School of Computer Science, McGill University

COMP-206 Introduction to Software Systems, Fall 2022

Mini Assignment 4: Intro to C and Arrays

Due Date November 16, 18:00 EST

This is an individual assignment. You need to solve these questions on your own.

You MUST use `mimi.cs.mcgill.ca` to create the solution to this assignment. An important objective of the course is to make students practice working completely on a remote system. Therefore, you must not use your Mac command-line, Windows command-line, nor a Linux distro installed locally on your laptop. You can access `mimi.cs.mcgill.ca` from your personal computer using **ssh** or **putty** as seen in class and in Lab A and use one of the command line editors (such as `vi`) available in `mimi`. **If we find evidence that you have been instead using your laptop, etc., to do some parts of your assignment work, you might loose ALL of the assignment points.** This restriction applies only to the editing (coding) and testing of your programs. You are free to (and encouraged to) backup your source code regularly to your personal computer as a precaution against any misadventures. Delays incurred because you accidentally deleted your programs and you had not taken backups will not get any considerations.

All of your solutions should be composed of source code and commands that are executable in `mimi.cs.mcgill.ca` and must be contained within the scripts that you submit. I.e., anybody else (such as your TA) should be able to just compile/run these scripts in mimi on their own without any modifications or additional setup (except giving execute permissions for scripts) and it should still work the same.

For this assignment, you will have to turn in one C source code and one shell script. Instructors/TAs upon their discretion may ask you to demonstrate/explain your solution. No points are awarded for commands that do not execute at all. (Commands that execute, but provide incorrect behavior/output will be given partial marks.) All questions are graded proportionally. This means that if 40% of the question is correct, you will receive 40% of the grade.**TAs WILL NOT modify your program or scripts in anyways to make it work.**

Please read through the entire assignment before you start working on it. <u>You can loose up to 3 points for not following the instructions</u> in addition to the points lost per questions.

Lab G and H provide some background help for this mini assignment.

Total Points: 20

Ex. 1 — Basic ASCII Drawing (16 Points)

Ever wondered how fancy GUI programs draw complex shapes and design patterns? There are specific libraries that are optimized by graphics experts that allows application programmers to directly draw such patterns without having to worry about the complex algorithms that work at pixel level.

In this assignment, we will do a sneak peek preview of the effort and complexity involved in building such libraries. However, in the interest of simplicity, we will stick with basic algorithms that you learned in your High School MATH Geometry classes. We will also treat a "character position" as an approximation of the pixel concept. As such, we are not going to get very beautiful looking art or elegant shapes, but nevertheless some interesting figures could be drawn using these concepts. In the below discussions, a "pixel" basically represents an entire "character position" in the screen.

You will be writing a C program `asciidraw.c` that would process some basic drawing commands to draw simple geometric shapes. It should be compilable exactly as in either :

```
$ gcc -o asciidraw asciidraw.c
```

or

```
$ gcc -o asciidraw asciidraw.c  -lm
```

Anything else, and the points for this exercise will be 0.
The program itself would be invoked the following manner

```
$ ./asciidraw
```

All commands are then typed into the standard input of the program. The program loops, reading these commands one after the other until it finds an `END` command.

```
END
```

At which point it will terminate. (We do not care about the termination code of the program).

1. **(1 Point)** *Setting a grid.* You can assume that our "drawing canvas" has a maximum size of 1000 x 1000 pixels. I.e., we are looking at a 1000 x 1000 character array where your program will "draw" shapes into (i.e., you are not drawing them directly to the screen, but storing the shapes internally). However, users often want to keep their "drawing area" smaller than the canvas and as a result would specifiy a grid size, as in.

   ```
   $ ./asciidraw
   GRID 40 30
   ```

   Which implies a drawing grid that is 40 pixels (characters) wide and 30 pixels tall. This will impact all drawing commands for geometric shapes, which we discuss below. Basically, you **should not draw** anything outside of the grid area into the canvas (but they can get truncated - i.e., ignoring the part of the shape that fell outside of the grid). Grid may be as big as the canvas itself.
   If the user has not specified a `GRID`, the program will refuse to process any commands whether they are valid / invalid (with one exception that is discussed later) till it sees a `GRID` command. The following message will be issued.

   ```
   Only CHAR command is allowed before the grid size is set.
   ```

   And will continue to accept new commands.
   If you try to set the `GRID` again at any later point in the program, after it was already set before, the program will refuse to do that

   ```
   GRID 20 20
   GRID was already set to 40,30.
   ```

   and will continue to accept new commands.

