Cairo University
Faculty of Engineering
CMP 102 & CMP N102

Spring 2018

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.

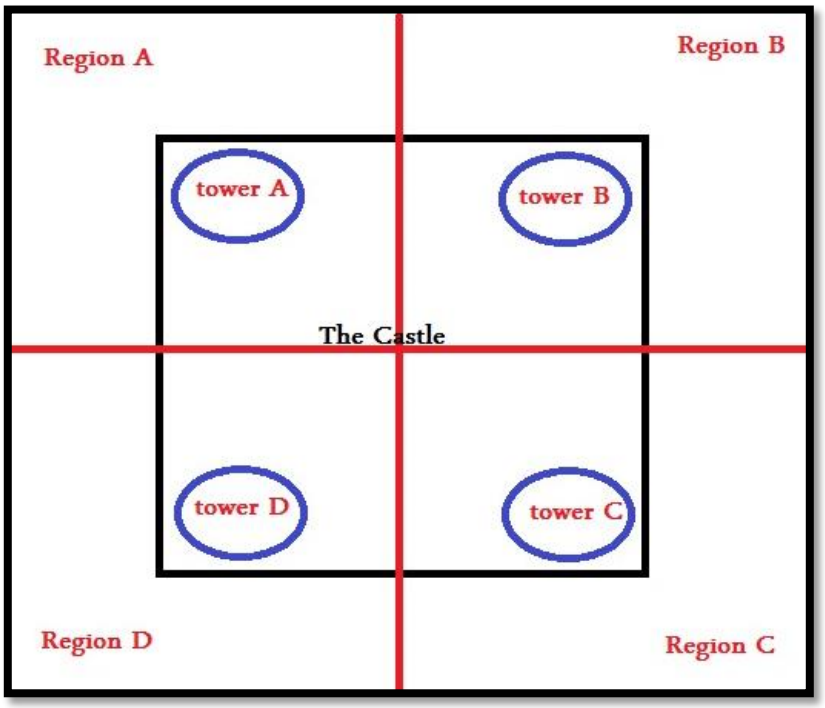

Figure 1 the castle

Project Phases

Project Phase	*%*	*Due Date*
Phase 1	**35%**	**Week 9**
Phase 2	**65%**	**Week 15**

NOTE: Number of students per team = **3 to 4 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 1/2 of the grade.

Two days late makes you lose 3/4 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.
- **Fire Power:** The shot hit power of the enemy.
- **Reload Period:** Time for an enemy to reload its weapon. During reload period, an enemy cannot fight but can move. After each *attack time*, the enemy have to wait the reload period to be able to attack again.
- **Type:** Three types of enemies: ***paver***, ***fighter*** and ***shielded fighter***
- **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.

Some Definitions and formulas

- **Enemy State:**
  At any time, an enemy should be in one of three states: ***inactive*** (not arrived yet), ***active*** (described below) or ***killed*** (health = 0). Only active enemies can fight.

- **Active Enemy** is an enemy with Arrival Time ≤ current time step & Health > 0.
  At each time step, each tower should choose ***N*** active enemies to shoot (***N*** is given in the input file).

- **Enemy distance** is the ***horizontal*** distance between the enemy and the tower of its region. All enemies start at **60 meters** distance from the castle. The minimum possible distance for any enemy to approach to the tower is **2 meters**. After that it may fight but not move.

- **Attack Time** is the time step before each reload period. **The attack time** of any type of enemies (including pavers) starts at the same time step of **their arrival**. Then, they will wait the reload time without attacking. After that they will attack in the next time step then wait the reload and so on.

  - **For example,** *Enemy e1 (arrival time = 5, reload period = 3)* will attack at time step 5 then wait without attacking at time steps: 6, 7 and 8 then it will attack at time step 9 then wait and so on.

- **Paver Enemy**
  All enemies can approach the castle **one meter** at every time step ***only if the next meter is paved***. At the start of the simulation, the last **30 meters** in the way to the castle (the 30 meters nearest to the tower of each region) are not paved. Only paver enemies can enter the unpaved distance (only in their attack time) to pave it so that enemies of other types can enter this paved distance later in the **next** time steps. Here are some notes about pavers:

  - A paver **does not shoot** the towers. It only paves. Its "Fire Power" represents the number of meters it can pave (and move) at each ***allowed*** attack time step (not reload time).

