Review Topics

LINUX: ls, mv, find, find -exec, chmod , wc, pipeline, sort, uniq, grep, cut (used to extract specific fields), scripts using control structures and use of test operators, [ ] , relational operators, arithmetic expressions evaluation, simple regular expressions to create patterns and use in find, grep, command to read (interactive via keyboard, command line argument or files), questions based on small scripts output, redirection operators in shell scripting.

C: evaluation of expressions and variables when compound operators and post and pre increment operators are used, find output of a given code, what does a given code do, fill missing statements is a given code to complete it, bitwise operators (& , |, not), logical operators (&&, ||), ternary operator, operator precedence applied to cases like x = y == 6 (can also include a post or pre increment operator within this assignment), precedence for membership operators and dereferencing, continue and break statements, switch statements and fall through, pass by value, pass by reference, questions on pointer arithmetic, arithmetic on the values of pointer variables themselves (memory addresses), questions on double pointers and their evaluation, accessing array elements (with and without pointers), arrays out of bounds, passing arrays/structures in a function, malloc syntax (all the forms), memory leak, free(), output of the programs having functions and having a proper/improper use of pass by reference/values/local variables, struct and malloc, multidimensional arrays, initialization, enumerations, string functions in C and associated precautions while using them, know runtime errors, compilation errors, and segmentation fault, command line arguments, array of pointers and double pointers.

Exam

    20 Linux
    40 C questions

Review -

- **Pointers**
- **Errors**
- **Pointer arithmetic ** important**
- **Double pointers ** important ?**
- **Linux commands**
- **malloc / realloc**
- **Memory leak**

- Chapter 1
  - C definition
    - General purpose language, compiled (not interpreted) language
    - Lower level language
    - Created to support implementation of Unix

        - Common in systems programming(**operating systems, file systems, drivers, embedded systems, etc**)
    - Syntax Errors (compile time)
        - Violation of C's programming rules, on how symbols can be combined to create program
    - Logic errors (run-time errors, bugs)
        - Happens when program runs
- Chapter 2
        - int (usually 4 bytes)
        - char (1byte)
        - Short int(usually 2 bytes)
        - Long int(4 or 8 bytes)
        - Long long int (8 or 16 bytes)
        - **There is no boolean type - 0 == false | 1 == true**
        - Double (~7 significant figures)
        - Float (~16 significant figures)
    - Compound operators
        - +=
        - -=
        - *=
        - /=
        - %=
    - Precedence rules (first to last)
        - () - evaluated first
        - Unary - used for negation
        - * / % - equal precedence among
        - += - equal precedence among
        - Left-right - when equal precedence
    - Double
        - Stores floating point number
        - Number with fraction part
        - %lf - specify a double type in the String, %d for int
    - Integer division and modules
        - 10 / 4 = 2
        - 3 / 4 = 0
        - (1 / 2) * b * h = Always zero
        - **10 / 4.0 - 2.5**
        - **10 / 4 = 2**
    - Type conversions
        - Double f + int i = **converted to float**

        - Char may be freely used in arithmetic conversions - converted to ASCII value
    - Bad code example
        - int a = 2,b = 3, c = 5, d = 7, e = 11, f = 3;
        - f += a/b/c ▶️ 3
        - d -=7+c*--d/e ▶️ -3
        - d = 2*a%b+c+1 ▶️ 7
        - a += b += c += 1 + 2 ▶️13
    - Type casting
        - Implicit (automatic conversion)
            - Int / double
        - Explicit (manual conversion)
        - Num = 10.5; num2 = (int) Num
    - ASCII Table
        - character encoding standard that represents text and control characters in computers and other devices that use text.
    - Character input
        - scanf(" %c", &bodychar)
    - Array
        - int Itemcount[3]
        - Itemcount[1] = 122;
        - Itemcount[2] = 119;
        - item count[3] = 117;
    - **Strings in C**
        - **No String type in C, only array of characters**
        - **char greetings[50] = "Hello"**
        - **String literal i surrounded in double quotes**
        - **Null character - serves as marker of endpoint of String ('\0')**
    - Integer overflow
        - Use long long, when int is not big enough
        - Use %lld - to print long long number
    - Scanf
        - Must #include<stdio.h>
        - First argument - format string that specifies the type of values read
        - Other arguments provide location of storage of values
- Chapter 3 - Branches
    - Control Statements
        - Conditional (if - else if - else)
        - Loop (while ; for ; do)

