Ryan Young

C Dirty Sheet 😉

**Libraries to include:**

Input and Output: <stdio.h>

Character Types: <ctype.h>

Strings and (byte) array: <string.h>

General Function: <stdlib.h>

Numerical Math Functions <math.h>

**Strings:**

Remember the array of chars will always have a 0 as the last value so if the string is “Hello” then the strlen(“Hello”) is 6.

**Memory:**

Three pools of memory: Static/global, stack, and heap

Static/Global:

All constants (including string literals) are held

Global variables

All variables declared “static” are held

Memory is allocated when the program starts and deallocated when the program terminates thus making it have a fixed size.

Stack:

This is where local variables are stored

Easy to manage because it is automatically handled and the scope is only in the function, the default stack size using gcc is 2 mb and needs to be increased when using large arrays and deep recursion.

Heap:

This is where memory that programmers manually set comes from, memory lasts here until it is freed.

To request memory use malloc(size of type x size of bytes you want)

Example: char \*P = malloc(100); // this requests a 100 bytes for the type char and the pointer p references it. Malloc can fail if for example, there is not enough memory to allocate and this this case malloc will return NULL.

Calloc: Same thing as malloc but initialized the content of the memory to 0

Adjusting the size of memory uses realloc!!!!!!!! You pass in the pointer to the memory you want to change and the value you want to adjust the size to.

Deallocation:

Free(ptr)

Rule 1: what ever you allocate manually you have to free

Rule 2: Only free what is allocated via malloc/calloc/realloc

If you do not follow these rules, then memory leaks occur and horrible crashes can occur with memory errors and corruption of the heap.

**Pointers and Arrays:**

You can pass an array into a pointer by just using the name because the name itself is the address of the first element in that array. Aka if you do arr[9] you can do int \* pt = arr and it will work.4

THEY ARE NOT THE SAME THING!!!!!!!!!!!!!!!

Int a[10]; // array of length 10

Int \*p = a; // a is converted to an int \* or integer pointer

&a; // pointer (the address of a) to an array of 10 int’s aka int \* p = a[10]

&p; // a pointer to a pointer to int aka int \*\*

So if you do sizeof(a) // will return 40 because 10 \* 4(size of an int)

Sizeof(p)// will return 8 because p is a pointer

Pox = malloc(6 \* sizeof(int));

Pox is now basically an array and you can use it as if it were an array of ints

Using a pointer as an array:

Int \* pox; // creating a pointer to something of type int

Pox = malloc(sizeof(int)\*n); // requesting bytes for n ints from the heap

\*pox = 0; set the int at the address pox to 0

Pox[0] = 0; // same thing

Pox[1] = 1; // the int after pox[0]

….

Free(pox); // will free the memory that pox was assigned to

Making an array of pointers:

Char \* p[3]; / initializing an array of length three for type char \*

P[0] = malloc (10);

P[1] = malloc(10);

P[2] = malloc(10);

Now each part of the array points to 10 bytes of memory

**Command Line Arguments:**argc: the number of arguments on the command line

Argv: array of pointers to characters

Aka the number of elements is argc and each element in an array points to null-terminated strings

**2D-Memory:**

You need to call malloc m times for m rows and n times for each element in that row

Arr = malloc(m \* n \* sizeof(int));

Arr[1][2] is the second row third column it would start at the 0 row and 0 column

**Pointers:**

The value of a pointer is a byte address

Unsigned integer used to number bytes in memory

Range is between 0x00000000 and 0xFFFFFFFF [32-bit systems]

Range is 0x0000000000000000 and 0xFFFFFFFFFFFFFFFF[64-bit systems]

Corollary:

If a pointer is an integer, you can do arithmetic….

So you can computer other addresses

Pointer addition:

Int \*p; // point p points to an int

The address in the pointer is 1000

P + 1 is the next byte address so p + 1 would give us 1004

Printf(“%d”, p); //would yield 1000

Printf(“%d”, p+1); //would yield 1004

Increases by 4 because the size of an integer is 4

If you add an integer to a pointer the result will be a pointer of the same type, but it is different from regular integer addition. The integer is automatically scaled by the size of the type pointed to. Have to remember that all a pointer is, is an address and then when you dereference that pointer that address turns into the thing at that address.

**Structures:**

Mechanism to define new types

If you pass in a pointer to a structure to a function you need to use the -> when referencing the values associated with that structure. If you are working directly on a “local” structure, then you use the . notation instead of the ->. The -> notation is actually a shortcut to dereference the pointer and then call the . on the structure.

Typedef struct{

Char name[32];

Int age;

Char gender;

} Tperson;

Now the when using this structure the type is Tperson