CSC 111 Cheat sheet

Stuff written in this colour are comments

Stuff written in this colour is in theory valid code

Standard library commands:

Printf

printf(" Text you want to print");

printf("Variables you want to print %c %d %f %f", char, int, float, double);

/* %c -> char, %d -> int %f -> float and double

\n to start a new line and %0.Xf to restrict the decimals of a float or double to X decimals

The command printf is a function call */

Pointers

/* pointers store arrows that can point at variables of the same type. Meaning a pointer with the type int* can store arrows of type int.

The character * means follow the arrow the character & means an arrow pointing at. These two characters cancel each other out just like to minus signs would.

Example of creating pointers and arrows:

int* p; // creating a pointer

int a = 1 // creating an int with value 1

p = &a // putting an arrow pointing at a into the pointer p

*p = 5 // following the arrow in p to a and the storing the value 5 into a

The arrows stored in a pointer can be changed in the same way as the values of variables can meaning if store an arrow pointing at a variable b then p now points at b as if a never existed.

You can also create pointers that can save arrows to other pointers by adding another * to their declaration. For example, int** s, now I could store a arrow pointing at p in the same way I stored an arrow point at a in p: s = &p.

This process of creating pointers that point at other pointers can be continued until infinity by just adding more stars to the function declaration.

Important int is not the same type as int* and int* is not the same type as int**. Meaning using the example above a = p is an incorrect statement. */

Scanf

```
double v w;

scanf("%lf %lf",&v,&w);
```

/* The stuff inside the "" are the items that scanf will scan when run in this example 2 doubles.

The &v,&w are pointers pointing at the variables, that will be used to save the 2 doubles that have just been scanned.

The command scanf is a function call

%c -> char, %d -> int, %f -> float, %lf -> double */

If statements

if (condition for the code in the if statement to run){

Code inside of the if statement

} else if (condition for the code in the else if statement to run) {

This code will only run if the condition for the if statement is not met but the condition for the else if statement is met

} else { this is the code that will run if all other conditions are not met }

Loops

while (condition for the loop to run){

Code that runs if the condition for the loop is met

i++; <- this represents a command that changes the conditions in some way often by increasing a number. If this command is forgotten the loop will run infinitely.

```
int n;

for ( n = 1; n <= 6; n++){
```

code that will run if condition for the loop is met}

/* n = 1 <- gives the condition variable a starting value

n <= 6 <- is the condition for the loop to run

n++ <- command that changes the condition variable in some way

don't forget to separate the different statements with ; */

Functions:

Function statement

```
int this_is_a_function (int b, double c){

code inside of the function

return b; }
```

/* this_is_a_function is the name of the function

The int before the function name is called the function type. It tells the program the type of the value you return. In the example above this means that b will be returned as an int. If you don't want your function to return anything then the function type needs to be void.

The variables inside the brackets are space holders for the numbers you give the function when you call it. In the example above this means the function expect you to give it an int and a double in that order. Make sure to separate multiple variables with commas.

main is a function with the type int that returns the value 0. */

Function call

```
int a;

int b = 1

double c = 2.1;

a = this_is_a_function( b, c);
```

/* this_is_a_function is the name of the function you are calling.

b and c are the values you are giving the function. Important these values are copies of the values saved in b and c meaning that what ever happens to the numbers in the function does not effect the values stored in b and c

with the example above the value returned from this_is_a_function is stored in a */

Arrays:

Declaring arrays

/* arrays are basically pointers that can only point at certain places. Instead of adding * to the type declaration as you would do with pointers you use [].

Example:

Int Array[];

Inside of the [] you have to put the size you want your array to be. If you set your array to size 5 then you are creating 5 boxes that can each store a number with the type of the array. You can then make the array point at the different boxes just like pointers would point at other variables.

Example:

Int Array [5]; //declaring an array with 5 boxes of type int

int a; //declaring a variable with type int

int b = 2; // declaring a variable with type int and giving it the value 2

Array [1] = b; // saving the value of b in the second box of the array

a = Array[1]; // putting the value of the second box into a. meaning a now equals 2

Important when declaring an array you need to put the number of boxes in the square brackets, but when you are later pointing at a box you have to remember that first box is box 0 meaning as in the example above you have to enter the value 1 instead of 2.

