Objective

- build a reactive grid-bot
- practice designing and building a finite state machine (FSM)
- practice using vectors
- practice using classes

Specifications

In this project you will write a program that uses the `Grid` class from Chapter 4 of the Mercer text. You will design and implement a program that reactively controls the actions of a software robot (grid-bot). To do this you will use a finite state machine (FSM) that controls the action of the grid-bot.

The `grid` is a 20 (rows) by 30 (columns) grid that looks like this:

```
    0 1 2 3 4 5 6 7 8 9 10  12  14  16  18  20  22 24  26  28 29
0   > . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
1   . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
2   . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
3   . . . . # # . . . . . . . . . . . . . . . . . . . . . . .
4   . . . . # # . . . . . . . . . . . . . . . . . . . . . . .
5   . . . . . . . . # # # # # . . . . . . . . . . . . . . . .
6   . . . . . . . . # . . # # . . . . . . . . . . . . . . . .
7   . . . . . . . . # # # . . . . . . . . . # # # . . . . . .
8   . . . . . . . . . . . . . . . . . . . . # . . # . . . . .
9   . . . . . . . . . . . . . . . . . . . . # . . # . . . . .
10  . . . . . . . . . . . . . . . . . . . . . # # . . . . . .
11  . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
12  . . . . . . . . . . . . . . . . # # # . . . . . . . . . .
13  . . . . . . . . . . . . . . . . # . # . . . . . . . . . .
14  . . . . . . . . . . . . . . . . # # # . . . . . . . . . .
15  . . . . . # # # . . . . . . . . . . . . . . . . . . . . .
16  . . . . . # . . # . . . . . . . . . . . . . . . . . . . .
17  . . . . . . # # # . . . . . . . . . . . . . . . . . . . .
18  . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
19  . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

The `Grid` class is found in your Mercer book on pages 123–127. Please refer to these pages when you are doing your project. You will need to make use of most

of the methods provided, including the other grid operations found on page 126. For this project we have produced a new `Grid` class implementation in the form of two files `grid.cc` and `grid.h`. There is also a `Makefile` which you can use. You can copy these files from the directory `~cs203/Projects/Project8`.

In addition to these methods, we have added two new methods, `getFacing()` and `cookieIsHere()`. The `getFacing` method returns the corresponding integer value for the specified direction:

0	north
1	east
2	south
3	west

depending on which way the grid-bot is facing. This function will be useful in order to get the grid-bot to return home. The method `cookieIsHere()` is a boolean method which returns `true` if the grid-bot's current square contains a cookie. The utility of this method will become clear shortly.

To construct the grid shown above, copy the file `~cs203/Projects/Project8/grid.dat` into your project directory. Notice that this file contains a list of all the blocks on the grid. Each line in the file contains a row number followed by a column number. You should not need to store these blocks since they can be placed directly into the grid from the file. The intention is that the file contains correct data, but you should check each input pair and ignore any pair whose row or column value is out of bounds.

As you input the block coordinates, you should create a vector of the block locations. This vector should take `Point` objects – remember the `Point` class from earlier in the semester. You will implement the `Point` class and then create the vector of blocks which are actually placed on the grid. This use of this vector is described in the next paragraph.

Your program will also take as input a list of cookies. These cookies have two integers as coordinates (just like the blocks). You may design your program to read the cookies location either from the keyboard or from a file. If you input from the keyboard then validate the input as usual (loop until a correct pair is entered) and if the input is from a file process as you do the blocks (described above). One additional requirement here is that a cookie cannot have the same position as a block. Part of the cookie validation is to only accept a pair if it doesn't coincide with a block location. You must do a search of the block vector to do this check.

Generating Moves

You should use a random number generator to determine a potential move. To do this, include the file `<unistd.h>` in your program. Then call the function `srandom` *once* at the beginning of your program as follows.

```
srandom(getpid());
```

Then you can use the function `random()` to produce random numbers. See the file `http://www.eg.bucknell.edu/~zaccone/csci203/largest.cc` for an example that uses `random()` and `srandom()`.

You will need to write a function to generate a potential move. You can represent your moves as integers. Use the mod function on your random value to determine which move to make.

