Table of Contents

Lecture 1: Introduction to Unix

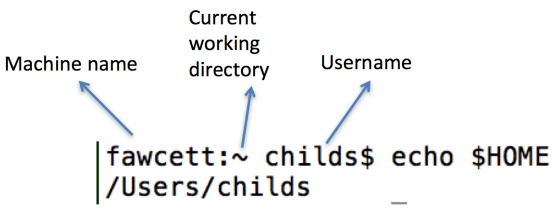
Unix
- Unix is an operating system that does multi-tasking and is multi-user
- Linux is an open source version of Unix
- OS X meets the Unix standards.. Macs use Unix.

Shells
- Shells are accessed through a terminal program
- Shells are interpreters (aka they interpret your code to be executed)
- The most common shell type is sh (which is bash and ash)
  - The different shell types differ in syntax. We are focusing on bash shell types.
- Environment Variables are stored by shell interpreter and some environment variables create side effects in the shell whereas other environment variables can be just for your own private purposes.
  - alyssa$ export CIS212=fun
    key word variable

Files
- Unix maintains a file system - File system controls how data is stored and retrieved
- Primary abstractions - Directories and Files
- Files are contained within Directories
- Directories can be places within other directories
- "/" -- The root directory
- "/dir1/dir2/file1" -- File file1 is contained in directory dir2, which is contained in directory dir1, which is in the root directory.
  root → dir1 → dir2 ↳ file1

Home Directory
- Unix supports multiple users and each user has their own directory that they control
- Location varies over Unix implementation, but typically something like "/home/username"
- Stored in environment variables
  - alyssa$ echo $HOME
    /Users/alyssa
- "~" (tilde) is shorthand for your home directory - and you can use it when invoking commands.

Anatomy of shell formatting



Unix commands

cd: change directory
- The shell always has a "present working directory" (pwd) - directory commands are relative to
- "cd" changes the present working directory
- When you start a shell, the shell is in your "home" directory

mkdir: makes a directory
- Two different ways:
  - Relative to current directory
    - mkdir dirNew
  - Relative to absolute path
    - mkdir /dir1/dir2/dirNew (Note: in this case, dir1 and dir2 already exist)

rmdir: remove directory

- Two different ways:
  - Relative to current directory
    - rmdir badDir
  - Relative to absolute path
    - rmdir /dir1/dir2/badDir (Note: This removes badDir, and leaves dir1 and dir2 in place)
- Note: This only works on empty directories - which means no files are in this directory

touch: "touch" a file

- Behavior:
  - If the file doesn't exist
    - Create it
  - If the file does exist
    - Update the time stamp

ls: list the contents of a directory

- A flag is a mechanism for modifying a Unix program behavior
- Convention of having hyphens to signify special status
- "ls" is also useful with "wild cards", which we will also discuss later

wc: word count

- Lines / words / characters

man:

- Man gives you a description of the command you put after it.
- Example:
  - alyssa$ man rmdir

Notes regarding VI/VIM (Not tested on)

- Command mode:
  - When you type keystrokes, they are telling vim a command you want to perform, and the keystrokes don't appear in the file
- Edit mode:
  - When you type keystrokes, they appear in the file.
- Transitioning from command mode to edit mode:
  - i: enter into edit mode at the current cursor position
  - a: enter into edit mode at the cursor position immediately to the right of the current position
  - l: enter into edit mode at the beginning of the current line
  - "A": enter into edit mode at the end of the current line
- Transitioning from edit mode to command mode:
  - Press escape "esc"
- Other commands:
  - yy - yank the current line and put it in a buffer
    - 2yy = yank the current line and the line below it
  - p - paste the contents of the buffer
    - 2pp = pasting the contents of the buffer two times
  - x - delete the character at the current cursor
  - ":100" = go to line 100 in file
  - Arrows can be used to navigate the cursor position (while in command mode)
    - h, j, k, l

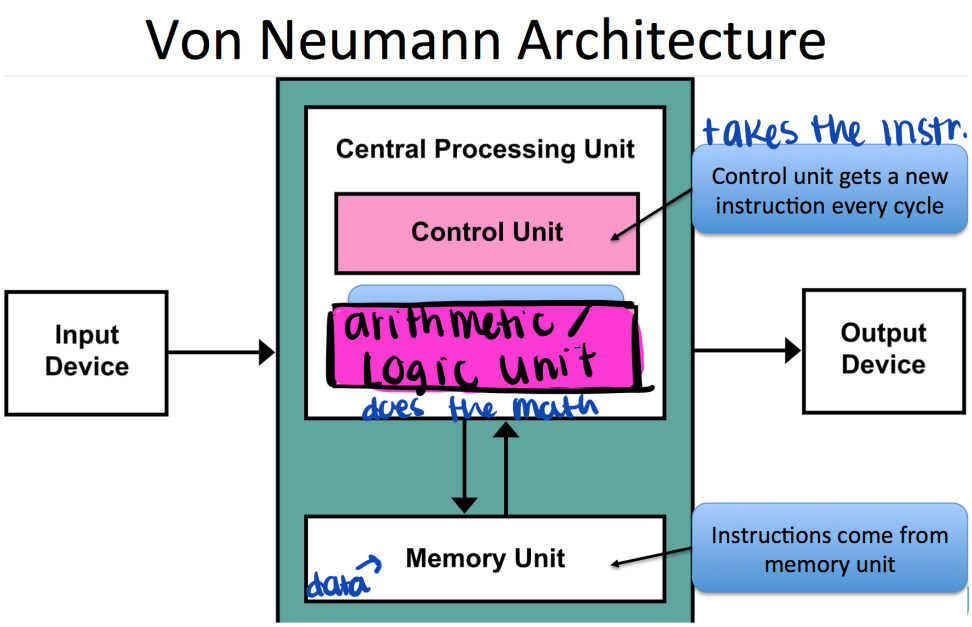
Lecture 2: Hello World in C

Conventions for a shell

- *: wildcard ... matches anything
  - h*k = matches hank, hulk, hack, etc
- ".": the current directory
  - "ls ." would list the current directory
- "..": the parent directory (one level up)
  - "ls .." would list the parent directory

How a computer works

- A computer operates in "cycles"
- The goal during a cycle is to compute one thing (*)
  - Various things may need to happen for the computer to carry out the cycle
- At the end of a cycle, the computer starts a new cycle, to carry out a new computation
  - Example: In a game, when Player1's turn is over, the game moves on to Player2's turn.
- A computer operates by loading a special kind of file called a "program"
  - This file has a sequence of instructions in it
- Instructions are very primitive - (assembly)
  - Add, subtract, looooots of math stuff
  - Load from memory, store to memory
  - Jump
  - Etc.
- Computer programs can do iteration - with iteration, we can write finite length programs that can run forever
  - Line 100: x = 0
  - Line 101: A = location in memory
  - Line 102: if X = 10, jump to line 106
  - Line 103: A = 2*A (aka *A = 2**A)
  - Line 104: X = X + 1 ...



How are instructions generated?

- Instructions come from programs, which are loaded in memory
- Programs contain a sequence of instructions that CPU can understand (machine code) and those instructions are fetched one by one.

The workflow with Compilers

- 1) Start with source code
- 2) Invoke compiler, which takes source code and generates executable
  - The executable is made up of instructions for the CPU (opcode)
- 3) invoke executable
  - The instructions for the CPU are fed to the CPU one at a time
- The compiler we are using to execute our C programs is gcc and this produces the executable a.out and then you can run a.out like this: ./a.out. You can change the output name to something else, and you would do this by adding an -o flag after the program name and then after the flag, specify your desired output name.
  - Note: "." Is the directory the shell is currently in. Therefore "./a.out" says run the program called a.out in the current directory.

- Note: Important C concept
  - If you do not return something in your functions, then C does not do this automatically for you. If you specify that you are going to return something, and you do not, then it will return a random memory location. Most of the time this will give you 0, but this will not always be the case. If you do not want your function to return anything, then the type in the function declaration should be void.

- More Deep Thoughts...
  - The compiler is a program. Its job is to make other programs
  - The operating system is a program. Its job is to run other programs (and provide an environment for them)
  - The first compiler came from machine code, and now new compilers are made from other compilers.

- C programs -> fast* -> compile -> turn into assembly code -> assembler -> program

- Etc. Notes
  - Assembly code varies from architecture to architecture
  - C does not
    - You can code in C, and the compiler will make your code work anywhere
  - Just about every command in C corresponds to a small handful of assembly instructions
    - This means C will be fast

Variable types

- When you declare variables, you have to declare their type
  - Example: int 9;
  - There are more types (floating point, etc)

Functions

- As you would expect, you can define your own functions and call them
- Every function has its own "scope" and its variable live within that scope.
  - Curly braces create scopes, and you can use more { and } within a function.
  - If you do not use curly braces within loops, then the function is not going to work as intended. Indentation means nothing in C, so it is important to have curly braces around what you want to occur for that loop

```
#include <stdio.h>

int main()
{
    int Y = 3;
    {
        int X = 2;
    }
    printf("X is %d\n", X);
}
Hanks-iMac:Downloads hank$ gcc scope2.c
scope2.c:9:25: error: use of undeclared identifier 'X'
    printf("X is %d\n", X);
                        ^
1 error generated.
```

X inside inner scope

the print is outside that scope w/ X so it cannot get X

in order to print x outside like this, you would need to initalize x outside the loop scope and inside the function scope then print it in the function scope. If you had another function you wanted x in, you would need to call function1 inside function2.

The for loop

- Three main components:
  - Initialization
  - Termination
  - What to do each step

```
for (i = 0 ; i < 10 ; i = i+1)
```

initialization ; termination ; What to do

- Often used with a loop variable (example: i)
  - Initialization: i = 0
  - Termination: i < 10
  - What do each step: i++ (this i++ is known as the increment operator)

Project 2A: A C program that finds the prime numbers between 5 and 100.

```c
//Author: Alyssa Kelley

//Class: CIS 212
//Project: 2A

#include <stdio.h>

int main()
{
    int prime_num, divisor; /* prime_num = the number we are checking to see if prime or not,
                * and divisor = the number it is trying to be divisable by.
                */

    for(prime_num=5; prime_num<=100; prime_num++)
    {

        /* this prime_num++ means that it is incrementing a in steps of 1
        so it will be counting up an integer each time to go up to 100
        */

        for(divisor=2; divisor<prime_num; divisor++)
        {
            /* this will have the divisor start at 2 and increment up and it
             *divides against the prime_num to see if there is a remainder or not
             */

            if(prime_num%divisor==0) /* if this is true then that means
                        * they are divisable and prime_num variable is NOT a prime number
                        */
            {
                break; /* Gets us out of the inner for loop and not the outer one */
            }
        }

        if(prime_num==divisor)
        {
            /*this is after that division for loop so if all of those went through
             *so this checks that if the two numbers are equal, and the remainder
             *was not equal to 0, then they are divisable and prime_num is a prime number.
             */

            printf("%d is a prime number\n", prime_num);

        }
    }
    return 0;
}
```

initialize in main what variables will be used.

can be = to 100.

wanting to check any #

once it can be divisable by a #, then you don't want to keep checking.

Lecture 3: More Unix. What is Data? Beginning on Memory

Files
- Unix maintains a file system
  - File system controls how data is stored and retrieved
- Primary abstractions
  - Directories
  - Files
- Files are contained within directories
- Directories are hierarchical

Permissions: System Calls
- System calls: a request from a program to the OS to do something [redacted] on its behalf
  - Including assessing files and directories
- System calls:
  - Typically exposed through functions in C library
  - Unix utilities (cd, ls, touch) are programs that call these functions
  - Note: Permissions in Unix are enforced via system calls

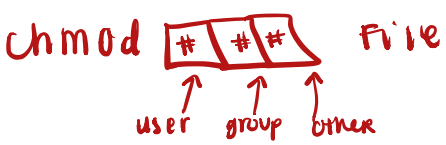
Permissions: Unix Groups
- Groups are a mechanism for saying that a subset of Unix users are related

Permissions
- Permissions are properties associated with files and directories
  - System calls have built-in checks to permissions
    - Only succeed if proper permissions are in place
- Three classes of permissions:
  - User: access for whoever owns the file
    - You can prevent yourself from accessing a file (but you can always change that back)
  - Group: allow a Unix group to access a file
  - Other: allow anyone on the system to access a file
- Three types of permissions:
  - Read
  - Write RWX
  - Execute

Executable Files
- An executable file: a file that you can invoke from the command line
  - Scripts
  - Binary programs compiler turns it into binary.
- The concept of whether a file is executable is linked with file permissions

Unix Command

chmod: change file mode
- chmod 750 <filename>
  - User gets 7 (rwx)
  - Group gets 5 (rx)
  - Other gets 0 (no access)



#	Permission	rwx
7	full	111
6	read and write	110
5	read and execute	101
4	read only	100
3	write and execute	011
2	write only	010
1	execute only	001
0	none	000

R/W/E

Unix commands for groups
- chgrp: changes the group for a file or directtory

    - chhgrp <group> <filename>
- Groups: lists groups you are in

Permissions and Directories
- You can only enter a directory if you have "execute" permissions to the directory

Unix Scripts
- Scripts
    - Use an editor (vi) to create a file that contains a bunch of Unix commands
    - Give the file execute permissions
    - Run it like you would any program
- Arguments
    - Assume you have a script named "myscript"
    - If you invoke it as "myscript foo bar"
    - Then
        - $# == 2
        - $1 == foo
        - $2 == bar

Project 1B: Write a script that will create a specific directory structure, with files in the directories, and specific permissions.

```
#!/bin/bash
mkdir $1
cd $1
mkdir Dir1
chmod 770 Dir1
mkdir Dir2
chmod 775 Dir2
cd Dir1
touch File1
chmod 400 File1
mkdir Dir3
cd Dir3
mkdir Dir4
chmod 750 Dir4
touch File3
chmod 200 File3
touch File4
chmod 666 File4
cd ..  #going back to Dir1
chmod 000 Dir3
cd ..
cd Dir2
touch File2
chmod 640 File2
```

To a computer, Data means lots of bits. So what is a bit?
- Short for "binary digit"
- 2 states:
    - On (1)
    - Off (0)
- Like a light switch
- Computers can turn their bits (light switches) on and off
- Computer implement this using electricity in a "capacitor"
    - Has electrical charge: on / 1
    - No electrical charge: off / 0

What is a byte?

- A group of 8 bits = byte
- We could write the values of the 8 bits in a row
- Example: 01000001 - this is a binary number

Binary Numbers
- Normal system (decimal) -- 10^n
  - Values for any digit are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - What we use every day
- Binary -- 2^n
  - Values for any digit are: 0, 1
  - What computers use

```
 0      1      1      0                binary
2^3    2^2    2^1    2^0
        4   +  2           = 6        decimal
```

Conventions Surrounding Binary Numbers
- Every character in this convention is one byte
- 01000001 = 65 = 'A' in ASCII

How do Bytes relate to computers?
- Computers have "memory"
  - Places to store data then retrieve it alter
- The memory is made up of many, many Bytes (Byte = 8 bits)

Reserving Memory
- Analogy: You go to the hotel front desk and ask for a room (malloc)
- When you are done, you check out and tell them you have left the room (free)

Automatic vs Dynamic Memory
- In C, there are two types of memory (usually 3, but ignoring static for now)
- Automatic - taken care of for you automatically
- Dynamic - you manage it

Minimum Reservation
- When reserving memory, the minimum request is 1 byte
  - You can access individual bits, but the hotel front desk will only let you reserve 1 byte at a time aka 8 bits

Types in C
-- (in 32 bit) --
- Char - ASCII character, 1 byte
  - Values -128 to 127 ---> can represent 256 #'s
- Unsigned char - 1 byte
  - Values 0 to 255
- Int - 4 bytes = 32 bits
  - -2^31 to 2^31 - 1
- Float - 4 bytes
- Double - 8 bytes
- Unsigned int - 4 bytes = 32 bits
  - Values 0 to 2^32 - 1

Will also need to know size of structs which differ.

pointer - 8 bytes

Examples of how much memory is being allocated:

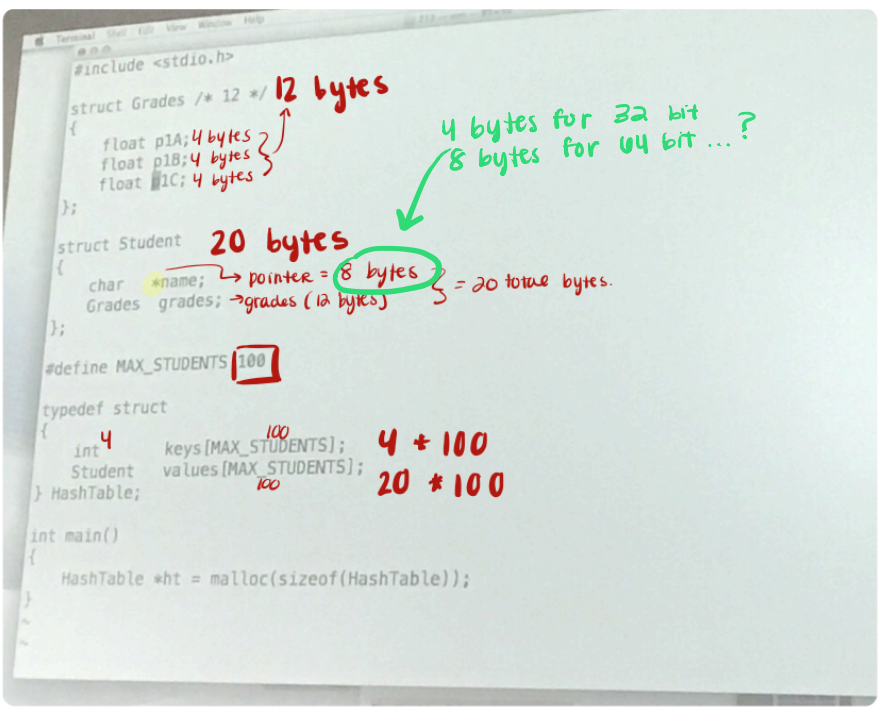
1) arrays:

Arrays

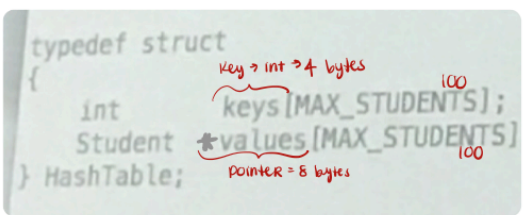
This is just 84 (4*21) bytes. We interpret these 84 bytes as 21 integers, following known conventions

```
#include <stdio.h>

int main()
{      4 bytes * 21 = 84
    int A[21];
    int i;
    for (i = 0 ; i < 21 ; i++)
        A[i] = 3;
    printf("Value of A[10] is %d\n", A[10]);
}
```

2) structs + arrays + global variables



NOTE:
If there is a function pointer (* function) in a struct, this would still count as a pointer → 8 B.



