Instructions

There is a single deliverable for this assignment: a C program named `schelling.c`. Please commit it to GitHub and submit via Gradescope. Only one member of each pair needs to do this, though do remember to add the other person's name to the submission. You are expected to practice pair programming, where both partners are working together on a shared screen to complete this assignment.

Schelling's Segregation Model

In 1971, the Nobel Prize winning economist Thomas Schelling proposed a simple mahematical model to study the dynamics of segregation [1]. To be clear, housing segregation is a complex problem, with many causes, including institutional racism (for example, redlining), organized discrimination, and economic inequity. Schelling, while acknowledging that these were far more important factors, was interested in better understanding a third cause: the role of individual choices. He wondered whether large-scale patterns of segregation could emerge from agents making decisions based on local preferences. On this assignment, you will implement a simulation of Schelling's segregation model to study this phenomenon. You should check out *The Parable of the Polygons* (link) for more intuition about Schelling's model and its implications, but note that details of the simulation we are implementing differ slightly from the ones shown there.

The simulation occurs on a two-dimensional array, representing a housing market. Each cell in the array—representing a house—is either vacant, or occupied by one of two kinds of agents, denoted by `.` and `$`. On every discrete time step ("round"), each agent considers its (up to) eight adjacent cells ("neighbors") and counts how many of them are of the same type as itself. If the proportion of similar neighbors is below some threshold (like 0.5), then the agent is dissatisfied and moves to a different vacant cell on the board; otherwise, the agent stays put. While there are many algorithms one could use when moving an agent, we'll use the following: a dissatisfied agent will move to the nearest vacant cell in *row major order*. In other words, an agent will scan its current row left-to-right, starting from its current position, and move to the first vacant spot it finds. If there are no vacant spots in the current row, then it will continue scanning from the start of the next row, and so on, wrapping around to the first row if it reaches the bottom-right corner of the array.

[1] T. C. Schelling, *Dynamic Models of Segregation*, Journal of Mathematical Sociology, 1971, Vol. 1, 143–186.

Note that moving an agent to a new location may cause a previously satisfied agent to now become dissatisfied. We will thus impose a system of priority when moving agents. On a given round, we will mark all dissatisfied agents first. *Only* these agents will be moved during the current round. Once the round is complete, a new round begins and we will once again mark all dissatisfied agents. Within a single round, agents will be moved in row major order. This process repeats until a specified iteration limit is reached, or when all agents are satisfied. An example will help clarify this process.

Consider the left panel in figure 1 that denotes the starting state of a simulation on a 5×4 array. For this example, we'll assume a satisfaction threshold of 0.5, i.e., an agent will stay put so long as at least 50% of its neighbors are of the same type as itself. The middle panel in figure 1 identifies in blue a satisfied agent: three of this agent's neighbors are \$s, while only two are .s, making for a ratio of $3/5 = 0.6$, which is above the 0.5 threshold. Note that we disregard the vacant spots when tallying neighbors. The right panel in the same figure identifies in red a dissatisfied agent: only one of this agent's neighbors is a \$, while two are .s, making for a ratio of only $1/3 = 0.33$.

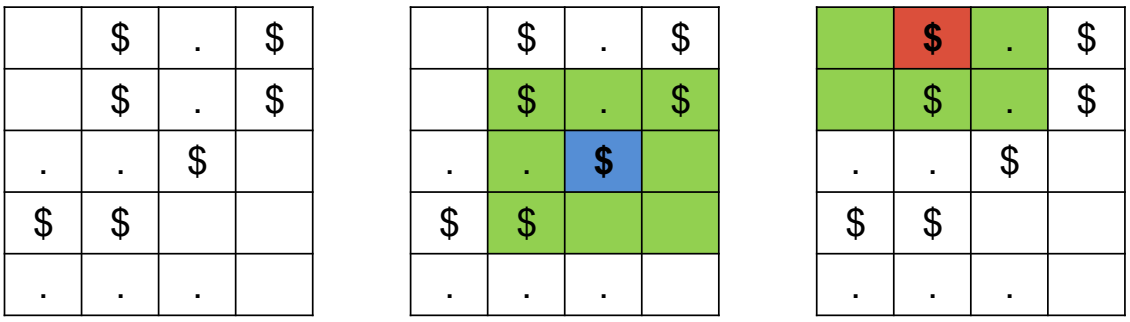

Figure 1: Left: the initial state of a simulation. Middle: a \$ agent (highlighted in blue) that is satisfied. Right: a \$ agent (highlighted in red) that is dissatisfied.

