COMP 206 Midterm Review

CSUS Helpdesk - Zoé McLennan & Daniel Busuttil
12/10/2018

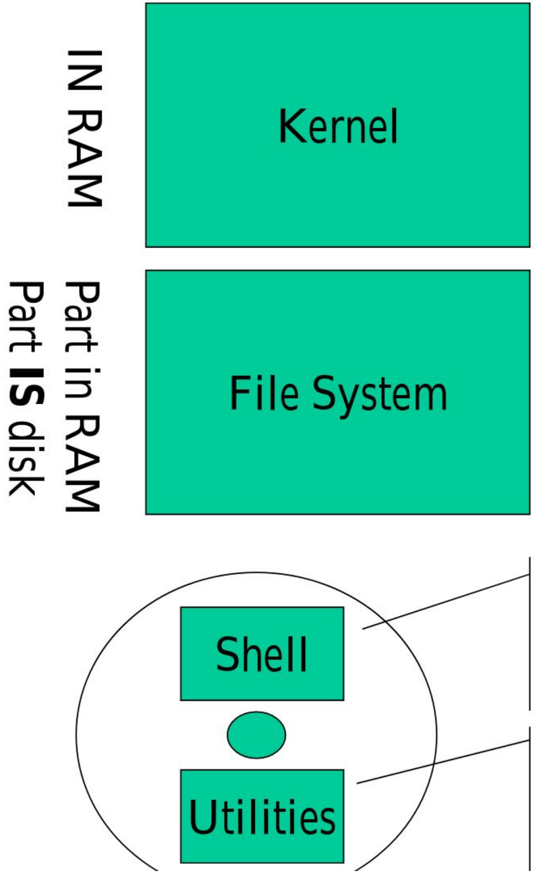
Outline

- Questions/Concerns
- Review
- Problems
- Questions again!

Operating System

- Piece of software that allows us to interact with a computer without needing to know the inner workings
- Manages resources
- Launches every other program

Unix OS Components



IN RAM

Kernel

- Login
- Knows how to run programs
- Basic common interface
- Memory management

Part in RAM
Part **IS** disk

File System

- Defines the way the disk drive is formatted
- The file allocation table (FAT)
- The data structure on disk that makes files real
- Reading and writing to disk and peripherals
- User commands to interact with files

Shell

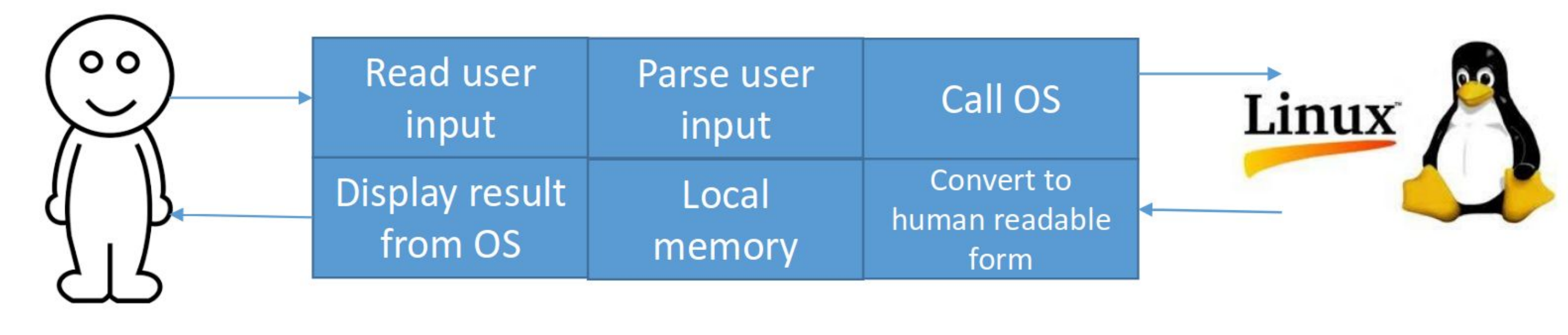
- A user interface
- Has a global memory
- Has commands to interact with OS

Utilities

- Additional OS commands and programs
- Third party commands and programs
- Drivers and managers

Shell

- Command interpreter: text → actions
- Bash
- Gets user input, displays OS information and stores session information



Read user input

Parse user input

Call OS

Display result from OS

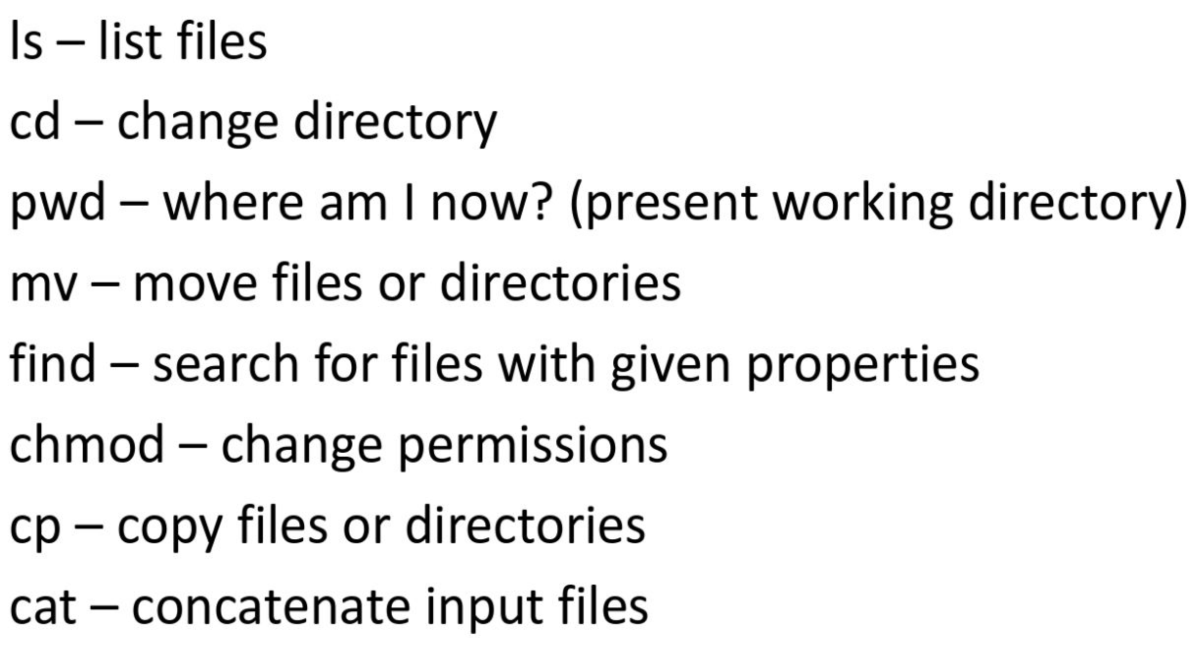
Local memory

Convert to human readable form

File System

- “/” → Root file system
- “~” → Current user's home directory
- “ . ” → Current directory
- “..” → Parent directory
- Using the “cd” command by itself from any location will move you to the home directory “~”

Some Commands



ls – list files

cd – change directory

pwd – where am I now? (present working directory)

mv – move files or directories

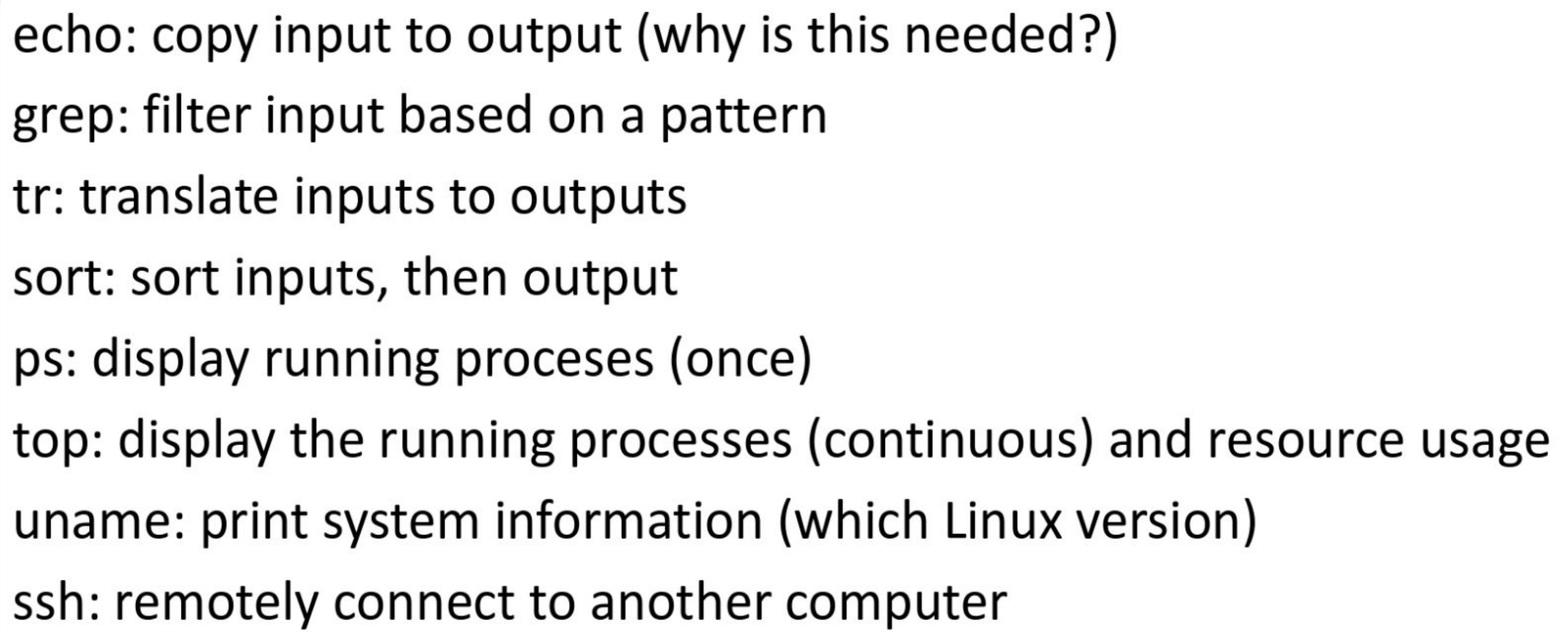
find – search for files with given properties

chmod – change permissions

cp – copy files or directories

cat – concatenate input files

More Commands



echo: copy input to output (why is this needed?)