4 * 100
8 * 100
= 1200 bytes allocated in memory

Important Memory Concepts in C:

Automatic vs Dynamic

- You can allocate variables that only live for the invocation of your function
  - Called automatic/stack variables
- You can allocate variables that live for the whole program (until you delete them)
  - Called dynamic/heap variables
- Automatic - taken care of for you automatically (static)
- Dynamic - you manage it (heap)

Pointers

- Pointer: points to memory locations
  - Denoted with '*'
  - Example: "int *p"
    - Pointer to an integer
  - You need pointers to get to heap memory
- Address of: gets the address of memory
  - Operator: '&'
  - Example:
    - int x;
    - int *y = &x;

Memory allocation

- Special built-in function to allocate memory from heap: **malloc**
  - Interacts with Operating System
  - Argument for malloc is how many bytes you want (usually use sizeof(type))
- Also built-in function to deallocate memory:
  - **free**
- Example of free/malloc:

New operator: &

- & == address of --> pointer - memory location
- This will tell you the address of a variable

Enables compiler to see functions that aren't in this file. More on this next week.

```
#include <stdlib.h>
int main()
{
    /* allocates memory */
    int *ptr = malloc(2*sizeof(int));

    /* deallocates memory */
    free(ptr);
}
```

sizeof is a built in function in C. It returns the number of bytes for a type (4 bytes for int).

don't have to say how many bytes to free ... the OS knows

Freeing the memory location for that int.

```
Hanks-iMac:Downloads hank$ cat t.c
#include <stdio.h>

int main()
{
    int A;
    A=4;
    printf("Value of A is %d, address of A is %p\n", A, &A);
    return 0;
}
```

p = pointer

```
Hanks-iMac:Downloads hank$ gcc t.c
Hanks-iMac:Downloads hank$ ./a.out
Value of A is 4, address of A is 0x7fff56486bc8
```

Lecture 4: Beginning on Memory

2B - Sorts 100 numbers in an array

And this is the correct output...

```c
#include <stdio.h>

/*
Author: Alyssa Kelley
Project 2b
*/

int main()
{
    int A[100] = { 252, 657, 268, 402, 950, 66, 391, 285, 133, 577, 649, 166, 987, 314, 954, 214, 920, 230, 904, 801, 40, 552, 369, 682, 202, 712, 395,
        517, 755, 603, 134, 385, 428, 941, 443, 477, 95, 647, 687, 737, 673, 19, 325, 697, 577, 181, 45, 964, 267, 600, 858, 145, 781, 760, 949, 508,
        673, 717, 446, 634, 635, 679, 466, 474, 916, 855, 216, 899, 804, 159, 237, 625, 963, 388, 437, 682, 821, 325, 805, 876, 968, 414, 190, 434, 902,
        794, 752, 729, 77, 243, 705, 953, 765, 637, 765, 158, 166, 599, 70, 927 };

    /* Using bubble sort. This is going to go through each number in the array to compare the two, and
     * possibly change positions so the int in the array that is larger goes behind
     * the other integer, this is putting them in lesser than to greater than order */

    int pos_i;
        while (1){
            int is_it_sorted = 0;   /* Having this remain 0 would indicate that it is sorted. */
            for (pos_i = 0; pos_i < 100-1; pos_i ++){
                if (A[pos_i] > A[pos_i + 1]){
                    int temporary_holder = A[pos_i];
                    A[pos_i] = A[pos_i + 1];
                    A[pos_i + 1] = temporary_holder;
                    is_it_sorted = 1; /* since we has to swap numbers, it is not sorted yet so this is 1 */
                }
            }

            if (is_it_sorted == 0){     /* 0 = it is sorted so we are done! */
                break;
            }
        }

    for (int i = 0; i < 100; ++i){  /* This is going through the array and printing a tab before the integer in the array
        and if the array can be divided by 10 that means we will need to start a new line so there are only 10 integers
        in each row that is printed. */
        printf("\t%i", A[i]);
        if ((i+1)%10 == 0){
            printf("\n");
        }
    }
    return 0;
}
```

```
19      40      45      66      70      77      95      133     134     145
158     159     166     166     181     190     202     214     216     230
237     243     252     267     268     285     314     325     325     369
385     388     391     395     402     414     428     434     437     443
446     466     474     477     508     517     552     577     577     599
600     603     625     634     635     637     647     649     657     673
673     679     682     682     687     697     705     712     717     729
737     752     755     760     765     765     781     794     801     804
805     821     855     858     876     899     902     904     916     920
927     941     949     950     953     954     963     964     968     987
```

Referring back to important memory concepts*

Automatic vs Dynamic memory

```c
#include <stdio.h>

void function1()
{
    int X = 4;
    printf("X is %d\n", X);
    /* X goes out of scope and the end of this function and thus
     * *automatically* disappears and is no longer accessible */
}

int main()
{
    int Z = 6;
    function1();

    /* NEW SCOPE */
    {
        int Y = 5;
    }
    /* Y no longer exists. Automatically allocated and de-allocated */

    /* only valid variable here is Z ... not X, not Y.  */
    printf("Z is %d\n", Z);

    /* Now Z goes away. */
}
```

Dynamic memory works differently

- You allocate it, it stays around until you deallocate it or the program ends
- Important: you need a way to keep track of memory
  - If not, the memory will be "leaked"
- So we need a way of managing dynamic memory
- The concept for doing this in C is POINTERS

Pointers

- Pointer: points to memory location
  - Denoted with '*'
  - Example: "int *p"
    - Pointer to an integer
  - You need pointers to get to dynamic/heap memory
- Address of: gets the address of memory
  - Operator: '&'
  - Example:
    - int x;
    - int *y = &x; <- this example is pointing to an automatic variable, not a dynamic variable

Dynamic Memory Allocation

- Special built-in function to allocate dynamic memory: malloc (Note: heap is the mechanism for how the memory is stored)
- Deallocate memory: free

Automatic vs Dynamic

- Automatic memory lives only for its current scope
- Dynamic memory lives until you free it, or until the program ends
- * see examples of this below*

This is just fine…

```
Hanks-iMac:Downloads hank$ cat scope.c
#include <stdio.h>
#include <stdlib.h>

int *foo1()
{
    int *X = malloc(sizeof(int)*2);
    X[0] = 1;
    X[1] = 2;
    return X;
}

int main()
{
    int *Y = foo1();
    printf("Y[0] is %d and Y[1] is %d\n", Y[0], Y[1]);
}
Hanks-iMac:Downloads hank$ gcc scope.c
Hanks-iMac:Downloads hank$ ./a.out
Y[0] is 1 and Y[1] is 2
```

memory only allocated here

When executing the function, you are giving up your room and you might still be ably to access that room later, but it is not guaranteed (like a hotel, and this is what is happening with allocated memory here)

This is not fine…

```
Hanks-iMac:Downloads hank$ cat bad_scope.c
#include <stdio.h>
#include <stdlib.h>

int *foo1()
{
    int X[2];
    X[0] = 1;
    X[1] = 2;
    return X;
}

int main()
{
    int *Y = foo1();
    printf("Y[0] is %d and Y[1] is %d\n", Y[0], Y[1]);
}
Hanks-iMac:Downloads hank$ gcc bad_scope.c
bad_scope.c:9:12: warning: address of stack memory associated with local variable 'X' returned
      [-Wreturn-stack-address]
    return X;
           ^
1 warning generated.
Hanks-iMac:Downloads hank$ ./a.out
Y[0] is 1 and Y[1] is 2
```

And here it goes wrong…

```
int *foo1()
{
    int X[2];
    X[0] = 1;
    X[1] = 2;
    return X;
}

int *foo2()
{
    int A[2];
    A[0] = 3;
    A[1] = 4;
    return A;
}

int main()
{
    int *Y = foo1();
    int *B = foo2();
    printf("Y[0] is %d and Y[1] is %d\n", Y[0], Y[1]);
    printf("B[0] is %d and B[1] is %d\n", B[0], B[1]);
}
Hanks-iMac:Downloads hank$ gcc bad_scope.c
bad_scope.c:9:12: warning: address of stack memory associated with local variable 'X' returned
      [-Wreturn-stack-address]
    return X;
           ^
bad_scope.c:17:12: warning: address of stack memory associated with local variable 'A' returned
      [-Wreturn-stack-address]
    return A;
           ^
2 warnings generated.
Hanks-iMac:Downloads hank$ ./a.out
Y[0] is 3 and Y[1] is 4
B[0] is 0 and B[1] is 0
```

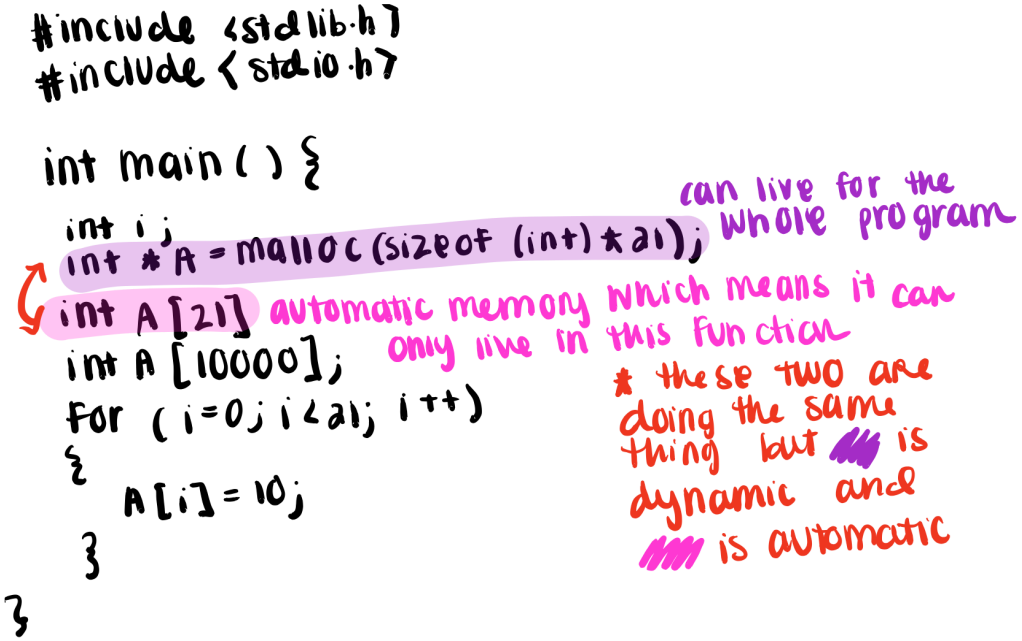
- You need to use malloc for this to run properly!

- Not correct answers.

Lecture 5: Beginning on Memory & character strings

What is an array?

- Block of contiguous memory
- If elements each have size N bytes and there are M elements, then N*M contiguous bytes
- Let A be address of the beginning of the array
- Then A[0] is at "A"
- And A[1] is at "A+N"
- A[2] is at "A+2*N"
- N = bytes, M = elements, N*M = total bytes



Once again... **Automatic memory** lives only for its current scope, and **Dynamic memory** lives until you free it, or until the program ends

Memory Segments

- Your program is divided into 4 segments:
  - Code segment: where the machine code instructions are
  - Data segment: where the global variable live... and other things
  - "Stack": where automatic memory lives
  - "Heap": where dynamic memory lives
  - Note: stack & heap are data structures
- If you access memory addresses outside your segments, you get a "**segmentation fault**" ... which causes a crash

Some special character

- New line = '\n'
- Tab = '\t'
- Backslash character = '\\'
- Null character = '\0' ----> This tells you the array is done!
- Note: %c prints out character
- Note: %s prints out strings (which is just an array of characters followed by \0 which the compiler adds for you)

Character strings

- A character "string" is:
  - An array of type "char"
  - That is terminated by the NULL character
- The C library has multiple functions for handling strings

- Example:

```
char name[7];       // this is 7 because it is allocating for 6 chars letters and
name = "Alyssa";    // 1 null char
printf("Name is %s\n", name);

---->

Name is Alyssa
```

- Note: If you override the null character, then you program won't stop because their is no \0 to indicate completion.
- An example of overriding would be `name[7] = '!';`

Characters are single quotes, Strings are double quotes

Useful C library string functions
- strcpy - string copy
- strncpy - string copy, but just first N characters
- strlen - length of a string

(dest, source)

(dest, source, size)

argc & argv
- Two arguments to every C program
- argc - how many commands line arguments
- argv - an array containing each of the arguments
- "./a.out alyssa kelley"
- --> argc == 3
- --> argv[0] = "a.out", argv[1] = "alyssa", argv[2] = "kelley"

Lecture 6: Wrap up character strings and structs

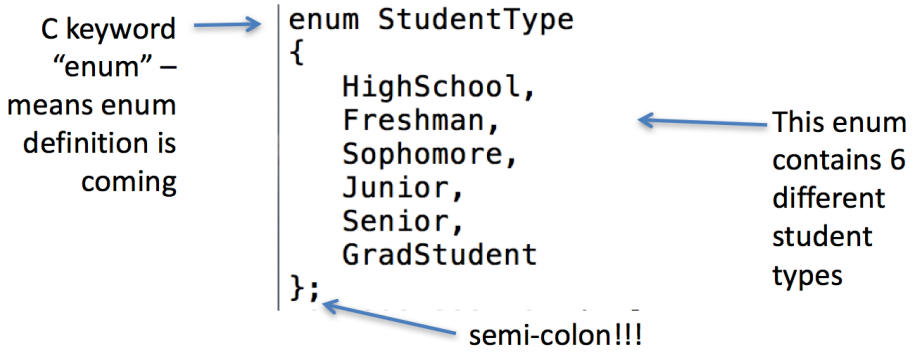
Project 2C - program that pareses 3 strings, then perform the operation.

String 1 = integer, complete this by reading one character at a time
String 2 = operation, all single characters and will be either + or -
String 3 = integer

```c
// Author: Alyssa Kelley
// Project 2C CIS 212

#include <stdio.h>
#include <stdlib.h>

int turn_into_int(char *string_number)
{
    int digit;
    int entire_integer = 0;

    for (digit = 0; digit < 10; digit++)
    {
        if (string_number[digit] != '\0')
        {
        int string_as_int = string_number[digit] - '0'; /* Converting to int */

        entire_integer = ((entire_integer * 10) + string_as_int);
        }

        else if (string_number[digit] == '\0')
        {
            return entire_integer;
        }
    }
    return entire_integer;
}


int main(int argc, char *argv[])
{
    char *number1 = argv[1];
    char *operation = argv[2];
    char *number2 = argv[3];

    int total_number;
    int num_1;
    int num_2;

    num_1 = turn_into_int(number1); /* Calling my function up above to make this into an int */

    num_2 = turn_into_int(number2);

    if (argc != 4)
    {
        printf("Usage: %s <integer> <+ or -> <integer>", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (operation[0] == 43)  /* This is the ASCII representation for + and - */
    {
        total_number = (num_1 + num_2);
    }
    else if (operation[0] == 45)
    {
        total_number = (num_1 - num_2);
    }

    /*printf("%s is the operation we are going to use.\n", operation);
    printf("I am about to do the total_number function call and operation!\n");
    printf("I am going to add %d and %d together!", num_1, num_2);
    These are print statements to check my code. They are commented out until needed.
    */

    printf("%d\n",total_number);
    return 0;
}

```

Enums

- Enums make your own type
  - Type is "list of key words"
- Enums are useful for code clarity
  - Always possible to do the same thing with integers
- Be careful with enums
  - ... you can "contaminate" a bunch of useful words



enum example

```
int AverageAge(enum StudentType st)
{
    if (st == HighSchool)
       return 16;
    if (st == Freshman)
       return 18;
    if (st == Sophomore)
       return 19;
    if (st == Junior)
       return 21;
    if (st == Senior)
       return 23;
    if (st == GradStudent)
       return 26;

    return -1;
}
```

But enums can be easier to maintain than integers

```
enum StudentType
{
   HighSchool,
   Freshman,
   Sophomore,
   Junior,
   Senior,
   PostBacc,
   GradStudent
};
```

```
int AverageAge(enum StudentType st)
{
    if (st == HighSchool)
       return 16;
    if (st == Freshman)
       return 18;
    if (st == Sophomore)
       return 19;
    if (st == Junior)
       return 21;
    if (st == Senior)
       return 23;
    if (st == PostBac)
       return 24;
    if (st == GradStudent)
       return 26;

    return -1;
```

If you had used integers, then this is a bigger change and likely to lead to bugs.

Structs: a complex data type

- Structs: a mechanism provided by C programming language to define a group of variables

- Variables must be grouped together in contiguous memory
- Also makes accessing variables easier... they are all part of the same grouping (the struct)

struct syntax

C keyword "struct" – means struct definition is coming

```
struct Ray
{
    double origin[3];
    double direction[3];
};

int main()
{
    struct Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
}
```

This struct contains 6 doubles, meaning it is 48 bytes

semi-colon!!!

Declaring an instance

"." accesses data members for a struct

typedef

- typedef: tell compiler you want to define a new type

```
struct Ray
{
    double origin[3];
    double direction[3];
};

int main()
{
    struct Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direct
}
```

```
typedef struct
{
    double origin[3];
    double direction[3];
} Ray;

int main()
{
    Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direct = 0;
}
```

saves you from having to type "struct" every time you declare a struct.

- Declare a new type for code clarity

- Also used for enums & unions
  - Same trick as for structs ... typedef saves you a word

Important: struct data member access is different with pointers!

- Pointers use "->" and instances use "."

* malloc

```
typedef struct
{
    double origin[3];
    double direction[3];
} Ray;

int main()
{
    Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
```

```
typedef struct
{
   double origin[3];
   double direction[3];
} Ray;

int main()
{
    Ray *r = malloc(sizeof(Ray));
    r->origin[0] = 0;
    r->origin[1] = 0;
    r->origin[2] = 0;
    r->direction[0] = 1;
    r->direction[1] = 1;
    r->direction[2] = 1;
}
```

Note: You can create a struct within a method, but then you would only be able to use that in that method.

Lecture 7: Finish Memory Stuff, Prep for 2D (pointer worksheet)

Important Memory Concepts in C:

Dereferencing

- There are two operators for getting the value at a memory locations: * and [ ]
  - This is called dereferencing
    - * = "dereference operator"
- `int *p = malloc(sizeof(int)*1);`
- `*p = 2;` // sets memory p points to to have value 2
- `p[0] = 2;` // sets memory p points to to have value 2

Example:

```
int main()
{
    int x;
    int *y = &x;
    x = 5;
}
```

Name	Location	Value
x	0xbec	5
y	0xbe0	0xbec

*y
y[0] ← same concept

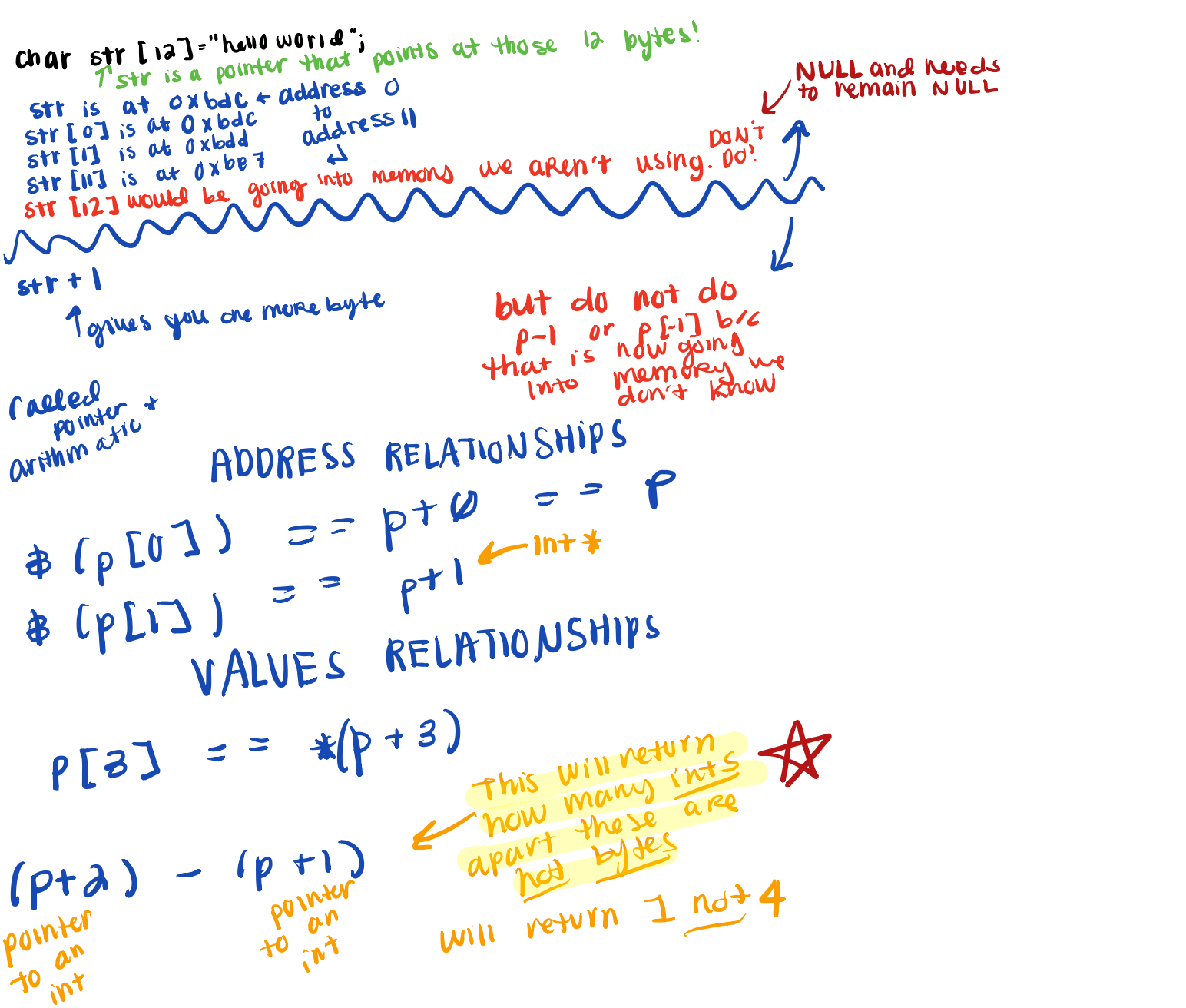
- By dereferencing y, it looks at the value of y and then looks at the value at that location
- *y ---> location y ---> that location ---> the value at that location
- Note: you can also use dereferencing to set a value.
  - Example: `*y = 8;` this will then change the value of `x = 8` because it goes to the value that y is pointing to (0xbec) and sets that value to 8.

More on Arrays

- Arrays lie in contiguous memory (this means consecutive with no gaps)
  - So if you know address to one element, you know address to the rest
- `int *a = malloc(sizeof(int) * 1);`
  - A single integer
  - ... or an array of a single integer
- `int *a = malloc(sizeof(int) * 2);`
  - An array of two integers
  - First integer is at 'a'
  - Second integer is at the address 'a+4'
    - Tricky point here, since C/C++ will refer to it as 'a+1'

Example: $a = 0x8000
$a[2] = 0x8004

this is because it is ints which are 4 bytes total and so it adds to the memory.



