CSCB09 Week 5 Notes

1. **Pointer to Pointer:**
   - Also known as a **Double Pointer**.
   - Syntax: type **var
   - Used to store the memory address of pointers.
   - **Note:** When dereferencing a double pointer, we do not get the final object. To get the final object, we must dereference it twice.
   - **Note:** We can also have triple pointer, quadruple pointers, etc, but we don't need to know them for CSCB09.
2. **Passing command line arguments to a C program:**
   - The command line arguments are handled using main function arguments.
   - We need the main function to take in 2 arguments:
     1. int argc: The number of command line arguments.
     2. char *argv[]: A pointer array which points to each argument passed to the program.
        a. argv[0] is the name of the program.
        b. The remaining elements of argv contain the arguments
   - **Note:** argv[1] is a pointer to the first command line argument supplied, and *argv[n] is the last argument.
   - If no arguments are supplied, argc is 1, and if you pass one argument then argc is 2.
   - E.g. Consider the C code below:

```
#include <stdio.h>
int main(int argc, char *argv[]){
  // Prints the value of argc.
  printf("The value of argc is %d\n", argc);
  printf("The value(s) of argv are:");
  // Prints out all  the values of argv.
  for (int i = 0; i < argc; i++){
     printf("%s\n", argv[i]);
  }
  return 0;
}
```

Suppose the function name is test.c. If I successfully compile test.c and do ./test a b c d e f g, then the output is:

```
The value of argc is 8 // This prints the value of argc, which is 8.
./test  // This is argv[0].
a // This is argv[1].
b // This is argv[2].
c // This is argv[3].
d // This is argv[4].
e // This is argv[5].
f // This is argv[6].
g // This is argv[7].
```

3. **Typedef:**
   - Typedefs allow us to define types with a different name - which can come in handy when dealing with structs and pointers.
     I.e. You can use typedef to give a name to your user defined data types.
   - typedef also allows you to define a short-hand.
   - E.g. Consider the C codes below:

```c
#include <stdio.h>
typedef struct Example1{
  int x;
  int y;
  int z;
} p1; // We can now use p1 as a data type.
int main(){
  // Using p1 as a data type.
  p1 P1 = {1,2,3};
  printf("P1.x is %d\n", P1.x);
  printf("P1.y is %d\n", P1.y);
  printf("P1.z is %d\n", P1.z);
  // Using Example1 as a data type.
  struct Example1 P2 = {1, 2, 3};
  printf("P2.x is %d\n", P2.x);
  printf("P2.y is %d\n", P2.y);
  printf("P2.z is %d\n", P2.z);
  return 0;
}
```

- If we use typedef and we use p1 as the data type, then we do not need to use struct. However, if we are using Example1 as a data type, we still need to use struct.
- Typedefs is good for large programs, so you don't have to repeatedly use structs.

4. **Organizing your program:**
    - C does not offer classes or similar concepts to organize your code.
    - Large programs are organized by breaking them down into multiple files.
    - Suppose you have these 3 files as part of a project:
        1. main.c
            ```
            #include "list.h"
            int main() {
            // Some code
            }
            ```
        2. list.c
            ```
            #include "list.h"
            int isEmpty(List *h) {
            // Some code
            }
            void add(List *h, int v) {
            // Some code
            }
            void remove(List *h, int v) {
            // Some code
            }
            ```
        3. list.h
            ```
            struct node {
                    int value;
                    struct node * next;
            } ;
            typedef struct node List;
            int isEmpty(List *);
            void add(List *, int);
            void remove(List *, int)
            ```
    - We can compile the above functions using **gcc main.c list.c –o myprogram**.
    - What gcc main.c list.c –o myprogram is doing is:
        1. First the **pre-processor** runs, it looks for example for #include directives and includes the corresponding .h file.

2. Then the **compiler** runs on each .c file. It produces machine code/object code for each .c file and places it in a .o file (main.o, list.o).
3. Then the **linker** takes all the object files and combines them into one executable (called myprogram in our example).