2. **(1 Point)** *Setting a drawing character.* By default, all shapes are drawn using the `*` character. However, you can change the drawing character by using the `CHAR` command. All shapes drawn after that command will then use this character (till the next `CHAR` command is encountered). This is also the ONLY command that could get executed before the grid has been set (as it is not impacted by the dimensions of the drawing area).

   ```
   CHAR +
   ```

   Here `+` becomes the character used in drawing for any future drawing commands. **Remember to use the default character** if no explicit character is set. If your program crashes because it is not able to draw a shape before a character is explicitly set, you would lose points for those test cases.

3. *Displaying the figure.* By using the `DISPLAY` command, the user can ask the program to show what the figure currently looks like. Please remember that it should ONLY produce the contents of the drawing area (as defined by the `GRID` command) on the output. Keep in mind that `DISPLAY` can be called multiple times (including before any shape was actually drawn). We primarily rely on this command to see your various other drawing commands are working. So irrespective of how accurately you did the source code logic for them, if the display command does not show the correct shapes, you will not get any points for them.
   The grid should also be bordered with numbering at the bottom (x-axis) and left (y-axis). With $0, 0$ being the left,bottom corner "pixel". This will help you verify if you are drawing the shapes in the right place, etc. Significant deductions / penalties will incur if this feature is not working correctly as it will hamper the TA's ability to judge if rest of your algorithms are working correctly (also see the section on additional restrictions).

4. **(2 Points)** *Drawing a Rectangle.* The `RECTANGLE` command can be used to draw rectangles bound by two sets of coordinates (x_1, y_1) and (x_2, y_2) as follows.
   Where (x_1, y_1) is the top left corner and (x_2, y_2) is the bottom right corner of the rectangle. (It will always be passed in that order).

```
RECTANGLE 4,15 20,1
```

You program should then "draw" this rectangle into the canvas, as restricted by the grid.

5. **(3 Points)** *Drawing a line.* The `LINE` command can be used to draw lines between two coordinates (x_1, y_1) and (x_2, y_2) as follows.

```
LINE 4,5 30,10
```

Your program should then "draw" this line into the canvas, as restricted by the grid. **Remember that the line drawing should work, irrespective of the order in which coordinates are provided.** i.e., the following is also valid and should result in the exact same line.

```
LINE 30,10 4,5
```

6. **(5 Points)** *Drawing a circle.* This command is used to draw a circle, with the center (x, y) and radius r as follows.

```
CIRCLE 10,20 6
```

Your program should then "draw" this circle into the canvas, as restricted by the grid. Now, because we are working with an integer plane, it is not possible to make the "origin" or center of the circle in-between two pixels. As such, for a circle of radius r, we will assume that the diameter is $2r + 1$. Valid values for r will be $r \geq 1$.

7. **(2 Points)** The program is of course expected to be able to draw complex figures that contains multiple drawing commands. Shapes can be overwritten over the previous shape that was drawn into the canvas.

```
GRID 70 30
LINE 10,10 40,20
RECTANGLE 40,30 45,25
CIRCLE 30,15 5
```

8. **(3 Points)** *Truncating figures.* The parameters to the drawing commands (grid dimensions, coordinates, radius, etc.) will always be valid values. Dimensions and radius will be above 0 and coordinates would be 0 and above). However, this does not mean that they will be contained (partially or completely) within the grid or even the entire canvas. In those circumstances, you must draw only the part that is contained within the grid (aka the drawing area) - truncating the figure.
For example, the below commands are valid combination.

```
GRID 70 30
LINE 10,10 80,20
RECTANGLE 4000,3000 4500,2500
CIRCLE 10,10 30
```

9. **(1 Point)** If the program encounters a command that is not valid, it should display an error message and continue to accept new commands.

```
CIRCA 10,10 30
Error did not understand CIRCA 10,10 30
```

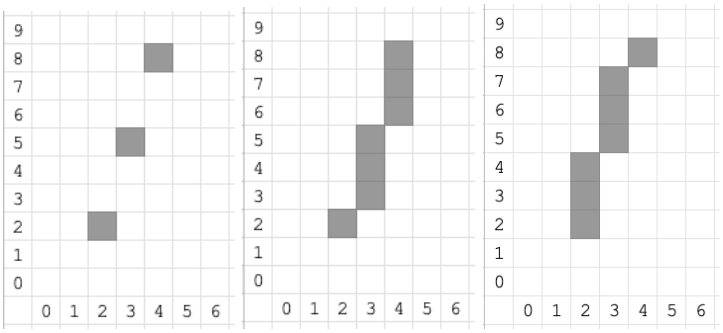
Ex. 2 — Making your own tester script (4 Points)

