Unix and C Programming COMP1000

S2 Assignment, 2018 – Report

Kevin Le – 19472960

<u>Contents</u>:

1. Main.c

1.1 main(int argc, char* argv[])

Function used to begin program, takes user command line argument input assigned to **`argc`** and **`argv[]`** char array. Validates number of arguments (2), if valid, calls **`readFile()`** in **`FileIO.c`** passing in the argument array. If not valid, output error message and exits program.

2. FileIO.c

2.1 readFile(char* argv[])

Reads file and stores commands. Function opens file, checks if successful/ not empty and initialises relevant data structures to begin reading file and storing into. Converts commands to upper case, then initializes appropriate structs to store command and value, which is then inserted into a linked list.

2.2 writeTo(double x1, double y1, double x2, double y2, int type)

Writes (appends) to a log file, "graphics.log". Writes points x1, y1, and destination points x2, y2 in the format **`COMMAND(x1, y1)-(x2-y2)`**. With the values being real values and having 3 decimal place accuracy. Uses type value to determine COMMAND that was run.

2.3 toUpper(char string[])

Converts all lower-case characters in a character array (String) to upper case. Required to ensure consistent case for all commands and **`strcmp`** compatibility in **`runCmd`** function.

3. RunCmd.c

3.1 runCmd(LinkedList* list, int numLines)

Tracks changing/ appended data using a struct. Iterates through linked list, validating, comparing strings to the current node in linked list to determine what command to run, if match is found, convert commands value (currently a string) to required datatype, running extra calculations if commands are MOVE/DRAW, then performing the operation. If invalid command found, modify loop condition to exit Linked List iteration, print an error and free all memory allocated. Points **`PlotFunc`** pointer to plotter function.

3.2 updateCoord(double d, Info* data, int type)

Takes in the command value, (e.g. DRAW 5, imports 5). Info struct, and type parameter. Updates/appends to the information struct the appropriate new coordinates. Storing the previous finished drawn point to a temporary x1, y1. Also stores the real valued calculations to be passed to **`writeTo`** function using static variables (as it stays isolated to the function).

3.3 plotter(void* plotData)

Simple `printf` statement to print out `plotData`, passed in by the calling function in `effects.c. Effects.c` function line must have pointer to plotter function. Declared in `runCmd().`

3.4 roundValue(double value)

Rounds real value to nearest integer. E.g. 1.2 rounds to 1, 2.6 rounds to 3. Returns rounded value.

3.5 validateSScanf(int success, int required)

Checks `sscanf` return value `success` against asserted value `required`. Returns 0 if invalid, 1 if valid. Can be checked in if in calling function. Sets value of while loop in `runCmd` to exit.

4. LinkedList.c

4.1 LinkedList* newList()

Allocates memory for a new Linked List, using struct LinkedList. Sets head and tail to NULL, returning initialized list to calling function.

4.2 insertLast(LinkedList* list, Data* cmdLine)

Inserts data struct onto the end of the Linked List. Declares a new node, using `struct LinkedListNode`. Pointing the `newNode` to the Data struct. Sets the `newNode` next to `NULL`. Then inserting it into the linked list. If the linked list is empty, then the `newNode` is made the head and tail, if there are existing nodes, it sets the current tails next node pointer to the `newNode`, then sets the `newNode` as the new tail.

4.3 printList(LinkedList* list)

Prints out the Linked List contents. Assigns head to a new node named current, iterates through Linked List until it hits the end. Printing each command and value pointed to by the `nodeData` in Linked List Node.

4.4 freeList(LinkedList* list)

Frees the linked list, nodes in the linked list, and any data in the linked list that requires freeing.

5. Conversion of input file into coordinate system

5.1 Implementation:

My method of turning the file commands into a system to calculate required coordinates heavily relied on the use of Linked Lists and structs. Reading in the file line by line using **`fgets`**, I split each command and value into strings using **`sscanf`**. Then converting that command to upper case to ensure uniformity. I then allocate memory for the Data struct itself which stores the command and value, both also requiring memory to be allocated. This runs through the entire file repeating the process inserting it into the end of the linked list. After the entire file is read, the Linked List contains all the data needed and can be simply passed to **`runCmd`** to begin performing the required operations.

runCmd initializes a new Linked List node called current, pointing at the start of the Linked List, a while loop begins going node by node through the linked list. Performing the required operation if the command matches with an if condition. Updating the coordinates when MOVE/DRAW is run shows the strength of structs, being able to compactly store data. I can just pass by reference the struct to the **`updateCoord`** function, which then can use the **`currAngle`** in the struct to modify/append the x/y calculated values in x1, y1, x2, y2 in the struct also. It saves the need for 5 imports. Once it has modified the values, it continues going through the loop or calls line. Once the loop finishes up, the image is successfully printed.