- We have recompile the program each time an edit is made because when we compile the C code, we are translating the C code to machine level code that runs directly on your hardware. Therefore, every time you make an edit a new executable has to be created.
- **Note:** To run the files together, they must be in the same folder/directory.
- **Note:** We did not have to compile each time a change was made in a shell script. This is because the commands inside a shell script are interpreted by the shell, they are not translated to a machine language program.
- The problem with manually calling gcc is that if you have a lot of files, the list can grow every long. Furthermore, this recompiles every module, even if it has not changed.
  E.g. gcc file1.c file2.c file3.c file4.c ……… –o myprogram
- We can also compile C code using this method:
  gcc –c list.c # this produces list.o
  gcc –c main.c # this produces main.o
  gcc –o myprogram main.o list.o # produces executable
- The above code only recompiles files that has changed by separating compilation & linking. However, this is still not efficient.
- We can use **Makefiles** to solve this problem.
- **Makefiles:**
  - Makefiles are processed by a program called **make**.
  - Contain information about "targets" and "dependencies".
    E.g. myprogram depends on list.o and main.o
    E.g. list.o depends on list.c and list.h
  - Contain information about "rules".
    E.g. To produce list.o run "gcc –c list.c"
  - **Make** looks at timestamps, and only recompiles a target if one or more of its dependencies are newer.
    I.e. Make allows a programmer to easily keep track of a project by maintaining current versions of their programs from separate sources.
  - Furthermore, make can automate various tasks for you, not only compiling proper branch of source code from the project tree, but

helping you automate other tasks, such as cleaning directories, organizing output, and even debugging.
    - Syntax:
      <target> : <dependencies>
      action
      Note: There can be more than 1 dependency.
    - **Note:** Makefile reads from top to bottom. If Makefile finds anything that was changed, then it will compile that part only. It won't touch the stuff that wasn't changed.
    - To run Makefile, use the **make** command.
    - To run clean, do **make clean**.
- **Pre-processor directives:**
    - Include Header Files:
        - #include: Inserts a particular header from another file.
    - Define Marcos:
        - #define: Substitutes a preprocessor macro.
        - **Note:** A macro is like a constant.
    - Conditional Inclusions:
        - #ifdef: Returns true if this macro is defined.
        - #ifndef: Returns true if this macro is not defined.
        - #undef: Undefines a preprocessor macro.
        - #endif: Ends preprocessor conditional.
        - #error: Prints error message on stderr.

5. **Error Handling:**
    - The return value of library functions tells you if there was an error.
      E.g. If malloc returns NULL, then there's an error.
    - Many library functions use a global variable called **errno** to store more information on what went wrong.
    - errno is declared in errno.h.
      I.e. #include <errno.h>
    - At process creation time errno is zero. A value of 0 indicates that there is no error in the program.
    - When a library call error occurs, errno is set.
    - Some possible values of errno:
        - ENOMEM: "Not enough space"
        - EDOM: "Domain error"
        - EACCESS: "Permission denied"
    - However, we need to be careful when we're using errno.

- Consider the C code below:

  ```
  if (somecall() == -1) {
          printf("somecall() failed\n");
          if (errno == ...)
          {
           ...
          }
  }
  ```

  The problem with this is that printf might change the value of errno if it encounters an error. If printf works successfully, then it returns a 0. Since errno looks at the last return statement, it will check the return statement of printf and see the 0, instead of the -1 from somecall(). Therefore, we need to save the value of errno before doing any further processing.
- **perror():**
  - Syntax: void perror(char *str)
  - perror displays str, then a colon(:), then an English description of the error as defined in errno.h.
  - Protocol:
    - check system calls for a return value of -1 or NULL
    - call perror() for an error description

6. **<u>I/O in C:</u>**
   - A file represents a sequence of bytes.
   - Two main mechanisms for managing file access:
     1. File descriptors (low-level):
        - Each open file is identified by a small integer.
        - Use for pipes, sockets (will see later what those are …)
     2. File pointers (regular files):
        - You use a pointer to a file structure (FILE *) as handle to a file.
        - The file struct contains a file descriptor and a buffer.
        - Use for regular files
   - **Standard streams:**
     - All programs automatically have three files open:
       - FILE *stdin;
       - FILE *stdout;

