Note: On this and subsequent assignments, you will be required to take responsibility for your own testing. As part of that requirement, this assignment is designed to get you into the habit of thinking about testing before you start writing your program. If you look at the deliverables and their due dates, you will notice that there is no C++ code due on Due Date 1. Instead, you will be asked to submit test suites for C++ programs that you will later submit by Due Date 2.

There will be a handmarking component in this assignment, whose purpose is to ensure that you are following an appropriate standard of documentation and style, and to verify any assignment requirements not directly checked by Marmoset. You are not expected to follow a specific style guide, however your code should be readable and you should could with good style. That is, abstract out functions instead of having repetitive code segments, follow good OOP style, as well as not writing overly complex code when unnecessary. Please be aware that if handmarking shows you have not followed the requirements of the question, correctness marks from test cases can partially or totally be taken away.

Some or all of these programs will ask you to develop C++ modules in order to make a provided test harness file which includes those modules work. Since you must use the main provided in the test harness to test your program, it is suggested that you first begin by providing empty implementations of all of the necessary functions. Then you will at least be able to compile, and as you implement each function you can recompile and test them immediately.

Note: All of your program must not leak resources. If your program produces the correct result but in doing so leaks resources then test case will not be considered as having been passed. We will use the tool valgrind to test your programs (which is available on the student servers) so you can too. Your code must compile and run correctly on the student server.

Allowed headers for this assignment: For this assignment you may use any of the standard library headers you would like.

1. Finding a way out of a maze is a classic sample problem used with learning algorithms. It’s especially common to see mazes used as simple examples of how Reinforcement Learning algorithms work. In these types of puzzles an “agent” exists in a maze, and can take actions to move through it to find its way out. In this question you’ll be using your knowledge of virtual functions to develop a class hierarchy for different types of mazes to be solved.

You will have to develop at least four classes, an abstract base class Maze, and concrete classes BasicMaze, TeleporterMaze, and IcyMaze. Your mazes represent a two-dimensional grid of tiles, and the types of tiles that are acceptable can vary between the maze types.

- Start Tile - Denoted by the character ‘$S$’, this denotes where the agent will begin in the maze when it is first created, or when it is reset. You may assume every maze has exactly one start tile.
• Goal Tile - Denoted by the character 'G', this denotes the “goal” or exit of the maze. When the agent reaches this location its task is done. There may be multiple goal states in a maze.

• Open Tile - Denoted by the character 'O', this denotes a normal tile of the maze that the agent is allowed to walk through.

• Wall Tile - Denoted by the character 'X', this denotes a solid wall tile of the maze that the agent is not able to walk through.

• Agent Tile - Denoted by the character 'P'. This is not actually a tile in your maze, but when outputting your maze whichever tile the agent is currently located on should be printed as a 'P' instead of its tile character.

• Teleporter Tiles - Denoted by any digit character ('0' through '9'), this denotes a teleporter tile. **Only available in the TeleporterMaze variant.** When the agent walks on a teleporter tile their location becomes that of the next matching teleporter tile “in order”. That is, if the agent walks on a teleporter tile denoted by '2' (we will call this the source), then after that move the agent is actually located at the next teleporter tile denoted by '2' (we will call this the destination). The definition of how to find the "next" here is interpreted the same way English is read, from left-to-right, top-to-bottom starting at the location immediately after the source. If you reach the bottom right corner of your maze and have not found a matching teleporter, than you should begin again looking at the top-left corner of your maze. If you are unsure about the behaviour, as always, test with the sample executable.

Here is an example of a basic maze, with the agent directly adjacent to the goal state, with the start state in the top left and the goal state in the bottom right:

```
SOOXO
OXOOO
OXXXP
OOXXG
```

Your program will be tested with the provided test harness a2q2.cc. As always for clarifications on these functions you should see the sample executable, in the case of a discrepancy between the assignment specification and the sample executable you should match the sample executable. The required functions (those used in a2q2.cc) are:

• Default Constructors - Each of the three classes must provide a default constructor.

• Overloaded Input Operator - The input operator should be overloaded to work with Maze reference parameters. This function should read characters from the stream using them to create your maze, until it reads the character 'Q' which is the signal to stop. Every time a new line character is read that means the user is inputting the next row of your maze. When a character that is not a valid tile of your maze is read it should be ignored. See the provided input file for an example of the formatting. Empty rows (when a newline is read when no valid characters have been read before it) should be excluded from your maze. In order to read in the new line characters you will need to use the noskipws flag with the provided stream, however you should make sure that after you are done the stream is returned to its intial state (e.g. if the noskipws flag was off when this function was called, it should be off when the function returns). See cplusplus.com for information on this flag.