            do{
            }while (<Test>);
        - Switch statements
            - Compare integer value against multipel options
    - Logical Operators
        - && - and
        - || - or
        - !a = NOT
    - Boolean Data Type
        - Must #include <stdbool.h>
    - Order of evaluation
        - ()
        - !
        - * / % + -
        - < <= > >=
        - == !=
        - &&
        - ||
    - String comparisons
        - Strcmp
            - Negative when str1 < str2
            - 0 when str1 == str2
            - Positive when str1 > str2
            - Must #include <string.h>
            - Also used for String indexes str[k] < str2[k]
    - Character operations
        - isalpha(c) - true if alphabetic a-z or A-Z
        - isdigit(c) - true if 0-9
        - isspace(c) - true if whitesapce
    - Conditional operation
        - Myvar = (condition) ? expr1 : expr2;
    - Floating - point comparison
        - if(fabs(numMeters - 0.0) < 0.001){
          \\Equals
        - %.valuelf - specify floating point format
            - %.25lf - prints next 25 digits after the

- Chapter 4 - Loops
    - While loops

        - while (<statement is true>){
          Args
          }
    - For loops
        - for(<initialize>; <check>; <update>){
          Args
          }
    - Placing ++ or –
        - Placing in front**(k=++i)**: increment or decrement occurs **before** value is assigned
        - Placing after variable (k=i++): increment or decrement occurs **after** value is assigned
    - Do-while loops
        - do{
          Args
          }while (<statement is true>)
    - break statement causes immediate exit of loop
    - continue statement causes a "jump" to the **loop condition check**
- Chapter 5 Arrays / Arrays with strings
    - String library function
        - Must #inlucde <string.h>
        - To copy string dont use str2 = str1
        - To compare Strings don't use (str1 == str2)
    - String methods
        - strcpy(destStr, soulrceStr) = Copies sourceStr(up to and including nul character) to destStr
        - strncpy(destStr,sourceStr,numChars) = Copies up to numChars
        - strcat(destStr, sourceStr) = concatinates up to and including null characters to destStr
        - Strncat = same thing as strcat, but has a numofChars parameter
        - strchr(sourceStr, searchChar) - Returns Null if character doesnt exist in sourceStr. Else returns address of first occurrence (**null is defined in string.h library**
        - strcmp(str1,str2) returns 0 if equal, else non-zero
        - fgets(str , num, stdin) - reads one line of characters from user input, ending with a newline. Writes those characters in str. If a newline char is read from the user input before **num** characters are read, newline character is also written in **str**. **num** is the maximum number of characters to be written into **str. <u>If num is 10 and the input line exceeds 10 characters,</u>**

<u>only the first 9 characters will be written (followed by a null character)</u>

- Summary
    - Arrays are zero-indexed
    - Arrays decay to pointers
    - Array size must be known at compile time
    - Arrays can be multidimensional
    - Arrays can be initialized with values
    - Arrays can be passed into functions
    - Arrays can be used with pointer arithmetic
    - Arrays canbe used with sizeof()
    - Arrays can be used with string literals
    - Arrays can be used with preprocessor macros

- Chapter 6 - User-Defined functions
    - You can call functions inside of functions
    - Functions help improve program readability
        - Modular development
        - Incremental development
        - Helps us avoid write repetitive code
    - Unit testing
        - Process of individually testing a small part or unit of program
        - Conducted by creating testbench / test harness
            - is a separate program whose sole purpose is to check if function returns correct output
        - Each unique set of input values is called test vector
        - assert() - must #include <assert.h> \
        - Example - assert(HrMinToMin(1,0) == 60)
            - **Assertion failed: (HrMinToMin(1,0) == 60), function main, file main.c, line 19**
        - Good test cases include normal cases, and weird/extreme cases
    - How functions work
        - Each function call creates new set of local variables, forming part of what is know as stack frame
        - return causes those local variables to be discarded
        - Compiler generates instructions to copy arguments to parameter local variables, and to store a return address
        - Jump intersection jumps from main to function's instructions
        - Function executes and stores results in designated return value location
        - When function completes, intersection jumps back to caller's location using the previously stored return address

