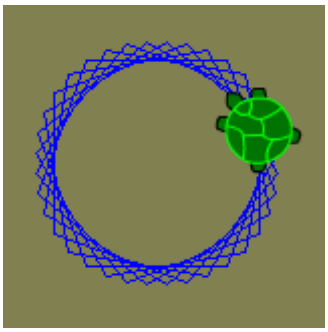
ECE220: Computer Systems & Programming **Fall 2022**
Machine Problem 12 **due: Wednesday 4 January (2023) at 11:59:59 p.m.**

Dancing Turtles



Your task this week is to write a short program to animate a turtle. Towards this end, you must open and read commands from an input file, then execute the commands frame by frame to animate the turtle. You will use the **`list`** template class (a doubly-linked list) from the Standard Template Library (STL) to keep track of line segments that must be drawn to the screen.

The objective for this week is to give you experience writing I/O code and an introduction to event-driven animation and the STL. Although your code will be in C++, you need use only a small amount of new syntax: method invocations for the **`TurtleScreen`** object and your list of line segments, as well as declaration and use of an iterator (but an example is given in the code).

Background

The screenshot above shows the turtle replaying an infinite loop of a few commands. As the turtle moves, it leaves trails in a color specified by the commands (it starts in white).

Ever hear of Logo? Check out http://www.calormen.com/jslogo/ for a more interesting version. We won't ask you to do more than implement a few simple commands (which aren't actually taken from that language—we just borrowed the turtle!).

Pieces

The code given to you includes a fairly substantial library as well as copies of header files and libraries for a graphics layer. You need read none of that code.

Note that you need only read and modify files in the **`jni`** subdirectory.

Your program will consist of a total of three files:

`mp12.h` This header file provides type definitions, function declarations, and brief descriptions of the subroutines that you must write for this assignment. **You should read through the file before you begin coding.**

`mp12.cpp` The source file for your code. A version has been provided to you with placeholders for the subroutines described in this document.

`TurtleScreen.h` Functions for your use in animating the turtle. Look at the list of methods starting around the middle of the file. Use them with the **`TurtleScreen*`** passed into your **`frameUpdate`** function.

A number of other files are also provided to you in the **`jni`** subdirectory:

`Makefile` A file that simplifies the building and visualization process. See the section below on Compiling and Executing Your Program.

`mp5.c` An empty file provided as a placeholder for your solution to MP5.

You can safely ignore the rest of the files, although, as always, you're welcome to look at them. Be warned that I stripped the old library down a little since the cross-development options have broken down in the last few years.

You should copy your **mp5.c** code into the **mp12/jni** directory over the placeholder provided to you. Please note that the **mp5.h** file is slightly different; do NOT replace it with the old one! The **draw_dot** and **set_color** routines are not the same as they were in MP5, but they work in the same way, so your code should be fine. We will not be checking the detailed image output, and only your **draw_line** function will be needed, so as long as that function is reasonably correct, the output should be acceptable.

The Task

The total amount of code needed in my version of this assignment was just over 100 extra lines.

You should first copy your **mp5.c** solution into your **mp12** directory.

You need to write five subroutines, of which two are likely to be more challenging that the others.

I suggest the following order:

1. Start with the following

   ```
   int32_t openInputStream (const char* fname);
   ```

   Open the file with name given by **fname** and store the input stream into the variable **input** (see **mp12.cpp**). Return 1 if successful, or 0 on failure.

2. Next, handle closing the stream:

   ```
   void closeInputStream (void);
   ```

   Close the input stream **input** (again, see **mp12.cpp**—it's a variable in that file).

3. Now write a function to use the MP5 functionality:

   ```
   void drawEverything (void);
   ```

   You should read about the STL list class template before you implement this function. An example of iterating over the list **lines** (a variable in **mp12.cpp**) is shown in the provided function **showLines** (just above the **drawEverything** function), which you can also use later to look at the list in GDB (using STL inside GDB can be painful otherwise).

At this point, you can build the program and run a couple of tests. If you move or remove the commands file from the top-level directory, your **openInputStream** should fail, causing the program to terminate with an error message. You can also check for an empty lines list (**lines.empty ()**) at the start of **drawEverything** and add a line or two by hand to check whether your drawing code works.

4. The last two routines work together to handle reading commands and animating the turtle. You may want to go back and forth, implementing reading the command and executing the animation for the command, then testing before moving on to another command.

   Command lines of 200 characters should be supported (use **fgets**). Commands are not case sensitive (use **strcasecmp**), nor does leading/trailing/extra space matter. Incorrect argument types and trailing non-space characters on a command's line are not allowed and should result in an error message printing the offending command to **stderr**.