• Overloaded Output Operator - The output operator should be overloaded to work with const Maze reference parameters. This function should print out the maze, with the location of the agent being represented by the character 'P'. There should be a newline after every row, including the last one.
• The member function `vector<int> MakeMove(char)` - This function is used to allow the agent to move through your maze. The input to this function is one of the characters 'N', 'E', 'S', or 'W', denoting the agent attempting to move North, East, South, or West respectively. The rules of movement vary for the types of mazes.
  - In any maze if the movement is off the bounds of the maze, or into a wall tile, then the agent remains in its current position.
  - In the `BasicMaze` if the tile is not a wall and in the bounds of the maze then the agent moves into that tile.
  - In the `TeleporterMaze` the rules of the `BasicMaze` are followed, but additionally if the tile is a teleporter tile then the agent is teleported to corresponding destination tile, as per the rules described above when defining teleporter tiles.
  - In the `IcyMaze` the agent will repeatedly move in the direction provided until it can no longer move in that direction (it reaches a wall or the bounds of the maze).

The return type of this function is a vector of ints which should have exactly two elements within. The first element should represent the agent’s new position in the x-axis, and the second element should represent the agent’s new position in the y-axis. **Unless** the agent moved into a goal state with this move, then both values in the vector should be -1.

• The member function `void reset()` - This function “resets” the maze, by placing the agent back at the starting tile.

**Deliverables:**

a) **Due on Due Date 1:** Design a test suite for this program. Call your suite file `suiteq1.txt`. Zip your suite file, together with the associated `.in` files, into the file `a3q1.zip`.

b) **Due on Due Date 2:** Write the program in C++. You must include in your submission (a3q1.zip) your header and implementation files as well as a `makefile` that builds your program with the executable named `a3q1`.

2. In this question you will be building a version of [Conway’s Game of Life](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life). The game of life is not really a “game”, but rather an automaton that simulates the “life” and “death” of cells as they progresses between “generations” according to a set of rules. In the game of life cells in a grid are either “alive” or “dead”, a given state of the grid is called a “generation”. Then to progress to the next generation a set of rules for which cells will become alive or dead the next generation are followed and the next generation is computed. In Conway’s Game of Life the rules are:

(a) Any cell with fewer than two living neighbours dies (due to **underpopulation**).

(b) Any cell with more than three living neighbours dies (due to **overpopulation**).

(c) Any dead cell with exactly three live neighbours becomes a live cell (due to **reproduction**).

However, in our version of the Game Of Life not every cell in the grid must follow the exact same rules. Instead rules can be added onto a cell at run time, and we will have some extra rules we can add to cells as well. Our rules will function as follows, rule 1 applies to all cells, the rest have to be added on to cells at run time. That is, all cells in the grid follow the underpopulation rule, all other rules are applied individually to cells. However, for convenience your program will also provide the option to default all the cells to one type (and then additional rules can be added onto cells individually)
1) **Only rule that applies to ALL cells:** Any dead cell with exactly three live neighbours becomes a live cell, unless another rule says it would die.

2) Any cell with fewer than two living neighbours dies.

3) Any cell with more than three living neighbours dies.

4) A periodic cell - this rule takes an additional integer parameter N denoting the period. When applied to the cell every N generations it will flip its state (if it is dead on the Nth generation it will become alive, if it is alive it will become dead).

5) A friendly cell - this rule takes an additional integer parameter N denoting how many friends this cell likes to have. If this cell is surrounded by exactly N neighbours then it will become (or stay) alive into the next generation.

The order the rules are added onto a cell matter. For example the “Friendly Cell” rule can directly conflict with the rules about overpopulation and underpopulation. In the case of a conflict, whichever rule was more recently added takes precedence.

Since the rules can be added at runtime to change the behaviour of the cells, you are required to implement this using the **Decorator pattern**. The test harness `a4q2.cc` simply instantiates a `LifeGame` object with a given width and height, and calls the `play()` method on it. You must implement this class, as well as any other classes you need to properly design your solution to match the behaviour required by the spec/sample executable. The `play()` method does the job of reading input from the user. As such a skeleton for the method has been given, which you will need to fill in to make it work with your implementation of the game. The comments in the skeleton explain the command structure for the game. However, the commands are also listed below.