Let us now simulate one round. The left panel in figure 2 shows the state of the array, with all dissatisfied agents highlighted in red. These five \$ and five . agents will all move to new locations on this round. The first agent to move is the first dissatisfied agent that is encountered when the array is traversed in row major order: this is the \$ agent at location $(0, 1)$. This agent will move to location $(1, 0)$, the first vacant spot that is encountered when we continue to traverse the array in row major order (see the middle panel of figure 2). The right panel of figure 2 shows the state of the array after this move. Note that the \$ agent we just moved will still be dissatisfied in its new location, but will not move again this round, having just relocated. Instead, the next agent to move will be the . agent at location $(0, 2)$,

who will move to cell $(2, 3)$ as shown in figure 3.

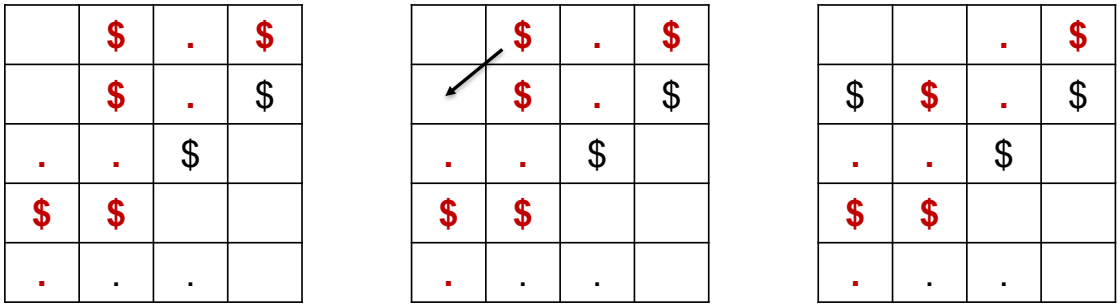

Figure 2: Left: the current state of the simulation, with all dissatisfied agents highlighted in red. Middle: the dissatisfied $ agent at location $(0, 1)$ is the first one we encounter when the array is traversed in row major order and is therefore the first to move. The cell at $(1, 0)$ is the first vacant spot we encounter when we continue to traverse the array in row major order, and is thus the destination for this agent. Right: the state of the array after the move.

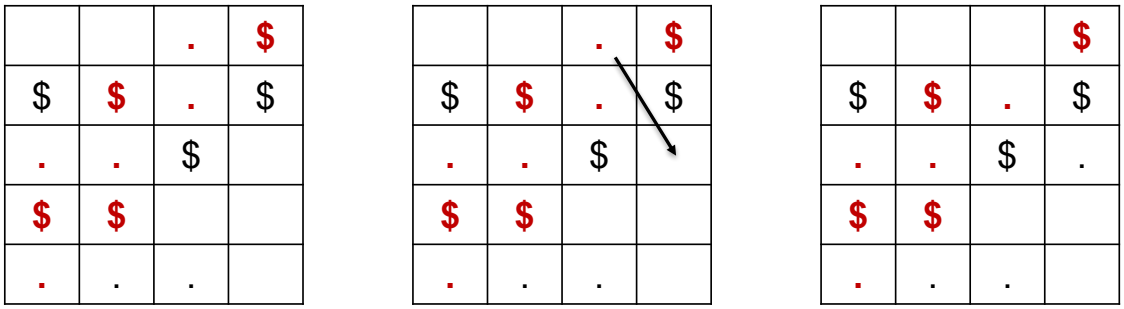

Figure 3: Left: the current state of the simulation, after the first dissatisfied agent has been moved. Agents in red are those that still need to be moved this round. Middle: the dissatisfied . agent at location $(0, 2)$ is next to move, and will relocate to $(2, 3)$. Right: the state of the array after this second move. Eight more agents will still need to be moved before the end of this round.

In all, eight more such moves will happen this round. The state of the simulation when all these moves are completed is shown in figure 4. This marks the end of the round. At this point, we will once again identify all dissatisfied agents and repeat the process described above. The simulation stops once all agents are satisfied, or the prescribed iteration limit is reached. The remainder of this document clarifies the program specification and provides some implementation tips.

.	.	$	$
$	.		$
		$	.
		$	$
	.	.	.