As you would notice when you are developing your code, repeatedly running the same test cases will be a tiring and error-prone effort. To make your developer life easy, you will write a test script, **drawtester.bash**. You can start by making a copy of the template given in **mini4_tester_template.sh** and modifying it suit your needs (remember to not use the template itself but the copy you created). I encourage you start writing this tester as you are developing your C program, to make it worth your time. Remember, you can get points for your tester script even if your C program does not work correctly.
Your tester script SHOULD NOT contain any commands that are not present in the template. And specifically, it must not contain any additional commands that modify the file system (i.e., `rm`, `mv`, `cp`, `mkdir`, etc ...). Violating this would result in a **2 points** penalty!

Your test script should contain at least 10 test cases, that test different aspects of the program. Make sure that your tester script prints as to what it is trying to "test", the commands it is issuing to the program, etc. - see the example template. In general, a person who knows the commands and knows what it is that the program should have drawn for it, must be able to look at the output of the tester and figure out whether the program is doing this correctly or not.
You are free to use some of the test cases given to you. But DO NOT skimp on thinking about other situations and writing your own test cases - remember, the TAs are going to come up with their own test cases.

ADDITIONAL RESTRICTIONS

- You must write a reasonable amount of comments (to understand the logic) in your C source code and shell script. You can lose up to **2 points** for not writing comments.
- If your C program crashes for ANY test case, it will result in a **3 points** deduction over and above whatever is already lost for individual parts.
- Your program **MUST NOT get into an infinite-loop sort of situation, hang, etc**. This would result in forfeiting any un-evaluated test cases from that point onwards in the TA test script. To give you an intuition, the template script test cases run under 0.3 seconds for a naive solution and the full TA tester runs under 0.7 seconds. If your program takes more than a minute, to finish all the test cases, TA will just stop it at that point.
- Your C program should be a single source code (file) and your tester also should be a single script (file), and not dependent on your personal directory structures, your own personal scripts, etc. I.e., the TA should be able to download your files, compile and run them with no other setup or modifications. Failure to comply with this can potentially result in a 0.
- Compilation of your C program SHOULD NOT produce any WARNINGS! This will result in a **2 points** deduction.
- The `DISPLAY` command **should only draw the canvas that is part of the display grid and NOT the entire canvas**. Dumping the entire canvas array into the output will automatically set your assignment to 50% penalty.
- **Make sure the grid numbers are present along both the axis** in the `DISPLAY` output. Not doing this will also result in 50% penalty. It is difficult to validate correctness of shapes without this. See given examples on what this looks like.
- The lines and rectangles you draw should definitely pass through the specific coordinates given in the command.
- For a circle of radius r and center (x, y), make sure that it passes through the following coordinates $(x - r, y)$, $(x + r, y)$, $(x, y + r)$, and $(x, y - r)$.
- There will be **1 point** deduction per type of shape, if there are cases for that shape where the above two rules are not maintained. There could be additional deductions if the shape does not make sense at all (e.g. circle looks like a rectangle, etc. because you used an inferior algorithm to what was given to you in hints.).
- **Your shape MUST NOT be "disconnected"**. I.e, make sure that "pixels" are "connected" to each other. Two pixels are considered connected if they are from immediate character positions (vertically/horizontally/-diagonally). Even if you have one test case where a shape is disconnected, the total points associated with that shape **will be reduced to half**.



Above, there are three examples of drawing lines from $(3, 2)$ to $(5, 8)$. The one on the left is not acceptable as it is just disconnected dots. Either of the other two are fine. The same rule is applicable to other shapes as well.

- DO NOT Edit/Save files in your local laptop, not even editing comments in the scripts. This can interfere with the file format and it might not run on `mimi` when TAs try to execute them. This will result in a 0. No Exemptions !!. TAs will not modify your script to make it work.

TESTING

There is no official testing script. Please use the template given to you and write your own testing script. Remember, TAs will use their own tester script (very similar in syntax to the template) to test your C program. They will separately verify your tester script for its capabilities.

You can look at `examples.txt` to understand what the program output is supposed to look like for some of the commands. This file was produced using the `script` command that we saw in the first half of the semester to record an interactive session with the program.