Commands include the following:

color <RGB>	Set the color of lines drawn by the turtle. **<RGB>** is a hex value (read it as an **int32_t** in hex).
move <dist> <frames>	Move **<dist>** pixels in the direction that the turtle is currently facing. Use **sin** and **cos** to calculate the final position, be sure to round (use **round**), and write the move command into **cmd**. The animation should require **<frames>** calls to **frameUpdate** to finish (smaller numbers are faster movement). Movement commands with non-positive distances or numbers of frames must be ignored (no error message should print, though).
restart	Go back to the beginning of the file and read it again. Use the **rewind** function on the input stream to implement this command.
turn <amt>	Turn the turtle by 10 degrees for each of the next **|<amt>|** frames. Positive values turn left, and negative values turn right. Ignore turn commands with amount 0.
wait <frames>	Wait for **<frames>** frames (pause). Non-positive wait commands must be ignored.

The frame update function for animation is

```
void frameUpdate (TurtleScreen* ts);
```

The function is called 25 times per second. A command variable **cmd** in **mp12.cpp** tells the update routine what to do in each frame. If the current command is **CMD_NONE**, the function should call **readNewCommand** to see whether a new command is available in the input stream. Then the function should switch based on the type of command (see the enumeration in **mp12.h** for the exact names). Turns and waiting are relatively easy. Movement requires creating or updating a line segment in the list of line segments being drawn. Be sure to call **makeVisibleChange** on the **TurtleScreen** to show changes to the set of lines, as otherwise your changes will not be shown up. Also be sure to change the command type to **CMD_NONE** when appropriate to initiate reading a new command from the input stream on the next call to **frameUpdate**.

Keep in mind that you must **read the TurtleScreen.h header file** to find the methods that you must use for executing the commands.

The function for reading new commands is

```
void readNewCommand (TurtleScreen* ts);
```

The function should read commands and execute them immediately, if possible (such as restart and color commands, as well as bad commands), stopping only after it either reaches the end of the file (leaving the command as **CMD_NONE**) or reads a command that requires animation (moves, turns, and waits).

To check for trailing garbage using **sscanf**, read an extra string. For example, if the string **buf** is supposed to contain an integer, scan **buf** using format string **"%d%1s"** and provide a 2-character array for the trailing garbage. In this example, **sscanf** will return 1 if no trailing non-white-space characters exist, or 2 if they do.

Note: For library functions mentioned, please check their usages by yourself if you are not familiar with them. Besides, calculating positions might be tricky, be careful.

Specifics

Be sure that you have read the type definitions and other information in the code and header files before you begin coding. **And remember that testing is your responsibility.**

- Your code must be written in C++ and must be contained in the files named **`mp12.cpp`** and **`mp12.h`** in the **`mp12`** subdirectory of your repository. We will NOT grade files with any other names.
- You may modify files as you see fit provided that your five functions operate correctly on the language defined in this specification. In other words, you may extend the language as you like, but you may not redefine the commands outlined here.
- You must write the **`openInputStream`**, **`closeInputStream`**, **`drawEverything`**, **`frameUpdate`**, and **`readNewCommand`** functions correctly.
- You may assume that the parameter values passed into your functions are valid, but be aware that the variables in **`mp12.cpp`** are managed solely by your code.
- You may assume nothing about the commands file. Your code must handle all forms of failure, including non-existent files, lack of permissions, and bad commands in all forms.
- Each movement command should create only one line segment in **`lines`**. When you add a new segment, use **`push_back`** to add it to the end of the list. Change the X and Y coordinates of the last segment in each frame update as the turtle moves.
- Your routine's return values, outputs, and calls to **`TurtleScreen`** methods must be correct.
- Your code must be well-commented. Follow the commenting style of the code examples provided in class and in the textbook, and be sure to add function headers containing the information that has been provided for you in previous assignments (inputs, outputs, return value, and any side effects, as well as a brief description).

Compiling and Executing Your Program

When you are ready to compile, type:

`make`

Warnings and debugging information are turned on in the **`Makefile`**, so you can use **`gdb`** to find your bugs (you will have some).

The executable is placed in the top-level directory, NOT in the **`jni`** subdirectory. If compilation succeeds, you can execute the program by typing, "**`./mp12`**" (no quotes). The **`commands`** file is also in the top-level directory.

To clean up, type "**`make clean`**" (no quotes), or to really clean up, type "**`make clear`**" (as usual, no quotes).

Grading Rubric

Since the end of the semester is nigh, we want to grade this MP quickly and easily.

Towards that end, we will try to avoid requiring a human to look at your code. In upper-level classes, this approach is the norm: 100% weight on functionality. You should still, of course, form good habits and continue to comment as you have learned this semester.

Functionality (100%)

- 5% - **`openInputStream`** function works correctly
- 5% - **`closeInputStream`** function works correctly
- 10% - **`drawEverything`** function works correctly
- 10% - **`color`** command works correctly
- 30% - **`move`** command works correctly
- 10% - **`restart`** command works correctly
- 20% - **`turn`** command works correctly
- 10% - **`wait`** command works correctly
- **-10% PENALTY** - compilation generates any errors or warnings; note that even a single warning incurs the full 10% penalty

Note that some categories in the rubric may depend on other categories and/or criteria. For example, if you code does not compile, you will receive no points. We may not know whether the color command works unless you can draw a line… As always, your functions must be able to be called many times and produce the correct results, so we suggest that you avoid using any additional static storage (or you may lose most/all of your functionality points).