- **update** command — if the user inputs the character `u` as a command this should update your Game of Life to the next generation. That is, each cell in the game should become either dead or alive based on the rules applied to that cell.

- **on** command — if the user inputs the character `o` as a command then two integers `x` and `y` are read in and the cell located at `(x, y)` on the grid is turned on. The upper left corner of the grid represents `(0,0)` and the x-coordinates get larger as you move right while the y-coordinates get larger as you move down. That is, the coordinate `(2, 5)` represents the cell in the third column (x-coordinate of 2), and the sixth row (y-coordinate of 5).

- **Rule two command** — if the user inputs the character `l` as a command then two integers `x` and `y` are read in and the cell located at `(x, y)` on the grid has rule two applied to it (it is decorated with a rule two object).

- **Rule three command** — if the user inputs the character `a` as a command then two integers `x` and `y` are read in and the cell located at `(x, y)` on the grid as rule three applied to it (it is decorated with a rule three object).

- **Rule four command** — if the user inputs the character `t` as a command then two integers `x` and `y` are read in and the cell located at `(x, y)` on the grid as rule four applied to it (it is decorated with a rule four object).

- **Rule five command** — if the user inputs the character `f` as a command then three integers `x`, `y`, and `N` are read in and the cell located at `(x, y)` on the grid as rule five parameterized by `N` applied to it (it is decorated with a rule five object).

- **default** command — if the user inputs the character `d` as a command then you must read in characters until anything other than a `l`, `a`, `p`, or `f` is read in (or an integer only if it immediately follows an f). Then, all cells in the grid must be set to off and have the rules read in applied to them. That is, if the user inputs `dla*`, then all cells in the game are set to dead and now follow rules 1, 2, and 3. Similarly, if the user inputs `daf7-`, then
all cells in the game are set to dead and now follow rules 1, 3, and 5 (parameterized with 7).

- **print command** — if the user inputs the character p as a command then the grid will be printed out as a 2D grid of characters. Living cells should be printed as the character “X”, while dead cells should be printed as the period character “.”.

- **quit command** — if the user inputs the character q as a command then the function will end.

**Important:** As the point of this assignment is to use the Decorator pattern, if your solution is found in handmarking to not employ the decorator pattern your correctness marks will be reduced up to 0.

**Hints:** Create a Cell abstract base class and have Decorator extend from that. As per the decorator pattern create a BaseCell class which implements the basic behaviour (and also stores the variables you need). Include a Grid class that stores your grid of Cells.

**Note:** Your program should be well documented and employ proper programming style. It should not leak memory. Markers will be checking for these things.

**Deliverables:**

a) **Due on Due Date 1:** Design a test suite for this program. Call your suite file suiteq2.txt. Zip your suite file, together with the associated .in and .args files, into the file a3q2.zip.

b) **Due on Due Date 2:** Write the program in C++. You must include in your submission (a3q2.zip) your header and implementation files as well as a makefile that builds your program with the executable named a3q2.

3. In this problem, you will use C++ classes to implement the game of Lights Out. (http://en.wikipedia.org/wiki/Lights_Out_%28game%29). An instance of Lights Out consists of an n × n-grid of cells, each of which can be either on or off. When the game begins, we specify an initial configuration of on and off cells. Lights Out is a one-player game. Once the cells are configured, the player chooses a cell and turns it on if it is off, and off if it is on. In response the four neighbouring cells (to the north, south, east, and west) all switch configurations between off and on as well. The object of the game is to get all of the cells turned off.

To implement the game, you will use the following classes:

- **class Cell** — implements a single cell in the grid (see provided cell.h);
- **class Grid** — implements a two-dimensional grid of cells (see provided grid.h);
- **class TextDisplay** — keeps track of the character grid to be displayed on the screen (see provided textdisplay.h).

**Note:** you are not allowed to change the public interface of these classes (i.e., you may not add public fields or methods), but you may add private fields or methods if you want.

Your solution to this problem must employ the Observer pattern. Each cell of the grid is an observer of all of its neighbours (that means that class Cell is its own observer). Thus, when the grid calls Cell::notifyNeighbours on a given cell, that cell must then call the notify method on each of its neighbours (each cell is told who its neighbours are when the grid is initialized). Moreover, the TextDisplay class is an observer of every cell. When a cell’s status changes, it must invoke TextDisplay::notify to publish its new state to the observer.
You are to overload `operator<<` for the text display, such that the entire grid is printed out when this operator is invoked. Each on cell prints as X and an off cell prints as _ (i.e., underscore). Further, you are to overload `operator<<` for grids, such that printing a grid invokes `operator<<` for the text display, thus making the grid appear on the screen.