grep: filter input based on a pattern

tr: translate inputs to outputs

sort: sort inputs, then output

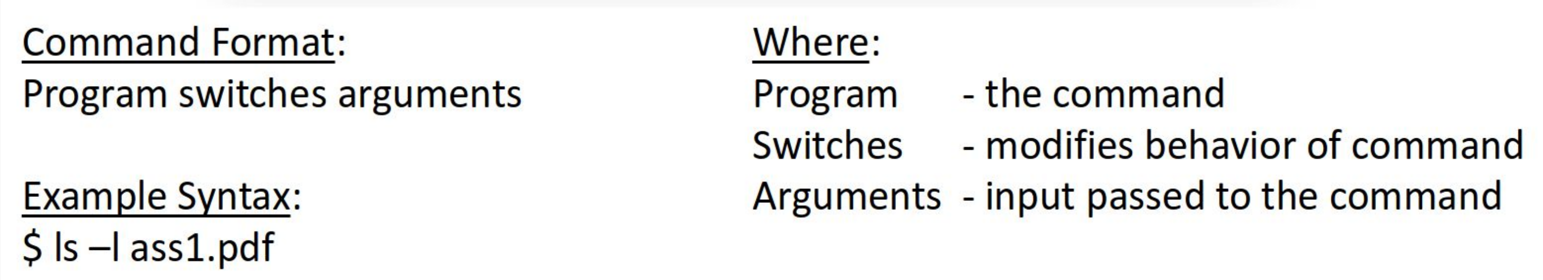
ps: display running proceses (once)

top: display the running processes (continuous) and resource usage

uname: print system information (which Linux version)

ssh: remotely connect to another computer

Command Format & Examples



Command Format:

Program switches arguments

Example Syntax:

```
$ ls –l ass1.pdf
```

Where:

Program	- the command
Switches	- modifies behavior of command
Arguments	- input passed to the command

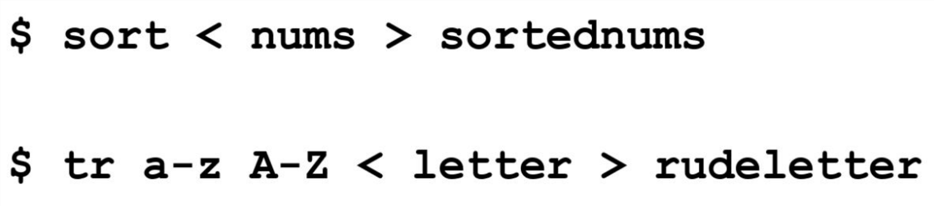
- mv file1.txt ./location
- cp file1.txt ./location/copy.txt
- ls -a
- rm file.txt

Redirection

- Change the standard I/O (keyboard/screen)
- Input can be redirected
  - From a file, “<”
    - **$ grep pattern < search_file.txt**
  - From the output of another program “|”
    - **$ cat test.txt sample.txt | more**
- Output can be redirected
  - To a file, “>”
    - **$ ls > file_info.txt**
  - As input to another file “|”
    - **$ cat test.txt sample.txt | more**

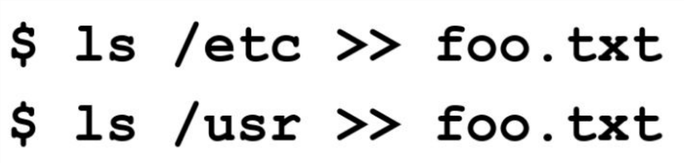
Redirection

- Do both



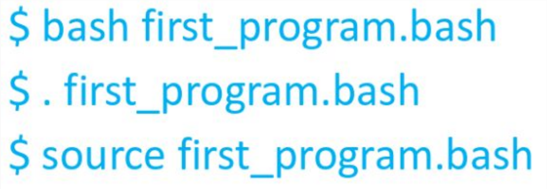
```
$ sort < nums > sortednums

$ tr a-z A-Z < letter > rudeletter
```

- Append



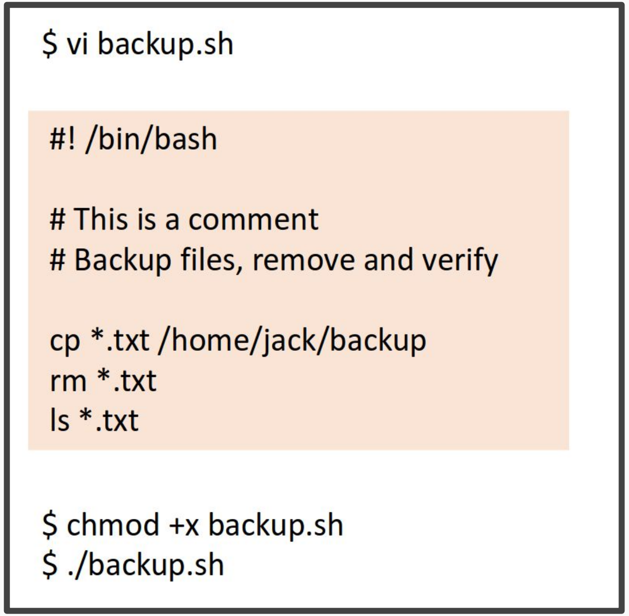
```
$ ls /etc >> foo.txt
$ ls /usr >> foo.txt
```

Bash Scripting

- Vim
- Shebang → **#!/bin/bash**
- Running
  - 

    ```
    $ bash first_program.bash
    $ . first_program.bash
    $ source first_program.bash
    ```

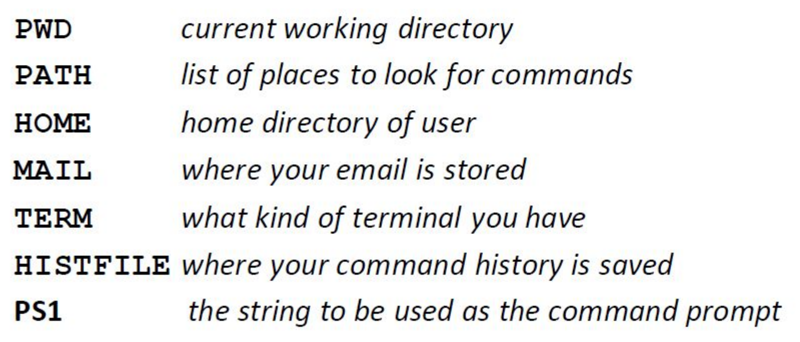
Simple example →