        - Instruction copies return value to appropriate value
    - Common Errors
        - Copy-paste code among functions
        - Return wrong variable
        - Failioing to return value
        - Incorrect output
- Chapter 7
- Chapter 8 - Pointers
    - Function can return only one value, pass by pointer can return multiple
    - Place *before parameter name to indicate parameter is a pointer
    - Prepending a * to a pointer variable's name accesses the value pointed to by the pointer
    - Pass-by-pointer should be only be used when the output values are tightly related
        - `int StepsToFeet(int baseSteps)` and `int StepsToCalories(int baseCalories)` **better than** `void StepsToFeetAndCalories(int baseSteps, int* feetTot, int* caloriesTot)`
    - void MyFct(int w, int x, int * y, int * z)
    - MyFct(a, b, &c, &d)
    - Pointer holds memory address, instead of actual value
    - Pointers have data types, like regular variables
    - reference operator (&) obtains a variable's address.
    - %p - can be used to print memory address
    - printf("%p", (void*) &sensorVal)
    - dereference operator (*) - used to retrieve data to which pointer variable points
    - You can assign pointer variable to Null - meaning nothing
        - int *maxValPointer = NULL;
    - Syntax Errors
        - * valPointer = &someInt; -> int value cannot be assigned int*
            - Correct
              valPointer = &someInt;
        - int* valPointer1, valPointer2;
          valPointer = NULL -> **int should not be assigned null**
          valPointer = NULL
            - Correct
              int* valPointer1;
              int* valPointer2;

    - Runtime Errors
        - Int * valPointer;

*valPointer = 4 -> **derefencing unknown address**

- Correct
  int someInt = 2;
  valPointer = &someInt;
  *valPointer = 4 -> **intitalize pointer before dereferencing**

- ➢ A syntax error results if valPointer is assigned using the dereference operator *.
- ➢ Multiple pointers cannot be declared on a single line with only one asterisk. Good practice is to declare each pointer on a separate line.
- ➢ Malloc functions
  - Must #include <stdlib.h>
  - pointerVariableName = (type*)malloc(sizeof(type));
  - Malloc returns a pointer to allocated memory, using void pointer. Void pointer stores memory address without referring to type of variable stored
  - Can allocate memory for any data type
  - **free(pointerVariable)** does the opposite of malloc() - deallocates memory
  - free() only works for memory allocated using malloc
  - Calling free with a pointer that wasn't previously allocated by malloc is an error
  - structPtr->memberName - called **member access** operator. Dereferences a pointer
    - myItemPtr1->num1 = 5 Equivalent to doing (*myItemPtr1).num1 = 5
    - use the dot operator when you are working with the actual structure, and use the arrow operator when you are working with a pointer to a structure.
- ➢ String functions with pointers
  - Each string library function operates on one or more strings, each passed as a char* or const char* argument.
  - Strings passed as char* can be modified by the function, whereas strings passed as const char* arguments cannot
  - **strcmp()** compares 2 strings. Since neither string is modified during the comparison, each parameter is a const char*.
  - **strcmp()** returns an integer that is 0 if the strings are equal, non-zero if the strings are not equal.
  - **strcpy()** copies a source string to a destination string. The destination string gets modified and thus cannot be const char*.
  - A char* can be passed as a const char* argument. The const just implies that the string will not be modified by the function.
  - **strchr(sourceStr, searchChar) -** Returns NULL if searchChar does not exist in sourceStr. Else, returns pointer to first occurrence.