Figure 4: The state of the simulation after one round of moves has been completed, when starting from the initial state depicted in the left panel of figure 1.

Command-Line Arguments

Implement your solution in a file named `schelling.c`. This program should accept **exactly** two *command-line arguments*. We've worked with command-line arguments before, but if you need a refresher, look back at section 2.9.2 of *Dive into Systems* (link).

The first command-line argument to your program will be the name of a configuration file (an ASCII text file) that specifies the dimensions and the initial state of the grid, as well as the number of simulation steps and the agents' satisfaction threshold. The second argument will be an integer value drawn from the set $\{0, 1, 2\}$ indicating the *verbosity level* of your program—a value of 2 for this parameter should cause your program to print out the state of the array after every iteration, a value of 1 should cause your program to *only* print the final state of the simulation, and a value of 0 should suppress all output. Here are a couple of examples of how your program will be executed from the terminal:

```
# run the simulation with configuration values read from small.txt
# and do not print any output
./schelling small.txt 0

# run the simulation with configuration values read from large.txt
# and print the board after every iteration
./schelling large.txt 2
```

Your program should gracefully handle excessive, missing or misspecified command-line arguments! For example, if the user fails to supply a valid filename, specifies an incorrect number of command-line arguments, etc., then your program should print out an error message (see sample message below) and terminate execution by invoking `exit(1)`[2]. For

[2]The convention is to use exit code 0 when a program terminates normally (this is why `main` has return type `int`; by default `main` returns 0) and non-zero values when a program terminates unexpectedly. This value is returned to the operating system, which can then decide how to respond.

example:

```
# ERROR! User forgot to provide *two* arguments when running the program
./schelling small.txt

Usage: ./schelling <filename> <verbosity>
       where <filename> = name of the configuration file.
             <verbosity> = 0 (no output), 1 (minimal output), or
                           2 (animated output)
```

Finally, note that the command-line arguments are provided as strings (i.e., `char` arrays) to `main`. However, you want to treat the second argument (the verbosity level) as an integer. In Python, you would use the `int()` function to convert a string into an integer—the `atoi()` ("ASCII to integer") function serves the same purpose in C. Refer to the Unix manual pages, accessed by the terminal command `man atoi` for more details on how to correctly invoke this function.

Configuration File Format

The configuration file that is provided to your program will consist of several lines of ASCII text. The first two lines specify the grid dimensions and the number of simulation iterations. The third line will contain a float denoting the agents' satisfaction threshold. This is followed by two sections of (i, j) (row, column) coordinates indicating which grid cells should be initialized to `$`s and `.`s respectively at start up—all other cells should be considered vacant. Here is an example configuration file (this is the `small.txt` provided in the repository):

```
5 4
100
0.5
7
0 1
1 3
...    # compressed in the interests of space
1 1
7
1 2
2 1
...    # compressed in the interests of space
4 0
```

Parsing these rows top to bottom, these numbers indicate that:

- the grid has dimensions 5×4 (i.e., 5 rows and 4 columns),
- the simulation is to run for a maximum of 100 iterations,
- each agent's satisfaction threshold is 0.5,
- there are seven grid cells to be initialized to `$`,
- those cells are $(0, 1), (1, 3), \ldots, (1, 1)$,
- there are seven grid cells to be initialized to `.`,
- and those cells are $(1, 2), (2, 1), \ldots, (4, 0)$.

This configuration file matches the worked example from earlier, so you may find it useful for debugging purposes. A second file `large.txt` is also provided to you for testing your simulation on a larger housing market. You are welcome (and indeed encouraged!) to create your own configuration files using a text editor of your choice to test other cases.

You may assume that the grid will have at least one row and one column. Further, you may assume that a cell won't appear in both the "occupied by `$`" and "occupied by `.`" sections of the configuration file. However, this input file may be misformatted in other ways—it may not adhere to the format described above, it may not contain valid integers or a valid value for the satisfaction threshold, etc., in which case your program should print an error message and exit with the error code 2. *Use defensive programming conventions when reading the configuration file (like testing the return value of `fscanf`) to detect and gracefully handle these errors!*

Use the C `FILE` interface (`fopen`, `fscanf`, `feof`, `fclose`) for handling file I/O. Section 2.8.2 of *Dive Into Systems* (link) covers file I/O.