5.2 Alternate Implementation:

An alternative approach for storing the commands is malloc'ing an array of strings (char array) that stores the entire command as a string (along with the value, e.g. COMMAND X is located in each element), then when running the commands, loop for the entire length of the array splitting the string at the array index into separate commands and values, then performing the required operations. An alternate way of keeping track of the data is also the use of standard variables, this is a less elegant solution then structs, but the variables can be passed by reference to the function **`updateCoord`** via pointer, which in turn can modify its values. This can lead to large imports and calls. This method should provide the same outcome.

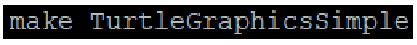
6. Program Demonstration

6.1 Compiling the program

There are 3 makefile targets for compiling.

- **`TurtleGraphics`** – Normal program, all features including appending to graphics.log
- **`TurtleGraphicsSimple`** – Equivalent to **`TurtleGraphics`**, however with FG being set to black and BG set to black. All FG/BG commands via file is ignored.
- **`TurtleGraphicsDebug`** – Equivalent to **`TurtleGraphics`**, however log file entries are printed out alongside the drawing whenever line is called.

Example Execution:



```
make TurtleGraphicsSimple
```

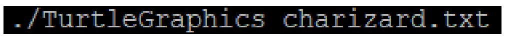
6.2 Running the program

Running the program works like so:

`./TurtleGraphics file.extension`

`file.extension` being the file to be opened with commands to read in and extension being the filetype. **`TurtleGraphics`** is the executable name.

Example execution:

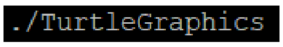


```
./TurtleGraphics charizard.txt
```

6.3 Error Handling

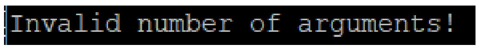
6.3.1 Invalid Number of Arguments

Run With:



```
./TurtleGraphics
```

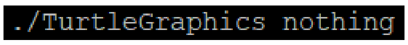
Error Message:



```
Invalid number of arguments!
```

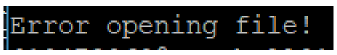
6.3.2 Invalid File (Empty/ Does Not Exist)

Run With:



```
./TurtleGraphics nothing
```

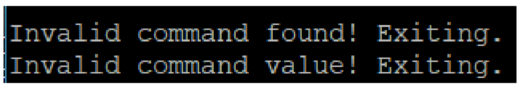
Error Message:



```
Error opening file!
```

6.3.3 Invalid File Contents (Invalid Command/ Value)

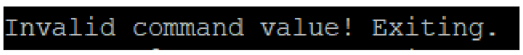
Invalid Command Message:



```
Invalid command found! Exiting.
Invalid command value! Exiting.
```