WHAT TO HAND IN

Upload your C program `asciidraw.c` and script `drawtester.bash` to MyCourses under the **mini 4** folder. Do not zip the file. Re-submissions are allowed. DO NOT turn in the executable. TAs will compile on their own.

You are responsible to ensure that you have uploaded the correct file to the correct assignment/course. There are no exemptions. **If you think it is not worth spending 5 minutes of your time to ensure that your submission that is worth 10% of your grade is correct, we won't either. NO Emailing of submissions**. If it is not in MyCourses in the correct submission folder, it does not get graded. **Because you do not know how to resubmit an assignment or run out of time to figure it out is not an excuse for emailing the assignment**. Imagine what will happen if all of you emailed your assignment instead of submitting it in MyCourses!

LATE SUBMISSIONS

Late penalty is -20% per day. Even if you are late only by a few minutes it will be rounded up to a day. Maximum of 2 late days are allowed. Do not email me asking for extensions because you have other midterms, assignments, etc.

Nevertheless, any requests made less than 48 hours from the deadline will be automatically denied. It would have been too late to start your work anyways. Extensions are given only for extenuating circumstances.

Late submission penalties are applied to the entire submissions and not just individual components.

ASSUMPTIONS

- You can assume that the GRID size will NEVER be larger than the canvas.
- You can assume that the GRID dimensions will NEVER be 0.
- For any valid commands to your drawing program, you can assume that its parameters (for those that require them) will be always correct.
- While there will be no "point" circles, it is very much valid to have a line or a rectangle that is only 1 pixel.
- You do not have to support case-insensitive processing for commands.
- All numbers used as input will fit in the `int` data type of C.
- You can assume that any command and its parameters typed in at the input will fit within a 100 character array.
- Due to possible differences in "rounding" logic, your shapes may not exactly overlap on the pixels of the example output given. That is fine as long as the other explicitly set rules in the assignment (such as given in the restrictions) are not violated while drawing the shape. I just want you to get the general logic correct. Of course I expect you to be able to draw shapes with simple straight lines (e.g. horizontal, vertical) correctly where there should be no rounding issues.

HINTS

It is highly recommended that you sit with a graph paper and work with it to figure out how your shape drawing algorithms "proceed" in order to understand the logic you need to develop and/or debug them. Do not have any graph paper handy? No worries, we have provided a printable pdf for graph paper along with this assignment.

Lines

Remember the equation for line, $y = mx + c$? Use that as your guide. Find out the value of m and c from your given sets of coordinates. Then you can either use x_1 through x_2 to find corresponding values of y or vice versa. Don't forget, your logic should work irrespective of $x_1 > x_2$ or $x_1 < x_2$ or $x_1 == x_2$, etc. Think about how to handle vertical or horizontal lines as well. You may have to do some sensible rounding to get "integer" values (i.e., the approximate pixel location in the grid).

You will need to put some extra thought on top of this simple logic in your "algorithm". Otherwise you will end up with a line made of disconnected dots, which is not acceptable. Can you think of a way in which you can "connect" these dots?

We are working on a discrete, integer plane instead of continuous plane. So your lines are going to look like "stair cases" or "double lines", etc (because of approximation/rounding)., and is not going to be the prettiest of lines. That is fine. This is a very simple logic to implement. Can you make it look any better?

Rectangles

This is more or less straight forward. I mean at the end of the day, they are but 4 lines.

Circles

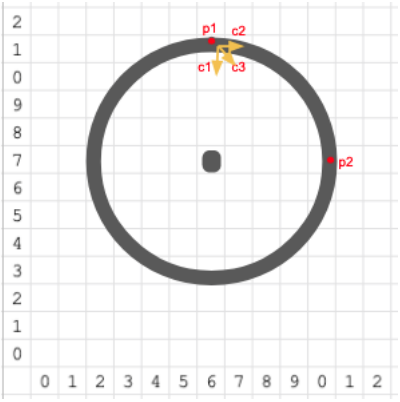
While this may look intimidating, let us tackle the problem one step at a time. Look at the figure below where we have drawn a contiguous circle over the grid. However, as we are working with a discrete, integer grid, we cannot draw a contiguous circle like this, and instead have to approximate.

To begin with, we will focus on the First Quadrant of the circle for now. Given the center (x, y) and radius r, we already know that the circle's outline in the first quadrant will begin with $p_1 = (x, y + r)$ on the top point and end at $p_2 = (x + r, y)$ on the far right.