        - FILE *stderr;

	stdio name	File descriptor	Default Location
Standard input	stdin	0	Comes in from keyboard
Standard output	stdout	1	Comes out on screen
Standard error	stderr	2	Comes out on screen

- **Opening Files:**
    - You can use the fopen function to create a new file or to open an existing file.
    - Syntax:
      FILE *fopen(const char *filename, const char *mode);
    - filename identifies the file to open.
    - mode tells how to open the file:
        - "r" for reading
          I.e. Opens an existing text file for reading purpose.
        - "w" for writing
          I.e. Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
        - "a" for appending
          I.e. Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here, your program will start appending content in the existing file content.
    - This returns a pointer to a FILE struct which is the handle to the file. This pointer will be used in subsequent operations.
- **Closing Files:**
    - Syntax: void fclose(FILE *stream);
- **Writing Files:**
    - fputc
        - Syntax: int fputc(int char, FILE *stream);
        - Writes the character value of the argument char to the output stream referenced by stream.
        - It returns the written character written on success otherwise EOF if there is an error.
    - fputs
        - Syntax: int fputs(const char *str, FILE *stream);

        - Writes the string str to the output stream referenced by stream.
        - Returns a non-negative value on success, otherwise EOF is returned in case of any error.
    - fprintf
        - Syntax: int fprintf(FILE *stream, const char *format, ...);
        - Stream: This is the pointer to a FILE object that identifies the stream.
        - Format: This is the C string that contains the text to be written to the stream. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested.
- **Reading Files:**
    - fgets:
        - Syntax: char *fgets(char *s, int size, FILE *stream);
        - fgets() reads up to size-1 characters from the input stream referenced by stream. It copies the read string into the buffer s, appending a null character to terminate the string.
        - **s:** Pointer to an array of chars where the string read is copied.
        - **size:** Maximum number of characters to be copied into str (including the terminating null-character).
        - ***stream:** Pointer to a FILE object that identifies an input stream.
          stdin can be used as argument to read from the standard input.
        - **returns:** The function returns s.
        - It is **safe** to use because it checks the array bound.
        - It keep on reading until new line character encountered or maximum limit of character array.
        - We can get fgets to read from the keyboard by using stdin for stream. E.g. fgets(s, size, stdin).
    - gets:
        - Another function to read from keyboard.
        - Syntax: char *gets(char *s);
        - Reads from keyboard until \n and stores results where buffer s points to.

        - We should never use gets because it does not check the array bound.
- **Binary:**
    - Block I/O allows you to read and write binary data, i.e. you read and write byte-for-byte rather than lines of characters.
    - Suppose you store the number 1,999,999 as follows:
      fprintf(fp, “1999999”);
      This takes 7 bytes because each character is a byte. However, by using block I/O, we can use 3 bytes to store it.
    - fread:
        - Syntax: size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
        - Reads data from the given stream into the array pointed to, by ptr.
        - Read nmemb * size bytes into memory at ptr.
        - Returns number of items read. If this number differs from the nmemb parameter, then either an error had occurred or the End Of File was reached.
        - Ptr: This is the pointer to a block of memory with a minimum size of size*nmemb bytes.
        - Size: This is the size in bytes of each element to be read.
        - Nmemb: This is the number of elements, each one with a size of size bytes.
        - Stream: This is the pointer to a FILE object that specifies an input stream.
    - fwrite:
        - Syntax: size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
        - Writes nmemb * size bytes from ptr to the file pointer stream
        - Returns number of items written. If this number differs from the nmemb parameter, it will show an error.
        - Ptr: This is the pointer to the array of elements to be written.
        - Size: This is the size in bytes of each element to be written.
        - Nmemb: This is the number of elements, each one with a size of size bytes.
        - Stream: This is the pointer to a FILE object that specifies an output stream.