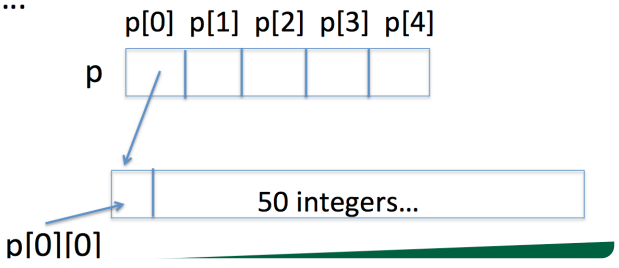
Pointer Arithmetic

- `int *a = malloc(sizeof(int) * 5);`
- C allows you to modify pointer with math operations
  - Called pointer arithmetic
  - "Does the right thing" with respect to type
    - `int *a = malloc(sizeof(int) * 5);`
    - p+1 is 4 bytes bigger than p!
- Then...
  - "p+3" is the same as "&(p[3])" (ADDRESSES)
  - "*(p+3)" is the same as "p[3]" (VALUES)

Pointers to pointers

- `int **p = malloc(sizeof(int*) * 5);`
- `p[0] = malloc(sizeof(int)*50);`
-

....



Null Pointer

- `int *p = NULL;`
- Often stored as address 0x00000000
- Used to initialize something to a known value
  - And also indicate that it is uninitialized

when using NULL you need to #include <string.h>

Project 2D - practice with pointers and dereferencing

CIS 212: Project #2D. Worth 4% of your grade.
Assigned: October 11, 2018
Due: 10am October 18, 2018. NO LATE PASSES, THIS IS DUE IN CLASS.
1001am is late (no credit)
Assignment: fill out this worksheet.

Location	0x8000	0x8004	0x8008	0x800c	0x8010	0x8014	0x8018
Value	0	1	1	2	3	5	8
Location	0x801c	0x8020	0x8024	0x8028	0x802c	0x8030	0x8034
Value	13	21	34	55	89	144	233
Location	0x8038	0x803c	0x8040	0x8044	0x8048	0x804c	0x8050
Value	377	610	987	1597	2584	4181	6765

B[0] = 0x8000
*B[0] = 0
B[1] = 0x801C
*B[1] = 13
B[2] = 0x8038
* B[2] = 377

Code:

```
int *A = 0x8000;
int *B[3] = { A, A+7, A+14 };
```

Note: "NOT ENOUGH INFO" is a valid answer.

Variable	Your Answer	Variable	Your Answer
A (pointing to location of array)	0x8000	(A+6)-(A+3) (A[6]-A[3] = 8-2 = 6; memory at A[5] is 8010, A[2] = 8008, and the value at 0x8010 = 3.)	3
&A (this would be the entire memory address of the array)	NOT ENOUGH INFO	*(A+6)-*(A+2) (A[6]-A[2] = 8-1 = 7)	7
A[2] (This would be the value in this position of the array)	1	A[5]-*(A+4) (A[5]-A[4] = 5-3 = 2)	2
*A (This is the value for A[0])	∅	(A+9)-B[0] (A[9]-A[0] = x8024 - x8000; this will give you 9 memory places forward (in int))	9
B[1][3] (This is the value of B[1] moved up 3 spaces. B[1] = A[7], A[7+3] = A[10] = 55)	55	B[0] (0x8000 = A[0] = 0)	0x8000
*(A+4) (This is the value of A[4] = 3)	3	B[2][2]-A[15] (B[2] = A[14], A[14] = 987; A[15] = 610; 987-610 = 377)	377
(A+6) (This is the location for A[6])	0x8018	&(B[2][3])-&(A[15]) (B[2]→[3] A[17]; &(A[15]) → A[15]; A[17]-A[15] two places in memory difference)	2
B[2][0]-B[1][6] (B[2] = 377; B[1] = A[7], A[7+6] = A[13] = 233; 377 - 233 = 144)	144	*(&A[6]) (Pointing to that memory address = value of A[6] = 8)	8
B[2][0]-B[1][7] (B[2] = 377; B[1] = A[7], A[7+7] = A[14] = 377; 377-377 = ∅)	∅	B[3]-B[1] (A[14] - A[7] = 7)	Not enough info.

how far apart they are (in ints)
value
how far apart they are (in ints)
value
because B is initialized as a pointer array to the values.
value
value
value
X

memory
value
memory
value
value

Project 2E - Extension of 2C (error checking bad inputs)

```c
// Author: Alyssa Kelley
// Project 2C CIS 212

#include <stdio.h>
#include <stdlib.h>

int turn_into_int(char *string_number)
{
    int digit;
    int entire_integer = 0; /* initializing as 0 */
    int sign_of_int = 1; /* defaulting it to postive unless otherwise changed. */
    int starting_digit; /* this will be 0 if number is pos, and 1 if number is neg */
    int first_digit = string_number[0] - '0';

// RE-CREATING STRLEN
    int len = 0;
    char * c = string_number;
    while(*c != 0)
    {
        len++;
        c++;
    }
    if (len > 10)
    {
        printf("ERROR: number has 10 or more digits\n");
        exit(EXIT_FAILURE);
    }

// CHECKING SIGN OF NUMBER

    if (first_digit < 0) /* this means it is negative */
    {
        sign_of_int = -1;
        starting_digit = 1;
    }
    else /* this means it is positive */
    {
        starting_digit = 0;
    }

// GOING THROUGH EACH NUMBER STARTING AT 0 (POS) OR 1 (NEG)
// can be 48-57 in ASCII

    for (digit = starting_digit; digit < 10; digit++)
    {
        if (string_number[digit] != '\0')
        {
            int string_as_int = string_number[digit] - '0'; /* Converting to int */
            entire_integer = ((entire_integer * 10) + string_as_int);
            if ((48 > string_number[digit]) || (string_number[digit] > 57))
            {
            printf("ERROR: number has character that is outside 0-9\n");
            exit(EXIT_FAILURE);
            }

        }
        else if (string_number[digit] == '\0')
        {
            return (entire_integer * sign_of_int);
        }
        // checking length of int
        else if (digit >= 10)
        {
            printf("ERROR: number has 10 or more digits\n");
            exit(EXIT_FAILURE);
        }
    }
    return entire_integer;
}
```

(Continued...)

```c
int main(int argc, char *argv[])
{
    char *number1 = argv[1];
    char *operation = argv[2];
    char *number2 = argv[3];

    int total_number;
    int num_1;
    int num_2;

    num_1 = turn_into_int(number1); /* Calling my function up above to make this into an int */

    num_2 = turn_into_int(number2);

    if (argc != 4)
    {
        printf("Usage: %s <integer> <+ or -> <integer>", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (operation[0] == 43)  /* This is the ASCII representation for + and - */
    {
        total_number = (num_1 + num_2);
    }
    else if (operation[0] == 45)
    {
        total_number = (num_1 - num_2);
    }

    else if ((operation[0] != 43) || (operation[0] != 45))
    {
        printf("ERROR: operation may only be + or -\n");
        exit(EXIT_FAILURE);

    }

    if ((operation[1]) != '\0')
    {
        printf("ERROR: operation may only be + or -\n");
        exit(EXIT_FAILURE);
    }

    printf("%d\n",total_number);
    return 0;
}
```

```
TEST CASES:

./proj2e 4 + 8                          -> 12                   SUCCESS
./proj2e 123 - 123                      -> 0                    SUCCESS
 ./proj2e 123 - -123                    -> 246                  SUCCESS
./proj2e 123 + 123                      -> 246                  SUCCESS
./proj2e 123 + -123                     -> 0                    SUCCESS
./proj2e 123 + 1234                     -> 1357                 SUCCESS
./proj2e 12334 - 789234                 -> -776900              SUCCESS
./proj2e 12sd334 - -789234              -> ERROR: 0-9           SUCCESS
./proj2e 123312323494 - 789234          -> ERROR: 10 digits     SUCCESS
./proj2e 7 -- 789234                    -> ERROR: opperation    SUCCESS
./proj2e 123394 ! 789234                -> ERROR: opperation    SUCCESS

*/
```

File I/O

Streams and file descriptors

- Two ways to access files:
  - File descriptors
    - Lower level interface to files and devices
      - Provides controls to specific devices
  - Streams
    - Higher level interface to files and devices
      - Provides uniform interface; easy to deal with, but less powerful
    - Type: FILE * ---> struct defined by C standard library.
- Note: Streams are more portable, and more accessible to beginning programmers.
- File I/O is a process for reading or writing
  - Open a file
    - Tells Unix you intend to do file I/O
    - Function returns a "FILE *"
      - Used to identify the file from this point forward
    - Checks to see if permissions are valid
  - Read from the file / write to the file
  - Close the file

Opening a file

- `FILE *handle = open(filename, mode);`
- When you are done, close the file with "fclose"

```
C02LN00GFD58:330 hank$ cat rw.c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
   char *hello = "hello world: file edition\n";
   FILE *f = fopen("330", "w");
   fwrite(hello, sizeof(char), strlen(hello), f);
   fclose(f);
}
C02LN00GFD58:330 hank$ gcc rw.c
C02LN00GFD58:330 hank$ ./a.out
C02LN00GFD58:330 hank$ cat 330
hello world: file edition
```

File Position Indicator

- File position indicator: the current location in the file
- If I read one byte, the one byte you get is where the file position indicator is pointing.
  - And the file position indicator updated to point at the next byte
  - But this can be changed.

Fseek

```
int fseek(FILE *stream, long offset, int whence);
```

SEEK_CUR = current position in file. To set this back to the front, set offset = 0.

Ftell

```
int ftell(FILE *stream);
```

- You can go to the end of the file with fseek and then it will tell you how long the file is with ftell and then you know the byte length for that file.

Return values in shells

```
C02LN00GFD58:330 hank$ ./a.out copy.c copy2.c
Copying 697 bytes from copy.c to copy2.c
C02LN00GFD58:330 hank$ echo $?
0
C02LN00GFD58:330 hank$ ./a.out copy.c
Usage: ./a.out <file1> <file2>
C02LN00GFD58:330 hank$ echo $?
1
```

$? is the return value of the last executed command

Printing to terminal and reading from terminal

- In Unix, printing to terminal and reading from terminal is done with file I/O
- Keyboard and screen are files in the file system.

Standard Streams

- Definition: Pre-connected input and output channels between a computer program and its environment (typically a text terminal) when it begins execution
- Three standard streams:
  - stdin (standard input)
  - stdout (standard output)
  - stderr (standard error)

strcpy - (dest, source)
strncpy - (dest, soure, size)
strlen - (str)

printf

- Print to stdout
  - printf("hello world\n");

fprintf

- Just like printf, but to streams
- fprintf(stdout, "helloworld\n);
  - Same as printf
- fprintf(stderr, "helloworld\n);
  - Prints to "standard error"
- fprintf(f_out, "helloworld\n);
  - Prints to the file pointed to by FILE *f_out

files

Buffering and printf

- Important: printf is buffered
- So:
  - Printf puts string in buffer
  - Other things happen
  - Buffer is eventually printed
- But what about a crash?
  - printf puts string in buffer
  - Other things happen... including a crash
  - Buffer is never printed!
- Solutions for this ---> 1) fflush, fprintf(stderr) always flushed.

Project 2F - Write a program that reads the file "2F_binary_file" and this file contains a two-dimensional array of 100 integers, that is 10x10. You are to read the 5x5 top left corner of the array. Aka indices 0-4, 10-14, 20-24, 30-34, and 40-44. You may only read 25 integers total and write out the new 5x5 array to the output file.

```c
// Author: Alyssa Kelley
// Project 2 Part F

#include <stdio.h>
#include <printf.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    FILE *originalFile;
    FILE *myFile;

    originalFile = fopen(argv[1], "r");

    if (argc != 3)
    {
        printf("Usage: %s <file1> <file2>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    myFile = fopen(argv[2], "w");
    fseek(originalFile, 0, SEEK_SET);

    int readback_ints[25];

    int *index_wanted = readback_ints;

    for (int row = 0; row < 5; row++)
    {
        for (int col = 0; col < 5; col++)
        {
            fread(index_wanted, sizeof(int), 1, originalFile);
            readback_ints[col+row*5] = *index_wanted;  /* This is important
            so we do not continue to repeat the column 0-4 and write over it */
            index_wanted++;
        }

        fseek(originalFile, (5*sizeof(int)), SEEK_CUR); /* Need to multiply by 5
        to get the entire bytes of all 5 integers. */
    }
    myFile = fopen(argv[2], "w");
    fclose(originalFile);
    for (int i = 0; i < 25; i++)
    {
        fprintf(myFile, "%d\n", readback_ints[i]);
    }
}
```

current position

moved to 0 → beginning.

Unix shells allows you to manipulate standard streams

- ">" redirect output of program to a file
  - Example:
    - ls > output
    - echo "this is a file" > output2
    - cat file1 file2 > file3
- "<" redirect file to input of program
  - Example:
    - Python < myscript.py
- ">>" concatenate output of program to end of existing file
  - (Or create file if it doesn't exist)
  - Example:
    - echo "I am starting the file" > file1
    - echo "I am adding to the file" >> file1
    - cat file1
      - I am starting the file
      - I am adding to the file
- What is happening here?

```
C02LN00GFD58:330 hank$ mkdir tmp
C02LN00GFD58:330 hank$ cd tmp
C02LN00GFD58:tmp hank$ touch f1
C02LN00GFD58:tmp hank$ ls f1 f2 > out
ls: f2: No such file or directory
C02LN00GFD58:tmp hank$ cat out
f1
```

make dir "tmp"
go into "tmp"
make f1
← since not f2 you get an error, and that error message outputs to the terminal (stderr) but the out file has "f1" in it.

ls is outputting its error messages to stderr

C functions: Fork and Pipe

- Fork: duplicated current program into a separate instance (Note: original - return 0, copy - return not 0)
  - Two running programs!
  - Only differentiated by return value of fork (which is original and which is new)
- Pipe: mechanism for connecting file descriptors between two forked programs (Note: output of 1 becomes the input for the other)
- Through fork and pipe, you can connect two running programs. One writes to a file descriptor, and the other reads the output from its file descriptor
- Only use on special occasion, and one of those occasions is with the shell
- Pipes in Unix shells
  - Represented with "|"
  - Output of one program becomes input to another program

Grep

- Keep lines that match pattern, discard lines that don't match pattern

Sed

- Replace pattern 1 with pattern 2
  - sed s/pattern1/pattern2/g
    - s means substitute
    - g means "global" ... every instance on the line

Etc useful shell things...

- 'Tab' = auto-complete
- Esc = show options for auto-complete
- Ctrl-A = go to the beginning of line
- Ctrl-E = go to end of line
- Ctrl-R = search through history for command

Lecture 9: Call by reference vs call by value - Enum/Struct/Union - Advanced Unix

Call by value / call by reference

- Refers to how paramours are passed to a function
  - **Call by value:** send the value of the variable as a function parameter
    - Side effects in that function don't affect the variable in the calling function

Call by Value

```
C02LN00GFD58:330 hank$ cat cbv.c
#include <stdio.h>

void foo(int x)
{
    x = x+1;
}

int main()
{
    int x = 2;
    foo(x);
    printf("X is %d\n", x);
}
C02LN00GFD58:330 hank$ gcc cbv.c
C02LN00GFD58:330 hank$ ./a.out
X is 2
```

C does not care that foo has an argument variable called "x" and main has an automatic variable called "x". They have NO relation.

- **Call by reference:** send a reference (pointer) as a function parameter
  - Side effects in that function affect the variable in the calling function

Call by reference

```
C02LN00GFD58:330 hank$ cat cbr.c
#include <stdio.h>

void foo(int *x)
{
    *x = *x+1;
}

int main()
{
    int x = 2;
    foo(&x);
    printf("X is %d\n", x);
}
C02LN00GFD58:330 hank$ gcc cbr.c
C02LN00GFD58:330 hank$ ./a.out
X is 3
```

Another example...

```
void Call by value (int x3)
{
    x3 = 3;   0xBBC
}

void  call by Reference (int * x4)   <- do not pass in a value, you pass in the * reference * to a value.
                          pointer
{
    int * x4 = 4;   0xBB8
}

int main()
    int x1 = 1;   = 0xBEC
    int x2 = 2;   0xBE8
    call by value (x1);   = 1
    call by Reference (&x2);   = 4
```

when you have an argument, that argument gets its own memory location.

so when we pass in the memory location of x2 inside call by R. then it copies the memory address of x4 from that file description. so then x2 → x4 and x2 memory goes away so we can't access it anymore

this doesn't happen to x1 in CBV b/c it just passed in the value

Revisiting **Structs: a complex data type**

- Structs: mechanism provided by C programming language to define a group of variables
  - Variables must be grouped together in contiguous memory
- Also makes accessing variables easier...they are all part of the same grouping (the struct)

Function Pointers

- You have a pointers to a function, this pointer can change based on circumstance, when you call the function pointer, it is like calling a known function.
- Example of a Function Pointer #1:

```
#include <stdio.h>
int doubler(int x) { return 2*x; }
int tripler(int x) { return 3*x; }
int main()
{
    int (*multiplier)(int);
    multiplier = doubler;
    printf("Multiplier of 3 = %d\n", multiplier(3));
    multiplier = tripler;
    printf("Multiplier of 3 = %d\n", multiplier(3));
}
128-223-223-72-wireless:cli hank$ gcc function_ptr.c
128-223-223-72-wireless:cli hank$ ./a.out
Multiplier of 3 = 6
Multiplier of 3 = 9
```

function pointer

- Example of a Function Pointer #2:

Function Pointer Example #2

```
128-223-223-72-wireless:cli hank$ cat array_fp.c
#include <stdio.h>
void doubler(int *X) { X[0]*=2; X[1]*=2; };
void tripler(int *X) { X[0]*=3; X[1]*=3; };
int main()
{
    void (*multiplier)(int *);
    int A[2] = { 2, 3 };
    multiplier = doubler;
    multiplier(A);
    printf("Multiplier of 3 = %d, %d\n", A[0], A[1]);
    multiplier = tripler;
    multiplier(A);
    printf("Multiplier of 3 = %d, %d\n", A[0], A[1]);
}
128-223-223-72-wireless:cli hank$ gcc array_fp.c
128-223-223-72-wireless:cli hank$ ./a.out
```

Function pointer — Part of function signature

Don't be scared of extra '*'s ... they just come about because of pointers in the arguments or return values.

- Tricky questions regarding function pointer declarations...

```
void (*foo)
(void);
```

function ptr, no input

```
void (*foo)(int **, char ***);
```

function ptr — ptr to ptr for int — ptr to ptr to ptr for char

```
char **(*foo)(int **, void (*)(int));
```

return type — function pointer — ptr to ptr for int — function pointer → returns void, takes in an int

Function Pointers vs Conditionals

```
128-223-223-72-wireless:cli hank$ cat function_ptr2.c
#include <stdio.h>
int doubler(int x) { return 2*x; }
int tripler(int x) { return 3*x; }
int main()
{
    int (*multiplier)(int);
    int condition = 1;

    if (condition)
       multiplier = doubler;
    else
       multiplier = doubler;

    printf("Multiplier of 3 = %d\n", multiplier(3));
}
```

```
#include <stdio.h>
int doubler(int x) { return 2*x; }
int tripler(int x) { return 3*x; }
int main()
{
    int val;

    if (condition)
       val = doubler(3);
    else
       val = tripler(3);

    printf("Multiplier of 3 = %d\n", val);
}
```

Callbacks

- Callbacks:

function that is called when a condition is met

- Commonly used when interfacing between modules that were developed separately
- ...libraries use callbacks and developers who use the libraries "register" callbacks
-

Background on Images

- Definitions:
  - Image: 2D array of pixels
  - Pixel: A minute area of illumination on a display screen, one of many from which an image is composed
- Pixels are made up of three colors: Red, Green, Blue (RGB)
- Amount of each color scored from 0 to 1
  - 100% Red + 100% Green + 0% Blue = Yellow
  - 100% Red + 0% Green + 100 %Blue = Purple
  - 0% Red + 100% Green + 100% Blue = Cyan
  - 100% Red + 100% Blue + 100% Green = White
  - 1 byte = unsigned char in C = 0 to 255
- Red = (255, 0, 0)
- Green = (0, 255, 0)
- Blue = (0, 0, 255)

- Colors are 0 ---> 1, but how much resolution is needed? How many bits should you use to represent the color?