  - In the ***"attack time"***, in addition to moving, *fighters and shielded fighters shoot the tower but pavers pave*.

  - The enemies can approach to the castle **one paved meter** at every **time step**. This includes **paver enemies during their reload period.**

  - **During the attack time of the paver (not in the reload period),** the paver can approach to the castle number of meters equal to its fire power **regardless of being in a paved or not paved area.**

- **Example 1,** if a paver enemy *p1 (arrival = 5, reload = 3, fire power = 2)*, p1 arrives at time step 5 to distance 60. The first attack time is the arrival 5, so p1 will move with his fire power (2) to reach the distance 58. Then, p1 will wait his reload period, time steps 6, 7 and 8 while moving one paved meter (like other enemy types) in each reload time step, so it will reach distance 55 at time step 8. The next attack time is 9 in which p1 will move with his fire power and so on.

- **Example 2,** if a region is paved until distance 25 by a paver and this paver is killed. Then, another paver enemy *p2 (reload = 3, fire power = 2)* arrives and moves until it reaches distance 25 which was the last paved distance in its region and this is its attack time, so it will move and pave with his fire power which will make him at distance 22 and make the last paved distance of its region is 22 and so on.

- **Damage to a tower by a certain enemy in its region**

$$DT = Damage\ (Enemy \rightarrow Tower) = \frac{K}{Enemy\_distance} * Enemy\_fire\_power$$

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

- **Damage to a certain enemy by the tower of its region**

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

Use K=2 for **shielded enemies** and 1 for other enemies, so shielded enemies do a larger damage on the towers (double damage) and are less damaged by the towers (half damage).

Note: Not all the active enemies are affected by this damage at each time step. Only the **N** active enemies shot by the tower in each time step are the enemies affected.

Note Formula **Damage(Enemy → Tower)** described above is not applicable for pavers. However, Formula **Damage(Tower → Enemy)** is applicable for them.

- **Enemy Priority**
  As mentioned before, a tower can shoot **N** active enemies **at each time step**.
  - If there are **no active shielded** enemies, the tower picks the next active enemy to shoot based on **its arrival time** according to FCFS (First Come First Serve) criterion.
  - **Active Shielded** enemies have higher priority than other enemies and should be shot first regardless of their arrival time.
  - If there are **more than one active shielded enemy**, each of them is given a **priority** according to the next formula:

$$Priority\ (Shielded\ Enemy) = \frac{Enemy\_Fire\_Power}{Enemy\_Distance} * C1 + \frac{C2}{Enemy\_remaming\_time\ \_to\_shoot+1} + Enemy\_health * C3$$

(where C1, C2 and C3 are three constants that are read from the input file.)
Shielded fighters with higher priority value should be shot first.

- **First-Shot Delay (FD)**
  The time elapsed until an enemy is first shot **by a tower**

$$FD = T_{first\_shot} - T_{arrival}$$

  Note: Not necessarily the enemy is first shot, by a tower, at the same time step it arrives because it may be not chosen as one of the N enemies that should be shot by the tower in this time step.

- **Kill Delay (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

$$LT = FD + KD = T_{enemy\_killed} - T_{arrival}$$

Assumptions

- Every tower can attack **only** the enemies of its region.
- Every enemy can attack **only** the tower of its region.
- At each time step, enemies:
  1. **Reload** if in reload period **OR Attack** (shoot or pave) if in the attack time.
  2. **Move** with the conditions mentioned before.
- The game is "**win**" if all enemies are killed.
- The game is "**loss**" if the all towers are destroyed.
- If all towers and all enemies become all destroyed at the same time step, you can assume that it is a "**win**" case too.
- **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 in the clockwise direction. (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. If such distance is not paved, it should be placed **at the closest paved distance** to the new region tower.

Bonus Criteria (maximum 10%)

- **[2.5%] Enemies speed:** Handling enemies with different speed; different number of meters they can move in one time step. Speed of each enemy should be added to the input file.
- **[2.5%] Using Sounds and Coloring the Unpaved Area:** Using sounds in the application, for example, when an enemy is killed or is paving, …etc. and Drawing the unpaved area of each region in a different color and updating it with each time step.
- **[5%] More enemy types:** Think about **two** more enemy types other than those given in the document. The load of the logic of the two enemies must be acceptable; not too trivial.

