Data Structures and Algorithms

Project Requirements

Objectives

By the end of this project, the student should be able to:

- Understand unstructured, natural language problem description and derive an appropriate design.
- Intuitively modularize a design into independent components and divide these components among team members.
- Build and use data structures to implement the proposed design.
- Write a **complete object-oriented C++ program** that performs a non-trivial task.
- Use third-party code libraries as parts of the project

Introduction

Back to the Middle Ages, assume you are the liege of a castle that is protected by 4 towers where every tower is required to protect a certain region (See Fig 1). Every day some enemies attack the castle and they want to destroy your towers. You need to use your programming skills and knowledge to data structures to write a ***simulation program*** of a game between your castle towers and enemies. You should simulate the war between the towers and the attacking enemies then calculate some statistics from this simulation.

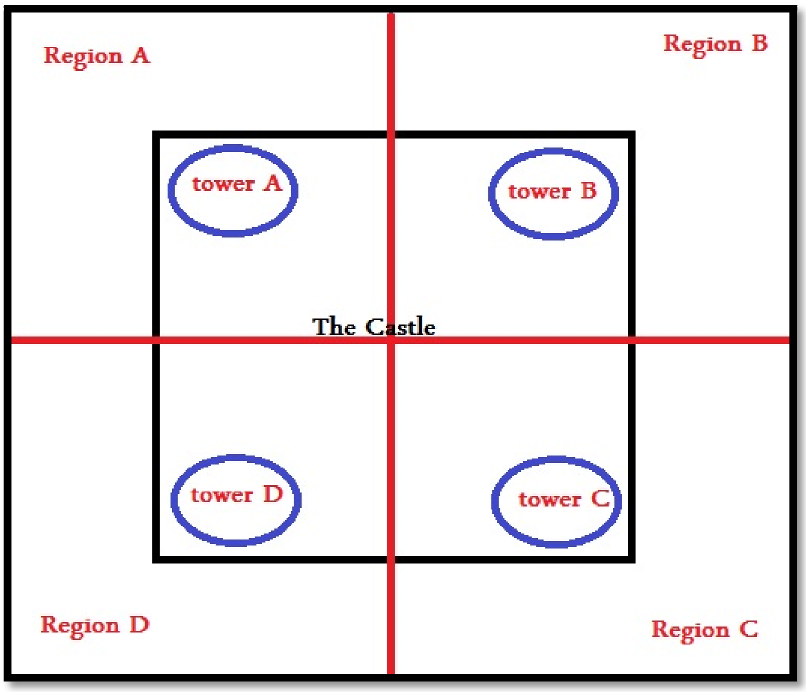

Figure 1the castle

Project Phases

Project Phase	*%*	*Deadline*
Phase 1	**35%**	**Week 11**
Phase 2	**65%**	**Week 14**

NOTE: Number of students per team = 3 students.

The project code must be totally yours. The penalty of cheating any part of the project from any other source is not ONLY taking ZERO in the project grade but also taking MINUS FIVE (-5) from other class work grades, so it is better to deliver an incomplete project other than cheating it. It is totally your responsibility to keep your code away from others.

Note: At any delivery,

One day late makes you lose 1/4 of the grade.

Two days late makes you lose 1/2 of the grade.

Problem description

Four towers are defending the castle. Each tower guards one region and can shoot only enemies **in its region**. Each tower has a starting health and can shoot at most **N** enemies at each time step.

Your system (your program) will receive the information of a list of enemies as input from an input file. This list represents the scenario to be simulated. For each enemy the system will receive the following information:

- **Arrival Time stamp (Enemy Arrival Time):** When the enemy arrives.
- **Health:** The starting health of the enemy.
- **Power:** The power of the enemy.
- **Reload Period:** Time for an enemy to reload its weapon. During reload period, an enemy cannot fight/heal but can move. After each *action time*, the enemy have to wait the reload period to be able to attack again.
- **Type:** ***fighter*** or ***healer (healers should be <20% of total enemies)***
- **Region:** The attack region of the enemy.

Simulation Approach & Assumptions

You will use incremental time simulation. You will divide the time into discrete time steps of 1 time unit each and simulate the changes in the system in each time step.

Definitions

- **Enemy Types:** There are three types of enemies: fighter, healer, and freezer
  - Fighter can attack tower and cause damage to tower.
  - Healer can heal other enemies and does not attack tower.
  - Freezer can attack the tower and cause it to freeze for some time.