How to organize a struct for an Image (3D arrays)

- 3D array: width * height * 3 color channels
- Color:
  - Choice 1 = RGB struct (*)
  - Choice 2 = just 3 unsigned chars
- Pixels:
  - Choice 1 = pointer per row
  - Choice 2 = just index it (*)

```
typedefs struct
{
 unsigned char r,g,b;
} Pixel;
```

```
typedef struct
{unsigned
  int height;
  int width;
  Pixel * colors
} Image;
```

Tradeoffs

- Most image formats are hard to read and write
  - So people use libraries to encapsulate reading and writing
  - These libraries are problematic for this class
    - Some will strike out with installing these
    - Different for different platforms
      - Differences in soaring them may lead to problems later

sscanf

- Like printf, but it parses from a string

```
sscanf(str, "%s\n%d %d\n%d\n", magicNum,
        &width, &height, &maxval);
```

assuming str has been read in previously as:

str="P6\n1000 1000\n255\n";

sscanf would give:

magicNum = "P6", width = 1000,

height = 1000, maxval = 255

Project 2G - Struct to contain image, read image from file, write image to file, and modify image

```c
// Author: Alyssa Kelley
// Project 2 G

// Note: This is re-submitted with freeing memory added for extra credit.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct
{
    unsigned char r, g, b;
} Pixel;

typedef struct
{
     unsigned int width, height;
     Pixel *ImageArray; // this would be an array of the colors for the pixels in the image.
} Image;

int GetPixelIndex(int width, int height, int row, int column)
{
    if (column < 0 || column >= width)
    {
        fprintf(stderr, "You asked for a pixel index for column %d, but the only valid values are between 0 and %d\n", column, width-1);
        exit(EXIT_FAILURE);
    }
    if (row < 0 || row >= height)
    {
        fprintf(stderr, "You asked for a pixel index for row %d, but the only valid values are between 0 and %d\n", row, height-1);
        exit(EXIT_FAILURE);
    }
    return row*width + column;
}

void
ReadImage(char *filename, Image *img)
{
    fprintf(stderr, "Reading image \"%s\"\n", filename);

    FILE *f_in;
    f_in = fopen(filename, "r"); // Note: rb is for read binary, I think just r would work too.
    char magicNum[128];
    unsigned int width, height, maxval;
    fscanf(f_in, "%s\n%d %d\n%d\n", magicNum, &width, &height, &maxval); //begining of pixels

    img -> width = width;
    img -> height = height;

    img -> ImageArray = malloc(width * height * sizeof(Pixel));

    Pixel p;

    for (int row = 0; row < height; row++)
    {
        for (int col = 0; col < width; col++)
        {
            fread(&p, sizeof(Pixel), 1, f_in);
            int index = GetPixelIndex(width, height, row, col);
            img -> ImageArray[index] = p;
        }
    }
    fprintf(stderr, "Done reading image \"%s\"\n", filename);
}

void WriteImage(char *filename, Image *img)
{
    fprintf(stderr, "Writing image \"%s\"\n", filename);

    FILE *f_out;
    f_out = fopen(filename, "w");

    fprintf(f_out, "P6\n");
    fprintf(f_out, "%d %d\n", img -> width, img -> height);
    fprintf(f_out, "%d\n", 255);

    fwrite(img -> ImageArray, sizeof(char), (img -> height * img -> width * sizeof(Pixel)), f_out);
    // printf("This is the size: %lu\n", (img -> height * img -> width * sizeof(Pixel)));
    fclose(f_out);
    fprintf(stderr, "Done writing image \"%s\"\n", filename);
}
```

```
void YellowDiagonal(Image *input, Image *output)
{
    fprintf(stderr, "Executing YellowDiagonal\n");

    unsigned int width, height;

    width = input -> width;
    height = input -> height;

    output -> ImageArray = malloc(width * height * sizeof(Pixel));

    output -> width = width;
    output -> height = height;

    for (int row = 0; row < height; ++row)
    {
        for (int col = 0; col < width; ++col)
        {
            int i = GetPixelIndex(width, height, row, col);
            if (row==col)
            {
                output -> ImageArray[i].r = 255;
                output -> ImageArray[i].g = 255;
                output -> ImageArray[i].b = 0;
                /* This 255 / 255 / 0 is the color yellow, and it will set only
                the pixels that are in a diagonal line to this color. */
            }
            else
            {
                /* This is keeping all of the pixels that are not in a diag the same.*/
                output -> ImageArray[i] = input -> ImageArray[i];
            }
        }
    }
    fprintf(stderr, "Done executing YellowDiagonal\n");
    return;
}

void LeftRightCombine(Image *input1, Image *input2, Image *output)
{
    fprintf(stderr, "Executing LeftRightCombine\n");

    unsigned int width1, width2, output_width, height;

    width1 = input1 -> width;
    width2 = input2 -> width;
    height = input1 -> height;
    output_width = width1 + width2;

    output -> ImageArray = malloc((width1 * height * sizeof(Pixel))
                           + (width2 * height * sizeof(Pixel)));

    output -> width = output_width;
    output -> height = height;

    for (int row = 0; row < height; row++)
    {
        for (int col1 = 0; col1 < width1; col1++)
        {
            int i = GetPixelIndex(width1, height, row, col1);
            /* This i is going to be the index of the pixel for Image1. */

            int j = GetPixelIndex(output_width, height, row, col1);

            /* j is going to be the pixel index of the new output image and this will be
            were we put the pixel from image1 at index i in the output ImageArray.

            We do not offset the output array here because this is the first
            image and it starts the output image. */

            output -> ImageArray[j].r = input1 -> ImageArray[i].r;
            output -> ImageArray[j].g = input1 -> ImageArray[i].g;
            output -> ImageArray[j].b = input1 -> ImageArray[i].b;
        }
    }
```

```
    for (int row = 0; row < height; row++)
    {
        for (int col2 = 0; col2 < width2; col2++)
        {
            int i = GetPixelIndex(width2, height, row, col2);
            /* This i is going to be the index of the pixel for Image2. */
            int j = GetPixelIndex(output_width, height, row, col2);

            /* j is going to be the pixel index of the new output image and this will be
            were we put the pixel from image2 at index i in the output ImageArray.

            Offset the output array by the width of the first image so it will
            start right when image1 finishes. */

            output -> ImageArray[j + width1].r = input2 -> ImageArray[i].r;
            output -> ImageArray[j + width1].g = input2 -> ImageArray[i].g;
            output -> ImageArray[j + width1].b = input2 -> ImageArray[i].b;
        }
    }
    fprintf(stderr, "Done executing LeftRightCombine\n");
}

void
VerifyInput()
{
    FILE *f_in = fopen("2G_input.pnm", "r");
    if (f_in == NULL)
    {
        fprintf(stderr, "Cannot open 2G_input.pnm for reading.\n");
        exit(EXIT_FAILURE);
    }
    fclose(f_in);

    FILE *f_out = fopen("2G_output.pnm", "w");
    if (f_out == NULL)
    {
        fprintf(stderr, "Cannot open 2G_output.pnm for writing.\n");
        exit(EXIT_FAILURE);
    }
    fclose(f_out);
}

int main(int argc, char *argv[])
{
    VerifyInput();

    Image source_image;
    Image yellow_diagonal;
    Image left_right;

    ReadImage("2G_input.pnm", &source_image);
    WriteImage("output_of_read.pnm", &source_image);

//#if 0
    YellowDiagonal(&source_image, &yellow_diagonal);
    WriteImage("yellow_diagonal.pnm", &yellow_diagonal);

//#endif

//#if 0
    LeftRightCombine(&source_image, &yellow_diagonal, &left_right);
    WriteImage("2G_output.pnm", &left_right);
//#endif
    free(source_image.ImageArray);
    free(yellow_diagonal.ImageArray);
    free(left_right.ImageArray);
    return 0;
}

// sudo pacman -Sy imagemagick. --> This will allow us to display the image in the VM with the display <image.pnm>
```

Lecture 12 What is a Data Structure?

What is a Data Structure?

- Data structure definitions
  - Textbook: "a systematic way to organize data"
  - Wiki: "data organization, management and storage format that enables efficient access and modification"
  - Note: stack and heap are data structures
  - Already familiar with some:
    - Arrays (built in in C)
    - Python (List, Tuple, Dictionary, Set)

Key Concept

- It organized data
- It enables efficient access

Gettimeofday

- This is a command that allows your to time your function to be able to compare optimization.

```
fawcett:330 childs$ cat timings.C
#include <sys/time.h>
#include <stdio.h>

int main()
{
    int num_iterations = 100000000;
    int count = 0;
    struct timeval startTime;
    gettimeofday(&startTime, 0);
    for (int i = 0 ; i < num_iterations ; i++)
        count += i;
    struct timeval endTime;
    gettimeofday(&endTime, 0);
    double seconds = double(endTime.tv_sec - startTime.tv_sec) +
                     double(endTime.tv_usec - startTime.tv_usec) / 1000000.;
    printf("done executing, took %f\n", seconds);
}
```

# of microseconds

the dot at the end makes it a double so you get a fraction

Big O Notation

Example (simple)

```
int IsStudentInClass(char *thisName, Student *students, int numStudents)
{
    int i;
    for (i = 0 ; i < numStudents ; i++)
        if (strcmp(thisName, students[i].name) == 0)
            return 1;

    return 0;
}
```

for loop = O(N)

- If input data size is "N", then you can describe how many operations occur using N
- If "numStudents" is N, and strcmp takes 50 operations, then 50N
- Idea 1: Don't worry about constants
- Idea 2: just say how many operations with respect to N (number of data elements)
- Answer: O(N)

Example (more complex)

```
int TwoStudentsWithSameName(Student *students, int numStudents)
{
    int i, j;
    for (i = 0 ; i < numStudents ; i++)
        for (j = 0 ; j < numStudents ; j++)
        {
            if (i == j)
                continue;
            if (strcmp(students[i].name, students[j].name) == 0)
                return 1;
        }

    return 0;
}
```

nested for loop = $O(n^2)$

- How many operations now?
  - numStudents * numStudents * 50 (about)
  - ----> O(n^2)

Can we get a better Big O for this one?

```
int TwoStudentsWithSameName(Student *students, int numStudents)
{
    SortNames(students, numStudents); /* how long does this take? */
    int i;
    for (i = 0 ; i < numStudents-1 ; i++)
    {
        if (strcmp(students[i].name, students[i+1].name) == 0)
            return 1;
    }

    return 0;
}
```

nlogn

- 'for' loop over numStudents: O(n)
- Combined: O(nlogn) + O(n) → O(nlogn)
  - (asymptotic analysis)

for loop — nested for loop

n	$\log n$	n	$n \log n$	n^2
4	2	4	8	16
8	3	8	24	64
16	4	16	64	256
32	5	32	160	1,024
64	6	64	384	4,096
128	7	128	896	16,384
256	8	256	2,048	65,536
512	9	512	4,608	262,144
1,024	10	1,024	10,240	1,048,576

2^n → Recursive.

Important: void *

- void * : pointer to memory
- Pointer arithmetic: 1 byte
- Just a location in memory
- Useless without a "cast"
    - "Cast" change to a different type
    - void * p = 0x7ffff;
    - Int *x = (int *) p; // now we can treat this like an int

Memcpy

- Function in C standard library
- Copies data
- We could write it ourselves, but this is easier

Project 2H - Implement 3 structs and 9 functions for Rectangle, Circle and Triangle

```c
// Author: Alyssa Kelley
// Project 2H

#include <stdio.h>

typedef struct
{
    double radius, originX, originY;
} Circle;

typedef struct
{
    double pt1X, pt2X, minY, maxY;
} Triangle;

typedef struct
{
    double minX, minY, maxX, maxY;
} Rectangle;

/* Define these 9 functions */
void InitializeCircle(Circle *c, double radius, double originX, double originY)
{
    c -> radius = radius;
    c -> originX = originX;
    c -> originY = originY;
};

void InitializeRectangle(Rectangle *r, double minX, double maxX, double minY, double maxY)
{
    r -> minX = minX;
    r -> minY = minY;
    r -> maxX = maxX;
    r -> maxY = maxY;

};

void InitializeTriangle(Triangle *t, double pt1X, double pt2X, double minY, double maxY)
{
    t -> pt1X = pt1X;
    t -> pt2X = pt2X;
    t -> minY = minY;
    t -> maxY = maxY;
};


double GetCircleArea(Circle *c)
{
    //Circle c;
    double CircleArea = 3.14159 * c -> radius * c -> radius;
    return CircleArea;
};

double GetRectangleArea(Rectangle *r)
{
    //Rectangle *r;
    double RectArea = (r -> maxX - r -> minX) * (r -> maxY - r -> minY);
    return RectArea;
};

double GetTriangleArea(Triangle *t)
{
    double TriangleArea = (t -> pt2X - t -> pt1X) * (t -> maxY - t -> minY) / 2;
    return TriangleArea;
};


void GetCircleBoundingBox(Circle *c, double *bbox)
{
    //(x-radius) to (x+radius) in X, and (y-radius) to (y+radius) in Y.

    bbox[0] = (c -> originX) - (c -> radius);
    bbox[1] = (c -> originX) + (c -> radius);
    bbox[2] = (c -> originY) - (c -> radius);
    bbox[3] = (c -> originY) + (c -> radius);

};

void GetRectangleBoundingBox(Rectangle *r, double * bbox)
{

    bbox[0] = r -> minX;
    bbox[1] = r -> maxX;
    bbox[2] = r -> minY;
    bbox[3] = r -> maxY;
};

```

```
void GetTriangleBoundingBox(Triangle *t, double *bbox)
{
    //pt1X to pt2X in X, and from minY to maxY in Y.

    bbox[0] = t -> pt1X;
    bbox[1] = t -> pt2X;
    bbox[2] = t -> minY;
    bbox[3] = t -> maxY;

};


/* DO NOT MODIFY AFTER THIS POINT */


void
PrintTriangle(Triangle *t)
{
    double bbox[4];
    double area;
    area = GetTriangleArea(t);
    GetTriangleBoundingBox(t, bbox);
    printf("Triangle has area %f and bounding box [%f->%f], [%f->%f]\n",
           area, bbox[0], bbox[1], bbox[2], bbox[3]);
}

void
PrintRectangle(Rectangle *r)
{
    double bbox[4];
    double area;
    area = GetRectangleArea(r);
    GetRectangleBoundingBox(r, bbox);
    printf("Rectangle has area %f and bounding box [%f->%f], [%f->%f]\n",
           area, bbox[0], bbox[1], bbox[2], bbox[3]);
}

void
PrintCircle(Circle *c)
{
    double bbox[4];
    double area;
    area = GetCircleArea(c);
    GetCircleBoundingBox(c, bbox);
    printf("Circle has area %f and bounding box [%f->%f], [%f->%f]\n",
           area, bbox[0], bbox[1], bbox[2], bbox[3]);
}

int main()
{
    Circle c;
    Triangle t;
    Rectangle r;

    InitializeCircle(&c, 1, 0, 0);
    PrintCircle(&c);
    InitializeCircle(&c, 1.5, 6, 8);
    PrintCircle(&c);
    InitializeCircle(&c, 0.5, -3, 4);
    PrintCircle(&c);

    InitializeRectangle(&r, 0, 1, 0, 1);
    PrintRectangle(&r);
    InitializeRectangle(&r, 1, 1.1, 10, 20);
    PrintRectangle(&r);
    InitializeRectangle(&r, 1.5, 3.5, 10, 12);
    PrintRectangle(&r);

    InitializeTriangle(&t, 0, 1, 0, 1);
    PrintTriangle(&t);
    InitializeTriangle(&t, 0, 1, 0, 0.1);
    PrintTriangle(&t);
    InitializeTriangle(&t, 0, 10, 0, 50);
    PrintTriangle(&t);
}

```

Lecture 13: Abstract Data Types / Stacks

struct name
{
  crap;
};
complex data type

64 bit — size of —

char	1
unsigned char	1
int	4
float	4
double	8
pointer	8

single data types

Data Types

- Simple data types
  - Float, double, int, char, unsigned char
- Complex data types
  - Defined with structs
  - Image, rectangle/square/circle
- Abstract data type
  - Accomplished through function calls
  - You don't have to know the details

Abstract data types

- Two pieces:
  - Define behavior (via function prototypes)
  - Define implementation (via functions)
- You can have more than one implementation for a given behavior

Example -> **Store/Fetch**

function call = abstract data type

- Abstract data type has two methods:
  - **Store**
    - Takes a "key" and a "value"
  - **Fetch**
    - Takes a "key", and returns a "value"
  - Example:
    - Key == UO ID
    - Value == student struct
- Two data structures that can do Store/Fetch: Array and HashTable

One Data Structure for Store/Fetch: Array

```
#define MAX_STUDENTS 1000

typedef struct
{
    int       keys[MAX_STUDENTS];
    Student  *values[MAX_STUDENTS];
    int       curID;
} StoreFetchArray;

void Initialize(StoreFetchArray *arr)
{
    arr->curID = 0;
}
```

- Why not pass in StoreFetchArray * instead of void *?
- A: need this later

- Observation:
- Not very generic (int key, Student value)

```
void ArrayStore(void *a, int key, Student *v)
{
    StoreFetchArray *arr = (StoreFetchArray *) a;
    if (arr->curID >= MAX_STUDENTS)
        exit(EXIT_FAILURE);

    arr->keys[arr->curID]   = key;
    arr->values[arr->curID] = v;
    arr->curID++;
}

Student *ArrayFetch(void *a, int key)
{
    StoreFetchArray *arr = (StoreFetchArray *) a;
    for (int i = 0 ; i < arr->curID ; i++)
        if (arr->keys[i] == key)
            return arr->values[i];

    return NULL;
}
```

for a single Store
O(1)
O(N)

HashTable

- Idea
  - Create a big array with keys and values
  - Don't insert starting from the beginning
  - Instead: Insert into "random" places in the array
  - Not truly random, as it needs to be reproducible
  - Typical: take key and perform some math operation on it

HashTable

Table index	0	1	2	3	4	5	6	7	8	9
Key	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
Student	NUL	NUL	NUL	NUL	NUL	NUL	NUL	NUL	NUL	NUL

Table index	0	1	2	3	4	5	6	7	8	9
Key	-1	-1	-1	-1	...34	-1	-1	-1	-1	-1
Student	NUL	NUL	NUL	NUL	xFF	NUL	NUL	NUL	NUL	NUL

Store:
UO_ID: 951001234
Student: "xFF"

Idea:
Turn UO_ID into an index.
In this case, %10.
(much more complex ideas)

you use % to wrap around

Table index	0	1	2	3	4	5	6	7	8	9
Key	-1	-1	-1	-1	...34	...44	..66	-1	-1	-1
Student	NUL	NUL	NUL	NUL	xFF	xAA	xAF	NUL	NUL	NUL

Store:
UO_ID: 951012344
Student: "xAA"

Idea:
Slot 4 is full ... just use the next slot (slot 5).
Keep going until you find one, including wraparounds

wraparound is an important concept!

Abstract and Concrete Types

- Abstract type: I can think about it abstractly, but it is *only* abstract.
- Concrete type: it is specific, and I can do things with it.

Abstract Type vs Abstract Data Type

- Abstract Type: a class without all the methods implemented
  - "Is A" test. A triangle "is a" shape
- Abstract Data Type (ADT): just an interface, with no notion of implementation
- But - the ideas are very similar (identical in some cases) and mechanism to implement both are very similar

Project 3A - implement an abstract type and a concrete type

Abstract type: Shape

Concrete type: Rectangle / Circle / Triangle

```c
// Author: Alyssa Kelley
// Project 3A

#include <stdio.h>
#include <stdlib.h>

typedef struct
{
    double radius, originX, originY;
} Circle;

typedef struct
{
    double pt1X, pt2X, minY, maxY;
} Triangle;

typedef struct
{
    double minX, minY, maxX, maxY;
} Rectangle;

typedef struct
{
    void *self;
    void (*GetBoundingBox)(void *self, double *bbox); // ERROR with Shape -> void, so this needs to be void *self.
    double (*GetArea)(void *self);
} Shape;

double GetCircleArea(void *circ)
{
    //printf("This is the GetCircleArea function!\n");
    Circle *c = (Circle *) circ; // casting void to be a circle type.
    double CircleArea = 3.14159 * c -> radius * c -> radius;
    //printf("This is radius: %f, X: %f, Y: %f\n", c -> radius, c -> originX, c -> originY);
    //printf("This is the circle area in GetCircleArea function: %f", CircleArea);
    return CircleArea;
};

