CMPUT275—Assignment 3 (Winter 2024)

X. Li R. Hackman

Due Date: Friday March 22nd, 8:00PM

Per course policy you are allowed to engage in *reasonable* collaboration with your classmates. You must include in the comments of your assignment solutions a list of any students you collaborated with on that particular question.

In this assignment and all following assignments you should test against the sample executables. The sample executables try to help you out and print error messages when receiving invalid input (though they do not catch *all* invalid input). You do not have to replicate any error messages printed, unless the assignment specification specifically asks for error messages. These messages are only in the sample executable to help you catch when you write an invalid test case.

Memory Management: In order to complete some of these questions you will be required to use dynamic memory allocation. Your programs must not leak any memory, if you leak memory on a test case then you are considerd to have failed that test case. You can test your program for memory leaks by using the tool `valgrind`.

Memory Requirements: In addition to not leaking memory your programs must not use at any one time more than double of the maximum amount of memory they require. That is if implementing a dynamic array you may should use the doubling strategy. If you simply allocate a very large array hoping input sizes will never exceed that then you will not receive marks for that question. For initializing dynamic arrays you may intialize them to have a capacity of 4.

Compilation Flags: each of your programs should be compiled with the following command:

```
g++ -std=c++17 -Wall -Werror
```

These are the flags we'll compile your program with, and should they result in a compilation error then your code will not be compiled and ran for testing.

Allowed libraries: `iostream`, `iomanip`, `string`, `sstream`, and for Q2 `cmath`. No other libraries are allowed.

1. **Conway's Game of Life**

In this question you will be implementing Conway's Game of Life. Conway's game of life is not really a game, it is a cellular automaton with very simple rules. Firstly, it takes place on a grid of squares. Each cell in the grid can be either "alive" or "dead" and begins with an initial value as the input. The cellular automaton then runs with rules to update each time step. Every time step cells will either become alive, become dead, or remain unchanged based on a four simple rules

(a) The *underpopulation* rule: Any live cell with fewer than two live neighbors dies

(b) The *overpopulation* rule: Any live cell with more than three live neighbors dies

(c) The *reproduction* rule: Any dead cell with exactly three live neighbors becomes a live cell

(d) Any live cell with two or three live neighbors stays alive in the next generation

In Conway's Game of Life a neighbour is any cell adjacent in the cardinal or ordinal directions, that is any other cell sharing a side or cells adjacent on the diagonal.

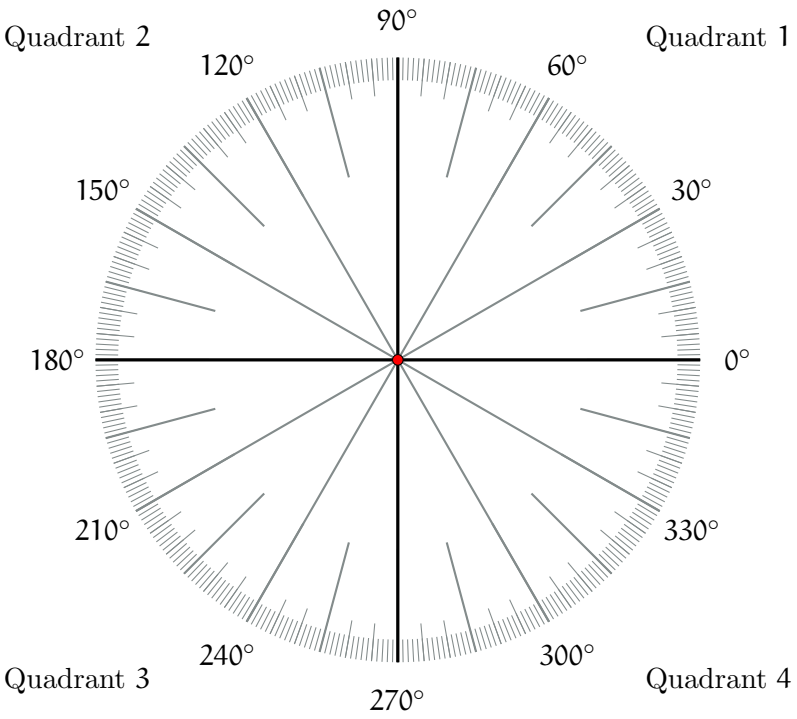
The first input to your program will be the grid, one row per line, with each alive cell being represented by the character `'O'` and each dead cell being represented by the character `'.'`. Once the grid is complete your program will receive the letter `'x'` alone on a line.