```
int i = random()%6;
```

Note: Each move does not have to have the same probability – for example, you could make 0–4 be a move-forward, 5 be a turn left, 6 and 7 be a turn right, and 8 be a pick-up cookie.

The meaning of a potential move will be determined by a finite state machine (FSM), which will be described in the next section. This means that you will have to first generate the potential move and then let the finite state machine interpret the move based on the current state of the machine and other factors. The moves which the grid-bot can carry out are:

1. turn left,
2. turn right,
3. move forward one space, and
4. pick-up a cookie.

The Finite State Machine

The FSM has six states and transitions are driven by the next move (randomly generated) and the status of the grid-bot's current grid square location.

FSM transitions are described as follows. Notice that for each state the possible transitions are described: what condition will cause a transition to be taken, the action to be carried out for that transition, and the next state to be visited.

1. `WANDER` — This is the machine's *start* state.

   If the current square contains a cookie then there is no action and the next state is the `PICKUP` state (no matter what the generated move is). Assuming the current square is empty (except for the grid-bot), if the generated move is "move forward one space" and the square in that direction is a block, then there is no action and the next state is the `AVOID` state; if the generated move is "pick-up a cookie" then no action occurs and the next state is the `WANDER` state. For all other moves, "turn left", "turn right", or "move forward one space", carry out the move and the next state is the `WANDER` state.

2. `AVOID` — If the grid-bot is in this state then there is a block blocking a move straight ahead (in the direction the grid-bot is currently pointed).

   If the generated move is a turn either to the left or right then the move is carried out and the next state is the `WANDER` state. For all other moves there is no action and the next state is the `AVOID` state.

3. `PICKUP` — If the grid-bot is in this state then it is "standing" on a cookie which must be picked up.

   If the generated move is "pick-up a cookie" then the action is carried out; if the cookie count is greater than zero then the next state is the `WANDER` state; if the cookie count is zero then the next state is the `RETURN` state. If any other move is generated, there is no action and the next state is the `PICKUP` state.

4. `RETURN` — If the grid-bot is in this state then it has picked up all the cookies and is trying to return to square (0,0).

   If the current grid square is (0,0) then no action is taken and the next state is the `DONE` state.

   If the generated move is a turn, either left or right, then carry out the action and the next state is the `RETURN` state. If the move is "move forward one space" and the direction of motion would cause the row or column number of the grid-bot to increase, then there is no action and the next state is the `RETURN` state. If the move is "move forward one space" and the row and col decrease or remain the same, then carry out the action if the facing square is not blocked and the next state is the `RETURN` state, but if the facing square is blocked then there is no action and the next state is `AVOIDonRETURN`.

5. `AVOIDonRETURN` — If the grid-bot is in this state then there is a block blocking a move straight ahead and the cookie-count is zero.

If the generated move is a turn either to the left or right then the move is carried out and the next state is the `RETURN` state. For all other moves there is no action and the next state is the `AVOIDonRETURN` state.

6. `DONE` — The finite state machine halts.

Design

Draw a finite state diagram of the FSM described above. In that diagram attempt to identify the conditions for each transition, and the actions that will take place on those transitions. Recall that a transition is the arc or edge that connects one state to another.

Implementation Ideas

In order to `verifyReturnMoves` you should consider which direction the grid-bot is facing, and its row and column. These will need to be compared with the origin.

You can use a global counter to figure out if the grid-bot has collected all the cookies. If the global counter has reached the the initial number of cookies , you are done.

After each move you will want to display your grid. This will allow you to see how the grid-bot is doing. Also output, just after the grid display, the number of moves so far (keep a counter), current state of the grid-bot, and the number of cookies remaining. [Write a function `displayState()` which displays the state name.]

Hand In

1. The state diagram for your FSM. This must be very neat! Use a drawing program if possible.

2. The final source code. Be sure to include appropriate pre and postconditions as well as assertions and/or `assert` calls, according to the style encouraged by your instructor.

3. A sample run from your program. To prepare your output for handing in, run the program and copy the output to a file (using `emacs`). Edit the file and remove the grid display for all moves EXCEPT the initial grid, the final grid, and the grid which is displayed on the step just after each cookie is picked up. Leave two blank lines between grid displays.