```

```c
double GetRectangleArea(void *rect)
{
    //printf("This is the GetRectangleArea function!\n");
    Rectangle *r = (Rectangle *) rect;
    double RectArea = (r -> maxX - r -> minX) * (r -> maxY - r -> minY);
    return RectArea;
};

double GetTriangleArea(void *tri)
{
    //printf("This is the GetTriangleArea function!\n");
    Triangle * t = (Triangle *) tri;
    double TriangleArea = (t -> pt2X - t -> pt1X) * (t -> maxY - t -> minY) / 2;
    return TriangleArea;
};


void GetCircleBoundingBox(void *circ, double *bbox)
{
    //printf("This is the GetCircleBoundingBox function!\n");
    //(x-radius) to (x+radius) in X, and (y-radius) to (y+radius) in Y.
    Circle * c = (Circle *) circ;
    bbox[0] = (c -> originX) - (c -> radius);
    bbox[1] = (c -> originX) + (c -> radius);
    bbox[2] = (c -> originY) - (c -> radius);
    bbox[3] = (c -> originY) + (c -> radius);

};

void GetRectangleBoundingBox(void *rect, double * bbox)
{
    //printf("This is the GetRectangleBoundingBox function!\n");
    Rectangle *r = (Rectangle *) rect;
    bbox[0] = r -> minX;
    bbox[1] = r -> maxX;
    bbox[2] = r -> minY;
    bbox[3] = r -> maxY;
};
```

```c
void GetTriangleBoundingBox(void *tri, double *bbox)
{
    //printf("This is the GetTriangleBoundingBox function!\n");
    //pt1X to pt2X in X, and from minY to maxY in Y.
    Triangle *t = (Triangle *) tri;
    bbox[0] = t -> pt1X;
    bbox[1] = t -> pt2X;
    bbox[2] = t -> minY;
    bbox[3] = t -> maxY;

};

/* Define these 9 functions */
Shape *CreateCircle(double radius, double originX, double originY)
{
    //printf("This is the CreateCircle function!\n");
    /*
    Creating the correct shape in the function definition (with a malloc).
    Connecting the shape's dispatch table to circle/rectangle/triangle.
    */
    Circle *c;
    c = malloc(sizeof(Circle));

    c -> radius = radius;
    c -> originX = originX;
    c -> originY = originY;

    Shape *circ_shape;
    circ_shape = malloc(sizeof(Shape));

    circ_shape -> self = c; // this is void

    circ_shape -> GetArea = GetCircleArea; // These are that functions parameters: (circ_shape -> self) but you do not use them here.
    circ_shape -> GetBoundingBox = GetCircleBoundingBox; // (circ_shape -> self, bbox);

    return circ_shape;
};
```

```
Shape *CreateRectangle(double minX, double maxX, double minY, double maxY)
{
    //printf("This is the CreateRectangle function!\n");

    Rectangle *r;
    r = malloc(sizeof(Rectangle));

    r -> minX = minX;
    r -> minY = minY;
    r -> maxX = maxX;
    r -> maxY = maxY;

    Shape *rect_shape;

    rect_shape = malloc(sizeof(Shape));

    rect_shape -> self = r;

    rect_shape -> GetArea = GetRectangleArea; // (rect_shape -> self); //(void *r);
    rect_shape -> GetBoundingBox = GetRectangleBoundingBox; // (rect_shape -> self, bbox);

    return rect_shape;
};
```

```
Shape *CreateTriangle(double pt1X, double pt2X, double minY, double maxY)
{
    //printf("This is the CreateTriangle function!\n");

    Triangle *t;
    t = malloc(sizeof(Triangle));

    t -> pt1X = pt1X;
    t -> pt2X = pt2X;
    t -> minY = minY;
    t -> maxY = maxY;

    Shape *tri_shape;
    tri_shape = malloc(sizeof(Shape));

    tri_shape -> self = t;

    tri_shape -> GetArea = GetTriangleArea; // (tri_shape -> self);
    tri_shape -> GetBoundingBox = GetTriangleBoundingBox; // (tri_shape -> self, bbox);

    return tri_shape;
};
```

```
162
163 double GetArea(Shape *s)
164 {
165     //printf("This is the GetArea function!\n");
166     double area = s -> GetArea(s->self);
167     return area;
168 };
169
170 void GetBoundingBox(Shape *s, double *bbox)
171 {
172     //printf("This is the GetBoundingBox function!\n");
173     s -> GetBoundingBox(s->self, bbox);
174 };
175
176 int main()
177 {
178     //printf("I am in main!\n");
179     Shape *shapes[9];
180     int    i;
181     shapes[0] = CreateCircle(1, 0, 0);
182     shapes[1] = CreateCircle(1.5, 6, 8);
183     shapes[2] = CreateCircle(0.5, -3, 4);
184
185     shapes[3] = CreateRectangle(0, 1, 0, 1);
186     shapes[4] = CreateRectangle(1, 1.1, 10, 20);
187     shapes[5] = CreateRectangle(1.5, 3.5, 10, 12);
188
189     shapes[6] = CreateTriangle(0, 1, 0, 1);
190     shapes[7] = CreateTriangle(0, 1, 0, 0.1);
191     shapes[8] = CreateTriangle(0, 10, 0, 50);
192
193     for (i = 0 ; i < 9 ; i++)
194     {
195         double bbox[4];
196         printf("Shape %d\n", i);
197         printf("\tArea: %f\n", GetArea(shapes[i]));
198         GetBoundingBox(shapes[i], bbox);
199         printf("\tBbox: %f-%f, %f-%f\n", bbox[0], bbox[1], bbox[2], bbox[3]);
200         free(shapes[i]); // freeing the memory we malloc in the Create functions.
201     }
202
203 }
```

Stacks

- A data structure
- 2 methods: push and pop
- Sometimes a third: peek

Example: Stack of Integers

```
#define MAX_STACK_SIZE 100

typedef struct
{
   /* your data members go here */
} Stack;

void Initialize(Stack *s)
{
}

void Push(Stack *s, int X)
{
}

int Pop(Stack *s)
{
}
```

```
int main()
{
    Stack s;
    int X;
    Initialize(&s);
    Push(&s, 5);
    Push(&s, 6);
    X = Pop(&s);
    printf("Stacked popped %d\n", X);
    Push(&s, 7);
    X = Pop(&s);
    printf("Stacked popped %d\n", X);
    X = Pop(&s);
    printf("Stacked popped %d\n", X);
}
```

```
Hanks-iMac:3B hank$ ./a.out
Stacked popped 6
Stacked popped 7
Stacked popped 5
```

Stack: Asymptotic Complexity (BigO)

- Push - O(1)
- Pop - O(1)
- Store - O(1)
- Fetch - O(n)
    - Pop each element and look

Project 3B - Implementing Stacks with "Reverse Polish Notation"

```c
// Author: Alyssa Kelley
// Project 3B – continuation of 2E
#include <stdio.h>
#include <stdlib.h>

#define MAX_STACK_SIZE 10 // this is for int int operation

typedef struct
{
    // data member #1
    // data member #2 = array
    int counter;
    int current_stack[MAX_STACK_SIZE]; // for the equation, could be 2 for just the ints in it

} Stack;

void Initialize(Stack *s)
{
    s -> counter = 0;
}

void Push(Stack *s, int x)
{
    // Two lines of code, with 5 more lines for error checking
    if ((s -> counter) == MAX_STACK_SIZE)
    {
        printf("Stack is full!\n");
        exit(EXIT_FAILURE);
    }
    else
    {
        s -> current_stack[(s -> counter)] = x;
        (s -> counter)++;
    }
    // printf("I am pushing %d\n", x);
}

int Pop(Stack *s)
{
    // Two lines of code, with 5 more lines for error checking
    --(s -> counter);
    //printf("I am pushing %d\n", s -> current_stack[(s -> counter)]);
    return s -> current_stack[(s -> counter)];

}
```

```c

int turn_into_int(char *string_number)
{
    int digit;
    int entire_integer = 0; /* initializing as 0 */
    int sign_of_int = 1; /* defaulting it to postive unless otherwise changed. */
    int starting_digit; /* this will be 0 if number is pos, and 1 if number is neg */
    int first_digit = string_number[0] - '0';

// RE-CREATING STRLEN
    int len = 0;
    char * c = string_number;
    while(*c != 0)
    {
        len++;
        c++;
    }
    if (len > 10)
    {
        printf("ERROR: number has 10 or more digits\n");
        exit(EXIT_FAILURE);
    }

// CHECKING SIGN OF NUMBER
```

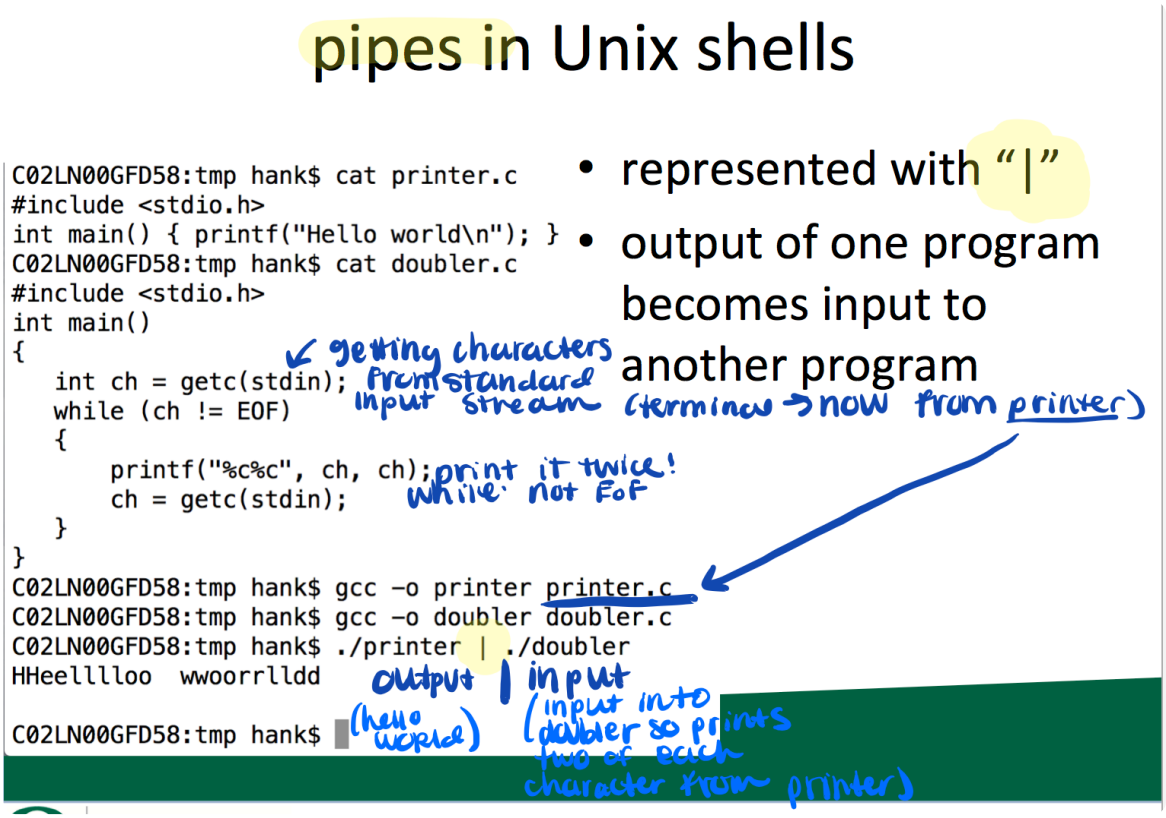
```
    if (first_digit < 0) /* this means it is negative */
    {
        sign_of_int = -1;
        starting_digit = 1;
    }
    else /* this means it is positive */
    {
        starting_digit = 0;
    }

    for (digit = starting_digit; digit < 10; digit++)
    {
        if (string_number[digit] != '\0')
        {
            int string_as_int = string_number[digit] - '0'; /* Converting to int */
            entire_integer = ((entire_integer * 10) + string_as_int);
            if ((48 > string_number[digit]) || (string_number[digit] > 57))
            {
            printf("ERROR: number has character that is outside 0-9\n");
            exit(EXIT_FAILURE);
            }
        }
        else if (string_number[digit] == '\0')
        {
            return (entire_integer * sign_of_int);
        }
        // checking length of int
        else if (digit >= 10)
        {
            printf("ERROR: number has 10 or more digits\n");
            exit(EXIT_FAILURE);
        }
    }
    return entire_integer;
}
int main(int argc, char *argv[])
{
    //char *number1 = argv[1];
    // char *operation = argv[2];
    // char *number2 = argv[3];
    int answer;
    int num_1;
    int num_2;
    Stack s;
    Initialize(&s);

    for (int i=1; i < argc; i++) // going through each argv
    {
        char *str = argv[i];
        // address of s creates the pointer.
        if (str[0] == '+' || str[0] == '-' || str[0] == 'x')
        {
            num_2 = Pop(&s); //popping off the most recently pushed int
            //printf("This is my num_2 popped off: %d\n", num_2);

            num_1 = Pop(&s); //popping off the first pushed int
            //printf("This is my num_1 popped off: %d\n", num_1);

            if (str[0] == '+')
            {
                answer = (num_1 + num_2);
                //printf("This is %d + %d = %d\n", num_1, num_2, answer);
            }

            else if (str[0] == '-')
            {
                answer = (num_1 - num_2);
                //printf("This is %d - %d = %d\n", num_1, num_2, answer);
            }

            else if (str[0] == 'x')
            {
                answer = (num_1 * num_2);
                //printf("This is %d * %d = %d\n", num_1, num_2, answer);
            }
            Push(&s, answer);
        }
```

```
148
149         else
150         {
151             // turn into int
152             int number = turn_into_int(&str[0]);
153             //printf("This is my number: %d\n", number);
154             Push(&s, number); // pushing the int to the stack
155         }
156     }
157     // if ((operation[1]) != '\0')
158     // {
159     //     printf("ERROR: operation may only be + or -\n");
160     //     exit(EXIT_FAILURE);
161     // }
162     printf("The total is %d\n",answer);
163     return 0;
164 }
165
```

Lecture 14: Advances Unix

Printing to terminal and reading from terminal
- In Unix, printing to terminal and reading from terminal is done with file I/O (fprintf)

Unix shells allow you to manipulate standard streams
- ">" redirect output of program to a file
- Example:
  - ls > output
  - echo "this is a file" > output2
  - Cat file1 file2 > file3
    printing file1
    printinf



Grep
- Keep lines that match pattern, discard lines that don't match pattern
  - Note: grep -v throws away the lines that matches this

Script Notes:

mkdir = make directory
touch = touch a file or make a file is it is not already existing
cd = change directory
chmod = change file name
    chmod 750 <filename>
    User gets 7 (rwx)
    Group gets 5 (rx)
    Other gets 0 (no access)
mv = moves or renames file and directories
cp = makes copies of files and directories
rm = remove file
rmdir = remove directory

chgrp = changes the group for a file or directory
    chgrp <group> <filename>

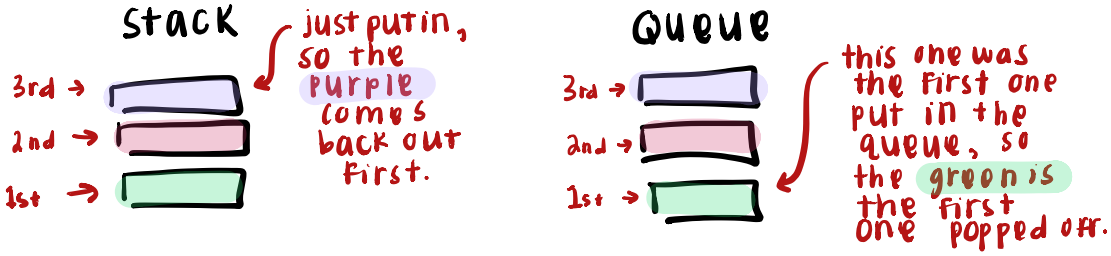
PERMISSIONS:

7 = full = 111
6 = R & W = 110
5 = R & X = 101
4 = R = 100
3 = W & X = 011
2 = W = 010
1 = X = 001
0 = none = 000

Lecture 15: Queues and more

Queues

- Definition: A line or sequence or data awaiting their turn to be attended to or to proceed
- It is just a word for a "line"



Queues vs. Stacks

- Queue: first in first out
- Stack: last in last out

Queue: methods

Initialize

Enqueue (enter the queue)
- Line up

- Dequeue (exit the queue)
  - Get called to the register

- Sometimes: Front
  - Look at the first person in the line, but don't get them out of line
  - Also useful to see if there is anyone in the line at all, IsEmpty

Queue: first pass at implementation

```
#define MAX_ELEMENTS 10   <-- Global variable

typedef struct
{
    int num_elements;
    int elements[MAX_ELEMENTS];
} Queue;   <--- Struct

void Initialize(Queue *q)
{
    q->num_elements = 0;
}

void Enqueue(Queue *q, int val)
{
    if (q->num_elements+1 >= MAX_ELEMENTS) { /* ERROR */ }
    q->elements[q->num_elements] = val;
    q->num_elements++;
}

int main()
{
    Queue q;
    Enqueue(&q, 10); /* q.elements[0] = 10; q.num_elements = 1; */
    Enqueue(&q, -5); /* q.elements[1] = -5; q.num_elements = 2; */
    Enqueue(&q, 14); /* q.elements[2] = 14; q.num_elements = 3; */
    Enqueue(&q, -8); /* q.elements[3] = -8; q.num_elements = 4; */
}
```

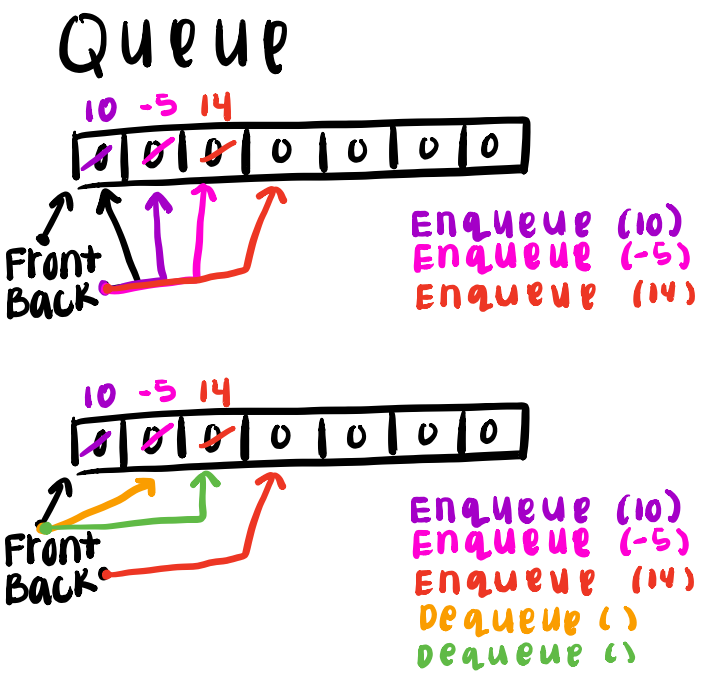
Complexity = O(1)

Dequeue

```
int Dequeue(Queue *q)
{
    if (q->num_elements == 0) { /* ERROR */ }
    int rv = q->elements[0];
    for (int i = 0 ; i < q->num_elements-1 ; i++)
        q->elements[i] = q->elements[i+1];
    return rv;
}

int main()
{
    Queue q;
    int   x;
    Enqueue(&q, 10); /* q.elements[0] = 10; q.num_elements = 1; */
    Enqueue(&q, -5); /* q.elements[1] = -5; q.num_elements = 2; */
    Enqueue(&q, 14); /* q.elements[2] = 14; q.num_elements = 3; */
    Enqueue(&q, -8); /* q.elements[3] = -8; q.num_elements = 4; */
    /* Now: q.elements = { 10, -5, 14, -8, ...}; q.num_elements = 4; *
    x = Dequeue(&q);  /* X == 10 */
    /* Now: q.elements = { -5, 14, -8, ...}; q.num_elements = 3; */
    x = Dequeue(&q);  /* X == -5 */
    /* Now: q.elements = { 14, -8, ...}; q.num_elements = 2; */
}
```

Complexity = O(n)

Visual Example of Queues:



Project 3C - Implement queues (Only O(1))
- Matching surgeries together with donors, recipients and hospitals

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

#define QUEUE_SIZE 10

// Author: Alyssa Kelley
// Project 3C

/*
Notes -> 5 queses: female donors, female recipients, male donors, male recipients, hospitals
*/

typedef struct
{
    int num_elements; // counter for the number of elements in each queue
    char *strings[QUEUE_SIZE];
    int front;
    int back;
} Queue;

void InitializeQueue(Queue *q)
{
    q -> num_elements = 0;
    q -> front = 0;
    q -> back = 0;
}

void Enqueue(Queue *q, char *string)
{ // see Lecture 15 Slide 16

    if (q -> num_elements+1 >= QUEUE_SIZE)
    {
        printf("Error 1\n");
        exit(EXIT_FAILURE);
    }

    q -> strings[q -> back%QUEUE_SIZE] = string;
    q -> back++;
    q -> num_elements++;
}

char * Dequeue(Queue *q)
{
    // Source: GeeksforGeeks
    // https://www.geeksforgeeks.org/circular-queue-set-1-introduction-array-implementation/

    char * data = q->strings[q->front%QUEUE_SIZE]; // First in, first out!
    if (q -> num_elements <= 0)
    {
        printf("Error 2 -- %d\n", q -> num_elements); // Error checking to see if empty
        exit(EXIT_FAILURE);
    }

    q -> front++; // always want to increase front here to get next value.
    q -> num_elements--; // taking something out of the queue so decrease the size.
    return data;
}

void *Front(Queue *q)
{
    void * front_person;
    if (q -> num_elements == 0) //if (q -> front == -1)
    {
        front_person = NULL;
        //printf("There are no unmatched entries.\n");
        return front_person;
    }
    else
    {
        front_person = q -> strings[q -> front%QUEUE_SIZE];
        return front_person;
    }
}

```

```c
void PrintQueue(Queue *q)
{
    int i;
    printf("Printing queue %p\n", q);
    printf("\tThe index for the front of the queue is %d\n", q->front);
    printf("\tThe index for the back of the queue is %d\n", q->back);
    if (q->front == q->back)
    {
        printf("\tThe queue is empty.\n");
        return;
    }
    int back = q->back;
    if (q->back < q->front)
    {
        // wrapped around, so modify indexing
        back += QUEUE_SIZE;
    }
    for (i = q->front ; i < back ; i++)
    {
        printf("\t\tEntry[%d] = \"%s\"\n", i%QUEUE_SIZE, q->strings[i%QUEUE_SIZE]);
    }
}

```

```c
void
PrettyPrintQueue(Queue *q, char *type)
{
    //printf("Checking queue %s\n", type);
    if (Front(q) == NULL)
    //if (q -> num_elements == 0)
    {
        printf("No unmatched entries for %s\n", type);
    }
    else
    {
        char *s;
        printf("Unmatched %s:\n", type);
        while (q-> num_elements > 0) // Changed part of this function so it checks if not zero
        {                            // first and then if goes to the if statement.
            if ((s = Dequeue(q)))
            {
                printf("\t%s\n", s);
            }
        }
    }
}

```