Program Interface

The program menu bar should contain **an icon** for each of the 3 modes of the application: **interactive mode**, **step-by-step mode** and **silent mode**.

Interactive mode and **step-by-step mode** allow you 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. In **interactive mode**, the program pauses for a user mouse click to display the output of the next time step. In **the step-by-step mode**, the program waits (sleeps) for 1 second to display the next step (without a user click).

In the status bar of the 2 modes mentioned above, the following information should be shown:

- Simulation Time Step Number
- **For each region,** print: ***[Note that the following information is for each region.]***
  - Current health of its tower
  - total number of active enemies
  - total number of killed enemies from the beginning of simulation in this region
  - unpaved distance to castle in this region

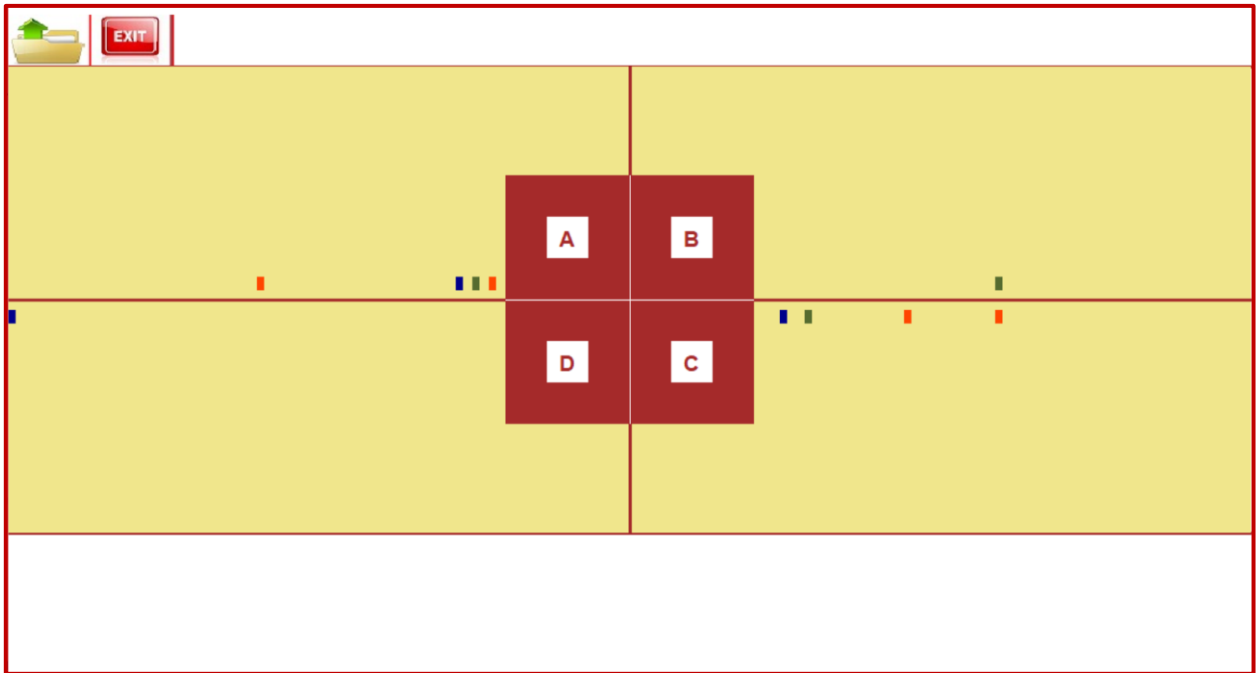

Figure 2 Program Output

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.

Note: Each enemy type should be drawn using a **different color** *(DARKBLUE for pavers, ORANGERED for shielded fighters and DARKOLIVEGREEN for normal fighters).*

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 contains three 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 fire power.
- Second line contains **C1 C2 C3** the constants of equation "**Priority (Enemy)"**
- Then the input file contains many lines (one line for each enemy) of the format
  S TYP T H POW RLD R
  where **S** is a sequence number that identifies the enemy (must be unique), **TYP** is the enemy type (*0:paver, 1:fighter and 2:shielded*), **T** is the enemy arrival time, **H** is the enemy health, **POW** is the enemy fire power, **RLD** is the enemy reload period and **R** 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.
- Your application should read the **input filename** from the user.