`MOV 3`

Invalid Value Message:



```
Invalid command value! Exiting.
```

`DRAW a`

6.4 Input file contents

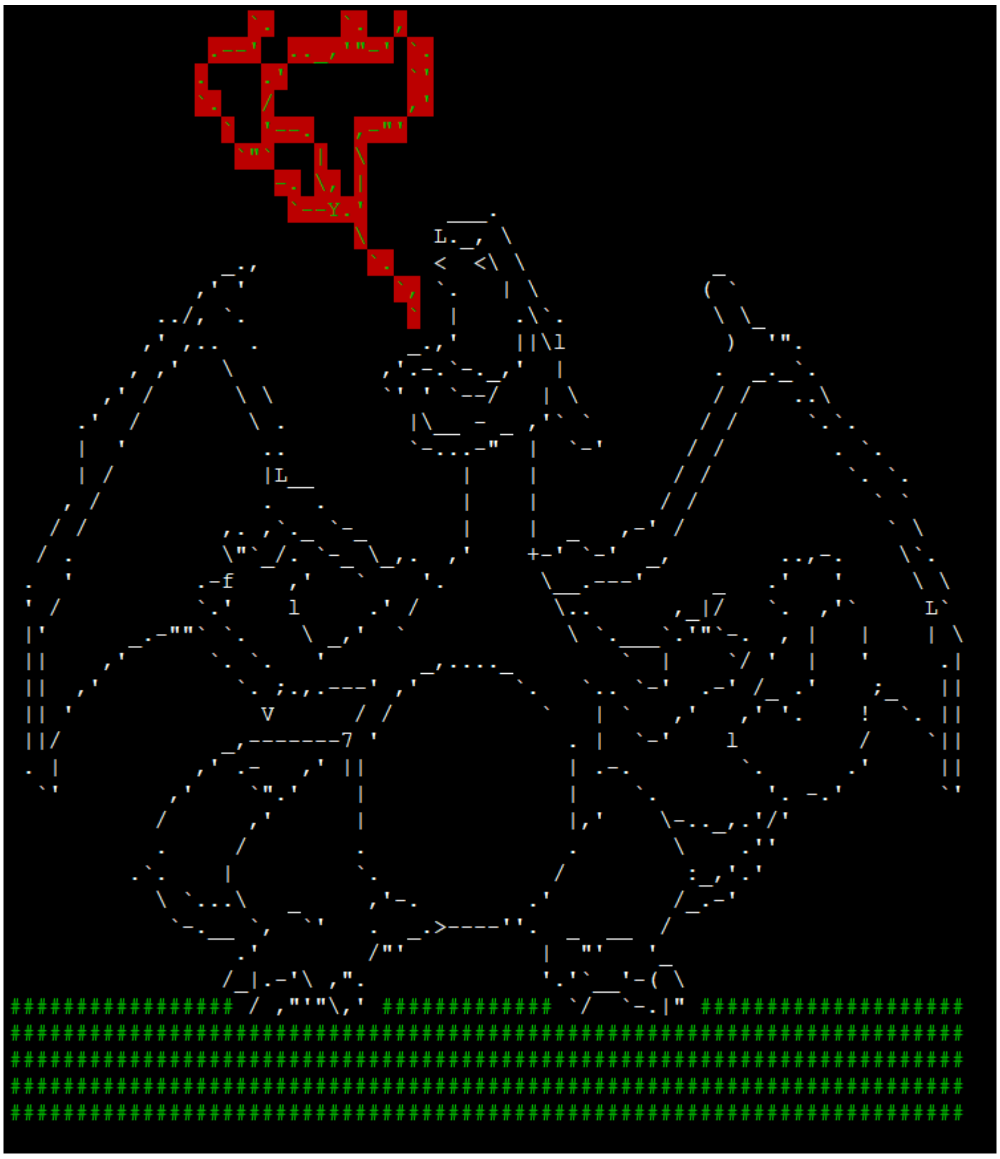
The input file snippet:

```
FG 2          DRAW 1        DRAW 1        DRAW 1        MOVE 25

BG 1          PATTERN "     PATTERN ,     PATTERN _     ROTATE 90     MOVE 18

MOVE 18       DRAW 1        DRAW 1        DRAW 2        MOVE 1        PATTERN `

PATTERN .     PATTERN -     PATTERN .     ROTATE 180    ROTATE 90     DRAW 1
```

6.4.1 Expected Output

The output is an image drawn using basic ASCII symbols.

Example Output to Screen:



Running the program also appends to a file graphics.log

Expected contents of graphics.log:

```
MOVE ( 12.000,  33.000)-( 16.000,  33.000)
DRAW ( 16.000,  33.000)-( 17.000,  33.000)
MOVE ( 17.000,  33.000)-( 26.000,  33.000)
DRAW ( 26.000,  33.000)-( 27.000,  33.000)
DRAW ( 27.000,  33.000)-( 28.000,  33.000)
MOVE ( 28.000,  33.000)-( 41.000,  33.000)
DRAW ( 41.000,  33.000)-( 42.000,  33.000)
```

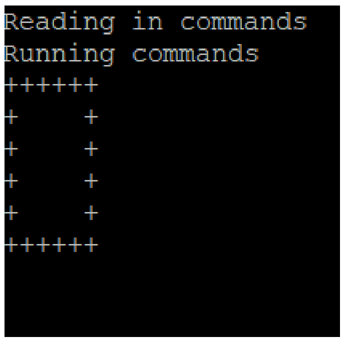
6.4.2 Example Output of Test File:

```
./TurtleGraphics test.txt
```

Test File Contents:

```
draw 5
rotate -90
draw 5
rotate -90
draw 5
rotate -90
draw 5
```

Test File Output:



```
Reading in commands
Running commands
++++++
+    +
+    +
+    +
+    +
++++++
```

graphics.log Contents:

```
DRAW (  0.000,   0.000)-(  5.000,   0.000)
DRAW (  5.000,   0.000)-(  5.000,   5.000)
DRAW (  5.000,   5.000)-(  0.000,   5.000)
DRAW (  0.000,   5.000)-( -0.000,   0.000)
```