- **Freezer Enemy:**
  This type of enemies throws ice one the tower it is attacking. When the accumulated ice on a tower reaches certain threshold, the tower is frozen **for one time step**.

- **Enemy State:**
  At any time, an enemy should be in one of four states: ***inactive*** (not arrived yet), ***active, frozen*** (both are described below) or ***killed*** (health <= 0). Only active enemies can act (fight/heal/freeze).

- **Active Enemy:** is an enemy with Arrival Time ≤ current time step & Health > 0.

- **Frozen Enemy:**
  Frozen enemies cannot move or attack towers. Any active enemy can be frozen for some time due to an attack from a tower.

- **Enemy distance:** is the ***horizontal*** distance between the enemy and the tower of its region.

- **Enemy Action step:** is the time step an enemy can attack tower or heal other enemies. The action step of any type of enemies starts at the same time step of **their arrival**. Then, they will wait the reload time without action.
  - **For example,** *Enemy e1(arrival time=5, reload period=3)* will act at time step 5 then wait to reload at time steps: 6, 7 and 8 then it will act again at time step 9 then wait.

Game Rules

- A tower can attack **only** the enemies of its region.
- An enemy can attack **only** the tower of its region.
- A tower can attack any enemy type.
- A tower can attack at most **N** enemies at each time step.
- A tower can either fire bullets or fire ice to freeze an active enemy.
- At ***random time steps*** the tower fires ice instead of bullets. The percentage of ice fire should ***never exceed 20%*** of all tower fires.
- If an active enemy is hit by tower ice, it is frozen for some time steps.
- A frozen tower cannot attack enemies.
- A frozen tower is affected by fighter attacks only. Freezer enemy has no effect on it.
- Number of healers should never exceed **20%** of total enemies.
- All enemies start at **60 meters** distance from the castle.

- In general, enemies move **one meter each time step**. But enemies with health less than half their original health move **one meter each 2 time steps**.
- The minimum possible distance for any enemy to approach to the tower is **2 meters**.
- Enemies can move at their reload period but cannot act. (cannot fire nor heal)
- Healer can heal only enemies that are at most 2 meters away (forward or backward)
- Healer cannot heal killed enemies.
- The game is "**win**" if all enemies are killed. The game is "**loss**" if the all towers are destroyed. Otherwise, the game is "**drawn**".
- **If a tower in a region is destroyed**, all enemies (current and incoming enemies) in that region should be transported to the next region. The next region means the adjacent region moving clockwise. (A → B → C → D → A).
- If an enemy is transported to the next region, it should be placed **at the same distance** it reached in the previous region.

Enemies picking criteria:

As mentioned in the game rules, a tower picks at most **N** enemies to attack at each time step. The criteria to pick an enemy to attack should depend on **at least three** of the following factors: Enemy distance, type, health, power, state(active or frozen), remaining time steps for an enemy to end reload period, and most recent enemies attacking tower.
Think about a suitable formula to give a priority for each enemy then choose N enemies with highest priority.

Formulas:

- **Damage to a tower by a fighter**

$$D_{ET} = \text{Damage (fighter} \rightarrow \text{Tower)} = \frac{K}{Enemy\_distance} * Enemy\_power$$

K = 1 for healthy enemies, **K=0.5** for enemies with health less than half their original health

Note: If an enemy is not allowed to fire at current step (during reload period), it will not cause any damage to the tower.

- **Think about a formula for amount of ice a freezer can throw on a tower.**
Take into account that the freezing threshold of a tower can never be reached by a single ice shot from an enemy.

- **Damage to an enemy by a tower bullet (not applicable for ice fires)**

$$D_{TE} = \text{Damage (Tower} \rightarrow \text{Enemy)} = \frac{1}{Enemy\_distance} * Tower\_fire\_power * \frac{1}{K}$$

Use **K=2** for healers and **K=1** for other enemies

- **Think about a formula for the time steps an enemy gets frozen for when it is hit by an ice shot from the tower.**

- **Think about a formula for healers to increase the health of other enemies**

- ❑ **First-ShotDelay (FD)**
  The time elapsed until an enemy is first shot **by a tower**
  $$FD = T_{first\_shot} - T_{arrival}$$
- ❑ **KillDelay (KD)**
  The time elapsed between first time a tower shoots the enemy and its kill time
  $$KD = T_{enemy\_killed} - T_{first\_shot}$$
- ❑ **Lifetime (LT)**
  The total time an enemy stays alive until being killed
  $$LT= FD+KD = T_{enemy\_killed} - T_{arrival}$$

Bonus Criteria (maximum 10%)

- **[5%] Different towers with reconstruction ability.**
  Each tower should have its own initial health, power and N (number of enemies to fight at each time step). When a tower is damaged, it can be **reconstructed** by redistributing health of all remaining towers among the four towers. In this case, enemies are NOT transported to another region when a tower is damaged.

