Assignment Report

COMP1000

Unix and C Programming

20/10/2018

Student ID: 19126296

Student Name: Syed Faraz Abrar

TABLE OF CONTENTS

Assignment Description

The Semester 2, 2018 UCP assignment tasked students with designing, implementing (in C89), and debugging a terminal-based turtle graphics drawing program.

The idea behind the program is that the terminal is to be interpreted as a sort of graph, with the very top left corner of the terminal being the coordinates (0, 0). A turtle is made to start at these coordinates and can be moved around and rotated. The turtle also has the ability to draw lines from one set of coordinates to another. For the purposes of this assignment, the X axis is positive going left to right, while the Y axis is positive going top to bottom.

The program reads in instructions from an input file. The input file is set out such that each line has a command, followed by an argument for the command. A valid list of commands and arguments are as follows (Note: commands are not case-sensitive):

- MOVE *d*
- ROTATE θ
- DRAW *d*
- FG *i*
- BG *i*
- PATTERN *c*

MOVE takes in a real number *d* and moves the turtle by *d* units across the screen in the direction it is facing.
ROTATE takes in a real number θ and rotates the turtle counter clockwise by that θ degrees.
DRAW takes in a real number *d* and draws a line of *d* units across the screen in the direction the turtle is facing.
FG takes in an integer *i* and sets the foreground color to that value. Valid *i* values range from 0-15 inclusive.
BG takes in an integer *i* and sets the background color to that value. Valid *i* values range from 0-7 inclusive.
PATTERN takes in a character *c* and sets the pattern for the turtle to draw on its next DRAW command.

The program is also required to output each DRAW and MOVE command followed by starting coordinates and ending coordinates to a log file named “graphics.log”.

There should be three different versions of the program. The main version should be named TurtleGraphics and should do exactly as the assignment specifications state. The second version should be named TurtleGraphicsSimple, which should set the foreground color to black, and the background color to, and ignore every FG and BG command in the input file. It should otherwise do everything the assignment specifications state. The final version should be named TurtleGraphicsDebug, which should do everything the assignment specifications state, while also printing out the log file entries to the stderr output stream after each DRAW or MOVE command is executed.

More detailed information about the assignment can be found within the assignment specifications, available on blackboard.

Files and functions

Below is the list of files that were written for this program. Each file name is followed by a description of each function contained within that file.

helper.h, helper.c

The helper.c file contains four helper functions that are required in various places within the program.

int round(double): This function imports a real number and proceeds to round it up or down depending on whether the value after the decimal point of the number is greater than or less than 0.5, then returns the rounded number. I have implemented it very similarly to the implementation of the round() function in C99's math.c file.

void convertToUpper(char *): This function imports a *char **, which is a pointer to an array of characters. It modifies the imported string, changing each lowercase character into its corresponding uppercase version.

double degToRad(double): This function imports a real number, treats it as being an angle in degrees, then proceeds to convert it into radians. It then returns the angle in radians. The formula used for this conversion is: **angle (in radians) = angle (in degrees) * PI / (180 degrees)**

void printError(int): This function imports an integer, treats it as an error code, then proceeds to print out an error to the stderr output stream depending on the error code that is imported. If the error code imported is not defined, the function does nothing.

io.h, io.c

The io.c file contains three functions that deal with File I/O.

int parseLine(char *, Node **): This function imports a *char **, which is a pointer to an array of characters, as well as a *Node ***, which is a pointer to the pointer that points to the head node of a linked list. The function proceeds to open the file whose name matches the string pointed to by the imported *char **, and then reads each line, uses the ***parseCommand*** and ***parseArgs*** functions to check that the line is valid (i.e that the line has a valid command and a valid argument for that command) and then stores the line in the linked list, whose head is pointed to by the *Node *** that is imported in. It returns an integer that corresponds to an error code. It is used to determine exactly what error occurred while trying to read the file.

char parseCommand(char *): This function imports a *char **, which is a pointer to an array of characters. The char * is a single line from the input file. The function proceeds to tokenize this line and perform a non-case-sensitive check to ensure that the command found within the file (i.e the first token) is one of the six valid commands that the assignment specifications specify. It returns a character that corresponds to the first character of each of the commands to specify which command is in the line (i.e 'd' would be returned for the DRAW command). If an invalid command is found, it returns an 'e' for error.

int parseArgs(char*, char): This function imports a *char **, which is a pointer to an array of characters, as well as a *char*. The *char ** is a single line from the input file, and the *char* is the character returned by the ***parseCommand*** function. It tokenizes the line from the file and ensures that there are two tokens in the line, otherwise the line is invalid. It then uses the *char* that is passed into it to determine whether the argument that immediately follows the command (i.e the second token in the line) is a valid argument. For example, if the *char* is a 'p', the function checks and makes sure the second token corresponds to a single printable character, as 'p' corresponds to the command "PATTERN". The function itself returns the number of tokens that is in the line of input. If this is not equal to 2, then an error has occurred.

linked_list.h, linked_list.c

The linked_list.c file contains four functions that deal with inserting into and freeing the linked list. The linked_list.h file contains definitions for the linked list struct as well as for the node struct.