```c
char *NewString(char *s)
{
    int len = strlen(s);
    char *rv = malloc(len);
    strcpy(rv, s);
    rv[len-1] = '\0'; /* take out newline */
    return rv;
}

```

```c
int main(int argc, char *argv[])
{
    // Queue q;
    // InitializeQueue(&q);
    // PrintQueue(&q);
    // Enqueue(&q, "hello");
    // PrintQueue(&q);
    // Enqueue(&q, "world");
    // PrintQueue(&q);
    // printf("Dequeue: %s\n", Dequeue(&q));
    // PrintQueue(&q);

    FILE *f_in;
    f_in = fopen(argv[1], "r");

    Queue female_donors;
    InitializeQueue(&female_donors);
    Queue female_recipients;
    InitializeQueue(&female_recipients);
    Queue male_donors;
    InitializeQueue(&male_donors);
    Queue male_recipients;
    InitializeQueue(&male_recipients);
    Queue hospitals;
    InitializeQueue(&hospitals);

    char line[256];
    char *name;
    char *hospital_name;
    int count = 0;
    while(fgets(line, 256, f_in) != NULL)
    {
        char * rv;
        rv = NewString(line); // using the rv variable to reflect the return of Hanks NewString function.
        name = (rv + 4); // the regular names would be +4 in order to get past -> Letter:Letter:[here is 4]

        if (rv[0] == 'R')
        { // this would ideally give you the name
            if (rv[2] == 'F')
            {
                Enqueue(&female_recipients, name);
                //printf("I am enqueueing for female_recipients: %s!\n", name);
                // put name into female recip queue
            }
            if (rv[2] == 'M')
            {
                Enqueue(&male_recipients, name);
                // put name into mail recip queue
            }

```

```
        }
        if (rv[0] == 'D')
        {
            if (rv[2] == 'F')
            {
                Enqueue(&female_donors, name);
                // put name into female donor queue
            }
            else if (rv[2] == 'M')
            {
                Enqueue(&male_donors, name);
                // put name into mail donor queue
            }
        }
        if (rv[0] == 'H')
        {
            hospital_name = (rv +2); // You use +2 instead of +4 since Hospitals only have H: before the hospital name.
            //printf("This is the hospital name: %s \n", hospital_name);
            Enqueue(&hospitals, hospital_name);
        }

        // then if based on first part of string, then put in correct queue

        // This is error checking I was doing that is commented out:
        // printf("This is the female recip array: %s, %s, %s\n", female_recipients.strings[0], female_recipients.strings[1], female_recipients.strings[2]);
        // printf("This is the female donor array: %s, %s, %s\n", female_donors.strings[0], female_donors.strings[1], female_donors.strings[2]);
        // printf("This is the male recip array: %s, %s, %s\n", male_recipients.strings[0], male_recipients.strings[1], male_recipients.strings[2]);
        // printf("This is the male donor array: %s, %s, %s\n", male_donors.strings[0], male_donors.strings[1], male_donors.strings[2]);
        // printf("This is the hospitals array: %s, %s, %s\n", hospitals.strings[0], hospitals.strings[1], hospitals.strings[2]);

        char * donor;
        char * recip;
        char * hosp;
```

```
        if (female_donors.num_elements >= 1 && female_recipients.num_elements >= 1 && hospitals.num_elements >= 1)
        {
            donor = Dequeue(&female_donors);
            recip = Dequeue(&female_recipients);
            hosp = Dequeue(&hospitals);
            printf("MATCH: %s donates to %s at hospital %s\n", donor, recip, hosp);
        }

        else if (male_donors.num_elements >= 1 && male_recipients.num_elements >= 1 && hospitals.num_elements >= 1)
        {
            donor = Dequeue(&male_donors);
            recip = Dequeue(&male_recipients);
            hosp = Dequeue(&hospitals);
            printf("MATCH: %s donates to %s at hospital %s\n", donor, recip, hosp);
        }
        // Error checking below:
        // printf("(1) This is the size of the queue: %d\n", female_donors.num_elements);
    }
    // Error checking below:
    //printf("(2) This is the size of the queue: %d\n", female_donors.num_elements);

    PrettyPrintQueue(&female_donors, "female donors");
    //printf("Done check for female donors.\n");
    PrettyPrintQueue(&female_recipients, "female recipients");
    //printf("Done check for female recipients.\n");
    PrettyPrintQueue(&male_donors, "male donors");
    //printf("Done check for male donors.\n");
    PrettyPrintQueue(&male_recipients, "male recipients");
    //printf("Done check for male recipients.\n");
    PrettyPrintQueue(&hospitals, "hospitals");
    //printf("Done check for hospitals donors.\n");
}
```

DRAM vs. NV-RAM

- DRAM: Dynamic Random Access Memory
  - Stores data
  - Each bit in separate capacitor within integrated circuit
  - Loses charge over time and must be refreshed
  - ---> volatile memory
- NV-RAM: Non-Volatile Random Access Memory
  - Stores data
  - Information unaffected by power cycle
  - Examples: Read-Only Memory (ROM), flash, hard drive, floppy drive, ...

Relationship to File Systems

- File systems could be implemented in DRAM
- However, almost exclusively on NV-RAM
  - Most often hard drives
- Therefore, properties and benefits of file systems are often associated with properties and benefits of NV-RAM

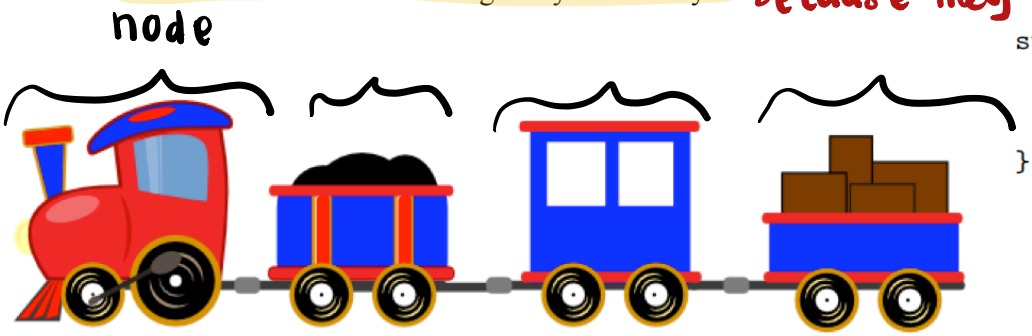
(probably not tested on ↑)

Lecture 16: Linked lists

Linked Lists

- Linked lists and doubly linked lists
- Linked lists are in project 3D where we implement a "map" using linked lists
- Doubly linked lists will tie into future data structures (Deques)
- A linked list is a linear collection of data elements, in which the linear order is not given by their physical placement in memory
- Instead, each node in the list points at the next element in the list
- Such a structure enables insertion or removal of elements without reallocation or reorganization of the entire structure, since the data items do not need to be stored contagiously in memory

because they are pointers.




```
struct node {
    struct node *next;
    char *value;
};
```

```
struct node *head;
```

```
int search(char *university, struct node *theList) {
    struct node *p;

    for (p = theList; p != NULL; p = p->next)
        if (strcmp(p->value, university) == 0)
            return 1;
    return 0;
}
```

- Compared to arrays, with linked lists, you can do insertions to the beginning, middle, and end, but the indexing is much slower, and linked lists require extra storage.

So storing in a Linked List is fast, but fetching from a linked list is slow.

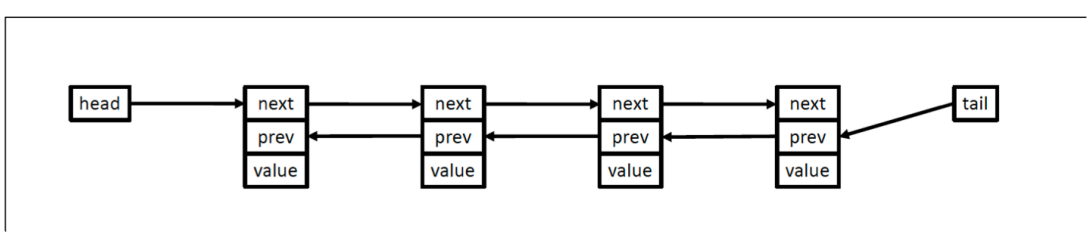
Doubly Linked List

```
struct node {
    struct node *next;
    char *value;
};
```

Linked List

```
typedef struct node {
    struct node *next;
    struct node *prev;
    void *value;
} Node;
```

Doubly Linked List



- Example of **doubly linked lists** would be an iPod playlist where you need to reverse to get to the previous song.

Lecture 17: Hash Tables, Maps, Finished Linked Lists

Note regarding strncmp: If you want to do it, then compare it to 0 or 1 since it returns a boolean. 0 = True, and 1 = False.

HashTable

- Idea: Create a big array with keys and values
- Don't insert starting from the beginning, instead, insert into "random" places in the array but not random entirely since it needs to be reproducible. Typically: take the key and perform some math operation on it.
- *Refer back to Hashtables from above*

Hash Tables: Dealing with Collision

- Collision
  - If you have two values both mapped to index/key 4, this is called a "collision"
  - You deal with a collision by...
    - Open addressing (what we are going to do) double hashing.
    - More ways ...

Probing

- When there is a collision, and you have to find an entry to store it
- "Probing" = how to look for the next open entry
- "Linear probing" = look at the next entry, then the next one, etc.

```
void HashTableStore(void *a, int key, Student *v)
{
    HashTable *ht = (HashTable *) a;
    int iter = 0;
    for (int i = 0 ; i < MAX_STUDENTS ; i++)
    {
        int idx = (i+key)%MAX_STUDENTS;
        if (ht->keys[idx] == -1)
        {
            ht->keys[idx]   = key;
            ht->values[idx] = v;
        }
    }

    exit(EXIT_FAILURE);
}

Student *HashTableFetch(void *a, int key)
{
    HashTable *ht = (HashTable *) a;
    int iter = 0;
    for (int i = 0 ; i < MAX_STUDENTS ; i++)
    {
        int idx = (i+key)%MAX_STUDENTS;
        if (ht->keys[idx] == key)
            return ht->values[idx];
        if (ht->keys[idx] == -1)
            return NULL;
    }

    return NULL;
}
```

Clustering

- Clustering happens with bad hash functions
- This makes store and fetch slow
- Ideas:
  - Probe more intelligently
  - Better hash function

Quadratic Probing

- Approach for avoiding clusters in hash table
  - H = HASH(key)
- Linear probing looks at
  - H, H + 1, H + 2, H + 3, etc
- Quadratic probing looks at
  - H, H + 1, H + 4, H + 9, H + 16, etc
  - * wrap arounds *

Double Hashing

- Idea: two hash functions.... Hash1 and Hash2
- For key K, apply Hash1 to get index Hash1(K)
- If Hash1(K) is not open (collision) then:
  - Apply Hash2 to get index Hash2(K)
  - Do probing with
    - Hash1(K) + i * Hash2(K)
      iterator
  - Need to be careful you cover all entries in the table

Chaining: Linked Lists

- Called: "Separate chaining"
- Idea: every element in the array is the head of a linked list
- When there is a collision, you put the new item at the front of the linked list
- Some linked lists may get long
- Requires linked list stuff (malloc, extra pointers, etc)

Deletions

- Chaining
  - Easy - Go to the linked list and find the item to remove
- Open Addressing
  - "Lazy Deletion" - don't delete it, but indicate that it is removed

Maps

- A map stores tuples (key, value)
- You store both key and value
- You retrieve with just the key (and it give the value)
- If the key is not in the map, then some appropriate value is returned
- ---> Python equivalent = Dictionaries {Key: Value}

How to Implement a Map - Linked Lists

- Make a tuple type (key, value)
- Make a linked list of that tuple type
- Store: insert the tuple into the list
- Fetch: iterate over the linked list, comparing the desired key with the key element of each tuple in the linked list

How to Implement a Map - Hash Tables

- Make a hash function to convert keys to indices
- Decide on hash table details (open addressing, chaining)
- Store: hash key, insert tuple into hash table
- Fetch: hash desire key, compare with key in tuple (repeat as appropriate)

Project 3D - Implement two types of maps (Linked list and Hash table)

```c
/*
 * I believe I have completed:
 *  Map based on linked lists: yes
 *  Double hashing: yes
 *  Performance study: yes
 */

// Author: Alyssa Kelley with starter code for hashtables and File I/O material from Hank Childs.
// Project 3D
// Goal: We are comparing two data structures and comparing their run time to see which is more
    optimized.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <math.h>

typedef struct
{
    char    *symbol;
    char    *name;
    float    lastSale;
    float    marketCap;
    int      IPOyear;
} Company;

void PrintCompany(Company *c)
{
    printf("%s:\n", c->name);
    printf("\tSymbol: %s\n", c->symbol);
    /* .2f: only print two digits after the period. */
    printf("\tLast Sale: %.2f\n", c->lastSale);
    printf("\tMarket Capitalization: %.2f\n", c->marketCap);
    printf("\tYear of Initial Public Offering: %d\n", c->IPOyear);
}

void ReadFile(const char *filename, Company **companies_rv, int *numCompanies_rv)
{
    int  i, j;

    if (filename == NULL)
    {
        fprintf(stderr, "No filename specified!\n");
        exit(EXIT_FAILURE);
    }
    FILE *f_in = fopen(filename, "r");
    if (f_in == NULL)
    {
        fprintf(stderr, "Unable to open file \"%s\"\n", filename);
        exit(EXIT_FAILURE);
    }

    fseek(f_in, 0, SEEK_END);
    int numChars = ftell(f_in);
    // printf("The number of characters is %d\n", numChars);
    fseek(f_in, 0, SEEK_SET);

    char *file_contents = malloc(sizeof(char)*numChars);
    fread(file_contents, sizeof(char), numChars, f_in);
    fclose(f_in);
    /* Note: the memory for this array is used to populate
     * the fields of the companies.  If it is freed, then
     * the company structs all become invalid.  For the
     * context of this program, this array should not be
     * freed. */
```

```
66
67      // Find out how many lines there are
68      int numLines = 0;
69      for (i = 0 ; i < numChars ; i++)
70          if (file_contents[i] == '\n')
71              numLines++;
72      // printf("Number of lines is %d\n", numLines);
73
74      int      numCompanies = numLines-1; // first line is header info
75      Company *companies    = malloc(sizeof(Company)*numCompanies);
76
77      /* strtok will parse the file_contents array.
78       * The first time we call it, it will replace every '"' with '\0'.
79       * It will also return the first word before a
80       */
81      int numColumns = 9;
82      int numberOfQuotesPerColumn = 2;
83      strtok(file_contents, "\"");
84      for (i = 0 ; i < numberOfQuotesPerColumn*numColumns-1 ; i++)
85           strtok(NULL, "\"");
86      for (i = 0 ; i < numCompanies ; i++)
87      {
88           companies[i].symbol = strtok(NULL, "\"");
89           strtok(NULL, "\"");
90           companies[i].name = strtok(NULL, "\"");
91           strtok(NULL, "\"");
92           companies[i].lastSale = atof(strtok(NULL, "\""));
93           strtok(NULL, "\"");
94           companies[i].marketCap = atof(strtok(NULL, "\""));
95           strtok(NULL, "\"");
96
97           /* Skip ADR TSO */
98           strtok(NULL, "\"");
99           strtok(NULL, "\"");
100
101          companies[i].IPOyear = atoi(strtok(NULL, "\""));
102          strtok(NULL, "\"");
```

```
103
104             /* Skip Sector, Industry, Summary Quote */
105             for (j = 0 ; j < 6 ; j++)
106                 strtok(NULL, "\"");
107
108             //PrintCompany(companies+i);
109         }
110
111         /* Set parameters to have output values */
112         *companies_rv    = companies;
113         *numCompanies_rv = numCompanies;
114     }
115
116     int hash(char *name, long array_size)
117     {
118         int hashval = 0;
119         char *p = name;
120         while (*p != '\0')
121         {
122             hashval = 31*hashval + *p;
123             p++;
124         }
125
126         return hashval%array_size;
127     }
128
129     int double_hash(char *name, long array_size)
130     {
131         int doublehashval = 0;
132         char *p = name;
133         while (*p != '\0')
134         {
135             doublehashval = 37*doublehashval + *p;
136             p++;
137         }
138
139         return 31 - doublehashval%array_size;
140     }
141
```

```
typedef struct
{
    int         numElements;
    char      **keys; //symbol
    Company    *companies;
} MapBasedOnHashTable;

void InitializeMapBasedOnHashTable(MapBasedOnHashTable *map, int numElements)
{
    map->numElements = numElements;
    map->keys = malloc(sizeof(char *)*numElements);
    map->companies = malloc(sizeof(Company)*numElements);
    for (int i = 0 ; i < numElements ; i++)
        map->keys[i] = NULL;
}

void StoreTo_MapBasedOnHashTable(MapBasedOnHashTable *map, Company *c)
{
    int hashval = hash(c->symbol, map->numElements);
    int hashval2 = double_hash(c->symbol, map->numElements);
    for (int i = 0 ; i < map->numElements ; i++)
    {
        int idx = (hashval+i*hashval2)%(map->numElements);
        if (idx < 0) idx+= map->numElements;
        if (map->keys[idx] == NULL)
        {
            map->keys[idx]      = c->symbol;
            map->companies[idx] = *c;
            return;
        }
    }
}

Company *
FetchFrom_MapBasedOnHashTable(MapBasedOnHashTable *map, char *key)
{
    int hashval = hash(key, map->numElements);
    int hashval2 = double_hash(key, map->numElements);
    for (int i = 0 ; i < map->numElements ; i++)
    {
        int idx = (hashval+i*hashval2)%(map->numElements);
        if (idx < 0) idx+= map->numElements;
        if (map->keys[idx] == NULL)
        {
            return NULL;
        }

        if (strcmp(map->keys[idx], key) == 0)
        {
            return &(map->companies[idx]);
        }
    }

    return NULL;
}

void FetchAndPrint(MapBasedOnHashTable *mfht, char *key)
{
    Company *c = FetchFrom_MapBasedOnHashTable(mfht, key);
    if (c == NULL)
    {
        printf("Key %s has no corresponding company\n", key);
    }
    else
    {
        PrintCompany(c);
    }

}

// Linked lists talked about at 12:23 in the YT video from Hank.
```

```c
// Linked lists talked about at 12:23 in the YT video from Hank.

struct ll_node
{
    Company        *companies; // value / data
    struct ll_node *next;
    int numElements;
    // key = symbol, value = company info
};

void initialize_ll(struct ll_node *cur_head, int numElements)
    {
        cur_head->numElements = numElements;
        numElements = 0;
        cur_head->companies = malloc(sizeof(Company)*numElements);
        for (int i = 0 ; i < numElements ; i++)
        {
            (cur_head->companies)->symbol = NULL;
        }
    }


struct ll_node *ll_store(Company *c, struct ll_node *cur_head)
{
    struct ll_node *rv = malloc(sizeof(struct ll_node));
    rv->companies = c; // node -> companies which is a company
    rv->next = cur_head;
    return rv;
}

struct ll_node *ll_fetch(struct ll_node *cur_head, char *key)
{
    //printf("Looking for %s!\n", key);

    if (cur_head == NULL)
        return NULL;

    /* special case for cur_head matching v */
    //printf("Comparing against %s\n", cur_head->companies->symbol);
    struct ll_node *curr = cur_head->next;
    if (strcmp(((cur_head->companies)->symbol), key) == 0)
        return cur_head;

    while (curr != NULL)
    {
        //printf("Comparing against %s\n", curr->companies->symbol);
        if (strcmp((curr->companies->symbol), key) == 0)
        {
            return curr;
        }
        curr = curr->next;
    }
    return NULL; /* nothing removed */
}
```

```
267 void ll_print(struct ll_node *head, char * key)
268 {
269     struct ll_node *curr = head;
270 
271     //printf("Comparing against %s and the key %s\n", head->companies->symbol, key);
272 
273     if (head == NULL)
274     {
275         printf("Key %s has no corresponding company\n", key);
276     }
277     else
278     {
279         PrintCompany(head->companies);
280         //head = head->next;
281     }
282 
283 }
285 int main(int argc, char *argv[])
286 {
287     Company *companies = NULL;
288     int      numCompanies;
289     ReadFile(argv[1], &companies, &numCompanies);
290     MapBasedOnHashTable mfht;
291     printf("num companies is %d\n", numCompanies);
292     InitializeMapBasedOnHashTable(&mfht, numCompanies*2);
293 
294     struct timeval startTime1;
295     gettimeofday(&startTime1, 0);
296 
297     for (int i = 0 ; i < numCompanies ; i++)
298     {
299         StoreTo_MapBasedOnHashTable(&mfht, companies+i);
300     }
301 
302     struct timeval endTime1;
303     gettimeofday(&endTime1, 0);
304     double seconds1 = (double)(endTime1.tv_sec - startTime1.tv_sec) +
305                 (double)(endTime1.tv_usec - startTime1.tv_usec)/1000000.;
306 
307 
308     struct timeval startTime2;
309     gettimeofday(&startTime2, 0);
310     FetchAndPrint(&mfht, "ZNWAA");
311     FetchAndPrint(&mfht, "Z");
312     FetchAndPrint(&mfht, "ZIOP");
313     FetchAndPrint(&mfht, "ZIOQ");
314     FetchAndPrint(&mfht, "YIOQ");
315     FetchAndPrint(&mfht, "QIOQ");
316     FetchAndPrint(&mfht, "WIOQ");
317     FetchAndPrint(&mfht, "XIOQ");
318     FetchAndPrint(&mfht, "TIOQ");
319     FetchAndPrint(&mfht, "UIOQ");
320     FetchAndPrint(&mfht, "VIOQ");
321     struct timeval endTime2;
322     gettimeofday(&endTime2, 0);
323     double seconds2 = (double)(endTime2.tv_sec - startTime2.tv_sec) +
324                 (double)(endTime2.tv_usec - startTime2.tv_usec)/1000000.;
```