```
$ vi backup.sh

#! /bin/bash

# This is a comment
# Backup files, remove and verify

cp *.txt /home/jack/backup
rm *.txt
ls *.txt

$ chmod +x backup.sh
$ ./backup.sh
```

Bash variables

- Bash has some shell variables that it creates on startup and they’re incredible useful:



PWD	*current working directory*
PATH	*list of places to look for commands*
HOME	*home directory of user*
MAIL	*where your email is stored*
TERM	*what kind of terminal you have*
HISTFILE	*where your command history is saved*
PS1	*the string to be used as the command prompt*

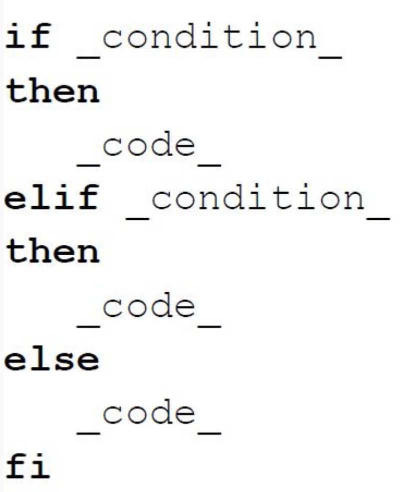
- You can use these in your scripts to great effect!

Important basics

- **$**
  - Access content of variables
    - **ls -al $dir**
- **Echo**
  - displays/prints variable
    - **echo $my_var**
- **Math**
  - $((computation))
    - **a=$((3+5))**

If statements pt. 1

Bash, like all languages, has its own control flow commands. The most important of these if the “if” statement and the syntax is as follows:



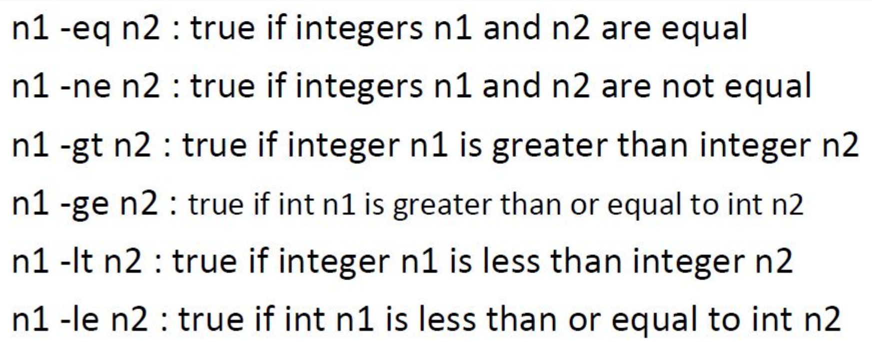
```
if _condition_
then
   _code_
elif _condition_
then
   _code_
else
   _code_
fi
```

In Bash, if-statements will check if ‘_condition_’s evaluate to zero (i.e. their return values) and execute the corresponding code if that is the case. There are many switches and commands that you can use to your advantage, and we’ll look at those next

If statements pt. 2

You can use these commands with the ‘test’ & ‘if’ commands to test for certain conditions, i.e.:

“ if [x -eq 4] “ == “ if test x -eq 4 “



n1 -eq n2 : true if integers n1 and n2 are equal

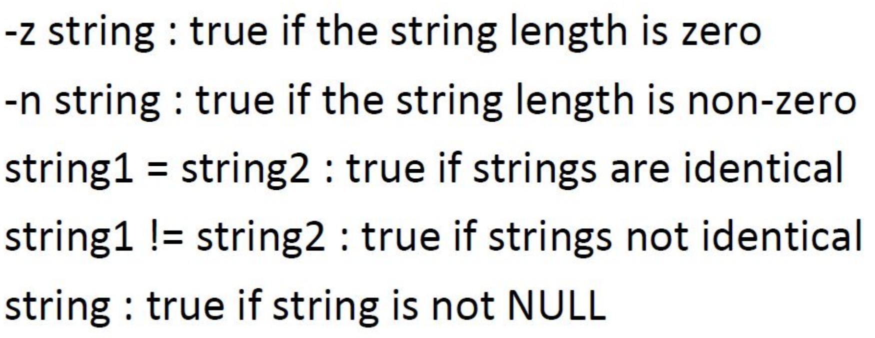
n1 -ne n2 : true if integers n1 and n2 are not equal

n1 -gt n2 : true if integer n1 is greater than integer n2

n1 -ge n2 : true if int n1 is greater than or equal to int n2

n1 -lt n2 : true if integer n1 is less than integer n2

n1 -le n2 : true if int n1 is less than or equal to int n2



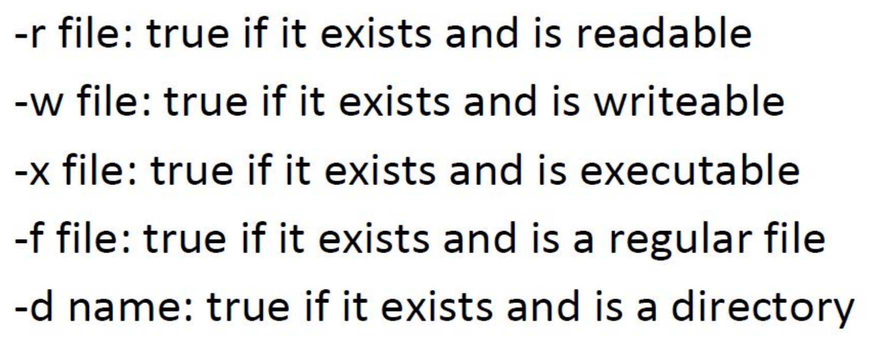
-z string : true if the string length is zero

-n string : true if the string length is non-zero

string1 = string2 : true if strings are identical

string1 != string2 : true if strings not identical

string : true if string is not NULL



-r file: true if it exists and is readable

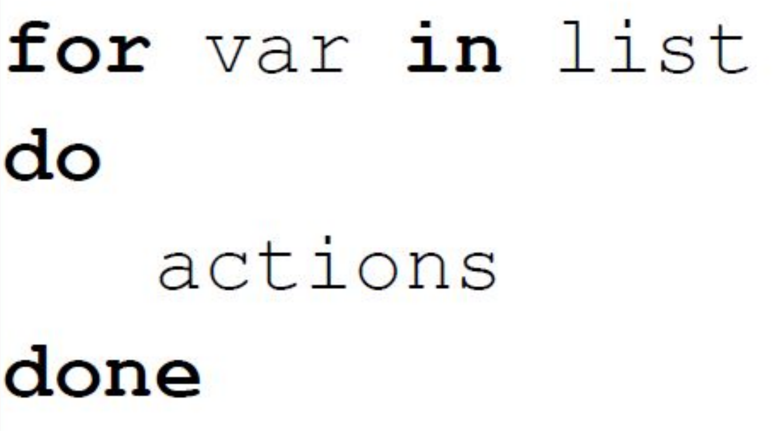
-w file: true if it exists and is writeable

-x file: true if it exists and is executable

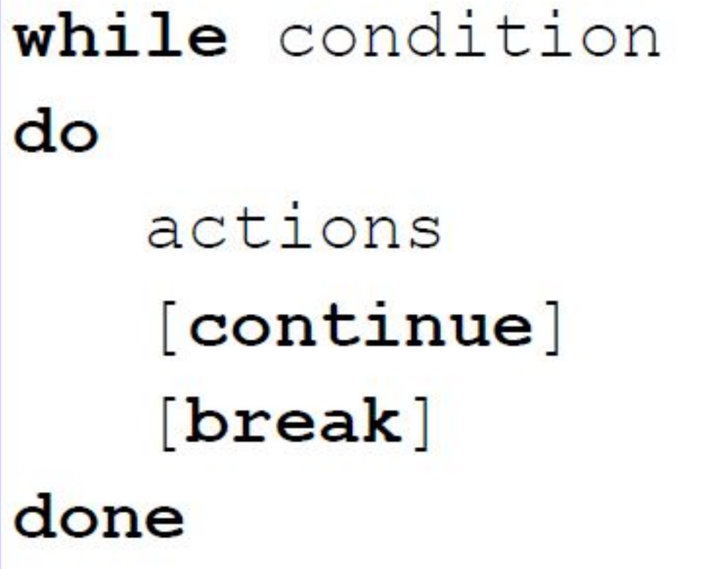
-f file: true if it exists and is a regular file

-d name: true if it exists and is a directory

For & While loops



```
for var in list
do
    actions
done
```



```
while condition
do
    actions
    [continue]
    [break]
done
```

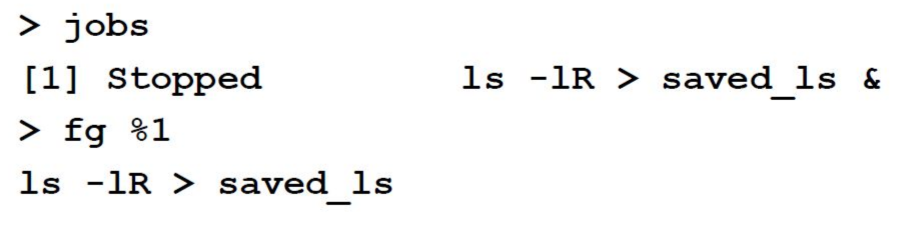
```
$ cat for2.sh
i=1
weekdays="Mon Tue Wed Thu Fri"
for day in "$weekdays"
do
 echo "Weekday $((i++)) : $day"
done

$ ./for2.sh
Weekday 1 : Mon
Weekday 2 : Tue
Weekday 3 : Wed
Weekday 4 : Thu
Weekday 5 : Fri
```

```
#!/usr/bin/bash

i=`wc -c < $1`;
while test $i -lt $2
do
    echo -n "0" >> $1;
    i=`wc -c < $1`;
done
```

```
% ./fill ass1.c 100
```

Job Control

- A shell has the capacity to manage ‘jobs’ which are processes that you run simultaneously; if you finish a command with ‘&’ the shell will run your process and, while it is not finished, it’ll run it concurrently with other processes you have running
- The shell will, appropriately, assign an ID number to each job it runs in the background. You can view these with the command “jobs” and suspend or kill any running jobs with ‘ctrl-Z’ and ‘ctrl-C’ respectively



```
> jobs
[1] Stopped            ls -lR > saved_ls &
> fg %1
ls -lR > saved_ls
```

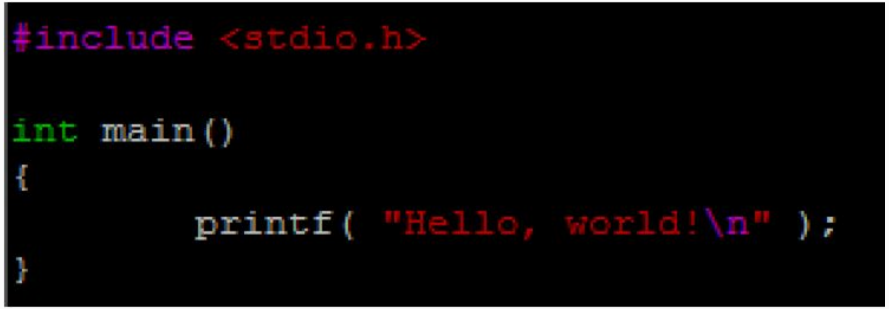
Wildcards

- * → any pattern
    - ls *.doc
        - List all documents with **.doc** extension
- **?** → any character
    - ls *.d?c
        - List all documents with **.d[any character]c** (.dac, .dzc, etc.)
- [ ] → Or
    - ls *.d?[acb]
        - List all documents with **.d[any character][a or b or c]** (.dza, .dmb, etc.)

End of Linux + Bash

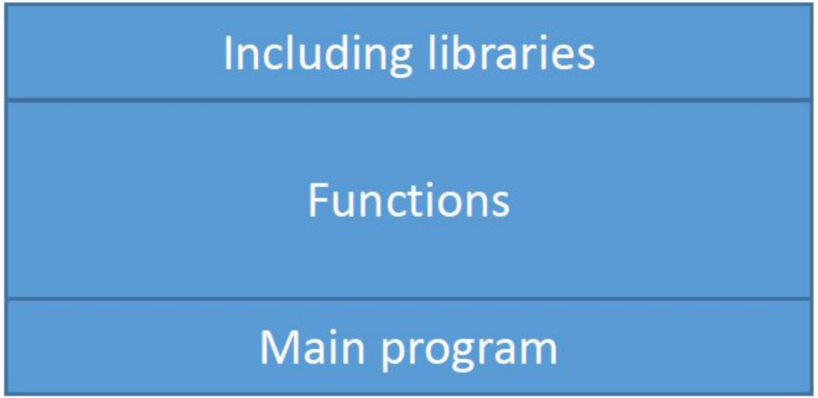
- Any Questions?

C Basics