typedef struct Node: This is the node struct, used to create nodes that are stored in the linked list. The nodes themselves contain a char[] variable that stores a string, and a pointer to the next node in the list.

typedef struct LinkedList: This is the linked list struct. It contains a pointer to the head node of the linked list.

void insertLast(Node **, char *): This function imports a *Node* **, which corresponds to the head node of the linked list to insert into, as well as a *char **, which is a pointer to an array of characters. The function proceeds iterate through the linked list, starting at the head, until it reaches the very last node. It then creates and allocates memory for a new node on the heap, ensuring that the node's char[] variable stores the string that is imported in (i.e the imported *char **). It then sets the last node's next variable to point to this newly created node and sets the new node's next to NULL.

void freeList(Node **): This function imports a *Node ***, which corresponds to the head node of the linked list that is to be freed. The function then recursively iterates through every node until it reaches the end node, then it recursively frees each node until it has finished freeing every node.

plotter.h, plotter.c

The plotter.c file contains three functions, one of which is draw(), which handles the main loop of the program and handles all the moving, rotating, and drawing that the turtle does, as well as keeping track of where the turtle currently is on the screen as well as the direction it is facing. It also keeps track of the current foreground color, the current background color, and the current pattern. The remaining two functions are helper functions used by the draw() function.

void draw(Node **head): This function imports in a *Node ***, which corresponds to a pointer to the head node of the linked list that stores all the lines from the input file. Every line within this linked list has already been error checked. The function proceeds to initialize all the variables (as required by the assignment specifications), opens the log file for writing to, and then iterates through the linked list, executing the command in each line with its corresponding argument as well as logging all MOVE and DRAW commands in the log file. After it has finished executing every line from the linked list, it closes the file stream used for the log file and simply returns.

void move(double *, double *, double, double): This function imports two *double ** variables, which correspond to pointers to the **x** and **y** variables of the turtle's location. It also imports two *double* variables, which correspond to the distance d to move the turtle by, as well as the angle θ to determine the direction the turtle is facing. It then modifies **x** and **y** variables that are pointed to by the *double ** to the new **x** and **y** values that correspond to the turtle being moved by d units in the direction of the angle θ. It also ensures to change the sign of the change in the **y** value, as the **y** axis is flipped for this program (i.e **y** increases as you go down the terminal).

void plot(void *): This function imports a *void **, which is always a pointer to a single character. This function is used by the line() function provided to us from within the effects.h file. The function simply takes the imported *void ** variable, typecasts it to a *char ** and dereferences it to get the character that is stored within it, and prints it to the screen. Note that the imported variable will always be a pointer to a *char* as that is how the code has been implemented. The user has no control over what the import is.

Implementation Details

When I first looked at the assignment specifications, it occurred to me that there are many different ways to actually implement a coordinate system for the terminal. I initially looked through the line() function within the effects.c file that is provided to us, and realized that it actually handled all the implementation of the coordinate system for me. No matter what initial and final x and y values I provided to the line() function, it would move the cursor across the screen to the initial x and y values for me, and then proceed to draw a line to the final x and y values.

Knowing this, the way I chose to implement the coordinate system was as follows. I decided that all I really needed to do was keep track of the x and y variables that correspond to the turtle's location. Whenever a MOVE or ROTATE command is initiated, I'd simply change the turtle's x and y values or its direction θ by the amount that is specified. I wouldn't need to do anything else, as the DRAW commands would do all the work of placing the cursor in the correct spot and drawing the line for me through the use of the provided line() function.

If I were to think of an alternative approach to implementing a coordinate system, I would have used a 2D char array implementation. I'd have a set number of max rows (corresponding to the x axis) and a set number of max columns (corresponding to the y axis). Within this array, I'd initialize every value to a blank space (' '). I would then have to use a different implementation of the line() function, as the current one wouldn't work for this specific implementation of the coordinate system. This new line() function would take the array, go from the initial x,y indexes (i.e array[x][y]) to the final x,y indexes, changing the character stored within those specific indexes of the array with the pattern that is specified. Finally, I would print out the contents of the array with a for loop onto the terminal. This would achieve the same result as my current implementation.

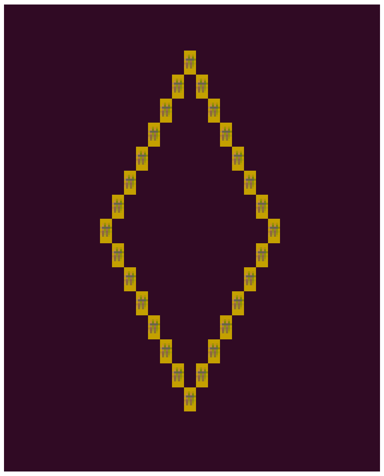
Test cases

Test case 1: The file is valid and draws a diamond shape on the terminal

Command used to execute the program: ./TurtleGraphics diamond.txt
User input provided by the terminal: N/A
Contents of the input file:

```
rotate -45
move 30
fg 8
bg 3
pattern #
draw 10
rotate 450
draw 10
rotate 90
draw 10
rotate 90
draw 10
```

Output to the screen:



