Curtin College

UNIX and C Programming (UCP1002)

Semester 1, 2016

Assignment

Weight: 20% of the unit mark.

Your task for this assignment is to design, code (in C89), test and debug a program to draw terminal-based "turtle" graphics.

In short, your program will:

- Read a series of graphics commands from a file;
- Convert angles and distances to (x, y) coordinates as needed;
- Record these coordinates in a log (output) file;
- Use pre-defined functions to execute the commands (drawing lines on the screen).

There is a lot of detail here. Read it *carefully* before you start anything.

1 Documentation

You must thoroughly document your code using C comments (`/* ... */`).

For each function you define and each datatype you declare (e.g. using `struct` or `typedef`), place a comment immediately above it explaining its purpose, how it works, and how it relates to other functions and types. Collectively, these comments should explain your design. (They are worth substantial marks – see Section 10.)

2 Input file

Your program should accept a command-line parameter: the name of the file containing graphics commands. Your program should read the contents of this file into a linked list.

The file will consist of a sequence of commands, one on each line. There is no length indicator. Each command may be one of the following:

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

θ is a real-valued angle, d is a real-valued distance, i is an integer-valued colour code and c is a non-space character. Command names may be in lowercase or uppercase (or any combination).

An example file might look like this:

```
rotate -45
move 30
FG 1
Pattern #
DRAW 10
Rotate 90
draw 10
ROTATE 90
dRAW 10
ROTATE 90
DRAW 10
```

(I have deliberately scrambled the casing here; this is still a valid input file.)

3 Turtle graphics

Your program must keep track of several changing pieces of information:

Current position: for each command, you start at a particular position on the screen, described by real-valued (x, y) coordinates, where x begins at zero on the left of the screen, and y begins at zero at the top, increasing downwards. The `MOVE` and `DRAW` commands will change the current position. The initial position is $(0, 0)$ (the top-left screen corner).

Current angle: for each command, you start at a particular real-valued angle, expressed in degrees, from 0° to 360°. 0° means "right"; 90° means "up". The `ROTATE` command will add or subtract from the current angle; i.e. rotating it anticlockwise or clockwise. The initial angle is 0°.

Current foreground and background colours: for each command, you start with two colour codes representing the foreground and background colours. The terminal has 16 foreground colours (0–15) and 8 background colours (0–7). These are used every time any character is displayed. The `FG` command changes the foreground colour, and `BG` the background. The initial foreground colour is 7 (white) while the initial background colour is 0 (black).

Current pattern: for each command, you start with a character that will be used to make up any lines you plot. The `PATTERN` command changes the current pattern. The initial pattern is the '+' character.

Your program must interpret and execute the sequence of commands, keeping track of the above information. Most importantly, the `DRAW` d command will draw a line d

units long, starting at the current position and heading at the current angle. After the line is drawn, the current position will move to the other end of the line. The `MOVE` command works in the same fashion, except it does not actually draw a line.

To do this, your program must calculate real-valued (x, y) coordinates from angles and distances. To this end, you will need to apply basic trigonometry (using `math.h`):

$$\Delta x = d \cos \theta$$
$$\Delta y = d \sin \theta$$

When outputting lines, your program must round its real-valued coordinates to the nearest integer values, as required below.

4 Existing functions

You must use the existing functions in the files `effects.c` and `effects.h`. The most important of these is `line()`, declared as follows:

```
void line(int x1, int y1, int x2, int y2,
          PlotFunc plotter, void *plotData);
```

This will draw a line from integer-valued coordinates $(\texttt{x1}, \texttt{y1})$ to $(\texttt{x2}, \texttt{y2})$. It has no idea of the "current position", "current angle", etc.

The `PlotFunc` type is defined as follows:

```
typedef void (*PlotFunc)(void *plotData);
```

For each discrete point on the line, the `line()` function will position the cursor at that point, then call the `*plotter` function. You must define this function yourself, and supply it to `line()`. The plotter function must generate the appropriate terminal output (i.e. the current pattern).

The final parameter to `line()` is a `void*`, which is simply passed on to the plotter function. This allows you to pass any required data between your algorithm and the plotter function.

Several other functions are also available:

- `void setFgColour(int)` — changes the foreground colour to a given colour code.
- `void setBgColour(int)` — changes the background colour to a given colour code.
- `void clearScreen()` — blanks the terminal.
- `void penDown()` — moves the cursor to the bottom of the screen (for when drawing is complete).