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 enemy are 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 should indicate the remaining unpaved distance in each region.
R1_Unpaved_Dist R2_Unpaved_Dist R3_Unpaved_Dist R4_Unpaved_Dist

Another line for string of **"Game is WIN" or "Game is LOSS"** for the game.

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**"
   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  0.03  0.01
1  0  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 with fire power is 14.
The second line means that constants C1=1, C2=0.03 and C3=0.01
Then enemies' details:

- An enemy number 1 of **type=0 (paver)** arrived at time step 1 in region A with Health =10 and Fire_power = 2 and Reload_period = 4.
- An enemy number 2 of **type=2 (shielded fighter)** arrived at time step 3 in region A with Health=15 and Fire_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 Fire_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
R1_Unpaved_Dist   R2_Unpaved_Dist   R3_Unpaved_Dist   R4_Unpaved_Dist
    30                30                25               2
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 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 you should add your phase 1 and phase 2 codes to. It is implemented using classes. You are required to write **object-oriented** code. The graphical user interface GUI for the project is almost all implemented and given to you. However, you need to add few more things mentioned below.

Phase 1:

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

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

1- Adding an icon for each of the 3 modes in the **Menu Bar**.

2- **Full data members** of Enemy, Tower, and Castle Classes.

3- **Full implementation data structures DS** 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 with better complexity for phase2.

Logically, any enemy should be in one of four states: ***inactive, active, high-priority active,*** or ***killed***.
So, when choosing the DS think about the following:

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. How will you store the **high priority enemies**?
d. Do you need to store **killed enemies**? In a **separate list or in one list**? When should you get rid of them to save memory?
e. **Which list type** is much suitable to represent the lists taking into account the **complexity of the main operations** needed for each list (e.g. insert, delete, retrieve, shift, sort …etc.). For example, if a list is sorted a lot, you should choose a data structure that is better in sort complexity. You may need to make a survey on the complexity of each of the main operations in each data structure. Then, for each enemy list, decide what is the most frequent operation needed in it according to the project description. Then, for this list, choose the DS with best complexity in this frequent operation.

You need to justify why you chose each DS and justify your decision about any list separation or joining. Selecting the appropriate DS for each list is the core target of phase 1 and the project as a whole. Most of the discussion time will be on that.

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). You are in data structures course where we teach you how to build data structures by yourself from scratch.

4- **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.

5- **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 active enemies and information of each one of them.
         3. Total number of killed enemies and information of each one of them.

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 in each time step.**
- **In this phase, you can go to the next time step by mouse click (until you implement the 3 modes in phase 2).**
- **No fighting, paving or moving logic is needed at this phase.**

Deliver a CD that contains Phase1 code and three sample input files.

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].
- **Three Sample input files** (test cases).
- **Phase 1 document** with 1 or more pages describing:
   1. each enemy list you chose
   2. the DS you chose for each list
   3. your justification of all your choices with the complexity of the most frequent or major operation for each list.
- Write your team number and team members name on the back of the CD cover.

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].
- **Six Sample input files (test cases) and their output files.**
- **A project document** with 2 or more pages describing your solution modules and any clever or innovative alternatives you followed in implementing the solution.
- Write your team number on the back of the CD cover.

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. You are not allowed to deliver the project with any compilation errors. Delivering the project with any compilation errors will make you lose a large percentage of your grade depending on the evaluation of the instructor.

- **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 ***templates***.

- **Interface modes**: Your program should support the three 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 **consistent** 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 in data structures course where we teach you how to build data structures by yourself from scratch.***

- 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 **friendships.**

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.
- ❑ **Each of the above requirements will have its own weight. The summation of them constitutes the group grade (GG).**

Individuals Evaluation:

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. Each member will get a percentage grade (**IG**) from the group grade (**GG**) according to this evaluation.

The overall grade for each student will be the product of GG and IG.

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
- ❑ 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 need to update the graphical user interface. 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 active enemies from ALL regions should be pointed 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.

There is an important general guideline for implementation. ***Do NOT allocate the same enemy more than once***. Allocate it once and makes whatever data structures you chose points to it (pointers). Then, to move it from a list to the other, just make the old list doesn't point to it and the new one points to it.