The square [] are the same thing as adding a *. Meaning that the array int array[] is of type int*

Just as with pointers you can create boxes in arrays that point at other boxes by adding more sets of square brackets.

Example: int Array[i][j];

A array with two square brackets is known a 2d array where the [i] represent rows while [j] represent columns (like a matrix). */

Ctype commands:

#include <ctype.h> // to get access to the library

char c = 'I' // use '' for c to equal the numerical value of I

isupper(c) // returns 1 if c is upper case returns, 0 if not

islower(c) // returns 1 if c is lower case, 0 if not

isspace(c) // returns 1 if c is a space, 0 if not

isalpha(c) // returns 1 if c is a letter, 0 if not

tolower(c) // if c is a upper character turn it into its lower counter part

toupper(c) // if c is a lower character turn it into its upper counterpart

String commands:

#include <string.h> // to get access to the library

char S1 = "Hello"; // if you put your string in "" it will automatically put into a correctly sized array with a null terminator at the end (only possible when string is first being initialized)

char S2 = "Hi";

strcmp (S1,S2); // returns 0 when strings are the same something else when not

strlen(S1); // returns the length of the string minus the null terminator so in this case it returns 5

strcpy(S2,S1); // copies the content of S1 into S2 as far as there is space (make sure S2 I big enough to store all values of S1)

/* strings are the same thing as 1 dimensional arrays and can be initialized and worked with in the same way. To indicate when the string ends you need to add a null terminator represented by \0

When working with letters always remember that all functions and commands return the value of the letter as a number. This number can be accessed by surrounding the letter by ‘’. If you are working with words you need to surround each letter in the word with ‘’ separately.

File commands:

FILE *input_file = fopen(“filename.txt”,”r”) // input_file represents the name of the file when using it later in your program, filename.txt the actual file name and type. “r” means you will be scanning stuff from the file (read). If you replace the r with w the program will open a new file with the filename given with purpose of entering things into the file later. When using w and the file you are opening already exists it will be replaced by an empty file.

if (input_file == NULL) {

printf(“unable to open file”);

return 1; } // use this if statement to see if your file opened correctly

fgetc(input_file) // gets the next character from the file if no more characters in file it will return EOF

fscanf(input_file, “%lf”,&d) //works like normal scanf just you need to first declare what file you use and it returns the amount of values scanned

fclose(input_life) //to close your file (always close your file at the end of your code)

Malloc:

/* Malloc is a command that allows you to store things outside of any scope by freeing a certain amount of memory somewhere and then giving back a pointer to that space. Depending on what you want to store in malloc you need to free a different amount of space since, for example on int takes up less space then two doubles. To find out how much space you need to store your data use the command sizeof((type name)). Storing things outside of any scope means that the data stored there will never be destroyed by a function ending. This for one means you can easily transfer data from one scope to another, but also means that you need to delete the data yourself in the end. This is done with the command free((pointer pointing at your data stored by malloc)).*/

// Example of malloc being used:

int* primes_up_to_k(int k){ //random function not all of this code is relevant, but I needed to get values that I could save in malloc and then some reason to use them to I took this from one of bills examples

int num_primes = 0;

for(int n = 2; n <= k; n++){

if (is_prime(n))

num_primes++; }

int* primes =malloc(num_primes*sizeof(int)); //primes is a pointer the will store the arrow to malloc num_primes is the amount of int’s you want to store. Sizeof(int) determents the

mount of space needed for one int and multiply that number amount of int's you want to store to give malloc the size of space you need

```
int output_idx = 0;
for(int n = 2; n <= k; n++){
if (is_prime(n)){
primes[output_idx] = n; //you enter data into malloc with the same rules as with an array.
output_idx++; } }
return primes; }
int main(){
int* pointer = primes_up_to_k(50);
free(pointer); } //at the end of the end on the program you use free to delete the space malloc made
```

Structures:

Creating Structures

```
#include <stdio.h>
typedef struct {          // defining a structure should always be done at the beginning of your code
int a_int;                // inside of the brackets you put the components of the structure
char a_char;
} A_struc_in_a_struc; // this is the name of the structure


typedef struct {
char a_string[5];
double an_array[7];
float a_float;
a_struc_in_a_struc another_struc; // you can put structures inside of other structures just as you
}a_structure;                     would other types
```