After your program has read in the grid the only non-whitespace characters your program will receive are `'s'` or `'p'`. When your program receives the command `'p'` it should print out the grid in its current state, with a line of pipe characters `'|'` above and below it. When your program receives the command `'s'` it should progress the grid one time step following the rules described above.

Deliverables For this question include in your final submission zip your C++ source code file named `conways.cc`

2. In this question you will be developing some very basic ADTs and operations to simulate very simplistic forces in a two-dimensional plane. You will have to develop the classes `Point`, `Force`, and accompanying operations in order to work with the provided test harness `a3q2.cc`.

   The `Point` class will model a point in a two-dimensional plane with a given `x` and `y` float value. The `Force` class will model a force in two-dimensional plane, with a given `angle` (in degrees), and `magnitude`. Angles in degrees start from the line out in the positive `x` direction of the cartesian plane, and wraps back around at 360 degrees. See below for an example:



   You have been provided a header `2DMotion.h`, which you may change as you like. However, you must implement all the functionality used by the provided test harness file `a3q2.cc`, as that file (or a similar one in how it uses the `Point` and `Force` classes) will be used to test your program. You may **not** change the test harness, as we will test you with our own copy of it, so any changes you make won't be reflected in how your solution is tested! You must implement the following functions for this question:

   - `Point` default constructor. Should initialize both `x` and `y` to 0.
   - `Force` default constructor. Should initialize both `angle` and `magnitude` to 0.
   - An overloaded input operator for `Point` objects. Should read in first the `x` field, then the `y` field.
   - An overloaded input operator for `Force` objects. Should read in first the `angle`, then the `magnitude`.
   - An overloaded output operator for `Point` objects. Should print them out in the format: "`(<x>, <y>)`" (Note `<variable>` is used to denote that the variables value should go there, so if the point has `x` value 4, and `y` value 5, you should print out "(4, 5)".
   - An overloaded output operator for `Force` objects. Should print them out in the format: "`<degrees> degrees with magnitude of <magnitude>`"
   - An overloaded addition operator between a `Point` object and a `Force` object. This should effectively create a new `Point` that is the result of "moving" the original by that `Force`. This requires a bit of trigonometry! You will require the `<cmath>` header, and

should use the `PI` constant defined in the provided `2DMotion.h` file. In order move a `Point` by a given `Force` you must determine the horizontal and vertical components of the given `Force`. Doing so is simple trigonometry. Consider that the magnitude of a `Force` is simply the hypotenuse of a right-angle triangle. Given the hypotenuse and angle of a right-angle triangle you can easily find out the length of the other sides (the horizontal and vertical components) using `sin` and `cos` provided in the `<cmath>` library - but be wary, those operations work on radians!

- An overloaded multiplication operator between a `Force` and an `int` scalar. This should simply produce a new `Force` which has a magnitude scaled by the given scalar.
- `int Point::quadrant()` - A member function that returns the quadrant (1, 2, 3, or 4) that the given point is in. See quadrants in diagram above.

A test harness is provided in the file `a3q2.cc`. **Make sure to read and understand this code, as you will need to know what it does in order to structure your test suite.** The test harness has commnets describing the commands.

Deliverables: For this question include in your final submission zip your C++ header file `2DMotion.h` and the implementation file `2DMotion.cc`

3. **Integer Sets — a full ADT this time**

For this question you may not use any STL container, nor may you use any STL smart pointers. You must implement the class in question by managing memory yourself. These headers are already banned from the assignment, but this is to remind you. You may create any helper classes you want to help yourself manage memory.

In this question, you will implement a class `intSet` that represents a mathematical set for integers (recall, a mathematical set means no duplicate values are included). The interface has already been given to you in the provided `intSet.h` file. You may add private helper methods or additional fields as you see fit, however you do not *need* to add additional fields (helper methods may be a good idea though).