    - **strstr(str1, str2) -** Returns NULL if str2 does not exist in str1. Else, returns a char pointer pointing to the first character of the first occurrence of string str2 within string str1.
- Malloc for arrays and Strings
    - statically allocated array - array of fixed size
    - dynamically allocated array - size can be changed after if needed
        - pointerVariableName = (dataType*)malloc(numElements * sizeof(dataType))
    - Creating a dynamically allocated copy of a string.
        - nameCopy = (char*)malloc((strlen(nameArr) + 1) * sizeof(char));
        - strcpy(nameCopy, nameArr);
- Realloc function
    - re-allocates an original pointer's memory block to be the newly-specified size
    - realloc() can be used to both increase or decrease the size of dynamically allocated arrays
    - Returns pointer to the re-allocated memory
    - The pointer returned by realloc() may or may not differ from the original pointer.
    - pointerVariable = (type*)realloc(pointerVariable, numElements * sizeof(type))
- Vector Abstract Data Type
    - data type whose creation and update are supported by specific well-defined operations
    - A key aspect of an ADT is that the internal implementation of the data and operations are hidden from the ADT user, a concept known as information hiding
    - Allows user to be more productive
    - Information hiding refers to hiding the underlying implementation from users, and instead allowing users to focus on higher-level concepts.
    - Users need only to know the functions supported by the ADT, not the underlying implementation.
    - unlike an array, a vector's size can be adjusted while a program is executing
    - Vector ADT naming convention
        - All functions supporting the vector ADT begin with "vector_", which is intended to indicate the functions are associated with the "vector" type.
    - must #include “vector.h”
- Vector ADT functions

void vector_create(vector* v, unsigned int vectorSize)	Initializes the vector pointed to by v with vectorSize number of elements. Each element with the vector is initialized to 0.	vector_create(&v, 20)
void vector_destroy(vector* v)	Destroys the vector by freeing all memory allocated within the vector.	vector_destroy(&v)
void vector_resize(vector* v, unsigned int vectorSize)	Resizes the vector with vectorSize number of elements. If the vector size increased, each new element within the vector is initialized to 0.	vector_resize(&v, 25)
int* vector_at(vector* v, unsigned int index)	Returns a pointer to the element at the location index.	x = *vector_at(&v, 1)
unsigned int vector_size(vector* v)	Returns the vector's size — i.e. the number of elements within the vector.	if (vector_size(&v) == 0)

void vector_push_back(vector* v, int value)	Inserts the value x to a new element at the end of the vector, increasing the vector size by 1.	// adds "47" onto the end of the vector
		vector_push_back(&v, 47);

void vector_insert(vector* v, unsigned int index, int value)	Inserts the value x to the element indicated by position, making room by shifting over the elements at that position and higher, thus increasing the vector size by 1.	// inserts "33" at index 2
		// Before: 18, 26, 47, 52
		// After: 18, 26, 33, 47, 52
		vector_insert(&v, 2, 33);

void vector_erase(vector* v, unsigned int index)	Removes the element from the indicated position. All elements at higher positions are shifted over to fill the gap left by the removed element. Thus, the vector size decreases by 1.	// erases element at index 0
		// Before: 18, 26, 47, 52
		// After: 26, 47, 52
		vector_erase(&v, 0);

- if a NULL pointer is passed to the vector_create() function, the expression v->elements would attempt to dereference a NULL pointer.

    This will returnan error referred to as a segmentation fault, access violation, or bad access, it varies on the computer system used
    - vector_size(NULL) returns the value -1, indicating an error, as a vector cannot have a negative number of elements.
    - **For vectors with thousands of elements, a single call to insert() or erase() can require thousands of instructions, so if a program does many inserts or erases on large vectors, the program may run very slowly.**
- Linked list (not needed)
- Double pointer
    - int ** - pointer to a pointer of type int
    - Useful for when you are working with 2dimensioanl structures (matrix)
- **Pointer Arithmetic**
    - **Only can do addition / subtraction**
    - U can assign array of int as int a[] || int *a
        - A[0] equivalent to *a
        - A[1] equivalent to *(a+1)
        - A[i] equivalent to *(a+i)
        - &a[i] equivalent to a + i
        - &a[0] equivalent to a
            - **The reason why is because the array elements are stored one after another, so for example if array a[0] = 1000, then a[1] = 1004, hence why u do *(a+1), because u are adding 4 bytes, which is the equivalent to an int, im assuming that its the same for different variables, just change the actual size of the thing ur adding**
            - **Because a points to an integer, it knows whe u do a + 1 you are adding 4 bytes**
- Mistakes when working with pointers
    - Unitialized pointers
    - Dangling pointers
    - Memory leaks
    - Incorrect pointer arithmetic
    - Using an invalid pointer
    - Mixing pointer types
    - Forgetting to allocate sufficient memory
- Memory leaks **(To do tomorrow add more to this)**
    - occurs when a program that allocates memory loses the ability to access the allocated memory