```
325
326     printf("\nNow running tests for Linked Lists!\n");
327
328     struct ll_node ll;
329     struct ll_node *cur_head = NULL;
330     // function header: void initialize_ll(struct ll_node *map, int numElements)
331     initialize_ll(&ll, numCompanies); // don't need *2 here since a linked list is a pointer to the
            companies
332
333     // get time of day start for store here!
334
335     struct timeval startTime3;
336     gettimeofday(&startTime3, 0);
337
338     for (int i = 0 ; i < numCompanies ; i++)
339     {
340         cur_head = ll_store(companies+i, cur_head); // Parameters: char *v, struct ll_node
                *cur_head)
341     }
342
343     struct timeval endTime3;
344     gettimeofday(&endTime3, 0);
345     double seconds3 = (double)(endTime3.tv_sec - startTime3.tv_sec) +
346                   (double)(endTime3.tv_usec - startTime3.tv_usec)/1000000.;
347
348     struct timeval startTime4;
349     gettimeofday(&startTime4, 0);
350
350
351     ll_print(ll_fetch(cur_head, "ZNWAA"), "ZNWAA");
352     ll_print(ll_fetch(cur_head, "Z"), "Z");
353     ll_print(ll_fetch(cur_head, "ZIOP"), "ZIOP");
354     ll_print(ll_fetch(cur_head, "ZIOQ"), "ZIOQ");
355     ll_print(ll_fetch(cur_head, "YIOQ"), "YIOQ");
356     ll_print(ll_fetch(cur_head, "QIOQ"), "QIOQ");
357     ll_print(ll_fetch(cur_head, "WIOQ"), "WIOQ");
358     ll_print(ll_fetch(cur_head, "XIOQ"), "XIOQ");
359     ll_print(ll_fetch(cur_head, "TIOQ"), "TIOQ");
360     ll_print(ll_fetch(cur_head, "UIOQ"), "UIOQ");
361     ll_print(ll_fetch(cur_head, "VIOQ"), "VIOQ");
362
363     struct timeval endTime4;
364     gettimeofday(&endTime4, 0);
365     double seconds4 = (double)(endTime4.tv_sec - startTime4.tv_sec) +
366                   (double)(endTime4.tv_usec - startTime4.tv_usec)/1000000.;
367
368     printf("\nPerformance Results: \n");
369     printf("This is the hashtable store time: %f\n", seconds1);
370     printf("This is the linked list store time: %f\n", seconds3);
371     printf("This means that linked lists is %f seconds faster for storing!\n", seconds1-seconds3);
372     printf("This is the hashtable fetch and print time: %f\n", seconds2);
373     printf("This is the linked list fetch time: %f\n", seconds4);
374     printf("This means that linked lists is %f seconds slower for fetching!\n", seconds4-seconds2);
375
376 }
377
```

Lecture 18: Cast, Dequeues Const, Static

Functions for the Map Data Structure

- Store
  - Put
- Fetch
  - Get
- Etc.
  - contrainsKey
  - putUnique
  - isEmpty
  - remove

Casting

- "Casting" coverts from one type to another
- Tells compiler that the programmer intends for the type to be changed
- Syntax
  - `type1 var1;`
  - `type2 var2 = (type2) var1;`
- Accomplishes this through extra variable declaration (var2)
  - Memory allocated for both var1 and var2
  - var1 will always be type1... no way to modify this type

```
void Function (shape *s) {
    circle *circ = (circle*) s;
}
```

Example 1

```
int main()
{
    double Y = 3.5;
    int X = Y; // The compiler will automatically cast here.
               // It invokes a non-trivial operation:
               // taking the binary representation of a double
               // precision value (Y) and converting it to
               // the binary representation of an integer.
               // This is real work!
    int X2 = (int) Y;  // this is doing the same cast
                       // operation explicitly
}
```

Example 2

```
Hanks-iMac:Downloads hank$ cat cast2.c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    void *p = malloc(12);
    int  *i = (int *) p;
    printf("p+1 is %p, i+1 is %p\n", p+1, i+1);
}

Hanks-iMac:Downloads hank$ gcc cast2.c
Hanks-iMac:Downloads hank$ ./a.out
p+1 is 0x7fe94bc02611, i+1 is 0x7fe94bc02614
```

(Singly) Linked List Vs Doubly Linked List

I am asked such question and I have my own sayings but I am not really sure what to say about cons and pros? Microsoft asked this question to one of its candidates.

Singly linked list allows you to go one way direction. Whereas doubly linked list has two way direction next and previous.

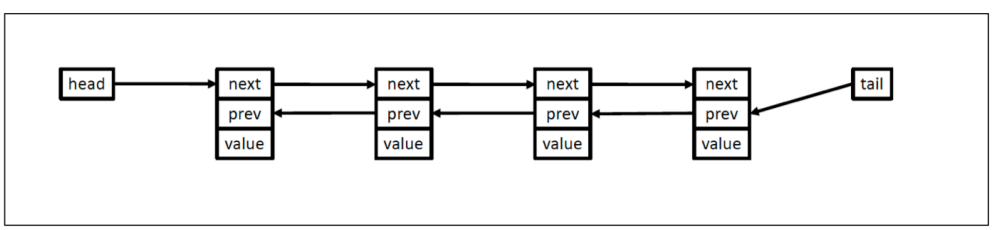
It all comes down to usage. There's a trade off here.

Singly linked list is simpler in terms of implementation, and typically has a smaller memory requirement as it only needs to keep the forward member referencing in place.

Doubly linked list has more efficient iteration, especially if you need to ever iterate in reverse (which is horribly inefficient with a single linked list), and more efficient deletion of specific nodes.

Deque - Double Ended Queue

- New data structure
- Pronounced "deck"
- Queue
  - O(1) insert
  - O(1) to remove head
  - O(n) to remove tail
- Deque
  - O(1) insert
  - O(1) to remove head
  - O(1) to remove tail
- Key methods:
  - Insert first
  - Insert last
  - Remove first
  - Remove last



- Head's prev is NULL
- Tail's next is NULL

Const

- This is a keyword in C
- Qualifies variables
- Is a mechanism for preventing write access to variables
- This creates more efficiency

Const arguments to functions

- Functions can use const to guarantee to the calling function that they won't modify the arguments passed in

```
struct Image
{
    int width, height;
    unsigned char *buffer;
};

ReadImage(char *filename, Image * );
WriteImage(char *filename, const Image * );
```

read function can't make the same guarantee

guarantees function won't modify the Image

Const pointers

- Assume a pointer name "P"
- Two distinct ideas:
  - P points to something that is constant
    - P may change, but you cannot modify what it points to via P

int x = 4;

Idea #1:
violates const:
"*P = 3;"
OK:
"int Y = 5; P = &Y;"

pointer can change, but you can't modify the thing it points to

  - P must always point to the same thing, but the thing P points to may change

int *P = &X;

Idea #2:
violates const:
"int Y = 5; P = &Y;"
OK:
"*P = 3;"

pointer can't change, but you can modify the thing it points to

Const usage

- class Image
- const Image *ptr;
  - Used a lot: offering the guarantee that the function won't change the Image ptr point to
- Image * const ptr;
  - Helps with efficiency. Rarely need to worry about this.
- const Image * const ptr;

Global variables

- You can create global variables that exist outside functions
- Global variables are initialized before you enter main
- Global variables are stored in special part of memory
  - "Data segment" (not heap, not stack)
- If you re-use global names, you can have collision

#define size = 10

Static Memory

- Static memory: third kind of memory allocation
  - Reserved at compile time
- Contracts with dynamic (heap) - aka when you use malloc - and automatic (stack) - aka when you do something like int array[3]; - memory allocations
- Accomplished via keyword that modified variables
- Two specific uses with static memory
  - Persistency within a function

```
fawcett:330 childs$ cat static1.C
#include <stdio.h>

int fibonacci()
{
   static int last2 = 0;
   static int last1 = 1;
   int rv = last1+last2;
   last2 = last1;
   last1 = rv;
   return rv;
}

int main()
{
   int i;
   for (int i = 0 ; i < 10 ; i++)
       printf("%d\n", fibonacci());
}
```

```
fawcett:330 childs$ g++ static1.C
fawcett:330 childs$ ./a.out
1
2
3
5
8
13
21
34
55
89
```

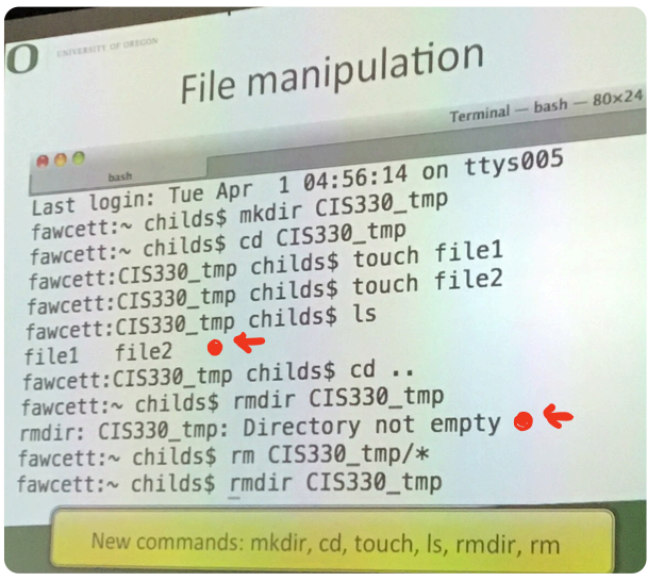
  - Making global variables be local to a file

```
fawcett:330 childs$ cat static1.C
#include <stdio.h>

int fibonacci()
{
   static int last2 = 0;
   static int last1 = 1;
   int rv = last1+last2;
   last2 = last1;
   last1 = rv;
   return rv;
}

int main()
{
   int i;
   for (int i = 0 ; i < 10 ; i++)
       printf("%d\n", fibonacci());
}
```

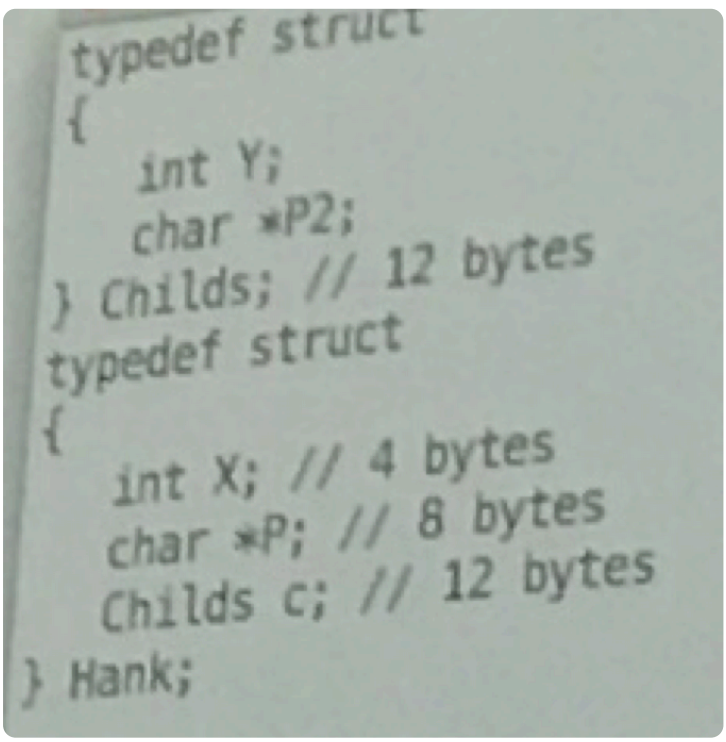
```
fawcett:330 childs$ g++ static1.C
fawcett:330 childs$ ./a.out
1
2
3
5
8
13
21
34
55
89
```

Lecture 19 - Final Review

- Memory is important, it will be heavily asked about on the final exam.
- Topics:
  - Directories are hierarchical (Unix)
  - Home directory (Unix)
  - File manipulation



  - File permission attributes
  - Translating R/W/E permissions to binary
  - Unix command - chmod
  - Sizeof - structs, char, unsigned char, float, int, double, pointers



  - Structs
  - Function pointers (with the tricky declarations)
  - Call by reference and call by value

○ Array

- Block of contiguous memory
- If elements each have size N bytes and there are M elements, them N*M contiguous bytes
- Let A be address of the beginning of the array
- Then A[0] is at “A”
- And A[1] is at “A+N”
- A[2] is at “A+2*N”
- and so on…

○ Dynamic memory

- You allocate it, it stays around until you de-allocate it or the program ends
- Important: you need a way to keep track of memory
  - If not, the memory will be “leaked.”
- So we need a way of managing dynamic memory.
- The concept for doing this in C is **POINTERS**

○ Pointers

- Pointer: points to memory location
  - Denoted with ‘*’
  - Example: “int *p”
    - pointer to an integer
  - You need pointers to get to dynamic memory
- Address of: gets the address of memory
  - Operator: ‘&’
  - Example:

    int x;
    int *y = &x; ← this example is pointing to an automatic variable, not a dynamic variable

○ Free/malloc example ---> every time you malloc, you need to free
○ Character strings

- A character “string” is:
  - an array of type “char”
  - that is terminated by the NULL character
- The C library has multiple functions for handling strings

```
C02LN00GFD58:212 hank$ gcc hello_world2.c
C02LN00GFD58:212 hank$ ./a.out
H is hello world!???$M
C02LN00GFD58:212 hank$ cat hello_world2.c
#include <stdio.h>

int main()
{
    char H[12] = "hello world";
    H[11] = '!';
    printf("H is %s\n", H);
}
C02LN00GFD58:212 hank$ gcc hello_world2.c
C02LN00GFD58:212 hank$ ./a.out
H is hello world!d?Ŧ?s → keeps going until you find
```

Specifics about the final

- Will be:
  - A question where you write a script in Unix
    - (Simple commands, not like 1C) Will be like 1B
  - A question like 2D (memory table)
  - One or more questions where I write code and you tell me what the program does
  - One or more questions where you write code most %
  - One or more questions where I infer your knowledge about data structures. Perhaps a simple implementation.
  - A true/false (or multiple choice) section 3 points out of 30
  - Almost certain to appear:
    - Character strings and their conventions (char * / '\0')
    - Structs
    - Malloc / free / understanding of memory / pointers

Lab Week 1: Virtual Machine and Introduction to Unix

Virtual Machine Configuration

1. What is a hypervisor? Computer software and/or hardware that hosts and manages one or more virtual machines
   (a) What hypervisor are we using in this class? VirtualBox
2. What does the acronym "VMDK" stand for? Virtual Machine Disk
   (a) What is stored in a VMDK file? All operating system, software & data files like a normal HD
3. What operating system (OS) should you select when creating the VM? Arch Linux 64bit
   (a) What other Linux distributions are supported? Ubuntu, Oracle, Debian, Red Hat, etc.
   (b) What will happen if you move the "`.vmdk`" file? The virtual machine will not run
4. How much RAM did you allocate to the VM? User specific. ≥4GB is reasonable.
   (a) Why did you select this quantity? It should be a function of your computer's total RAM. 25-50% of your RAM is a good rule of thumb.
   (b) What would be the effect if this quantity was increased? decreased? Increased: may take too much RAM from the host OS, Decreased: Reduced VM performance
5. What is the Arch Linux VM username? `me`
6. What is the Arch Linux VM password? `me`

Introduction to Unix

This portion of the lab should be done on the class virtual machine. If it is run on a different machine, you may get different answers.

1. What is a terminal? A text-based tool for executing and seeing the results of OS commands
   (a) How do you open an Arch Linux terminal? Start → System → UXTerm or XTerm
   (b) Is it possible to open multiple terminals simultaneously? Yes
2. What Unix command prints the active username? `whoami`
   (a) What is the output when you run this command? `me`
3. What is the path to the user's home directory? `/home/me`
   (a) How is this path related to your username? The last directory in the path matches your username
4. What does the command "`cd /home/me`" do? Changes the active directory to your home directory
   (a) How about "`cd ~`"? Same as above

5. Change into your home directory. Next, using `vi` write a shell script that creates a folder named "`labDebug`", changes into directory "`labDebug`", and then prints the working directory path (hint). Write the file contents of this script below.

```
mkdir labDebug
cd labDebug
pwd
```

   (a) What is the typical file extension for a shell script? `.sh`

   (b) What command would be used to run this shell script? `sh example.sh`

   (c) What is output of the script? `/home/me/labDebug`

   (d) Would the output change if you had created and run the script in a different directory? If yes, how would the output be different? Yes. The output would have been `<OtherDirectoryPath>/labDebug`

6. What is a package manager? A terminal-based too used to automatically download and install external tools

   (a) Why is it important to update the package manager? To ensure the package manager itself and the remote sources are the latest

   (b) What command updates the Arch Linux package manager? `sudo pacman -Syu`

   (c) What command in Arch Linux installs new packages? `sudo pacman -S PACKAGE_NAME`

   (d) List three routinely used packages: `vim`, `git`, `gcc`

   (e) Install `vim` (vi Improved) on your VM. What command did you use? `sudo pacman -S vim`

7. How do you retrieve a Unix command's documentation? `man COMMAND`

   (a) What are three dash options for Unix command **`grep`**? Explain what each option does.
   `-help`: Output a usage message and exit
   `-i`: ignore case distinctions in search
   `-e PATTERN`: search for the PATTERN in a file specified

8. When zipping a directory, why is option "`-r`" required? Recursively zip files in subdirectories

   (a) What are other shell commands accept option "`-r`"? `rm`, `grep`

9. What is a tarball file? A compressed file that uses a different compression algorithm than `zip`

   (a) What are the common file extensions of a tarball? `.tar` and `.tar.gz`

   (b) What command (with necessary options) decompresses a tarball? Is this related to the file extension? `tar -xzf example.tar.gz` – Yes the `.gz` affects the dash options (see below)

   (c) Explain each of the dash options needed to decompress a tarball?
   `-x`: extracts files from tarball
   `-z`: gzip/ungzip
   `-f`: specifies tarball file to do actions on

Lab Week 2: C Libraries and Data Types and Conditionals and Loops and Functions

C Libraries

1. Why are libraries needed? Libraries provide access to commonly used, helpful, and standardized functions/classes.

   (a) Name some commonly used libraries. `math.h`, `stdlib.h`, `stdio.h`, `vector.h`, etc.

2. What is the extension of a library and what does it stand for? ".h" – Header files

3. What is the difference between `#include <file>` and `#include "file"`? Libraries included with "<...>" are standard libraries built into standard C/C++. Custom (i.e., user or third-party) libraries are included with quotation marks.

Data Types

float = 4, double = 8

[0 = false
1 = true

1. What is the difference between a `float` and a `double`? A `double` has *double* the precision (i.e., bits) of a `float`. On most systems, a `float` is 32 bits, and a `double` is 64 bits.

2. What is the value of `true`? Syntactic sugar for an integer value of "1".

   (a) How is this different in C than Python? Boolean variables are objects in Python. In C/C++, Booleans are primitives, which means they can not be inherited from nor do they have methods.

3. Will the following code give an error? If so, where? If not, why? Yes. Variable `x`'s scope is limited to the inner curly brackets. The error will be on the third 'printf' since variable 'x' is destroyed.

Figure 1: C-Code for Data Types Problem 3

```
#include <stdio.h>
int main()
{
  int a = 1;
  int b = 3;
  int c = 4;
  printf("%d\n", a + b + c);
  {
    int x = 4;
    printf("%d\n", x);
  }
  printf("%d\n", a + b + c + x);
}
```

Conditionals

1. What must a condition evaluate to? A condition evaluates to true/false. A false equals integer 0, and any value that is not false (i.e., not equal to zero) is treated as true.

   (a) What will happen if it does not evaluate to anything? A compile or runtime error

2. Consider that you have an integer value stored in variable `x`. Write a conditional that will print the value of `x` any time `x` does not equal 5.

```
if (x != 5)
    printf(''%d'', x);
```

Loops

1. When should you use loops? When performing a repetitive action with little variation between each repetition

   (a) What are your two choices in loops? `while` (including `do/while`) & `for`

   (b) What is the difference between these two loops? `for` generally run for a specified number of iterations with a logical increment. While loops match a condition which may have an indeterminate number of iterations.

2. How many times is the loop executed if the starting value is `x=0` and condition is `x<4`, incrementing by 1? Four

   (a) How many times is the loop executed if the starting value is `x=1` and condition is `x` $\leq$ 4, incrementing by 1? Four

   (b) Is there a difference doing one way or the other? If the loop uses the value of "`t`", then yes. Otherwise, no.

Functions

1. What are the parts to create a function? Prototype (e.g., return type, function name, arguments) and the body (e.g., commands, statements)

2. What is the return type of a function with no return value? `void`

3. Create a working function that calculates the area of a circle. (Practice good naming).