The important part of memory management in this class is that your `add` method, which adds an integer to the set, must follow the following memory management scheme:

- A default constructed `intSet` should have an array large enough to store 4 `ints`, the `capacity` field should reflect this (your `size` field represents how many integers are actually in the array, and thus should be 0 for default constructed objects).
- When `add` is called and there is no more space in the current array (that is `capacity==size`), then you must **double** the size of the array. That is, you must allocate a new array twice the size of the old array, copy over all the old elements, and finally add your new `int` to the array. Of course, this must also update the `size` and `capacity` fields correctly.
- Of course, if `add` is called and there is still space in the array you simply need to add the `int` to the array and update the `size` field.

Recall though that `add` only actually adds the integer to the set if it doesn't already exist in the set. This can be achieved either by not adding the `int` when it already exists, or adding it but ignoring duplicates in future functions. The behaviour is up to you.

You must implement all the functions in the provided header, that is:

- A copy constructor, which must perform a deep copy so that each set maintains its data independently
- A copy assignment operator, which must perform a deep copy so that each set maintains its data independently
- A move constructor, which must efficiently steal data and not perform a deep copy
- A move assignment operator, which must efficiently steal data and not perform a deep copy
- A destructor, which must free all memory allocated within the `intSet`
- `operator|` which consumes two `intSet` objects and returns a new `intSet` object which represents the set union of those two sets. Set union of two mathematical sets is defined as a set which contains all elements which occur in **either** set
- `operator&` which consumes two `intSet` objects and returns a new `intSet` object which represents the set intersection of those two sets. Set intersection of two mathematical sets is defined as a set which contains all elements which appear in **both** sets
- `operator==` which consumes two `intSet` objects and returns true if they are equal sets, and false other. Two sets X and Y are equal if there does not exist an element in X that doesn't exist in Y, and there also does not exist an element in Y that does not exist in X. That is, they contain exactly the same elements - though ordering does not matter

- `isSubset` which is a method called with an `intSet` parameter. It returns true if the parameter is a subSet of the `intSet` the method was called on, and false otherwise. A set X is a subset of another set Y if every element that occurs in X also occurs in Y
- `contains` which is a method called with an `int` parameter. It returns true if the `int` parameter is a member of the set
- `add` which is a method called with an `int` parameter. It adds the `int` to the set (if the set already contained that `int` then it doesn't need to do anything), it must follow the memory management scheme specified above.
- `remove` which is a method called with an `int` parameter. It removes the `int` parameter from the set (if the set didn't contain that `int` then it doesn't need to do anything). You never need to shrink your array, so no matter how many elements are removed from a set you only ever change the contents of your array and your `size` variable, never changing `capacity`.
- A friend `operator<<` function which first prints a left parenthesis, then each integer in the set delimted by a comma and a space in ascending order and ends with a right parenthesis. For example the set containing 3, 5, and 2 is printed as: `(2, 3, 5)`

A test harness is provided in the file `a3q3.cc`. **Make sure to read and understand this code, as you will need to know what it does in order to structure your test suite.** The test harness has commnets describing the commands.

Memory Management Requirements: You must manage your own memory in this assignment, using `new` and `delete`. All copies of objects must be deep copies (can test this behavior in sample executable). Additionally your move constructor and move assignment operator must be constant time! You must also follow the memory scheme for growing your array as outlined above. Your code will be handmarked for these requirements, if you fail to meet them then you will lose some or all of your correctness marks.

Deliverables: For this question include in your final submission zip your C++ header file `intSet.h` and the implementation file `intSet.cc`

How to submit: Create a zip file `a3.zip`, make sure that zip file contains your C++ implementation and header files `conways.cc`, `2DMotion.h`, and `2DMotion.cc`. Assuming all five of these files are in your current working directory you can create your zip file with the command

```
$ zip a3.zip conways.cc 2DMotion.h 2DMotion.cc intSet.h intSet.cc
```

Upload your file `a3.zip` to the a3 submission link on eClass.