```
#include <stdio.h>

int main()
{
        printf( "Hello, world!\n" );
}
```

1. vim program.c
2. gcc program.c
3. ./a.out



Recommended layout

C Basics: Datatypes

Built-in types:

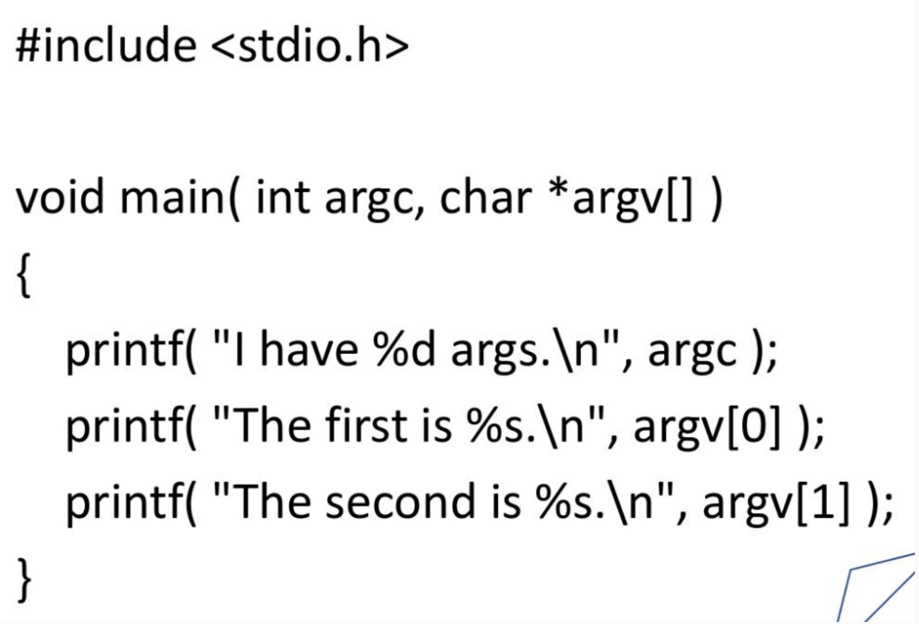
- int → 16/32 bit integer
- float → 32 bit floating point number
- double → 64 bit floating point number
- char → 8 bit character

Modifiers:

- short/long: 16 vs 32 bit int
- signed/unsigned: positive vs negative
- Pointers: int *, char *

Accessing Arguments

- **argc** → argument count
    - Good for condition statements
- **argv[i]** → access ith argument
    - argv[0] is always program name
        - Ex: ./a.out



```
#include <stdio.h>

void main( int argc, char *argv[] )
{
   printf( "I have %d args.\n", argc );
   printf( "The first is %s.\n", argv[0] );
   printf( "The second is %s.\n", argv[1] );
}
```

C Basics: control flow

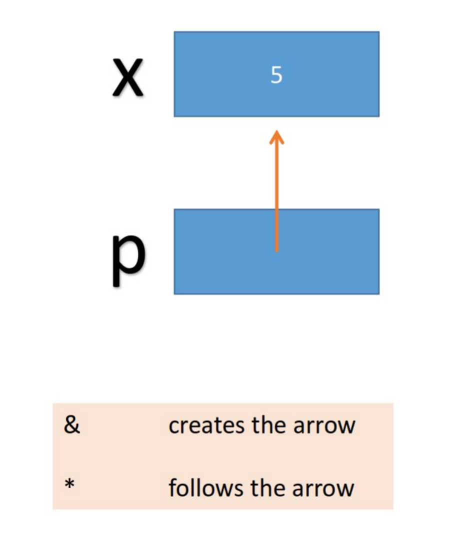
```
if ( COND ) STATEMENT;


if ( COND )
{STATEMENTS;}


if (COND1) {CODE}

else if (COND2) {CODE2}

else {CODE3}
```

```
switch (VAR) {

        case VAL1:   CODE

                     break; // not optional

        case VAL2:   CODE

                     break; // not optional

…

        default:     CODE

}
```

```
While ( COND ) STATEMENT;

while( COND ) { STATEMENTS; }


do STATEMENT; while (COND);

do {STATEMENTS;} while (COND);


for ( START; COND; EXPRESSION ) STATEMENT;

for ( START; COND; EXPRESSION )
{STATEMENTS;}
```

Pointers

- Pointers are variables which allow you to access (typed) areas of memory
  - Referencing
    - **&p** → return the address of the structure
  - Dereferencing
    - ***p** → get the at location of p
- Example:
  - int x=5;
  - int *p;
  - p = &x;
    - printf(“%d”, x); // prints 5
    - printf(“%d”, *p); // prints 5



&	creates the arrow
*	follows the arrow

Pass by value vs. Pass by reference

```
int increment(int n) {
        n++;
        return n;
}
```

```
int main(){
        int a = 5;
        a = increment(a);
        printf(“The value of a is now %d.\n”, a);
}
```

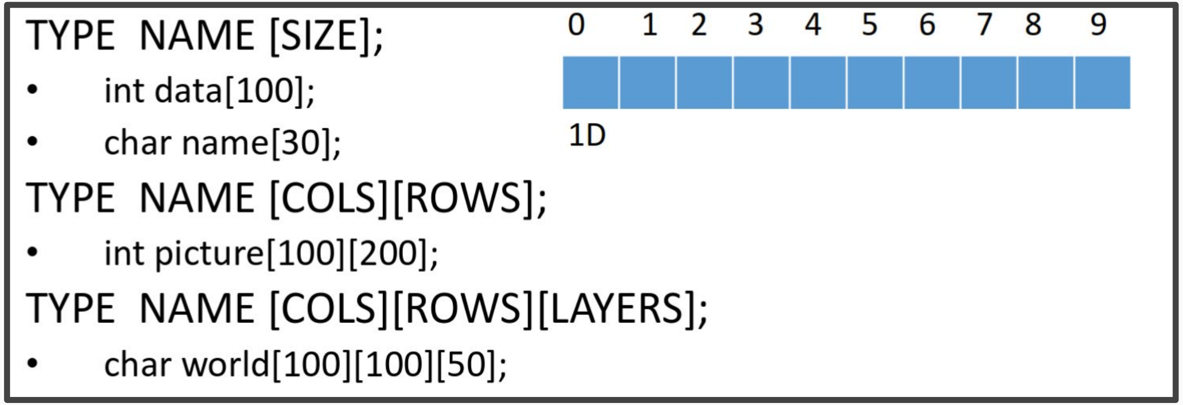
```
int increment(int *m) {
        (*m)++;
}
```