Output to the log file:

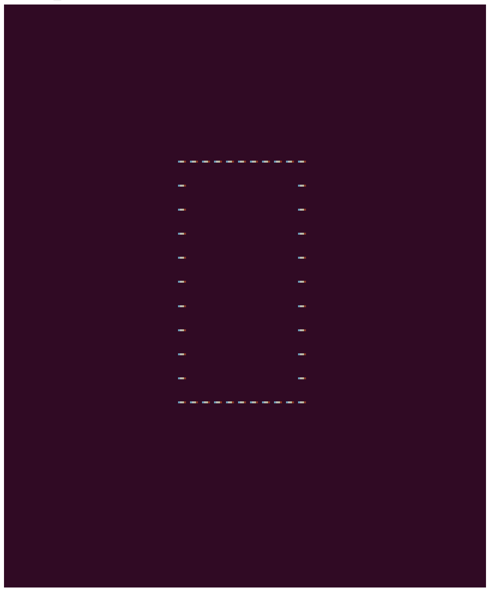
```
---
MOVE (   0.000,    0.000)-(  21.213,   21.213)
DRAW (  21.213,   21.213)-(  28.071,   28.071)
DRAW (  28.071,   28.071)-(  35.071,   20.929)
DRAW (  35.071,   20.929)-(  27.929,   13.929)
DRAW (  27.929,   13.929)-(  20.929,   21.071)
```

Test case 2: The file is valid and draws a rectangle shape on the terminal

Command used to execute the program: ./TurtleGraphics rectangle.txt
User input provided by the terminal: N/A
Contents of the input file:

```
rotate -45
Move 30
Pattern -
ROTATE 45
DRAW 10
rotate -90
draw 10
rotate -90
draw 10
rotate -90
draw 10
```

Output to the screen:



```
-----------
-         -
-         -
-         -
-         -
-         -
-         -
-         -
-         -
-         -
-----------
```

Output to the log file:

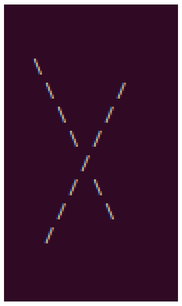
```
---
MOVE (   0.000,    0.000)-(  21.213,   21.213)
DRAW (  21.213,   21.213)-(  31.000,   21.000)
DRAW (  31.000,   21.000)-(  31.000,   31.000)
DRAW (  31.000,   31.000)-(  21.000,   31.000)
DRAW (  21.000,   31.000)-(  21.000,   21.000)
```

Test case 3: The file is valid and draws somewhat of an X symbol to test intersecting lines

Command used to execute the program: ./TurtleGraphics intersect.txt
User input provided by the terminal: N/A
Contents of the input file:

```
PATTERN \
ROTATE -45
MOVE 10
DRAW 10
ROTATE 135
MOVE 6
ROTATE -225
PATTERN /
DRAW 10
```

Output to the screen:



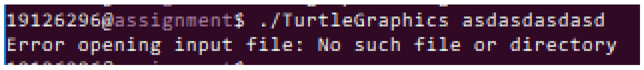
```
\
 \     /
  \   /
   \ /
    /
   / \
  /   \
 /
```

Output to the log file:

```
---
MOVE (   0.000,    0.000)-(   7.071,    7.071)
DRAW (   7.071,    7.071)-(  14.071,   14.071)
MOVE (  14.071,   14.071)-(  14.000,    8.000)
DRAW (  14.000,    8.000)-(   6.929,   15.071)
```

Test case 4: The file is missing

Command used to execute the program: ./TurtleGraphics asdasdasdasd
User input provided by the terminal: N/A
Contents of the input file: N/A (file does not exist)
Output to the screen:

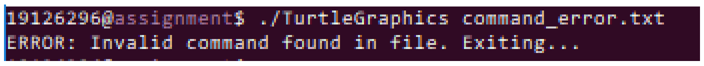


```
19126296@assignment$ ./TurtleGraphics asdasdasdasd
Error opening input file: No such file or directory
```

Output to the log file: N/A (file does not exist)

Test case 5: The file has an invalid command

Command used to execute the program: ./TurtleGraphics command_error.txt
User input provided by the terminal: N/A
Contents of the input file:
FG 1
BG 2
MoVEa 23
ROTATE -45
DRAW 10
Output to the screen:



```
19126296@assignment$ ./TurtleGraphics command_error.txt
ERROR: Invalid command found in file. Exiting...
```

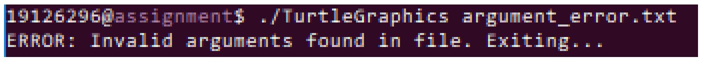
Output to the log file: N/A (file is invalid)

Test case 6: The file has an invalid argument

Command used to execute the program: ./TurtleGraphics argument_error.txt
User input provided to the terminal: N/A
Contents of the input file:

FG 1
BG 2
MOVE ad
DRAW 20

Output to the screen:



```
19126296@assignment$ ./TurtleGraphics argument_error.txt
ERROR: Invalid arguments found in file. Exiting...
```

Output to the log file: N/A (file is invalid)

<END OF REPORT>