5 Log file (Consider this when developing test cases!)

Your program must also *append* to (not overwrite) a log file called `graphics.log`.

The first line written to the log file, after its existing contents, should be "`---`". This will serve to separate the new contents from the old.

The log file must show the actual, real-valued (x,y) coordinates calculated for the `DRAW` and `MOVE` commands, in the following format: "`command (`x_1`,`y_1`)-(`x_2`,`y_2`)`". All coordinate values should be printed such that they line up vertically.

An example `graphics.log` file (generated after only one execution) would be:

```
---
MOVE (  0.000,  0.000)-( 50.000,  33.333)
DRAW ( 50.000, 33.333)-(100.000,  50.000)
DRAW (100.000, 50.000)-( 50.000,  66.667)
DRAW ( 50.000, 66.667)-( 50.000,  33.333)
```

(This is not related to the previous example.)

If the program were run a second time with the same input, `graphics.log` would become:

```
---
MOVE (  0.000,  0.000)-( 50.000,  33.333)
DRAW ( 50.000, 33.333)-(100.000,  50.000)
DRAW (100.000, 50.000)-( 50.000,  66.667)
DRAW ( 50.000, 66.667)-( 50.000,  33.333)
---
MOVE (  0.000,  0.000)-( 50.000,  33.333)
DRAW ( 50.000, 33.333)-(100.000,  50.000)
DRAW (100.000, 50.000)-( 50.000,  66.667)
DRAW ( 50.000, 66.667)-( 50.000,  33.333)
```

The second half of the log file would be added on the second execution.

6 Makefile (or "How to Actually Get Marks, part 1")

You must create a makefile, and it must actually work. That is, the marker will type:

```
[user@pc]$ make
```

This is the only way the marker will attempt to compile your code. If your makefile does not work, then, according to the marking guide, *your code does not compile or run*. (The marker will delete all existing .o files and executable files beforehand.)

Your Makefile must be written by you, and *not* automatically generated. It must be structured properly in accordance with the lecture notes.

7 Testing (or "How to Actually Get Marks, part 2")

To begin with, construct a small-scale test: an input file with only commands. You should manually determine what the output should be, so you know if your code is doing the right thing. Ensure you test *all* the different kinds of commands. Ensure also that you do, eventually, test a reasonably complex test case.

Use `valgrind` early and often! You *will lose marks* for failing to fix issues reported by `valgrind`, and your program could crash unpredictably anyway.

8 README.txt

Prepare a text file called `README.txt` (*not* using Word or any other word processor), that contains the following:

- A list of all files you're submitting and their purpose (if it's not obvious).
- A statement of how much of the assignment you completed; specifically:
  - How much of the required functionality you attempted to get working, and
  - How much *actually does* work, to the best of your knowledge.
- A list of issues reported by `valgrind`, if any, along with any other bugs or defects you know about, if any.
- A statement of which computer you tested your code on.

9 Submission

You must submit the following electronically inside a single .zip or .tar.gz file:

- Your `makefile` and `README.txt`.
- All .c and .h files (everything needed for the `make` command to work).
- A completed Declaration of Originality (whether scanned, photographed, or filled-in electronically).

10 Mark Allocation

Here is a rough breakdown of how marks will be awarded. The percentages are of the total possible assignment mark:

10% – Using code comments, you have provided good, meaningful explanations of all the files, functions and data structures needed for your implementation.

10% – Your have followed good coding practices, and your code is well-structured, including being separated into various, appropriate .c and .h files.

20% – You have correctly implementated the required functionality, according to a visual inspection of your code by the marker.

20% – Your program compiles, runs and performs the required tasks. The marker will use test data, representative of all likely scenarios, to verify this. You will not have access to the marker's test data yourself.

You may lose this entire component of your mark by either (a) not having a working makefile, OR (b) failing to solve issues raised by `valgrind`.

40% – For the all practical signoffs put together (i.e. 5% per prac signoff).

11 Academic Misconduct – Plagiarism and Collusion

If you accept or copy code (or other material) from other people, websites, etc. and submit it, **you are guilty of plagiarism**, unless you correctly cite your source(s). Even if you extensively modify their code, it is still plagiarism.

Exchanging assignment solutions, or parts thereof, with other students is **collusion**.

Engaging in such activities may lead to serious penalties.

You are expected to understand this at all times, across all your university studies, *with or without* warnings like this.

End of Assignment