```
int main(){
        int a = 5;
        increment(&a);
        printf(“The value of a is now %d.\n”, a);
}
```

Both print 6!

Arrays

- Arrays are -in C- a series of contiguous variables of the array type with the array variable being a pointer to the first variable, i.e.
  - array[0] == *array
  - array[i] == *(array+ i)
  - &(array[j]) == array + j



TYPE NAME [SIZE];

- int data[100];
- char name[30];

TYPE NAME [COLS][ROWS];

- int picture[100][200];

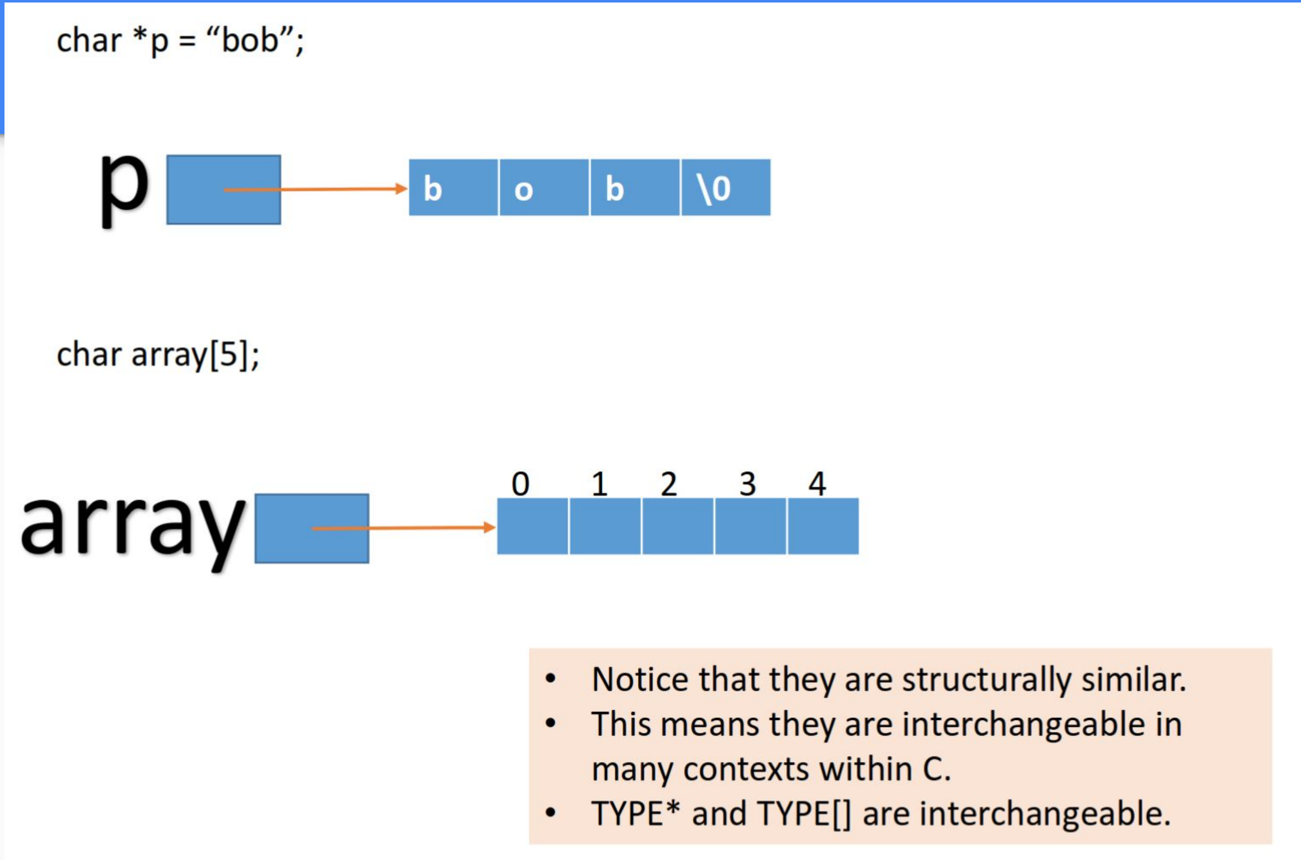
TYPE NAME [COLS][ROWS][LAYERS];

- char world[100][100][50];

- No safety mechanisms for string length!!
- Best practice to store length of array in variable

Strings

- Strings are simply char arrays with a ‘terminating’ null character: ‘\0’

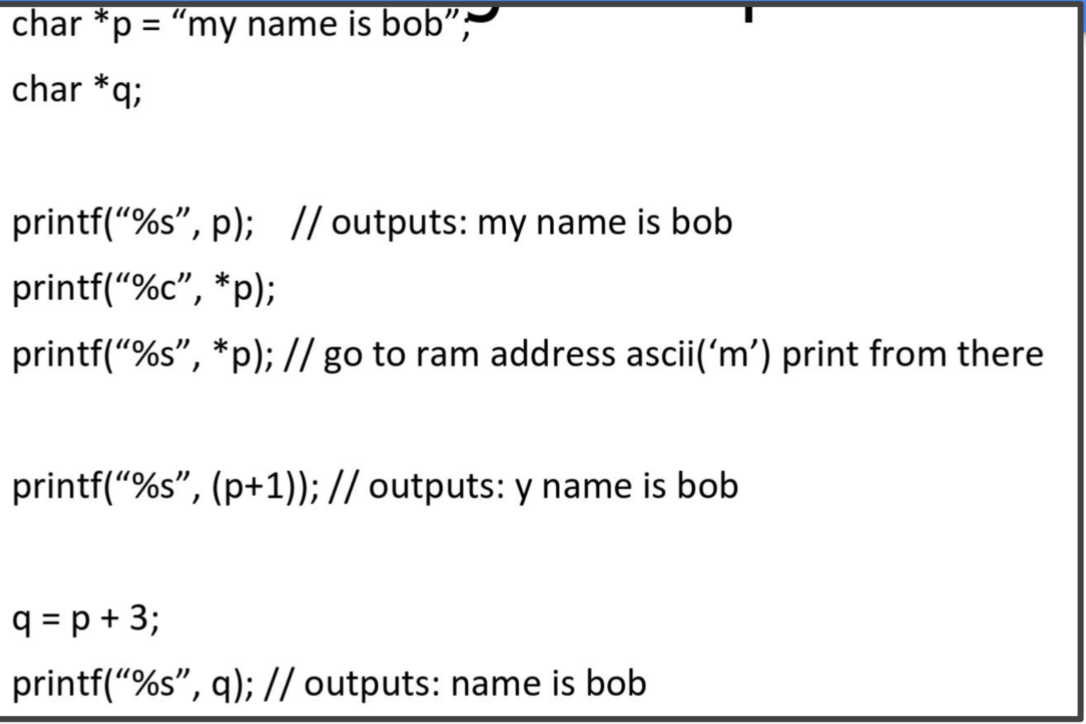


```
char *p = “bob”;
```

```
char array[5];
```

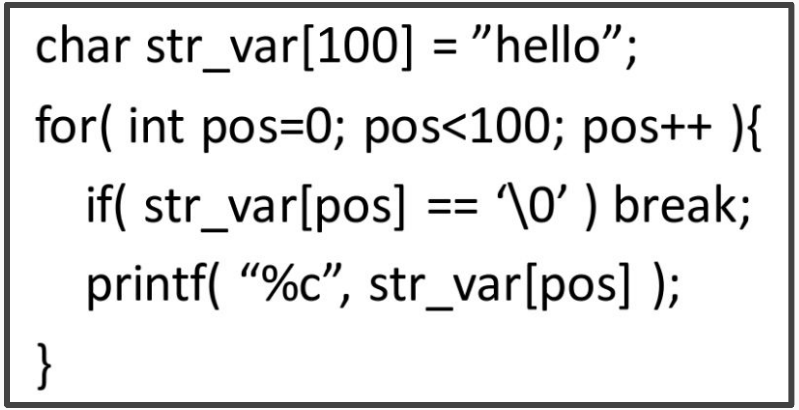
- Notice that they are structurally similar.
- This means they are interchangeable in many contexts within C.
- TYPE* and TYPE[] are interchangeable.

String Manipulation



```
char *p = “my name is bob”;
char *q;


printf(“%s”, p);    // outputs: my name is bob
printf(“%c”, *p);
printf(“%s”, *p); // go to ram address ascii(‘m’) print from there


printf(“%s”, (p+1)); // outputs: y name is bob


q = p + 3;
printf(“%s”, q); // outputs: name is bob
```

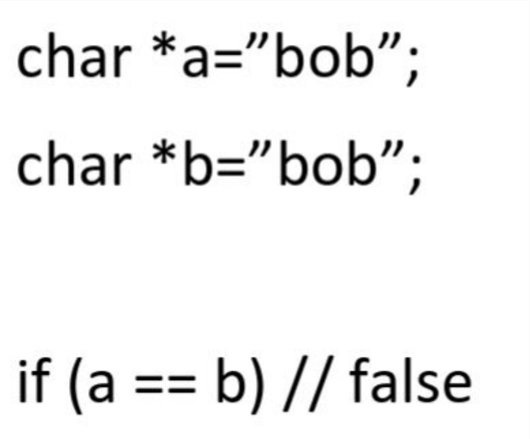
Iterate through string:



```
char str_var[100] = ”hello”;
for( int pos=0; pos<100; pos++ ){
   if( str_var[pos] == ‘\0’ ) break;
   printf( “%c”, str_var[pos] );
}
```

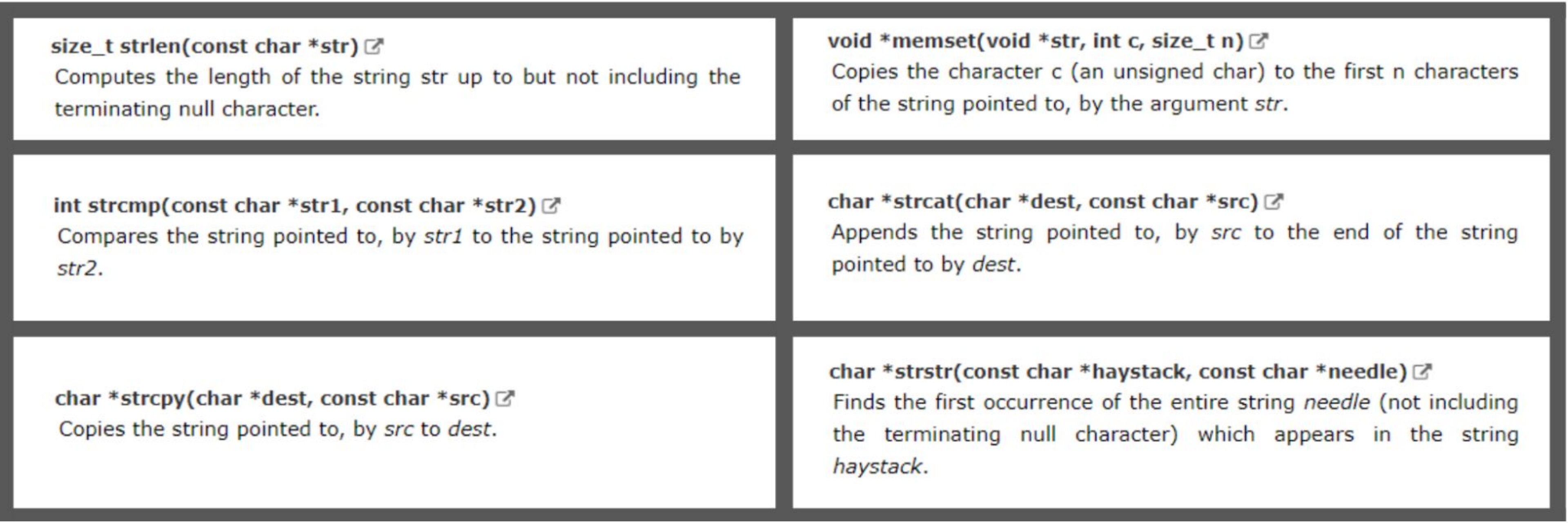
Logic

- Important to remember that logic is based on pointer position
- Need to iterate through both
- Or use built in library...