Now that we have our starting point, which is p_1, we need to figure out what is the next point (cell in the graph) that we should "hop" onto. If we look at the circle, we can see that as we "travel" from p_1 to p_2, the only valid directions would be points that fall below or to the right to the current point. In our current example, these are c_1, c_2 or c_3. So we need to figure out which of these cells we should "hop" onto next. How do we do that?

Remember, the idea of a point p being on the circle is that, it means that p is at a distance of r from the center of the circle. So what we have to do is figure out what is the distance of c_1, etc., from the center of the given circle (Pythagorean theorem). Let us call them r_1, etc. Since we are approximating the circle due to our plane being integer based, we will pick the point c_i such that it has the r_i that is the "closest" to r.

Congratulations!, we have found the next point to draw for our circle. Now standing on this point, we will look at its neighbours to see which one we should hop onto (exact same logic). Once we have reached p_2, we have finished drawing the first quadrant!



I highly recommend that you first fix your logic to draw this one quadrant correctly before venturing out drawing the full circle.

So how do you draw the remaining points of the circle in the other three quadrants? Look at the diagram, they are surprisingly simple and involves very little math. They are some form of "mirror images" of the points in the first quadrant and therefore, you can calculate them directly without repeating all the above logic.

How to "truncate"

A naive, but effective logic would be, you figure out in your shape drawing algorithm as to what is the next coordinate (x, y) of the "point" you should be drawing if it was an infinite grid/canvas. Once that information is computed, check if that point will fall within the "grid". If not, skip updating the canvas. Continue with the logic to find the next point to draw. But can you improve on this logic?

COMMANDS ALLOWED

You may not use any commands / functionality that are not listed here or explicitly mentioned in the assignment description. **Using commands not allowed will result in 2 points deduction per each violation.**

asciidraw.c

- All of your logic should be within a SINGLE function, `main`. You may not write any functions of your own.
- General C program features variables, arrays, control statements, etc., are fair game to use.
- From the C libraries, you may use functions from `stdio.h`, `string.h`, and `math.h`. NO OTHER libraries are allowed.

drawtester.bash

For your tester script, you may not use ANY command/concept that is already NOT present in the template given to you. And I most definitely do not want to see anything with respect to making files, directories, removing them, etc. They are totally not required for this assignment.

QUESTIONS?

If you have questions, post them on the Ed discussion board and tag it under mini 4, but do not post major parts of the assignment code. Though small parts of code are acceptable, we do not want you sharing your solutions (or large parts of them) on the discussion board. If your question cannot be answered without sharing significant amounts of code, please make a private question on the discussion board or utilize TA/Instructors office hours.

Please remember that TA support is limited to giving any necessary clarification about the nature of the question or providing general advice on how to go about identifying the problem in your code. You are expected to know how to develop a high level logic, look up some syntax/options and most importantly, debug some aspects of your own code. We are not testing your TA's programming skills, but yours. Do not go to office hours to get your assignment "done" by the TAs.

Emailing TAs and Instructors for assignment clarifications, etc., is not allowed. TAs and instructors may convert private posts to public if they are not personal in nature and the broader student community can benefit from the information (to avoid repeat questions). Also check the pinned post "Mini 4 General Clarifications" before you post a new question. If it is already discussed there, it will not get a response. You can email your TA only if you need clarification on the assignment grade feedback that you received.

The ART Gallery!

Need a bit more challenge? See if you can draw interesting figures using the commands that you have implemented.

Make a copy of your `asciidraw.c` to `asciidraw_x.c`

Extend your commands (`RECTANGLE` and `CIRCLE`) to implement solid shapes. It is a lot easier than you think!

```
RECTANGLEFILL 10,10 30,30
CIRCLEFILL 20,20 10
```

Would you like to implement colors? (HINT:- You might need a 3D array, where the 3rd dimension can be used to store some encoding to help you decide what colors you should implement). You can also use separate 2D arrays to keep track of colors - more than one way to implement this.

This page provides information about the control-characters required to get colors. The page is specifically for java, but it is the same color scheme for all console-based programs.

Implement additional commands such as

```
COLOR FOREGROUND RED
COLOR BACKGROUND GREEN
```

etc.

Show off your artistic skills! Post your art work in Ed, under the pinned thread "Mini 4 ART Gallery". Vote (heart icon) the arts you liked most from your classmates.

Remember **DO NOT** turn in the `asciidraw_x.c` for your assignment submission. Only your original program `asciidraw.c`. Be careful not to modify your original assignment work and only play around with a copy.