Using Structures

```
Int main () {
a_structure test_struc; // in main or any function you declare a structure just like any other type
test_struc.an_array[1] = 5; // use dots to access the different variables of each structure
test_struc.a_float = 2;
test_struc.another_struc.a_int = 3; // use multiple dots if you want to access
return 0; }
```

/* Structures are basically file cabinets that allow you to store data of different types in the same place. When using structures in your program you can use them just like every other type. Just like a normal type you can assign a variable of a structure. This can be seen in the example with test_struc. test_struc is a variable that is assigned with the type a_structure, which is one of the structs in the example. However since structures are made up out of multiple variables you can not just assign a value to the structure itself, but only to the variables in the structure individually. Just like the file cabinet you need to put the file in a certain slot and not just put it on the top. To access the different variables in the structure you put a dot between the name of the structure and the name of the variable and then use it normally. An example used above is test_struc.a_float = 2;. If you are trying to access the variables of a structure that is inside another structure you continue the name then . pattern until you reach the final variable as seen in the example test_struc.another_struc.int = 3.

Structures and Pointers

/*The pointer rules for structures are the same as for every other type. There is however a small problem since the dot operator has a higher priority then the * operator the command to follow a arrow from a structure has to be written like this:

```
(*test_struc).a_float = 4;
```

Since this can get complicated there is a operator that does the same thing as (*(name)). This is the arrow operator: -> . rewriting using the -> operator the code line above will look like this:

```
Test_struc -> a_float = 4;
```

The arrow basically means follow back an arrow to the (in this case a_float) variable of the structure. */

Linked lists:

/* linked lists are a certain type of structure. Their function is similar to the one of arrays, but with the important difference that they are easier to modify. The variables inside of a linked list structure can be split into 2 parts. The first part are the variables that the list is storing, the second part are the pointers pointing to the next entry in the list and the last entry of a list. */

//Example:

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

//Definitions for a doubly linked list
typedef struct Node{            // normal structure definition
   int element;                 // the variable that stores the data inside of the linked list
   struct Node* previous;       // Pointer pointing at the previous entry of the list
   struct Node* next;           // Pointer pointing at the next entry in the list
} Node;

typedef struct{        // this is another structure that contains pointers to the beginning and end of
                          your linked list
   Node* head;         // this is the pointer to the front of your linked list
   Node* tail;         // this is the pointer to the back of your linked list
```

```
} LinkedList;
```

/* entering values into a linked list is a bit more complicated compared with an array. This is because every time you want to add a new value you have to declare a new node and then enter pointers linking the node with the previous and next entry additionally to entering the value. The next shows the code needed to do this: */

Adding to the front of the list

```
void(add_front(LinkedList* L, int new_value)){        //to add a node to the front of your linked list
Node* new_node = malloc (sizeof(Node));   //create a space in malloc big enough for the node
new_node -> element = new_value ;        //set the value of your node to your desired value
new_node -> previous = NULL;              //making sure the node points to nothing in the beginning
new_node -> next = NULL;

if(L -> head == NULL){              //this if statement is if there is no node in your linked list yet
L -> head = new_node;               //since there is only one node in your list its both the start and end of
L -> tail = new_node;               your list meaning both the tail and head pointer have to point at it
}else{              //this else statement is used when there is at least one other node already in your list
Node* old_first_element = L->head; //this gets the former first entry from your linked list
old_first_element -> previous = new_node; //this sets the previous pointer of the former first entry
                                            to point at your new node
new_node->next = old_first_element;     //this sets the next pointer of you new node to point at the
                                          former first entry
L ->head = new_node;                //this makes the header of you linked list to now point at the new
                                      node. These last three steps cause the new node to become the new
                                      first entry of the linked list
}}
```

/* since linked lists are stored “in malloc” they do not get destroyed when the function ends. This means you don’t have to worry about returning them from your function. However you do need to delete them yourself at the end of your program (or when ever you don’t need them anymore). This next example shows how you can delete a node. */

Deleting lists

```
void delete_first(LinkedList* L){
Node* node_to_delete = L->head; //make node to delete point at the first node in your list
L->head = node_to_delete->next; //make the start of your list point at the value after the one you
                                    are deleting.
free(node_to_delete);}              //deleting the node
```

Operators:

Operator	Description	Associativity
() [ ] . -> ++ --	Parentheses: grouping or function call Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement	left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right