**C Programming Notes**

Compiling:

Without C99 – gcc –o myProgram myProgram.c

With C99 – gcc –std=c99 –o myProgram myProgram.c

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));

Example, realloc():

void \*realloc(void \*ptr, size\_t size)

char \*str = (char \*)malloc(10 \* sizeof(str));

str = realloc(str, 25 \* sizeof(str)); // changes the size to 25

// Can also be written

int \*mate = (int \*)realloc(str, 20 \* sizeof(mate)); // data type now an ‘int’ and size ‘20’

\*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;

Forking and Creating a separate Process

#include <sys/types.h>

#include <stdio.h>

#include <unistd.h>

int main() {

pid\_t pid; // creating a process id

pid = fork(); // Fork a child process

/\* If pid is less than 0, then there’s an error \*/

if (pid < 0) {

printf(stderr, “Fork Failed”);

return 1;

/\* If pid is 0, then this is the child process \*/

} else if (pid == 0) {

printf(“I am the child process”);

/\* If pid is greater than 0, then it is the parent process\*/

} else if (pid > 0) {

wait(NULL); // parent will wait for the child to complete before executing

printf(“I am the parent and I waited for the child process!”);

}

return 0;

}

If you want to use forking you will need to include all of this:

#include <unistd.h> // Symbolic Constants

#include <sys/types.h> // Primitive System Data Types

#include <errno.h> // Errors

#include <stdio.h> // input, output

#include <sys/wait.h> // Wait for Process Termination

#include <stdlib.h> // Standard Library

Make file:

To create a make file, create a file called ‘Makefile’

To compile one c program:

myVariable: myFile.c

gcc -std=c99 -o file myFile.c

To compile a bigger project: https://www.youtube.com/watch?v=\_r7i5X0rXJk

fib: fib.o main.o

gcc fib.o main.o -std=c99 -o fib

fib.o: fib.c fib.h

gcc -c -std=c99 fib.c

main.o: main.c

gcc -c -std=c99 main.c

clean:

rm \*.o fib

If you have multiple make files you can specify the -f option to make a specific make file, for example:

Make -f fibfork.Makefile

Piping:

**\*Note\*** It is important to create a pipe first before forking, so the parent process creates a message via [1], so if one or more child processes can read the message via [0].

*On success 0 is returned, On error -1 is returned.*

Remember that ‘1’ is for writing and ‘0’ is for reading.

char \*message = “This is a message”;

int fd[2];

char buf[2048]; // size of char array (string)

Writing example: write(fd[1], message, strlen(message) + 1);

// Parameter ‘fd[1]’ is for writing, param 2 ‘message’ is the message, and param 3 is the length

Reading example: read(f[0], buf, 2048);

// Parameter ‘fd[0]’ is for reading, param 2 ‘buf’ stores the message received, and param 3 is the length;

Header files for piping:

#include <unistd.h>

#include <stdio.h>

#include <string.h>

**Piping Example -> Next page**

Piping example:

#include <unistd.h>

#include <sys/types.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

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

char \*myMessage = “I will divorce your mum!”;

int fd[2];

int p;

char \*buf; // can be char buf[] or char buf[1024], the number inside is the size

pid\_t childID;

p = pipe(fd); // creates a pipe for the ‘fd’ array

if (p != 0) {

// If result is 0, it’s a success, if not error

}

childID = fork(); // creates a child process (new process)

if (childID == 0) {

printf(“This is the child ID”);

write(fd[1], myMessage, strlen(myMessage) + 1); // writes message

exit(0);

} else {

printf(“I’m the parent!”);

read(fd[0], buf, 1024); // reads the message and stores in variable ‘buf’

printf(“\n\nMessage Sent is %s”, buf);

}

return 0;

}

Threading - pthreads

To compile using pthread, make sure to include -pthread. For example:

gcc -std=c99 -o myThread myThread.c -pthread

#include <pthread.h>

void\* sum\_runner(void\* arg) {

long long \*limit\_ptr = (long long \*)arg;

long long limit = \*limit\_ptr;

long long sum = 0;

for (long long i = 0; i <= limit; i++) {

sum += i;

}

pthread\_exit(12); // This exits the thread, and whatever value is passed inside the parameter will be the return value

}

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

long long limit = atoll(argv[1]);

pthread\_t threadID; // Create thread id

pthread\_attr\_t attr; // Thread attribute initialize

pthread\_attr\_init(&attr); // Thread create

pthread\_create(&threadID, &attr, sum\_runner, &limit); // create thread

// Param 1 is thread id, P2 is the attribute, P3 is the called function, P4 if any argument/variable // you want to send to the called function (P3)

pthread\_join(tid, NULL); // This waits till thread is done working, second param is the

// return value from pthread\_exit(12);

// if second param is null, it doesn’t catch the return value

}

Sockets

Header includes:

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

Server:

* Create a socket with the socket()
* Bind the socket to an address using the bind()
* Listen for connections with the listen
* Accept a connection with the accept()
* Send/Receive data using read()/write()

Client:

* Create a socket with the socket()
* Connect the socket to the address of the server using connect()
* Send/Receive data using read()/write()