Printing the Grid

As stated earlier, when run with verbosity set to 2, your program should print out the state of the grid after each simulation step, starting with the initial state. This means that for an n-step simulation, your program should print out $(n + 1)$ grids in all (including the start and end states). Below is an example of how you should print out the current state of the simulation—this corresponds to the initial state described in the `small.txt` file. Note that

vacant spaces should be printed out using the space character, but these have been denoted below using the ␣ character for clarity.

```
␣$.$
␣$.$
..$␣
$$␣␣
...␣
```

To animate the simulation, you should pause the program after printing the current state of the grid, before clearing the screen and printing the next state of the grid. This will cause every iteration's output to be written to the same spot on the screen, providing a dynamic visualization of the simulation as it proceeds. You can use the `usleep` function for pausing program execution—the call `usleep(200000)` will pause the program for 0.2s. This function is declared in the `unistd.h` header file, so you will need to `#include` this at the top of your program. You can call `system("clear")` to clear the terminal of the previous iteration's output. *Make sure not to clear the very last iteration's printed result.* Note that running your program with a verbosity level of 0 should execute a "pure" simulation that prints no output to the screen and makes no calls to `usleep` or `system`. Running your program with verbosity level 1 should cause it to behave identically to verbosity level 0, except that it should print the final state of the grid alone.

Other Notes and Constraints

- The space for the board should be dynamically allocated on the heap since the grid dimensions only become known at runtime when you have parsed the configuration file. Section 2.5.2 of *Dive into Systems* (link) explains how to dynamically allocate two-dimensional arrays. You are required to use "Method 1: Memory Efficient Allocation," which allocates a single block of memory for the two-dimensional array, in your solution.

  Remember to check the return value of `malloc` to verify that the space you requested was indeed granted to your program—otherwise, your program should print an error message and exit with the error code 3. Also remember to `free` dynamically allocated memory when done, and to run `valgrind` to check for memory access errors and memory leaks.

- To prevent division-by-zero errors, you may assume that an agent completely surrounded by vacant spaces is satisfied.

- Your solution should *not* use any global variables. Use local variables that are passed to functions that need them.
- Your program should demonstrate evidence of thoughtful design—*do not write all your code in one large function*! Use helper functions to decompose the task at hand. Each function should solve one discrete, well-defined task.
- You can use `CNTRL-C` to kill your program if it appears to be stuck in an infinite loop.

Code Style Guidelines

We will be emphasizing good coding style in the grading of this assignment. Your code should abide by the guidelines outlined in the *C Style Guide* document. The following (non-exhaustive) checklist may be handy when preparing your code for submission.

Checklist

- Correctness and Defensive Programming
  - ☐ The file `schelling.c` implements Schelling's segregation simulation as outlined in the specification.
  - ☐ The `main` function accepts and processes command-line arguments as outlined in the specification.
  - ☐ The source file compiles with no error messages when using the `-Wall` flag.
  - ☐ The program runs without generating any `valgrind` errors or warnings about memory leaks.
  - ☐ The program checks the return values of all calls to the library functions listed in the style guide.
  - ☐ All open files are closed before the program terminates.
- Documentation
  - ☐ A comment header at the top of your file has a brief description of your program, the names of you and your partner, and the time you spent on the assignment.
  - ☐ Every function has its own comment header that briefly describes its purpose, parameters, and return values.
  - ☐ Comments inside functions describe the sub-tasks solved by the function, when appropriate.
- Readability
  - ☐ Variable and function names are meaningful and make their purpose clear to the reader.
  - ☐ Local variables are used to the maximum extent possible, without resorting to unnecessary global variables.

- ☐ The code is indented neatly and consistently.
- ☐ "Magic numbers" and other constants have been eliminated using the `#define` directive.
- ☐ Boolean expressions are used in the simplest form possible.

- Structure and Parsimony
  - ☐ The code is free of unnecessary complications and redundant fragments.
  - ☐ The program displays evidence of thoughtful design, with appropriate decomposition and helper functions.

Submission

Submit the file `schelling.c` by pushing to GitHub and uploading to Gradescope. Only one member of your team needs to do this, though you should remember to attach your partner's name to the submission. Note that not following the given specifications—misnaming your C file, misnaming your function, having your function accept an incorrect number of parameters or parameters in the incorrect order, not returning the correct value in your function, etc.—will result in significant autograder failures, so read the handout again carefully if this happens.