- **[5%] More enemy types:** Think about **two** more enemy types other than those given in the document (**and other than shielded fighter or paver**). The load of the logic of the two enemies must be acceptable; not too trivial.

Program Interface

The program can run in one of two modes: **interactive mode** or **silent mode**.

When the program runs, it should ask the user in the console window to select the program mode.

Interactive mode allow user to monitor the fighting between the active enemies and the castle as time goes on. At each time step, the program should provide output similar to that in the following figure (Figure 2) on the screen. The program pauses for <u>a user mouse click</u> to display the output of the next time step.

At the bottom of the screen, the following information should be printed:

- Simulation Time Step Number
- **<u>For each region,</u>** print: ***[Note that the following information is <u>for each region</u>.]***
  - Current health of its tower
  - Total number of active enemies (how many fighters, healers)
  - Total number of killed enemies from the beginning of simulation in this region

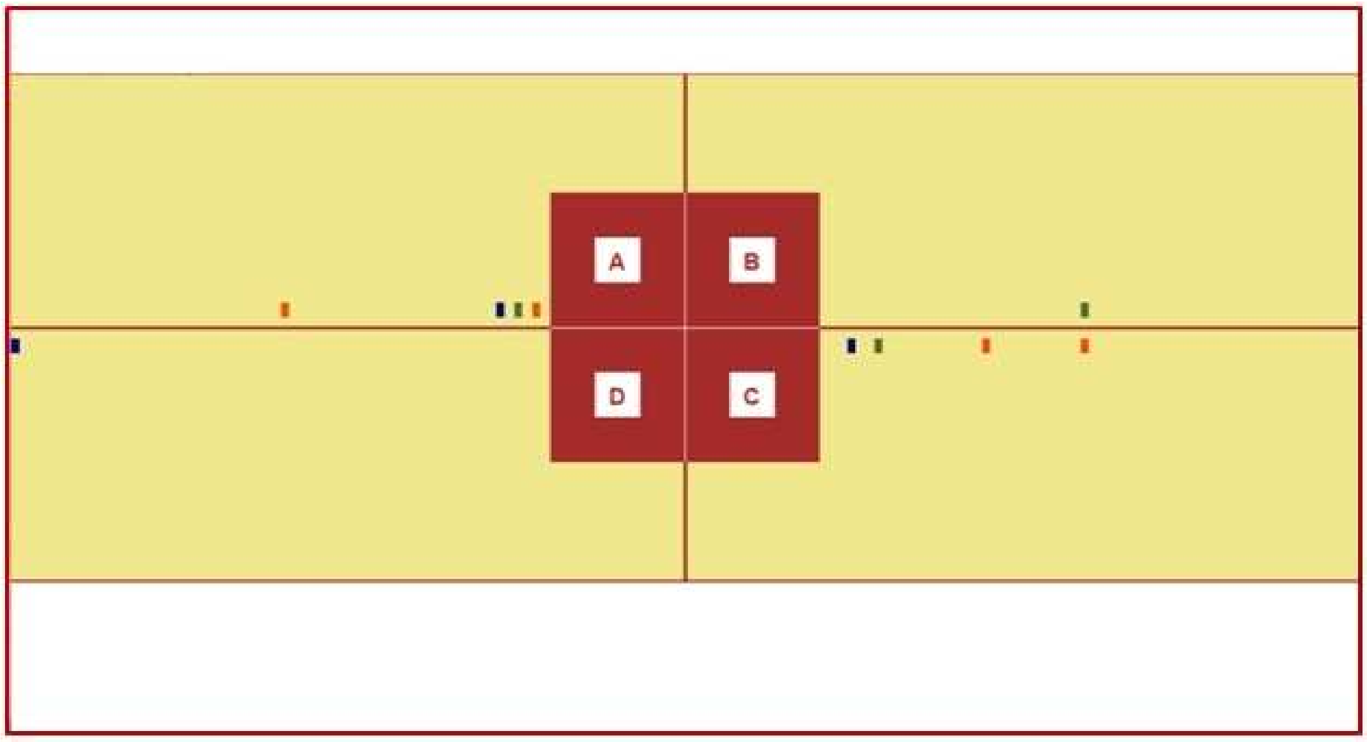

Figure 2Program Output

In **silent mode**, the program produces only an output file (See the “File Formats” section). It does not draw the enemies on the screen or simulate the fighting graphically.
No matter what mode of operation your program is running in, **the output file** should be produced.