```
char *a=”bob”;

char *b=”bob”;



if (a == b) // false
```

<string.h> - Important functions

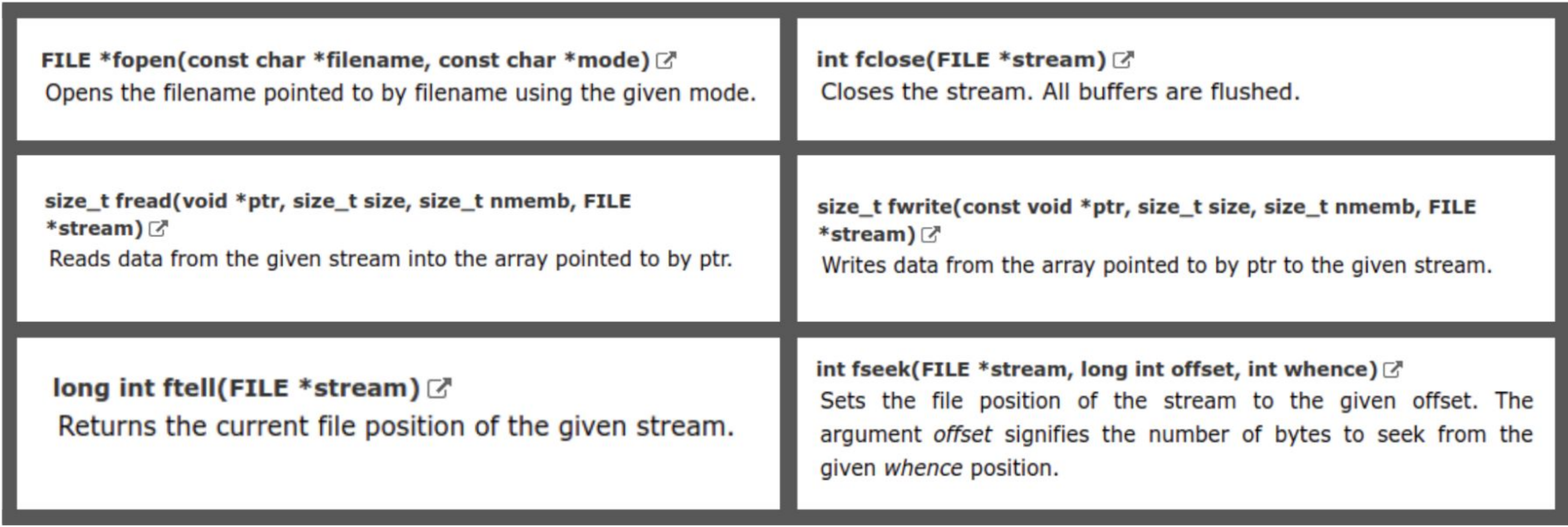


size_t strlen(const char *str) Computes the length of the string str up to but not including the terminating null character.	**void *memset(void *str, int c, size_t n)** Copies the character c (an unsigned char) to the first n characters of the string pointed to, by the argument *str*.
int strcmp(const char *str1, const char *str2) Compares the string pointed to, by *str1* to the string pointed to by *str2*.	**char *strcat(char *dest, const char *src)** Appends the string pointed to, by *src* to the end of the string pointed to by *dest*.
char *strcpy(char *dest, const char *src) Copies the string pointed to, by *src* to *dest*.	**char *strstr(const char *haystack, const char *needle)** Finds the first occurrence of the entire string *needle* (not including the terminating null character) which appears in the string *haystack*.

<stdio.h>

- 3 types of I/O
  - Console
    - Keyboard, screen
    - stdin, stdout, stderr
  - Stream
    - Constant stream of data from logical/physical device
  - File
    - Reading or writing to file
- <stdio.h provides built in functions to deal work with I/O

<stdio.h> - Important functions



FILE *fopen(const char *filename, const char *mode)
Opens the filename pointed to by filename using the given mode.

int fclose(FILE *stream)
Closes the stream. All buffers are flushed.

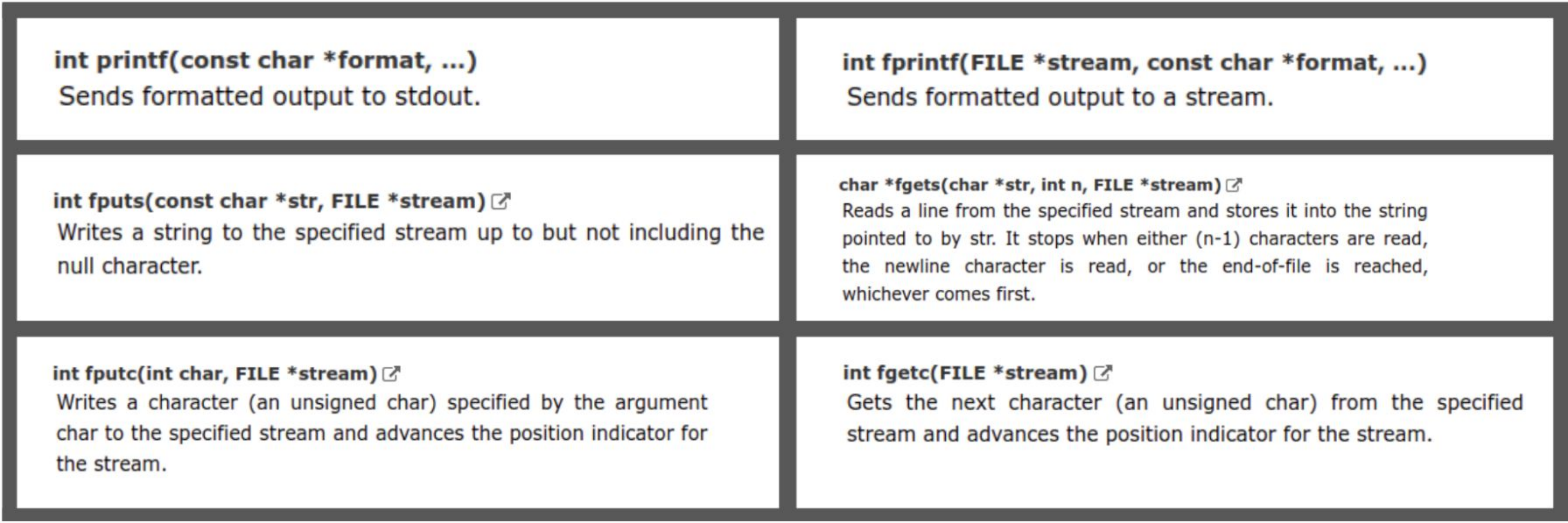
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
Reads data from the given stream into the array pointed to by ptr.

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
Writes data from the array pointed to by ptr to the given stream.

long int ftell(FILE *stream)
Returns the current file position of the given stream.

int fseek(FILE *stream, long int offset, int whence)
Sets the file position of the stream to the given offset. The argument *offset* signifies the number of bytes to seek from the given *whence* position.

<stdio.h> - Important functions



int printf(const char *format, ...)

Sends formatted output to stdout.

int fprintf(FILE *stream, const char *format, ...)

Sends formatted output to a stream.

int fputs(const char *str, FILE *stream)

Writes a string to the specified stream up to but not including the null character.

char *fgets(char *str, int n, FILE *stream)

Reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.

int fputc(int char, FILE *stream)

Writes a character (an unsigned char) specified by the argument char to the specified stream and advances the position indicator for the stream.

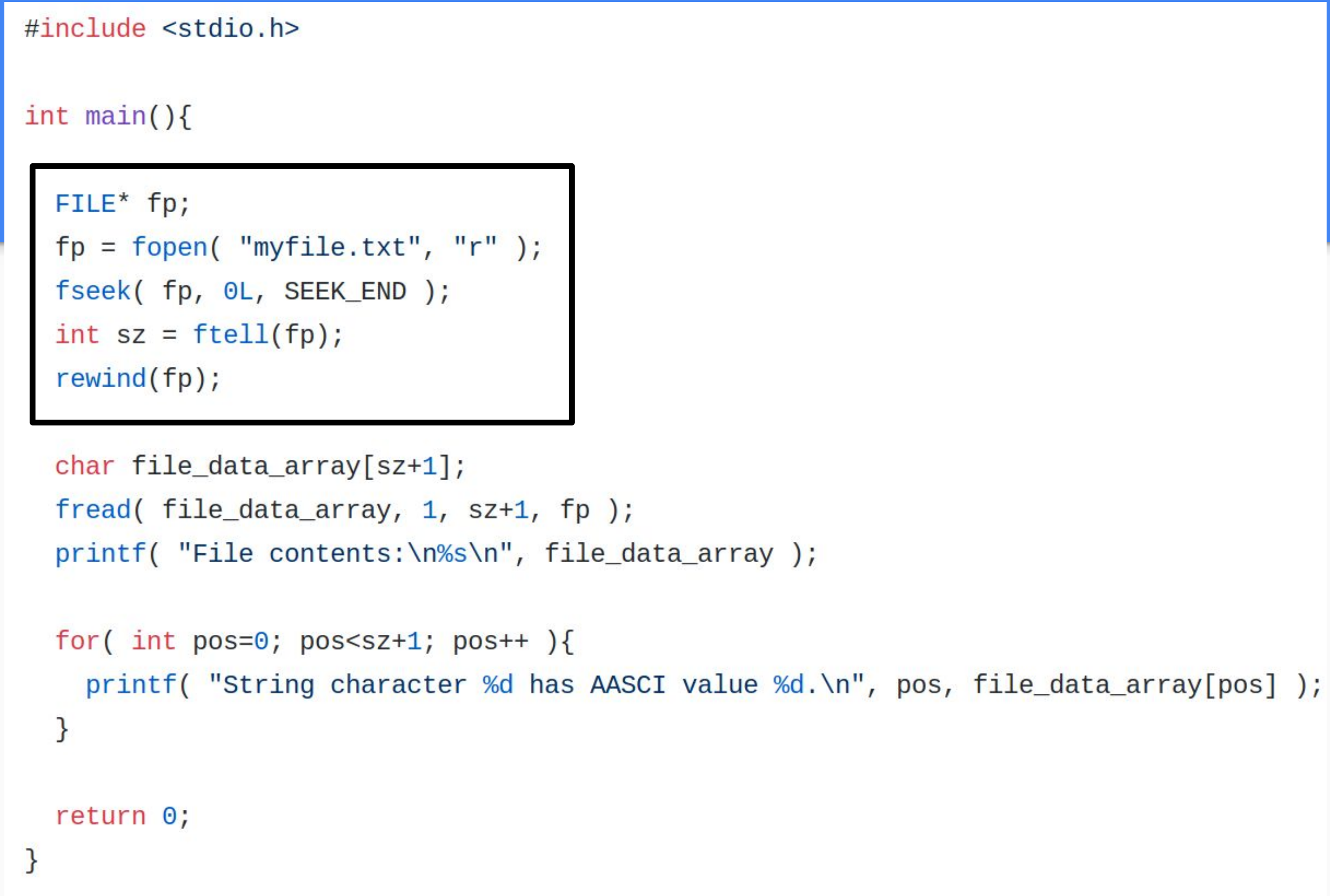
int fgetc(FILE *stream)

Gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.

fopen()

- FILE *fopen(const char *filename, const char *mode)
- FILE → built in pointer type
- Always check if NULL pointer after opening
- Modes:
  - r → read
  - w → write
  - a → append

fseek()



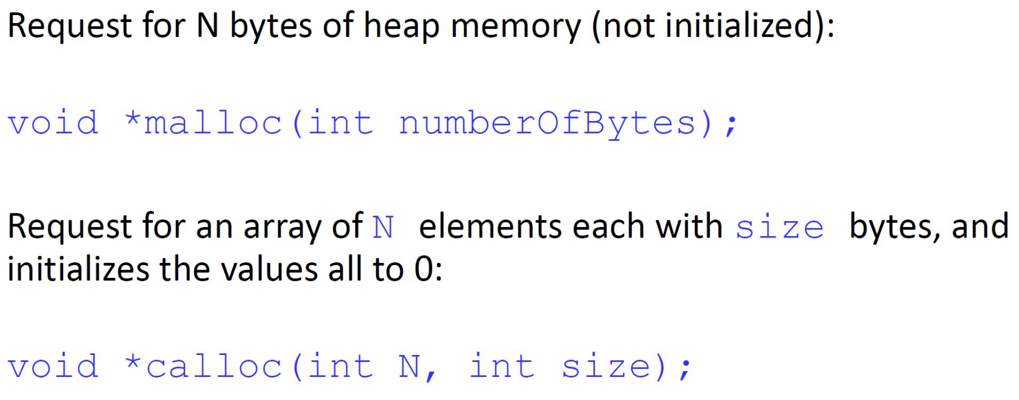
```
#include <stdio.h>

