# CS247—Assignment 1 (Spring 2024)

R. Evans

Due Date 1: Friday May 24th, 5:00PM
Due Date 2: Friday May 31st, 5:00PM

**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 code 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.

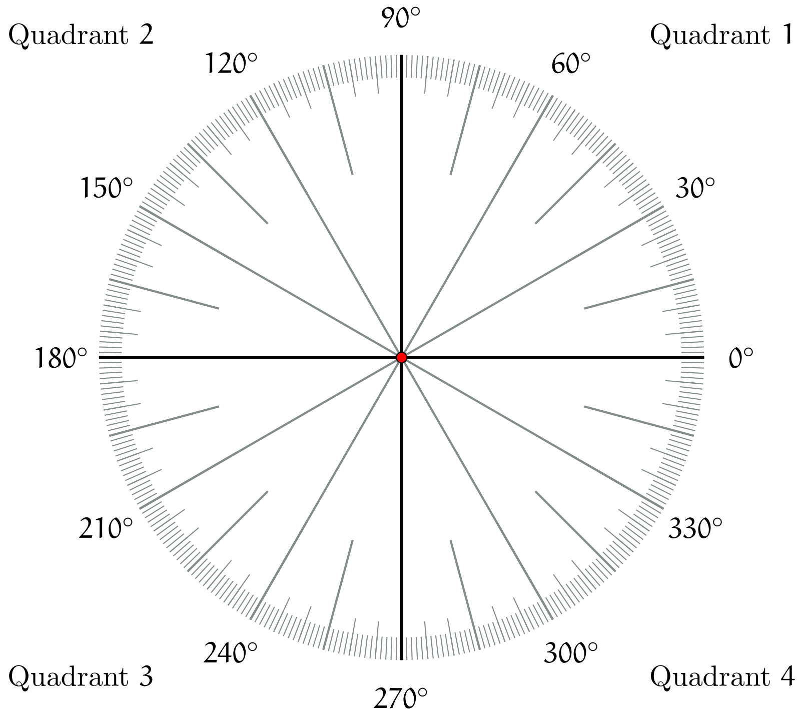
Some or all of these programs will ask you to develop C++ modules in order to make a provided test harness file which includes those modules work. Since you must use the main provided in the test harness to test your program, it is suggested that you first begin by providing empty implementations of 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.
Your programs will be tested against the provided sample executables, so you can do the same! Use the provided `produceOutputs` and `runSuite` tools to check if you are producing the same output as the sample executable. These tools are located in the `tools` folder in the course git repository. A PDF documenting their use is also included there.
It is suggested you create many input files for your specific test cases, and use this procedure to test your program. Even better, a bash script could be written to automate checking all of your tests! These assignments are in testing as much as they are in coding! Without thinking of the appropriate test cases and checking them it will be hard to get full marks! If you didn't think about a test case, even if it is not explicitly pointed out to you in the specification, that doesn't mean you deserve those marks.

**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. You code must compile and run correctly on the student server.

**Allowed headers for this assignment:** For this assigment you may only use the headers `<iostream>`, `<utility>`, and `<cmath>`. Inclusion of any other headers (not created by yourself) may result in a loss of some or all of your marks.

1. 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 `a1q1.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 $magnitude$ (both of which are also float values). 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 `a1q1.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`. Essentially, the force must be written in terms of its x and y components, and then added to the point. 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 to 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. Coordinates on the y-axis are considered to be in quadrants 1 or 4. Coordinates on the x-axis are considered to be in quadrants 1 or 2.

a) **Due on Due Date 1:** Design a test suite for this program. The test suite must be compatible with the `runSuite` format. Call your suite file `suiteq1.txt`. Zip your suite file, together with the associated `.in` files, into the file `a1q1.zip`.

b) **Due on Due Date 2:** Write the program in `C++`. You must include in your submission (`a1q1.zip`) your files `2DMotion.h` and `2DMotion.cc`, no other files other than our own test harness will be used to compile your solution - so do not create any additional files.

2. **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 yourself to help you manage memory.

Battleship is a two-player game in which players take turns guessing coordinates of a grid. Each player lays out rectangular "ships" on their own grid, and attempt to guess where the other player's ships are located.

In this assignment you will create some data structures to model Battleship. You will do so by implementing the `Battleship` class.

You have been provided a header for the `Battleship` class, which you may change as you like. However, you must implement all the functionality used by the provided test harness file `a1q2.cc`, as that file (or a similar one in how it uses the `Battleship` class) 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!

The required functions are listed below, for their exact behaviour refer to the sample executable:

- `Battleship(int, int)` - A basic constructor that takes in two positive integer parameters, the number of rows and the number of columns for the board size.
- A copy constructor, which must be a deep copy.
- A move constructor, which must be a constant time operation with regards to the size of the board.
- A copy assignment operator, which must be a deep copy.
- A move assignment operator, which must be a constant time operation with regards to the size of the board.
- A destructor, which must free all resources the object holds.

- An overloaded output operator that prints out the board, row by row, one row per line. It should use pipe characters '|' at both the left and right edges of each row, as well as inbetween spots. An untouched spot should be printed as a space character, a hit spot should be printed as 'X', and a missed spot should be printed as 'o'. Each row should end with a new line, including the last one.
- An overloaded input operator that is used to set up ship locations on the board. Ship locations are given by 4 space separated coordinates: `r1 c1 r2 c2`. `(r1, c1)` denotes the row and column where the ship starts, and `(r2, c2)` denotes where the row and column where the ship ends (inclusive). For example, a user may enter `1 3 1 5` to create a ship lying on row 1 column 3, row 1 column 4, and row 1 column 5. Ships are expected to have width 1 and height `s`, or width `s` and height 1 (where `s` is a positive integer). Ships must be placed entirely on the board, either horizontally or vertically, and may not intersect. Error checking must be performed for each of these constraints. The operator should stop reading input when either 5 ship coordinates have been read in, or when a non-numeric character is entered. Each time the input operator is called, all ships should be cleared, and the user should start with a "clean" board.
- `void Battleship::playTurn()` - A member function that allows one to take a turn by guessing coordinates. A user can enter pairs of coordinates to shoot at. Error checking must be performed to ensure that the coordinate is on the board and has not been guessed before. If a ship is hit, it is marked, and the user can guess again. The function returns either on the first miss, or when a non-numeric character is entered.

**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! 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.

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` files, into the file `a1q2.zip`.

b) **Due on Due Date 2:** Write the program in `C++`. You must include in your submission (`a1q2.zip`) your files `Battleship.h` and `Battleship.cc`, no other files other than our own test harness will be used to compile your solution - so do not create any additional files.