You are provided a code library (set of functions) for drawing the above interface.(see Appendix A)
Note: Each enemy type should be drawn using a **different color**

File Formats

Your program should receive all information to be simulated from an input file and produces an output file that contains some information and statistics about the simulation. This section describes the format of both files and gives a sample for each.

The Input File

- First line containsthree integers: **TH N TP**
  TH is the starting health of all towers; **N** is the maximum number of enemies a tower can attack at any time step and **TP** is the tower power.
- Then the input file contains many lines (one line for each enemy) of the format **S TYP T H POW RLD REG**
  where **S** is a sequence number that identifies the enemy (must be unique),**TYP** is the enemy type (*1:fighter and 2:healer*),**T** is the enemy arrival time, **H** is the enemy health, **POW** is the enemy power, **RLD** is the enemy reload period and **REG** is the enemy region. The input lines are **sorted by arrival time** in ascending order.
- The last line in the input file should be **-1** which indicates the end of input file.

The Output File

The output file you are required to produce should contain **M** output line of the format **KTS S FD KD LT**
which means that the enemy identified by sequence number **S** is killed at time step **KTS** and its first-shot delay is **FD** and kill delay is **KD** and total enemy lifetime is **LT**.
The output lines **should be sorted** by **KTS** in ascending order. If more than one enemyare killed at the same time step, **they should be ordered by FD**.

A line at the end of the file should indicate the total damage for each tower by the attacking enemies.
T1_Total_Damage T2_Total_Damage T3_Total_Damage T4_Total_Damage

Another line for string of **“Game is WIN/LOSS/DRAWN”**.

Then the following statistics should be shown at the end of the file

1- In case of game “**win**”
   a. Total number of enemies
   b. Average “First-Shot Delay” and Average “Kill Delay”
2- In case of game “**loss/drawn**”
   a. Number of killed enemies
   b. Number of alive enemies (active and inactive)
   c. Average “First-Shot Delay” and Average “Kill Delay” for killed enemies only

Sample Input File

```
200  3     14
1   1      1      10     2      4      A
2   2      3      15     5      4      A
3   1      7      15     2      3      B
-1
```

The above file initializes the towers with health 200, each tower can attack at most 3 enemies at every time step and their power is 14.
Then enemies' details:

- An enemy number 1 of **type=1 (fighter)** arrived at time step 1 in region A with Health =10 and power = 2 and Reload_period= 4.
- An enemy number 2 of **type=2 (healer)** arrived at time step 3 in region A with Health=15 and power = 5 and Reload_period=4
- An enemy number 3 of **type=1 (fighter)** arrived at time step 7 in region B with Health=15 and power = 2 and Reload_period=3.

Sample Output File

The following numbers are just for clarification and are not produced by actual calculations.

```
KTS    S       FD   KD      LT
  5      1       0     5      5
 10     2       4     4      8
 15     3       5     2      7
 T1_Total_Damage   T2_Total_Damage   T3_Total_Damage   T4_Total_Damage
 33.5                    12.5                   55                    200


 Game is WIN
 Total Enemies                    = 50
 Average First-Shot Delay         = 4.5
 Average Kill Delay               = 12.36
```

The second line in the above file indicates that enemy with sequence number 1 killed at time step=5 and it took FD=0, KD= 5 and **LT=5**

The last four lines indicate that you won the game, total enemies=50, average First-Shot Delay= 4.5, and average kill delay=12.36

Project Phases

You are given a partially implemented code that is implemented using C++ classes. You are required to extend the given code and write an **object-oriented** code to complete the project. The graphical user interface GUI for the project is almost all implemented and given. You just need to know how to call its functions to draw the interface items.