When you run your program, it will listen on stdin for commands. Your program must accept the following commands:

- **new n** Creates a new \( n \times n \) grid, where \( n \geq 1 \). If there was already an active grid, that grid is destroyed and replaced with the new one.
- **init** Enters initialization mode. Subsequently, read pairs of integers \( r \ c \) and set the cell at row \( r \), column \( c \) as on. The top-left corner is row 0, column 0. The coordinates \(-1 \ -1\) end initialization mode. It is possible to enter initialization mode more than once, and even while the game is running. When initialization mode ends, the board should be displayed.
- **game g** Once the board has been initialized, this command starts a new game, with a commitment to solve the game in \( g \) moves or fewer.
- **switch r c** Within a game, switches the cell at row \( r \), column \( c \) on or off, and then redisplays the grid.

**Note:** The commands for this grid refer to row and column, rather than \( x \) and \( y \) as in Q2. It’s important to note that a coordinate \((x, y)\) is equivalent to a \((row, column)\) pair when \(row == y\) and \(column == x\). That is, your row represents your \( y \) coordinate and your column represents your \( x \) coordinate. Can you think of a clever way to reuse some parts of your code for class `Grid` despite these interface differences?

The program ends when the input stream is exhausted or when the game is won or lost. The game is lost if the board is not cleared within \( g \) moves. You may assume that inputs are valid.

If the game is won, the program should display `Won` to stdout before terminating; if the game is lost, it should display `Lost`. If input was exhausted before the game was won or lost, it should display nothing.

A sample interaction follows (responses from the program are in italics):

```
new 5
init
1 2
2 2
3 2
-1 -1
-----
__X__
__X__
__X__
-----

---X---
---X---
---X---

3 moves left
switch 2 2
-----
-----
__X_X__
-----
-----

2 moves left
```
switch 3 1
-----
-----
___X_
XXX_
__X__
1 move left
switch 3 3
-----
-----
-----
XX_XX
__X_X
0 moves left
Lost

Note: Your program should be well documented and employ proper programming style. It should not leak memory. Markers will be checking for these things.

Deliverables:

a) Due on Due Date 1: Design a test suite for this program. Call your suite file suiteq3.txt. Zip your suite file, together with the associated .in files, into the file a3q3.zip.

b) Due on Due Date 2: Write the program in C++. You must include in your submission (a3q3.zip) your header and implementation files as well as a makefile that builds your program with the executable named a3q3.

4. In this problem, you will adapt your solution from problem 3 to produce a graphical display. You are provided with an SDL wrapper (files sdl_wrap.h and sdl_wrap.cc), to handle the mechanics of getting graphics to display. Declaring a Screen and SDL_Runner object (e.g., SDL_Runner r; Screen s;) causes a window to appear. When the object goes out of scope, the window will disappear (thanks to the destructor). The class supports methods for drawing rectangles.

For this assignment, you should only need black and white rectangles. To make your solution graphical, you should carry out the following tasks:

- add fields for the x- and y-coordinates, as well as width and height, and a pointer to a window in the Cell class.
- add a method setCoords to the Cell class, whose purpose is to set the above fields. The Grid object will call this method when it initializes the cells.
- add a field to the Grid class representing the pointer to the window, so that it can be passed on to the cells. Change Grid’s constructor so that it can initialize the window field.
- add draw and undraw methods to the Cell class to draw either a black or white rectangle to the correct spot on the board (as determined by each cell’s coordinates).
- When a cell goes from off to on, it should call its draw method.
- When a cell goes from on to off, it should call its undraw method.

The window you create should be of size 500×500, which is the default for the Xwindow class. The larger the grid you create, the smaller the individual squares will be.

Note: to compile this program, you need to pass the option -lSDL to the linker for linking your program. For example:
g++ *.o -o lights-graphical -lSDL

**Note:** Your program should be well documented and employ proper programming style. It should not leak memory (note, however, that the given SDL library leaks a small amount of memory; this is a known issue). Markers will be checking for these things.

**Deliverables:**

a) **Due on Due Date 2:** Write the program in C++. You must include in your submission (a3q4.zip) your header and implementation files as well as a `makefile` that builds your program with the executable named `a3q4`. 