**C Programming Notes**

Type casting a void pointer looks like, \*((int \*) p)

#include <stdio.h>

int main(void) {

void \*ptr;

char a = 'A';

int b = 2;

float c = 3;

ptr = &a;

printf("\nThe value of a = %c\n",\*((char\*)ptr)); // type casts ptr to char

ptr = &b;

printf("\nThe value of b = %d\n",\*((int\*)ptr)); // type casts ptr to int

ptr = &c;

printf("\nThe value of c = %f\n",\*((float\*)ptr)); // type casts ptr to float

return 0;

}

\*\* To print the memory address of something \*\*

int a = 45;

int \*myPointer = &a;

printf(“Number of a is %d and the memory address is %p”, \*myPointer, myPointer);

// %p is what you use to display a memory address

**W1**

To compile a C file: gcc -o HelloWorld HelloWorld.c

To run your C file: . /HelloWorld

To get input from a user:

int myNum;

printf(“Enter your favourite number”);

scanf(“%d”, &myNum); // This is the correct syntax for storing the number

Functions:

A function will always return something:

1. For void functions, it returns a return;
2. For non-void functions, it returns a return expression, i.e. return true;

It is a good practice to create a function prototype whenever you create a function. It looks like:

int myFavouriteNum(int a, int b); /\* This is a function prototype \*/

int main(int argc, char \*\*argv) {

printf(“The sum of my two favourite numbers are %d”, myFavouriteNum(1, 4));

return 0;

}

int myFavouriteNum(int a, int b) {

return a + b;

}

Arrays:

Array declarations:

int grades[100]; // a one dimensional array

int twoGrades[2][3]; // a two dimensional array

int threeGrades[2][3][4]; // a three dimensional array

Array initializations:

int c[] = {21, 103, 221};

int c[3] = {4, 5, 8};

int a[2][3] = {{2, 4, 6}, {1, 3, 5}};

#define N 100

int a[N], sum, eye, \*p;

The following statements are equivalent:

p = a; **AND** p = &a[0];

p = a + 1; **AND** p = &a[1];

sum = 0;

for (eye = 0; eye < N; eye++) { **AND** for (eye = 0; eye < N; eye++) {

sum += a[eye]; sum += \*(a + eye);

} }

**\*Note\*** To copy another array, you must use a for loop to copy each element of the array every time it loops.

In C, a string is a one-dimensional array of type *char*.

String declarations and initializations:

Example 1:

char w[100];

w[0] = ‘L’;

w[1] = ‘O’;

w[2] = ‘S’;

w[3] = ‘T’;

w[4] = ‘\0’ // this is the end of the line

Example 2:

char w[] = {‘L’, ‘O’, ‘S’, ‘T’};

char w[] = “LOST”;

char \*w = “LOST”;

**Understanding main(int argc, char \*\*argv):**

Suppose we have a file called ‘dog.c’ and we give the command: dog cool

argc = 2 // this is how many words we type in the terminal

argv[0] = dog; // this is the first word

argv[1] = cool; // this is the second word

Another file called ‘bitch.c’ and we give the command: master mind is fat

argc = 4;

argv[0] = master;

argv[1] = mind;

argv[2] = is;

argv[3] = fat;

Structures:

Structures are a means of aggregating a collection of data items of possibly different types.

Example:

struct dog { //This is how you create a structure using the keyword *‘struct’*

char \*name;

char \*eyeColor;

int age;

}

struct dog myDog; // creating a struct variable called *‘myDog’*

To access a member:

myDog.name = “Brownie”;

myDog.eyeColor = “Brown”;

myDog.age = 12;

**Dynamic Memory Management:**

C provides calloc() and malloc(), and the function prototypes are in stdlib.h

When you use calloc(), the storage are automatically initialized to zero, while malloc() the items in the storage are not initialized.

Example, malloc():

void \*p = malloc(10 \* sizeof(int)); // this reserves 10 memory for an int variable

// 10 is the number of elements, and sizeof(int) is the size of each element

\*Note\*

It is important to typecast, because malloc and calloc are void pointers, for example:

int \*p = (int\*)malloc(10 \* sizeof(int)); // This is the right way of using malloc

Example, calloc():

calloc – void\* calloc(size\_t num, size\_t size)

void \*p = calloc(10, sizeof(int));

The right way to use calloc is:

int \*p = (int\*)calloc(10, sizeof(int));

\*Important\*

It is important to free the memory when using malloc() and calloc().

To free memory:

free(p); // This frees the reserved memory of the pointer \*p

And it’s a good practice to set the pointer address as NULL after it’s been freed.

p = NULL;