int main(){

  FILE* fp;
  fp = fopen( "myfile.txt", "r" );
  fseek( fp, 0L, SEEK_END );
  int sz = ftell(fp);
  rewind(fp);

  char file_data_array[sz+1];
  fread( file_data_array, 1, sz+1, fp );
  printf( "File contents:\n%s\n", file_data_array );

  for( int pos=0; pos<sz+1; pos++ ){
    printf( "String character %d has AASCI value %d.\n", pos, file_data_array[pos] );
  }

  return 0;
}
```

Example

```
#include <stdio.h>
#include <stdlib.h>
void copyFile (FILE *source, FILE *destination) {
    char c;
    while(!feof(source)) {
        c = fgetc(source);
        fputc(c, destination);
    }
}
void main() {
    FILE *s = fopen(“letter.txt”,”rt”), *d = fopen(“copy.txt”,”wt”);
    if (s == NULL || d == NULL) exit(1); // terminate with an error code
    copyFile(s, d);
    fclose(s); fclose(d);
}
```

Heap Memory

- What to do when you don’t know how much memory you’ll need ahead of time?
  - Use dynamically allocated memory
  - Heap memory can allocate and deallocate memory dynamically many times



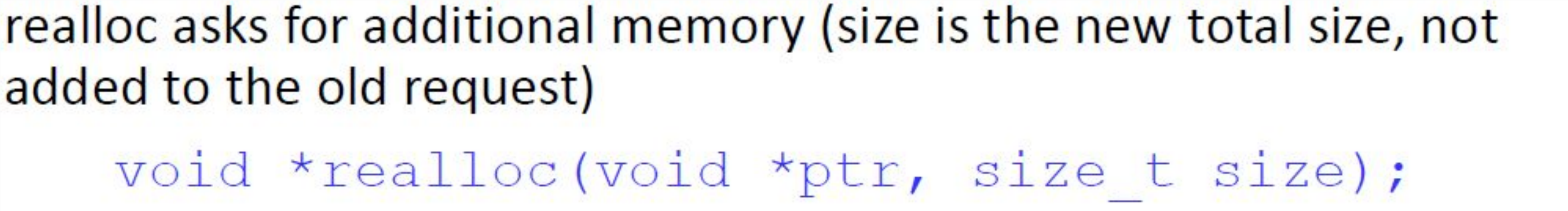
Request for N bytes of heap memory (not initialized):

```
void *malloc(int numberOfBytes);
```

Request for an array of `N` elements each with `size` bytes, and initializes the values all to 0:

```
void *calloc(int N, int size);
```

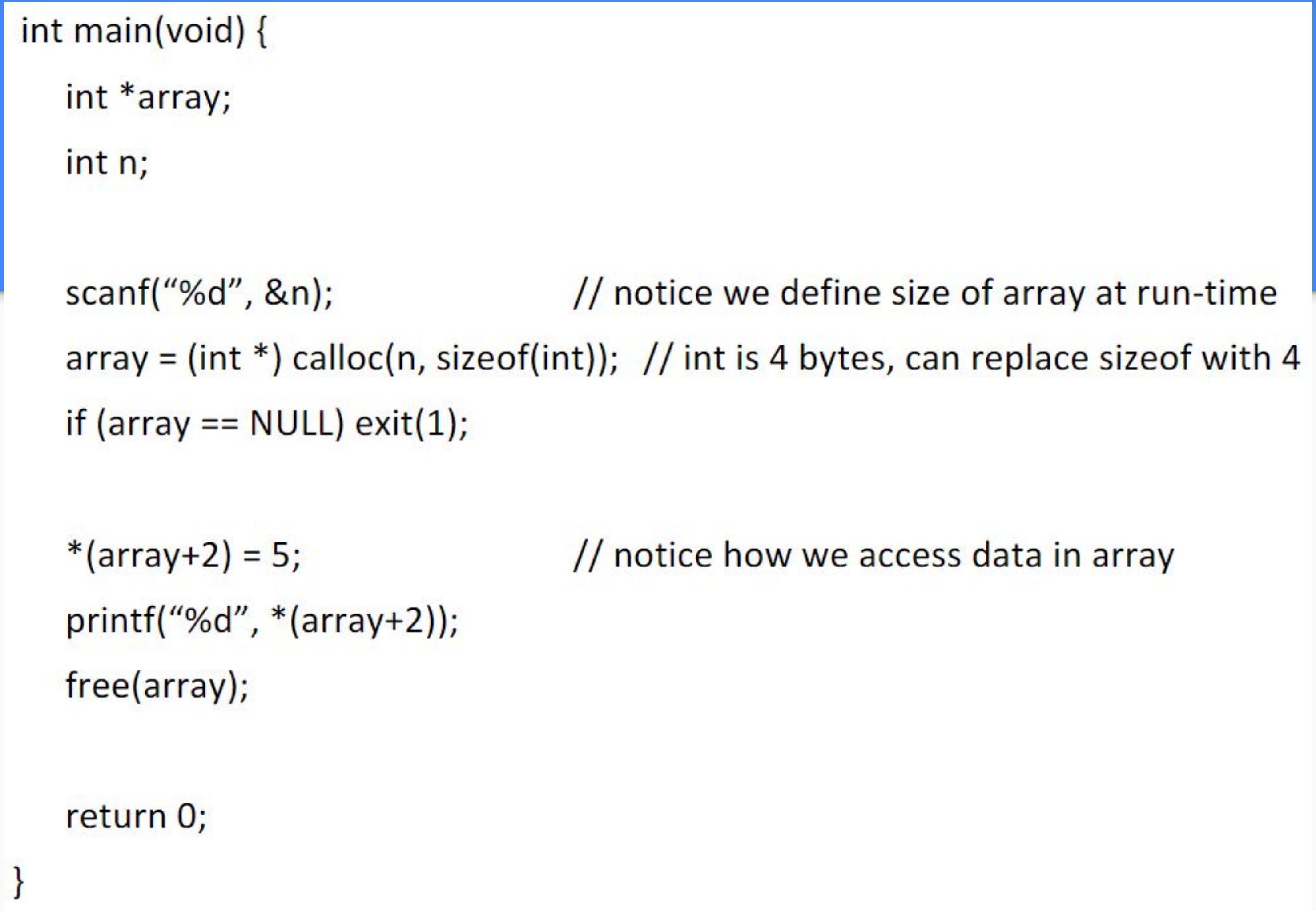
Heap Memory



realloc asks for additional memory (size is the new total size, not added to the old request)

```
void *realloc(void *ptr, size_t size);
```

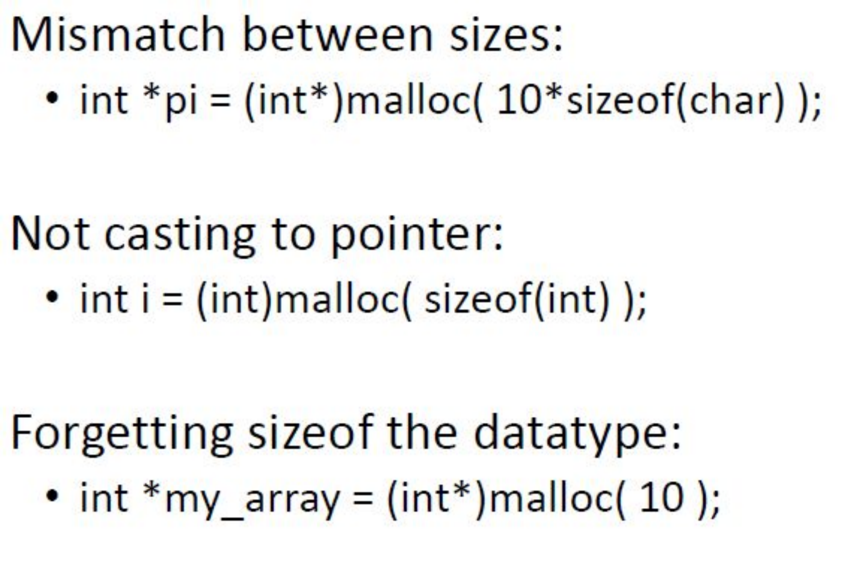
- Might not have enough space at original address
- Beware, data may be moved to new location
- After using heap memory:
  - free(void *ptr)
  - ptr= NULL;