```
// Put at top of file
#include <math.h>
...
double calculate_area(const double radius) {
   return M_PI * pow(radius, 2);
}
```

Lab Week 3: Exit Status Codes and Printing with Error and Debugging & GDB

Exit Status Codes

1. What is an exit status code, and how does a programmer use them? An integer value between 0 and 255 (inclusive) returned by the program to provide information about the program's exit state.
2. What is the standard exit status code for a successful program execution? 0
3. If you do not specify an exit status code, will your program compile? run? Yes, it will compile and run. Upon termination, the program automatically returns exit code 0 per the C99 standard.
4. What Unix command prints the last exit status code? `echo $?`

Printing with Error

1. What can cause an executed **`printf`** command to never actually write to the screen? Anything that can cause the program to crash before the OS writes what is on the buffer, like a segmentation fault.
2. What is the command to force clear the screen write buffer? `fflush(stdout)`
3. What is **`stdout`**? What is the command to write to **`stdout`**? `stdout` refers to the standard output device, which in our case is the terminal. The command that we use to write to `stdout` is `printf`.

Debugging & GDB

1. What are some features of a general debugger? It allows you to step through the code line by line, insert breakpoints, check the state/value of a variable, and many other features.
2. What is a breakpoint? Who defines where breakpoints are placed? A breakpoint is a user defined point in the code where the program will pause its execution.
3. What is a backtrace? The set (stack) of all function calls that lead to the failure.
4. What is GDB? How do you install it? GDB stands for the GNU Project debugger. Install it on the VM using `sudo pacman -S gdb`
5. Is every compiled binary file fully compatible with GDB? If not, what must be included during compilation to ensure compatibility with GDB? No, we must include the flag `-g` when compiling.
6. What command opens GDB? Which command closes GDB? Open: `gdb` Close: `quit` or `q`
7. What command selects the file to debug? What type of file do you select, e.g., `*.c`? Why? `file` The type of file we need to select is a `.out` file, because the debugger requires the compiled code with the debugging information in order to retrieve and interpret any exit status codes created.
8. The following questions relate to the provided **`factorial.c`**.
   (a) What function is used to read user input from the console? `scanf()`
   (b) What command is used to place a breakpoint at the start of the program? `break main`

(c) When stepping through the program line by line starting from the beginning, `printf` does not write to the screen immediately upon being executed. Why? Because the OS buffers the print statements and decides when the statements get written to the screen.

(d) When stepping through the code line by line, how many times will GDB print line 13 (i.e., the `for` loop) if `num = 3`? Why? Four times. GDB first prints line 13 (`for`) when $\mathbf{i = 0}$. GDB prints line 13 the second and third times when `i` equals 1 and 2 respectively. When `i = 3`, GBD prints line 13 a fourth time since the `for` loop's condition (`i <` num) must be checked before terminating the loop.

(e) What GDB command allows you to check the value of the variable `num`? `print num`

(f) Try changing the value of `factorial` during debug. What command did you use? Why did you select this value? `print factorial = 1` You should use 1 because that is the value of "1!"

(g) What is the bug in this program? On what line does it appear? The initialization of `factorial` on line 12 does not have a starting value, so it does not know what value to initialize it to.

(h) What GDB command did you find most useful to debug this program? Why? Answers may vary. Common answers include, `break`, `next`, and `print`.

9. These questions relate to the provided `fib.c`.

(a) What is the Fibonacci sequence? How is it calculated? It is the sequence of numbers where each number is the result of adding the two previous numbers, starting with 1 and 2.

(b) Why can variable `i` be declared twice in `main` without causing an error? Because each `i` is visible only in the scope of the `for` loop in which it is declared.

(c) Why does the second `for` loop not require any curly brackets? Because there is only one statement in the `for` loop.

(d) When you run the provided version of `fib.c`, what happens? What does this mean? An error that an unallocated pointer is being freed. This indicates a logic error in one of the `free` statements.

(e) Use GDB to find the line(s) causing the faults. Which line(s) did you have to comment out? (Hint: step line by line.) Line 26, `free((int)num);`

(f) Why does the code print a newline on line 24? (Hint: there is an additional side effect beyond just ending the line) To force flush the screen write buffer. This may not apply when writing to a file or an output device other than the terminal.

(g) Once you have fixed the faults, you will see that the printed sequence is still wrong. Use GDB to find the other bug(s). What were they?
Line 15: `if` statement condition should be `i <2`
Line 18: `else` statement should be `A[i] = A[i-1] + A[i-2]`
Line 20: `free(A)` should be removed

(h) **Bonus Question**: `A` is an array of unsigned integers. If `A[0] = 1`, what does `A[0] - 10` equal? Why? Since `A` contains only unsigned numbers, `-9` is not possible. Instead C rolls over the value meaning: `A[0] - 10 = MAX_UNSIGNED - 9 = 4294967287`.

Arrays

1. At what index do C arrays start? Zero
2. If you declare an array as follows: `int A[10];` what is in A? The contents of `A` are not initialized by default. Hence, the values of the 10 integers should be considered undefined (like random) values.
3. What function returns the number of elements in an array? For some array, `A`, the number of elements is:
   `sizeof(A) / sizeof(A[0])`
4. Can you extend an array beyond its initialized size? If so, how? No. You must allocate new memory and copy the contents of the original array into this new memory. The command `realloc` can do this for you automatically.
5. For a 2D array, which bracket refers to the rows? Which refers to the columns? When indexing a matrix element in linear algebra, the first index represents the row number while the second index is the column number. Two dimensional array indexing in C is similar with the only difference being that row and column indexing begins at zero.

Strings

1. What are three ways to create a string? Which are the advantages of each approach?
   Option #1: `char* str = "Hello World";`
   Option #2: `char str[] = "Hello World";`
   Option #3: `char str[5] = "Hello World";`
   Options #1 and #2 are functionally the same. The compiler determines the exact amount of memory needed and allocates it. Option #3 allows the user to specify the amount of memory needed. If the user specifies too little memory (as in the above example), the compiler issues a warning and uses the smaller size. There is the side benefit that the user can allocate more memory than required to hold this initial string, which may be useful later in the program.
2. What is the only macro in the library `string.h`? `NULL`
3. Assuming `str1` and `str2` are valid strings, what does `strcmp(str1, str2)` return? What does each value mean? `strcmp` returns a **signed** integer. If `str1` has the same contents as `str2`, 0 is returned. If `str1` is alphabetically before str2, the return value is negative. Otherwise, the return value is positive. Note that `strcmp` is case sensitive with all uppercase letters treated as alphabetically before all lowercase letters.
4. What function returns the length of a string? `strlen`
5. For the following questions, please attempt to create code to do the following. If unsure, refer to the starter code `week04practice.c` on Canvas:

(a) Given a file path, i.e. "~/cis212/practice.txt", find and return the file extension.

Figure 1: Extract file extension from file path

```
#include <string.h>
...
char* find_file_extension() {
  char* path = "~/cis212/practice.txt";

  /* Must use last period since may have periods in filename */
  char* per_loc = strrchr(path, '.');

  if (per_loc == NULL)
    return NULL;

  /* Increment by one since no period in file ext */
  int ext_len = strlen(path) - (per_loc + 1 - path);
  char * ext = (char *)malloc(sizeof(char) * ext_len);
  strcpy(ext, per_loc + 1);
  return ext;
}
```

(b) For a specific file path, e.g., "~/cis212/proj2a.c", refactor (i.e., expand) the "~" to the complete, unabbreviated path by replacing "~" with "/home/hank".

Figure 2: Refactor abbreviated path to full path

```
#include <string.h>
...
char* refactor_path() {
  char* path = "~/cis212/practice.txt";

  char* full_path = (char*)malloc(sizeof(char) * 100);
  strcpy(full_path, "/home/hank");
  /* Add 1 to remove tilde */
  strcat(full_path, path + 1);
  return full_path;
}
```

(c) Given a file path, find and return the folder path, i.e., everything before the last "/".

Figure 3: Extract folder path

```
#include <string.h>
...
char* find_folder_path() {
  char* path = "~/cis212/practice.txt";
  char* end_folder = strrchr(path, '/');

  if (end_folder == NULL)
    return NULL;

  /* Increment by one since no period in file ext */
  int path_len = end_folder - path;
  char* folder = (char*)malloc(sizeof(char) * path_len);
  strncpy(folder, path, path_len);
  return folder;
}
```

GDB Review

1. List four common features of a debugger.
   1. View, change value of variables
   2. Step through program line by line
   3. Set breakpoints
   4. Set watch points
2. GDB does not work with all compiled binaries. What steps are required to make the binary compatible with GDB? Why are these steps required? We need the dash option `-g` in our compile command, because just `gcc` by itself strips all of the debugging information.
3. What is the difference between GDB commands "`step`" (`s`) and "`next`" (`n`)? `step` is like the `next` command, except that `next` will not "enter" functions and show the steps taken in that function.

Breakpoints and Watchpoints

1. How do you set a conditional breakpoint on line 11 of file `main.c` for the condition `x < 10`?
   `b main.c:11 if x<10`
2. What are the commands to save and load the breakpoints from disk?
   `save breakpoints <outputFile>`
   `source <inputFile>`
3. What is the command to print all active breakpoints? `info breakpoints`
4. What requirements must be met to set a watchpoint? The program must be running and the variable to watch must be in the current scope.

Hands-On GDB Practice

Find the answer to all questions in this section using **only** GDB. Although these questions could be answered using `printf` statements, you should avoid using them so as to gain experience working with GDB.

1. These questions relate to the program "`string_fun.c`". This program does basic string manipulations.
   (a) Step through the program line by line. On which line does the variable value not match your expectations? (Note that if you run this program on a Mac, it may abort with an error) On line 22, `strcpy(str1, str3)` copies a string of 33 characters into a string that has a maximum capacity of 15.
   (b) What is the value of variable `x`? What does `x` represent? `x` has the value of 33, which represents the number of `char` in `str2`.
   (c) What is the effect of the call to `memcpy`? Explain how each input parameter affects this result. The `memcpy` call puts the 3rd argument, 6, number of bytes from the 2nd argument, `str2`, starting at the 12th index, and puts them into the destination pointer of the 1st argument, `buf2`, giving `buf2` the value "CIS212".
   (d) Load the saved breakpoints in file "`breakpoint_list`". How many breakpoints are there? On what lines are there a breakpoint? There are 4 breakpoints, each on line 5, 22, 17, and 26.

2. These questions relate to the program "`summation.c`". This program is supposed to calculate the summation $\sum_{i=1}^{n} i$.

   (a) Does this summation have a *closed form* solution? If so, what is it? Yes. sum $= \frac{n(n+1)}{2}$

   (b) What is the output of the program when you run it with $n = 5$? Is the result consistent? What is the cause root cause for this type of behavior? The calculated value is not consistent. You might see answers such as -1483143146 or -185615658, but not the expected value of 15. The cause of this is on line 17: `int sum, i;`, because `sum` is not initialized to a value.

   (c) Describe all bugs discovered in the source code. When referencing a bug, be sure to include the line number. line 17: `sum` is not initialized to a value and will therefore take the value of whatever is at that address in memory.

3. These questions relate to the program "`bit_manipulations.c`". This program is intended to give you experience with GDB watchpoints and C bit manipulations.

   (a) Set a watchpoint to trigger whenever `x` is less than zero. What command did you use? `watch x if x<0`

   (b) Which line triggers the watchpoint from the previous question? Line 9

   (c) What functions do operators "`<<=`" and "`>>=`" perform? Taking the variable's original value and bit shifting it left or right and setting the variable equal to the new value.

   (d) What function does operator ~ (tilde) perform? Negation, or flipping the values of all the bits from 0 to 1 and vise versa.

   (e) What function does operator ^ (carrot) perform? XOR

Structs

1. What is the purpose of a `struct`? User-defined, *composite* data type, i.e., encapsulates into a single object multiple, related variables of *different* types. Structs improve code readability and reduce bugs

2. What is the difference between defining a struct using `struct` versus `typedef struct`? When "`typedef`" is not included in the definition of a `struct`, then "`struct`" must be included any time a struct **object** is created in your program.

3. How is a `struct` stored in memory? One contiguous piece of memory like an array

4. The following questions relate to creating a struct "`Book`."

   (a) What field(s)/attribute(s) could the `Book struct` encapsulate and what would be each field's type? Answers may vary. Valid answers include: title (`char*`), number of pages (`unsigned int`), author (`char*`), price (`double`), etc.

   (b) What functions may need to be defined for objects of this type? (Hint: What may need to be done with books?) Answers may vary. Valid answers include: `create_new_book` – Returns new `Book` objects. `change_price` – Updates the `Book`'s price

5. The following questions relate to creating a struct "`Bookstore`."

   (a) What fields/attributes could the `Bookstore struct` encapsulate and what would be each field's type? Answers may vary. Valid answers include: address (either `char *` or `struct` of type Address), store name (`char *`), Inventory (`dict` of Book objects), Cash on hand (`double`)etc.

   (b) What functions may be defined for the `Bookstore struct`? Answers may var. Valid answers include: `sell_book`: Removes book from inventory and increases cash amount. `purchase_inventory`: Adds new books to the inventory and decreases cash amount. `open_new_store`: Creates a new bookstore object.

6. How do you read/set a `struct`'s fields? Show all possible approaches and explain when each would apply. Demonstrate using the `Book struct` from above.

```
Book* x = malloc(sizeof(Book));
x->title = ''If You Give a Mouse a Cookie'';
char* a = x->title;
Book y;
y.author = ''Laura Numeroff'';
char* b = y.author;
```

7. Some languages (e.g., Java) have the restriction that arrays can only hold primitives. In C, why can you create an array of structs, a composite data type? Since `structs` are stored as a single piece of contiguous memory, their encapsulation is no different than a primitive.

8. What is the difference between a member access operator (.) and a structure pointer (–>)? The period operator(.) is used to access the fields of a **concrete** instance of `struct`. In contrast, the arrow operator (–>) is used to access the fields of a **pointer** to an instance of a `struct`.

Debugging Structs

The following questions are about the file "`file_fun.c`":

1. "`file_fun.c`" has a compile error. Explain the cause of the compile error and how it should be fixed. Field "`fileNameDir`" is an array of characters. C does not allow the assignment operator (=) to set a `char` array. Instead use the command "`strcpy(aEntity->fileName, fileNameDir);`" Do not forget to include `string.h`

2. After fixing the compile error, run the program. What happens? A segmentation fault

3. Use GDB to debug the new error. What is the line number of the problematic statement? What is the cause of the issue? Line #: 16. Object `aEntity` is a pointer to an `entity` object. The pointer is not initialized and points to an unknown (random) address. This leads to a segmentation fault.

4. What edits must be made to the code to fix this error? Allocate heap memory to `aEntity` by changing line #14 to "`entity* aEntity = malloc(sizeof(entity));`" To prevent a memory leak, "`free(aEntity);`" should be added before the `return`.

Shell is a Programming Language

1. What error will the shell report if you use an uninitialized/undefined variable? No error. The shell will treat the undeclared variable as if it is an empty string.
2. Provide the syntax to store the output of a Unix command in variable `$VAR1`.
   `VAR1=$( <command> )`
3. Write an `if` statement that prints the value of variable "`X`" when `X` is less than or equal to 5.
   ```
   if [ $X -le 5 ]; then (also acceptable if (( $X <= 5 )); then)
    echo $X
   fi
   ```
4. How do `for` loops in the shell differ from C `for` loops? Like in Python, shell `for` loops are *range-based*. This means that the `for` loop iterates over a list/set/container of values rather than relying on an iterator/counter.
5. How are arguments specified in user-created shell functions? Arguments do not appear in the function definition. Instead, function arguments are treated similar to command line arguments where `$1` is the first argument, `$2` is the second argument, etc.

Coreutils

1. What is the command to output the contents of a file? How is the command name related to the command that outputs the file lines in reverse order? `cat`, which is related to the command that prints the file contents in the reverse order, `tac`, by being the reverse of each other.
2. What is a PID? Provide two ways to find the PIDs. PID stands for "Process Identification Number". The Unix commands "`top`" and "`ps`" list the PIDs of the running programs.
3. When can the total CPU usage reported by `top` exceed 100%? The reported percentage represents the CPU usage for a single CPU/core. The quantity can exceed 100% when utilization is high on multiple core systems.
4. Briefly define regular expressions. Why are they important? It is a pattern-based scheme to match strings. They are concise representations of complex, even infinite, strings.
5. Write a command to replace all instances of "foo" with "bar" in file "input.txt".
   `sed 's/foo/bar/g' input.txt`
6. Why is command `uniq` often preceded by the command `sort`? Lines need to be adjacent to each other for `uniq` to consider them duplicates. `sort` rearranges the lines so duplicates ones are adjacent.
7. Write a command that removes the first five characters from each line in file "test.txt".
   `cut -c 6- test.txt`
8. What operation is performed by the command `grep`? Provide an example where you may use it. `grep` searches an input for a given regular expression and returns the lines in the input that contain the expression. You can use `grep` when you want to know how many people with the name "Potter" there are in a phone book.

Pipes

1. When are pipes used? Pipes route the output of one Unix command to the input of another.
2. What is the difference between pipes "|" and "|&"? "|" pipes (receives) standard out, while "|&" pipes standard error.
3. Provide the syntax to pipe the output of `tail` to `head`? When may you want to do this?
   `tail | head`
   It may make sense to do this if you wanted to grab just the lines in the middle of the file.

Programming Exercises

For all the exercises below, you will use the CSV (comma separated variable) file "exercise.csv". The column definitions are specified in the file.

1. Remove the header row from the CSV file then output the remaining lines in **reverse** order to file "reverse.csv".
```
sed '1d' exercise.csv | tac > reverse.csv
```

2. Output to standard out the last name of all **students** with the first name "John".
```
grep -wi "John" exercise.csv | grep "student" | cut -d ',' -f 3
```

3. For each employee only, print to the console "<FirstName> <LastName> is <Age> years old." (Hint: You will need to use a `for` or `while` loop)
The following is one of many approaches.
```
grep -wi "employee" exercise.csv |
cut -d ',' -f 2,3,5 -output-delimiter=' ' > temp_employee.csv
while IFS=' ' read -r first last age ; do
   echo $first $last is $age years old.
done < temp_employee.csv
```

Valgrind

1. What is valgrind? What can you do with it? Valgrind is a framework for building dynamic analysis tools. Common uses include: detecting memory management and threading bugs, as well as profiling your programs in detail

2. How is valgrind different from gdb, i.e. when would you use valgrind over gdb? GDB is used for finding "logic" bugs.

3. According to the official valgrind documentation at valgrind.org, there are 6 production-quality tools. What are they? 1. A memory error detector
   2. & 3. Two thread error detectors
   4. A cache and branch-prediction profiler
   5. A call-graph generating cache and branch-prediction profiler
   6. A heap profiler

4. List some options that you can use with the tool **memcheck**. Which do you find most useful?
   `-v` : Verbose output of messages
   `--leak-check=yes` : Check for memory leaks unfreed memory
   `--track-fds=yes` : Check for any files open (via `fopen`)
   `--read-var-info=yes` : Print variable information with backtrace when an error occurs

5. What issues may you run into if you run Valgrind outside the VM? Valgrind is not supported at all on PC. If Valgrind is run on a Mac, many false memory errors will be reported due to implementation issues in Mac's C/C++ compiler.

Programming Exercises

1. What is needed to compile a C file, so that it can be used with valgrind? `gcc -g <filename>.c` – The "`-g`" flag tells the compiler to include debug information in the compiled binary

2. What is the command to use valgrind on the executable file "`mem_leak`"?
   `valgrind ./mem_leak` or `valgrind --leak-check=full ./mem_leak`

3. Using valgrind on the file `factorial.c`, what are the errors messages that it gives? Were there any memory leaks? How would you fix the errors? Valgrind reports that on line 15 of the program, an uninitialized variable is being used. In the program source code, you can see that variable `factorial` is not initialized. This can be easily fixed by changing line 12 to:
   `int factorial = 1;`

4. Using valgrind on the file `fib.c`, what are the errors messages reported by valgrind? Were there any memory leaks? How would you fix the errors? Valgrind reports that "definitely lost: 20 bytes in 1 block". This indicates that memory is allocated but not freed. In the source code, you can see that variable `A` receives a `malloc` that is never freed. This can be fixed by calling "`free(A);`" right before the `return`.

5. Using valgrind on the file `mem_array.c`, what are the errors messages that it gives? Were there any memory leaks? How would you fix the errors? Valgrind reports two errors, namely:
   "`Invalid write of size 4`" & "`Invalid read of size 4`"
   These error messages usually indicate that your program is writing and reading past the end of an array respectively. In the source code, observe that the `for` loops are iterating from array index 0 to 10. However, the array pointed to by `a` only contains ten elements so the last read and write are invalid. This can be fixed by change the for loops to:
   `for(i=0;i < 10; i++)`