        - A program's leaking memory becomes unusable, much like a water pipe might have water leaking out and becoming unusable
        - memory leak may cause a program to occupy more and more memory as the program runs, which slows program runtime.
        - Even worse, a memory leak can cause the program to fail if memory becomes completely full and the program is unable to allocate additional memory.
        - A common error is failing to free allocated memory that is no longer used, resulting in a memory leak. Many programs that are commonly left running for long periods, like web browsers, suffer from known memory leaks — a web search for "<your-favorite-browser> memory leak" will likely result in numerous hits.
    -
- Chapter 9 & 10
    - cd allows you to change directory
    - pwd prints the working directory
    - mkdir makes a directory
    - cp copy a directory
    - mv allows you to move to another location
    - rm allows you to remove a file
    - find tries to find a file using regex
        - * any sequence of characters
        - ? any one character
        - [abc] one of any of the listed character
        - Option (exec) allows you to execute a command when a file is found using find command
    - File Permission using chmod:
        - Owner / Group / Others
        - Read r / Write w / Execute x
    - **Porbably no links on the test**
    - Links
        - Symbolic: file that points to another name
            - Breaks when either file are moved, or rtenamed i believe
        - Hard link: another name taht points to the same file
            - Its “global”
    - Streams
        - Standard input
        - Standard output
        - Standard error: &2
    - Redirection

        - Output:
            - Command > file # Direct output to file
            - Command 2> file # Direct error to file
            - Command > &2 # Direct output to std error
            - Command 2 > &2 # Not too useful but redirects error to std error
            - Command >> file # append output to a file
        - Input
            - Command < file # Direct file to input
            - Command << delimiter
                - This allows us to direct a specific file to input until we read a delimiter or usually EOF (End Of File)
    - Variables
        - PATH: directories for applications
        - USER: the name of the current user
        - SHELL: the current shell (usually bin/bash)
        - HOME: the home directory (usually just .)
        - PWD: the current working directory
- Chapter 11
    - wc command
        - -l count the lines
        - -w count the words
        - -c count the characters
        - -m count the bytes
    - Conditions & multiple commands
        - The response code for each command is:
            - 0: successful
            - Any other value: failure
        - Could be obtained by $? for the last response command.
        - Multiple commands
            - cm1 ; cm2 ; cm3 #run in sequence regardless of the state of completion
            - cm1 & cm2 & cm3 #run in parallel at the same time
            - cm1 && cm2 && cm3 #run if previous is successful
            - cm1 || cm2 || cm3 #run if previous failed
        - Testing []:
            - [condition] # spaces are needed
                - Option f - file is regular
                - Option z - value is empty
                - Option n - length of string is not 0
    - GREP - Global Regular Expression Print

            - Searching & filtering text
            - grep [options] pattern [files]
                - Option i - case insensitive
                - Option r - recursive search in all files
                - Option v - inverts the match
                - Option n - line numbers with matching lines
                - Option c - counte the number of lines
                - Option A n - Display n lines after each match
                - Option B n - Display n lines before each match
                - Option E - Extended regex

- Chapter 12
    - IFS: Input Field Separator
        - How to read a file:
            -
              ```
              IFS=“,”
              While read -r field1 field2 field3; do
                 Echo “Field 1: $field1”
                 Echo “Field 1: $field1”
                 Echo “Field 1: $field1”
              Done < input.csv
              ```
            - There is an assumption when reading this type of file: each line is separated by 2 commas creating 3 fields.
    - Creating a function:
        -
          ```
          funcname(){
             # commands
          }
          ```
    - Interactive scripts have a standard input (stdin) and standard output (stdou).
        - You can read from stdin
        - From file
        - From command line arguments
    - No sed command on the final exam
    - Formatting patterns:
        - \? → 0 or 1 occurrence
        - \+ → 1 or more
        - *   → 0 or more