Phase 1:

In this phase you should finish all simple functions that are **NOT involved in fighting logic nor statistics calculations.**

The required parts to be finalized and delivered at this phase are:

1- **Full data members** of Enemy (and subclasses), Tower, and Castle Classes.
2- **Full implementation data structures** that you will use to represent **the lists of enemies**.

Important: Keep in mind that you are **NOT** selecting the DS that would **work in phase1**. **You should choose the DS that would work efficiently for both phases.**

Guidelines for suitable data structure selection:

a. Do you need a **separate** list for each enemy state? Why?
b. Will you use **one** list for all regions or **a separate** list for each region? Why?
c. Do you need to store **killed** enemies? In separate lists or in one list? When should you get rid of them to save memory?
d. **Which list type** is much suitable to represent the lists taking into account the **complexity of the operations** needed for each list (e.g. insert, delete, retrieve, shift, sort …etc.).You may need to make a survey on the complexity of the operations of each list type. Then, for each enemy list, decide what are the most frequent operations needed according to the project description. Then, for this list, choose the DS with best complexity regarding those frequent operations.

Selecting the appropriate DS for each list is the core target of phase 1 and the project as a whole. Most of phase1 discussion time will be about it.

Note: you need to read *"File Format"* section to see how the input data and output data are sorted in each file because this will affect your DS selection.

All the above data structures should be fully implemented at this phase. You are not allowed to use STL or the code of any external resource (considered as cheating). One basic objective of the course is to build and maintain data structures from scratch.

3- **File loading function.** The function that reads input file to:

a. Load Towers data and constants values

b. Create and populate (fill) inactive enemies list.

4- **Simple Simulator function for Phase 1**. The main purpose of this function is to test your data structures and how to move enemies between lists of inactive, active and killed lists if any. This function should:
   a. Perform any needed initializations
   b. Call file loading function
   c. At ***each time step*** do the following:
      i. Move active enemies from inactive to active list(s) according to arrival time.
      ii. Pick at most **4 random** active enemies to kill.
      iii. Remove *killed enemies* (the enemies killed randomly from the previous step) from the active list(s) and move them to the killed list(s) or just delete them (according to your previous decision about keeping them or deleting them).
      iv. **For each region**, print in the status bar:
         1. Information of its tower.
         2. Total number of current active enemies.
         3. Total number of killed enemies so far.

Notes about phase 1:
- No output files should be produced at this phase.
- You should draw the enemies in the GUI and print the required information on the status bar at each time step.
- In this phase, you can go to the next time step by mouse click.
- No fighting or healing logic is needed at this phase.

Phase 1 Deliverables:
Each team is required to deliver a **CD** that contains:
- A text file named ***ID.txt*** containing team members' names, IDs, and emails.
- **Phase 1 full code** [Do not include executable files, .sdf files nor ipch folders].
- **Three Sample input files** (test cases).
- Write your team number team members name on the CD cover paper.

Phase 2: In this phase, you should extend code of phase 1 to build the full application and produce the final output file. Your application should support the different operation modes described in "Program Interface" section.

Phase 2 Deliverables:
Each team is required to deliver a **CD** that contains:
- A text file named ***ID.txt*** containing team members' names, IDs, and emails.
- **Final Project Code** [Do not include executable files, .sdf files nor ipch folders].
- **Six** Sample input files to cover different cases ranging from **weak-enemy-weak-castle case** to **strong-enemy-strong-castle case** and their output files.
- **A project document** describing workload distribution among team members.
- Write your team number on the CD cover paper.

Project Evaluation

Evaluation Criteria

These are the main points that will be graded in the project:

- **Successful Compilation:** Your program must compile successfully with zero errors. Delivering the project with any compilation errors will make you lose a large percentage of your grade.

- **Object-Oriented Concepts:**
  - **Modularity:** A **modular** code does not mix several program features within the same unit (module). For example, the code that does the core of the simulation process should be separate from the code that reads the input file which, in turn is separate from the code that implements the data structure. This can be achieved by:
    - adding classes for each different entity in the system, for example, each DS has its class and each enemy type is in a different class that inherits from Enemy class, …etc.
    - dividing the code in each class to several functions. Each function should be responsible for a single job. Avoid writing very long functions that does everything. Divide each task into sub-tasks, add a function for each sub-task and make the major-task functions calls the functions of each sub-task in the appropriate order.
  - **Maintainability:** A maintainable code is the one whose modules are easily modified or extended without a severe effect on other modules.
  - **Separate each class in .h and .cpp.**
  - **Encapsulation.**
  - **Class Responsibilities:** Each class does its job. No class is performing the job of another class.
  - **Polymorphism:** use of pointers and virtual functions.

