) or trunk(1.2)→tip(1.3), tip is the most recent version, trunk is intermediate, root is original and branch is a separate version/direction.

```
# Move to directory with code to manage with git
cd /project/ass1/source
git init #Initializes repository
# For remote repo download:
git clone git@github.com:repo.git
# Pick files (Stage)
git add f1.c f2.c
git commit # Checks in
```

Basic Git Commands:

Command	Info

git init	setup repo
git add	add files for next commit
git commit	commit queued files
git commit -m “Message”	Add message to commit
git push	push commit(s) to remote repo
git pull	fetch changes from remote
git clone	clone repo into local dir
git status	show uncommited changes
git rm	remove file from repo
git mv	move file within repo
git diff	differences between multiple commits
git log	log of commits

Add files to .gitignore to ignore specific files.

Branching

Command	**Info**
git branch new_branch	create new branch
git branch -b new_branch	create and switch to new branch
checkout master	switch back to master
git merge branch2	merge work, all commits from branch2 to one commit in current
git rebase branch2	rebase your current branch off updated branch2, incase unrelated things updated in branch2. Good to rebase before merging, to see if anything breaks before merging.
git branch -d branch2	delete branch

Tagging (adding releases)

Command	**Info**
git tag -a name	Add tag with this name
git tag -l	Lists tags
git push –tags	Push tags to remote

Modifications:

Command	Info
git commit –amend	Change last commit
git reset HEAD file_name	Unstage staged file
git checkout –file_name	Unmodify modified file

GNU Debug (GDB) A software bug is an error/flaw/mistake/failure/fault in program preventing it from working as intended. Bugs can exist at design level or source code level. Side-effects: incorrect answer, crash application, loss of data, loss of money, loss of life.
Debugging = finding the source of a bug & fixing it. Hardest part is finding the problem (becomes increasingly difficult when source is large). Analyzing code sometimes won't cut it. You might need to run the program. Debuggers help the debugging process. Allows programmer to run program in different mode. Many new features available to programmer. Can see what line causes a program crash, analyze application line by line, consult content of memory. Most languages have a debugger. printf is useful, but can't do everything that a debugger can (pause program, list source code, print data, jump to a line of code).
GDB is part of the GNU OS, can be used for C, C++, Objective-C, Fortran, Java and Assembly programs.
To use GDB, compile executable with the -g flag. Compiles the program with extra info, like source code and symbol table.

```
gcc -g -o HelloWorld HelloWorld.c
# Then call gdb on the executable
gdb HelloWorld
# You will now be in "gdb" mode
(gdb) help # Gets list of available commands
# Can also get help on specific command
(gdb) help breakpoints
```

To look at Source code, use the list command. Takes either a filename and line bumer or function name or just line number.

```
(gdb) list main.c:10
set listsize # Changes size of a list
(gdb) run # Runs code until crash
# Can also run line by line
(gdb) step
(gdb) next
```

Step & next both do the same thing, but step will step into a function call, next will execute the function call, but not step through.
More GDB commands

Command	**Info**
quit	end gdb
list	shows 10 lines (default)
list n,m	show lines n to m
list function	show function
run	runs program
ctrl-c (during run)	interrupt program
run -b <in> out	redir in & out to prog
backtrace	see run-time stack
whatis x	show x's declaration
print x	show value of x
print fn(y)	execute fn with y
print a @ n	print n elements of array a
break n	puts a breakpoint at line n, stops there during run
break n if EXPR	break at line n if true (i.e $i = 99$ for loop), can do same for fn
break fn	interrupt program at function call
break filename:lineno	break at line number of source file
continue	continue after break
watch EXPR	stop program when expr is true
set variable NAME = VALUE	change contents of variable
ptype NAME	pretty print of NAME (as struct)
call fn(y)	execute fn with y
info breakpoints	gives info about breakpoints
delete n	deletes breakpoint n
delete	deletes all break points
clear n	clear break or watch on line n
disable n	disables break on line n
enable n	enables break on line n
enable once n	turn on once
where	produce backtrace

DDD A graphical front-end for command-line debuggers like GDB, DBX, JDB, Python debugger, etc. Can do four things:

- Start program
- Make program stop on conditions
- Examine what happened
- Change things in program

Lint Lint originally flagged suspicious/non-portable constructs/bugs in C. Now the name signifies any tool reporting suspicious behavior like:

- Variables being used before set
- Conditions that are always true/false
- Calculations whose result is likely to be out of range of values representable by type used

GProf Program can be unacceptably slow, takes longer to complete operation than desired. Program needs to be optimized, so it can accomplish same tasks with less resources. Behavior of program should be unchanged. We need to be able to identify portion that is slow.

Definitions of Slow

- **Big-Oh** Logical speed of algorithm. Pro: Independent of hardware. Con: Long to do on large pieces of code.
- **Clock Time** Actual speed corresponding to hardware (MHz, RAM, devices). Pro: Can be automated. Con: Hardware dependent.