```
int main(void) {
    int *array;
    int n;

    scanf(“%d”, &n);                        // notice we define size of array at run-time
    array = (int *) calloc(n, sizeof(int));  // int is 4 bytes, can replace sizeof with 4
    if (array == NULL) exit(1);

    *(array+2) = 5;                         // notice how we access data in array
    printf(“%d”, *(array+2));
    free(array);

    return 0;
}
```

Heap Memory

- Common Errors:



Mismatch between sizes:

- int *pi = (int*)malloc(10*sizeof(char));

Not casting to pointer:

- int i = (int)malloc(sizeof(int));

Forgetting sizeof the datatype:

- int *my_array = (int*)malloc(10);

Review Questions!

a) List all files/directories in current directory that contain upper case letters
b) List all files/directories in current directory using upper case letters

Review Questions!

Answer:

a) `$ ls | grep [A-Z]`

b) `$ ls | tr [a-z] [A-Z]`

Review Questions!

Assume that in your current directory there is a file called “passwords.txt”. Write a script that takes in 1 cmd-line argument and checks if it exists (ignoring case) inside the text file.

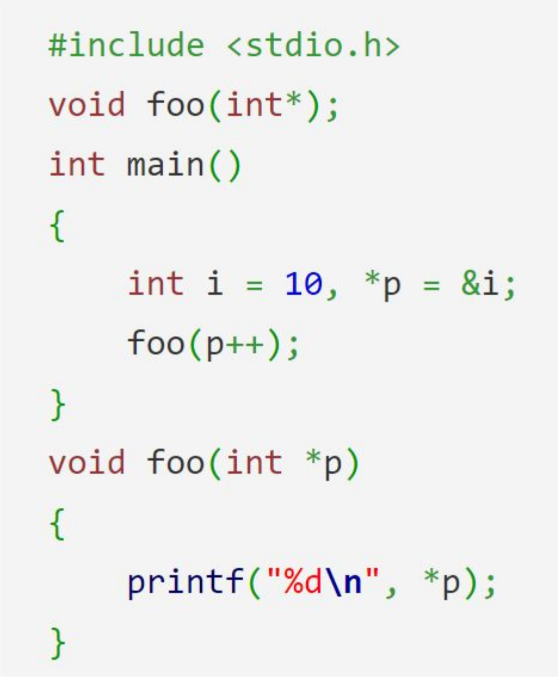
Review Questions!

Answer:

```
if [ `grep -i -c `echo $1` passwords.txt` -ge 1 ]

then

        Echo “Found it!”

else

        Echo “I couldn’t find it”

fi
```

Review Questions!

- What prints?
  a. A really long value
  b. 10
  c. Compilation error
  d. Segmentation fault

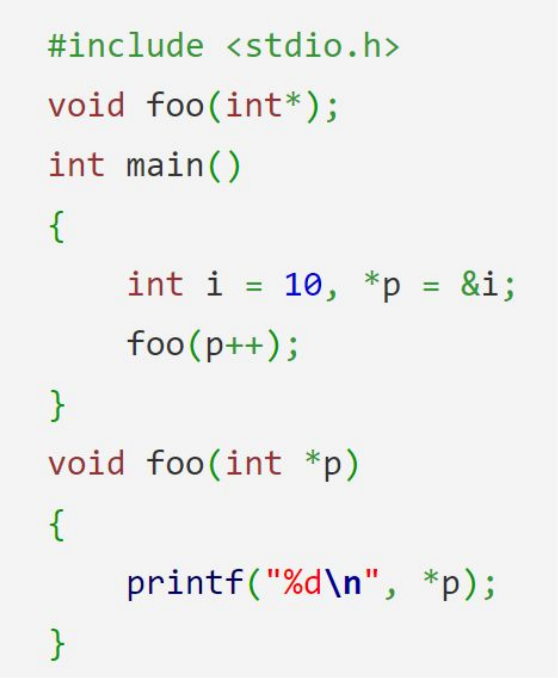


```
#include <stdio.h>
void foo(int*);
int main()
{
    int i = 10, *p = &i;
    foo(p++);
}
void foo(int *p)
{
    printf("%d\n", *p);
}
```

Review Questions!

Answer

→ B



```c
#include <stdio.h>
void foo(int*);
int main()
{
    int i = 10, *p = &i;
    foo(p++);
}
void foo(int *p)
{
    printf("%d\n", *p);
}
```

Review Questions!

Write a program that allows a user to input how many students are in a class. You must prompt the user for the number of students and then store them in an array.

- Hint: use heap memory

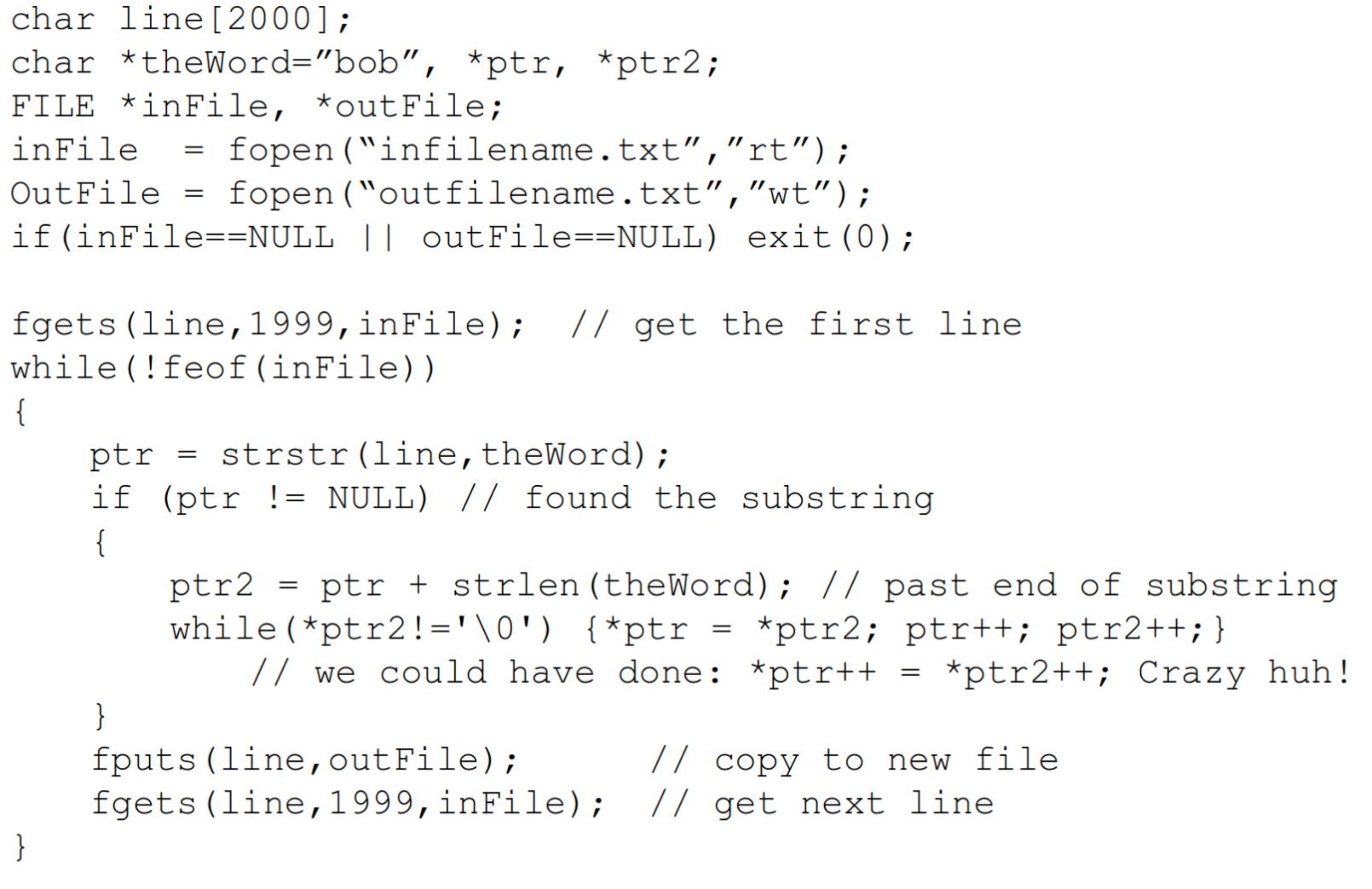
Review Questions!

Answer:

```
int main(){
    char *students, *studentName;

    printf(“Enter number of students: “);
    scanf(“%d”, &n);

    students = (char *) calloc(n, sizeof(char));
    if(students==NULL) exit(1);

    for(x=0; x<n; x++) {
        studentName = students+x;
        scanf(“%s”, studentName);
    }
}
```

Review Questions!

Remove all occurences of the word “bob” in a file called “inputfilename.txt” and write it to “outputfilename.txt”



```
char line[2000];
char *theWord=”bob”, *ptr, *ptr2;
FILE *inFile, *outFile;
inFile  = fopen(“infilename.txt”,”rt”);
OutFile = fopen(“outfilename.txt”,”wt”);
if(inFile==NULL || outFile==NULL) exit(0);

fgets(line,1999,inFile);  // get the first line
while(!feof(inFile))
{
    ptr = strstr(line,theWord);
    if (ptr != NULL) // found the substring
    {
        ptr2 = ptr + strlen(theWord); // past end of substring
        while(*ptr2!='\0') {*ptr = *ptr2; ptr++; ptr2++;}
            // we could have done: *ptr++ = *ptr2++; Crazy huh!
    }
    fputs(line,outFile);      // copy to new file
    fgets(line,1999,inFile);  // get next line
}
```