- **Data Structure & Algorithm**: After all, this is what the course is about. You should be able to provide a concise description and a justification for: (1) the data structure(s) and algorithm(s) you used to solve the problem, (2) the **complexity** of the chosen algorithm, and (3) the logic of the program flow. If any data structure is used for different types, write them once by using ***<u>templates</u>***.

- **Interface modes**: Your program should support the two interface modes described in the document. The existence of one mode does not compensate the absence of another.

- **Test Cases**: You should prepare comprehensive test cases (at least 6) that range from weak to strong castle and enemies (e.g. weak-castle-moderate-enemy case). Your program should be able to simulate different scenarios not just trivial ones.

- **Coding style**: How elegant and **<u>consistent</u>** is your coding style (indentation, naming convention ...etc)? How useful and sufficient are your comments? This will be graded.

These are NOT allowed to be used in your project:

- You are not allowed to use **C++ STL** or any external resource that implements the data structures you use.

- You are not allowed to use **global variables** in your implemented part of the project, use the approach of passing variables in function parameters instead. That is better from the software engineering point of view. Search for the reasons of that or ask your TA for more information.

- You need to get instructor approval before using **class friendship.**

Notes:

- ❑ The code of any operation does NOT compensate for the absence of any other operation.
- ❑ There is no bonus on anything other than the points mentioned in the bonus section.

Individuals Evaluation (IG):

Each member must be responsible for writing some project modules (e.g. some classes or some functions) and must answer some questions showing that he/she understands both the program logic and the implementation details.

You should **inform the TAs** before the deadline **with a sufficient time (some weeks before it)** if any of your team members does not contribute in the project work and does not make his/her tasks. The TAs should warn him/her first before taking the appropriate grading action.

Note: we will reduce the IG in the following cases:

- ❑ Not working enough (unfair load distribution)
- ❑ Neglecting or preventing other team members from working enough

See Appendix A in the Next Page

Appendix A–Guidelines on the Provided Framework

The main objects of the game are the castle, the towers and the enemies. There is a class for each of them.**Castle class** contains an array of the 4 towers. **Enemy class** is the base class for each type of enemies. You should add a class for each enemy type and make it inherit from the enemy class. Enemy class should later be an abstract class that contains some pure virtual functions (e.g. move, attack, …etc.).

Battle class contains an object of the castle and an array of pointers for all enemies in all regions, *BEnemiesForDraw*. This enemies array will be used just for drawing. We will explain this more below. Battle class is ***the controller*** of the game. It will contain the functions that connect the classes with each other and the sequence of calls that perform the game logic.

GUI class contains the input and output functions that you should use to show game progress. The given code already draws the regions, castle and enemies. It contains a function called, "*DrawEnemies*", which uses the enemies array of pointers, *BEnemiesForDraw,* existing in the Battle class to draw all the enemies. Here is the description of this function:

void GUI::DrawEnemies(Enemy* enemies[],int size) const

This function draws ALL active enemies in the game in all regions. It handles when there are more than one enemy in the same region in the same distance by drawing them vertical to each other.

Inputs: enemies [ ] ➔ an array of enemy pointers. ALL <u>active</u> enemies from ALL regions should be pointed to by this array in any order. It exists inside class Battle. This array needs to be updated each time step to draw the current active enemies after the last changes.
size ➔size of the enemies' array (Total number of active enemies).

[Important Note]: Taking an array of pointers to enemies does NOT necessarily mean to choose the array of pointers as the data structure for the enemies' lists. At every time step, you should **update** those pointers of the array to point to the current active enemies that exist in whatever data structure you chose (pointing to the already allocated enemies not reallocating them or changing the data structure) then pass the pointers' array to *DrawEnemies* function. No matter what data structure you are using to hold enemies list, you must pass the enemies to the above function as an array of enemy pointers.

Finally, a demo code (***Demo_Main.cpp***) is given just to test the above functions and show you how they can be used. It does noting with phase 1 or phase 2. You should write your main function of each phase by yourself.

Important implementation guideline:

<u>Do NOT allocate the same enemy more than once</u>. Allocate it once and makes whatever data structures you chose points to it (pointers). Then, to move it from a list to another, just don't make the old list point to it and the new one point to it.