Profilers Dynamic performance analysis tools. Lint are static analysis tools. Profilers record data as application executes. Usually tells you what part is slow/uses a lot of memory.

```
gcc -pg file.c #Compile file with timechecks
```

After running the program, a gmon.out binary file will be created.

```
gprof -b a.out gmon.out > statsfile # Read results
```

The output will be messy/long, so better to redirect to a text file.
Options:

- -b note verbose
- -s merge all gmon files
- -z table of functions never called
- -a don't print statically declared (private) functions
- -e funcname, don't print info about this function
- -E funcname, like -e, but time spent in function won't be printed
- -f funcname, only print specified function and children
- -F funcname, only time spent

Flat Files Here's a GProf flat file sample, running bubble sort on 100,000 numbers on a relatively slow computer.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 63.31     63.07    63.07        1    63.07    99.32  bubbleSort
 36.39     99.32    36.25 2498010236     0.00     0.00  swap
  1.02    100.34     1.02        2     0.51     0.51  printArr
  0.01    100.35     0.01        1     0.01     0.01  genArray
```

Column	**Info**
% time	Percentage of time program spent in this function, should add to 100.
cumulative seconds	Total number of seconds spent executing this function plus time spent in functions above this one.

self seconds	Number of seconds(or milliseconds) form this function itself. Flat profile sorted by this number.
calls	Number of times the function was called, blank means none.
self s/call	Average number of seconds (or ms) spent in the function per call
total s/call	Average number of seconds spent in this function and its descendants per call.
name	Name of function. Sorts this alphabetically after self seconds is sorted.

Call graph

```
granularity: each sample hit covers 2 byte(s) for 0.01% of 100.35 seconds

index % time    self  children    called     name
                                             <spontaneous>
[1]    100.0    0.00  100.35                 main [1]
               63.07   36.25       1/1          bubbleSort [2]
                1.02    0.00       2/2          printArr [4]
                0.01    0.00       1/1          genArray [5]
-----------------------------------------------
               63.07   36.25       1/1          main [1]
[2]     99.0   63.07   36.25       1          bubbleSort [2]
               36.25    0.00 2498010236/2498010236     swap [3]
-----------------------------------------------
               36.25    0.00 2498010236/2498010236     bubbleSort [2]
[3]     36.1   36.25    0.00 2498010236          swap [3]
-----------------------------------------------
                1.02    0.00       2/2          main [1]
[4]      1.0    1.02    0.00       2          printArr [4]
-----------------------------------------------
                0.01    0.00       1/1          main [1]
```

```
[5]     0.0    0.01    0.00        1          genArray [5]
```

```
Index by function name

   [2] bubbleSort              [4] printArr
   [5] genArray                [3] swap
```

Primary Line: Describes function

Column	**Info**
index	Entries numbered with integers.
% time	Percentage of total time spent in function (includes time spent in subroutines called by function, main should be 100).
self	Total amount of time spent in function, sohuld be indetical to number in seconds field of flat profile.
children	Total amount of time spent in calls made by function.
called	# times function was called/# of non-recursive calls form all callers

Function's Callers: Function has a line for each function it was called by.

5.12 Invoking Programs

Already saw that we can call a new shell with the system command. But we can also use int fork();, need #include<unistd.h>. Fork will make child process, duplicates current program and runs from after the line that forks, with concurrent execution of parent and child processes. Returns PID of child for parent, 0 for child. If parent dies before child, child may terminate prematurely. Need to use wait function with #include<sys/wait.h>.

Main Program (Driver) Creates & manages children, but they do all the work. Fork copies all variables except return val of fork, so you can use if statements to check if child or parent. Since previous variables are maintained, parents can speak to children, but not the other way around (unless you use something like a textfile to communicate).

```
int status=0;
int pid = fork();
if(pid == 0){ // Child
    //Do stuff
}
else if(pid<0){ //For failed
    status =-1;
}
else{//Parent
    while(waitpid(pid,&status,0)==0); // Loop until done
}
```

5.13 C Data Structures

struct, creates a type. Kind of like a class without functions (unless you implement pointers to functions).

```
struct STUDENT{
    // Fields
    char name[30];
    int age;
    double gpa;
}x; // Variable name is optional, will create one if specified
// Semi-colon is mandatory though
```

We can then use the dot operator.

```
x.age=19;
```

To declare a struct, we can use a pointer or a literal.

```
struct STUDENT z;
struct STUDENT x[30]; //For 30 students
z.age=20;
// or, using pointers:
struct STUDENT *p;
// Allocates memory the size of STUDENT structure
p = (struct STUDENT*)malloc(sizeof(struct STUDENT));
// Cannot use dot operator here
p->age=20; // Assign age of struct to 20
```

```
printf("%d",p->age); // Print age of struct
```

We can also nest structures, defining another struct inside the definition of a struct.

Unions For polymorphic things, where we represent the same data as multiple types. All data is stored in the same spot (whereas for structs, everything has its own memory). Useful when you want to store something that can be multiple types. Merges variables into a single variable. Can declare unions and structs inside of a function too.

```
union NUMBER{
    short int a;
    int b;
    float c;
    double d;
}number;
```

In this case,

```
number.a=5;
number.b=6;
```

will both overwrite the same memory (you should not be modifying multiple parameters of a union, only using the respective one for that case).
We can put unions in structs and structs in unions and whatnot.

typedef Declaration Can make your own type, i.e.
typedef int boolean;
It can also be used with structs so you don't need to specify structs.

```
typedef struct COURSE{
    ...
}nameOfVar;
// Then when creating a COURSE
nameOfVar c206;
```

Enumerated Types Contains a list of constants that we can refer to as integers.

```
enum days {mon, tue, wed, thu, fri, sat, sun};
```

mon has value 0, tue has value 1 and so on... Can change to your own values.

```
enum days{mon = 1, tue, wed, ...}
// or
enum days{mon = 10, tue = 20, ...}
// then
int x = mon+tue; // x = 30;
```

5.14 Memory