University of Science and Technology
Communication and Information Engineering Program
CIE 205: Fundamentals of Computer Programming

Spring 2021

Data Structures and Algorithms
CIE 205

Battle Game

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 you should protect against enemies attacks. You need to use your programming skills and knowledge of data structures to write a ***simulation program*** for a battle between your castle and enemies. You should simulate the battle between them then calculate some statistics from this simulation.

Project Phases

Project Phase	*%*
Phase 1 (Week 9)	**30%**
Phase 2 (Week 15)	**35%**
Individual Discussion* (week 15)	**35%**

Late Submission is not allowed

*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. The work load between team members must be almost equal.

NOTE: Number of students per team = 3-4 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 rather than cheating it. It is totally your responsibility to keep your code away from others.

Problem description

Your system (your program) will receive the information about 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.
- **Speed**: number of meters an enemy can move in a single time step.
- **Type:** ***fighter, freezer*** or ***healer***

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 castle and cause damage to castle.
  - Healer can heal other enemies and does not attack castle.
  - Freezer can throw ice on the castle to frost it for some time.

- **Enemy Status:**

At any time, an enemy should be in one of four states: ***inactive*** (not arrived yet), ***active, frosted*** (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.

- **Frosted Enemy:**

Frosted enemies cannot move or attack castle. Any active enemy can be frosted **for some time steps** (**freezing time**) due to an attack from the castle.

- **Enemy distance:** is the ***horizontal*** distance between the enemy and the castle.

- **Reload Period**: it is an interval where enemies are reloading their weapons. During reload period enemies cannot act (i.e. cannot attack or heal) but can move. Once reload period is over, enemies can act again.
  - **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 to reload at 10, 11 and 12 then act at 13 and so on.

Game Rules

- Each enemy has its own speed and its own fire power.
- All enemies start at **60 meters** distance from the castle.
- In general, an enemy moves at **its original speed each time step**.
  But an enemy with (health $< \frac{1}{2}$ its original health) moves at its **half original speed**.
- The minimum possible distance for any enemy to approach to the castle is **2 meters**.
- Enemies can move at their reload period but cannot act. (cannot fire nor heal)
- **Movement Patterns**: By default all enemies move in the forward direction (approaching castle) but only healers can move backward. When a healer reaches end of forward direction, it starts moving backward until it reaches end of backward direction where it turns back and moves forward again and so on.
- Healer can heal only enemies that are at most 2 meters ahead in his moving direction.
- Healer cannot heal killed enemies. Healer cannot heal frosted enemies.
- A castle can attack any enemy type.
- Both castle and enemies can attack each other at the same time step.
- A castle can attack at most **N** enemies at each time step. (N is loaded from input file).
- A castle can either fire bullets or throw ice to freeze an active enemy.
- At ***random time steps*** the castle throws ice instead of bullets. The percentage of ice fire should be ***around 20%*** of all castle fires.
- ☞ If Castle kills a healer within a distance of 5 meters from the castle, it uses healer's tools to recover its health by a percentage of 3%.
- If an active enemy is hit by castle ice, it gets frosted for some time steps until ice melts (See formulas section below).
- A frosted enemy is affected by castle fire but is not affected by castle ice.
- A frosted castle cannot attack enemies.
- A frosted castle is affected by fighter attacks only. Freezer enemy has no effect on it.
- The game is "**win**" if all enemies are killed. The game is "**loss**" if the castle is destroyed. Otherwise, the game is "**drawn**".

Enemies picking criteria:

As mentioned in the game rules, a castle picks at most **N** enemies (in total) to attack at each time step. Attack means to fire bullets or to throw ice with certain probability. The criteria to pick an enemy to attack are as follows:

1. First attack fighters. The castle should pick fighters depending on their distance, power, and health, status (active or frosted), remaining time steps for an enemy to end reload period. **Think about a suitable formula to give a priority for each fighter according to at least four of the above factors. Then pick higher priority fighters first.**

2. Then (if N is not exhausted yet), pick healers to attack. A fresh healer who has just joined the forces is picked first. This means that healers are picked in the reverse order of their arrival (**Last-Come-First-Serve**)

3. Finally (if N is not exhausted yet) pick freezers to attack. A freezer that arrived first should be attacked first (**First-Come-First-Serve).**

Formulas:

- **Damage a fighter causes to Castle (D_{FC})**

$$D_{FC} = Damage\ (fighter \rightarrow Castle) = \frac{K}{Enemy\_distance} * Enemy\_power$$

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 castle.

- **Think about a formula for amount of ice a freezer can throw on a castle.**

Take into account that the freezing threshold of a castle can never be reached by a single ice shot from an enemy. When the accumulated ice on a castle reaches certain threshold, the castle is frosted **for one time step**.

- **Damage to an enemy by a castle bullet (not applicable for ice throws)**

$$D_{CE} = Damage\ (Castle \rightarrow Enemy) = \frac{1}{Enemy\_distance} * Castle\_fire\_power * \frac{1}{K}$$

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

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

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

The formula should depend on healer health, enemy to be healed health, distance between healer and enemy

- **First-ShotDelay (FD)**

The time elapsed until an enemy is first shot **by castle**

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

- **KillDelay (KD)**

The time elapsed between first time a castle 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

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

Program Interface

The program can run in one of three modes: **interactive, step-by-step** or **silent mode**. When the program runs, it should ask the user to select the program mode.

Both Interactive and step-by-step modes allow user to monitor the battle between 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 1) on the screen.
In interactive mode, the program pauses for a mouse click to advance to the next time step.
In step-by-step mode, the program sleeps for one second then advances to the next time step.

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

- Current Time Step
- Current health of the castle and whether it is frosted or not
- Number of active fighters, healers and freezers and their total
- Number of frosted fighters, healers and freezers and their total
- Number of killed fighters, healers and freezers and their total

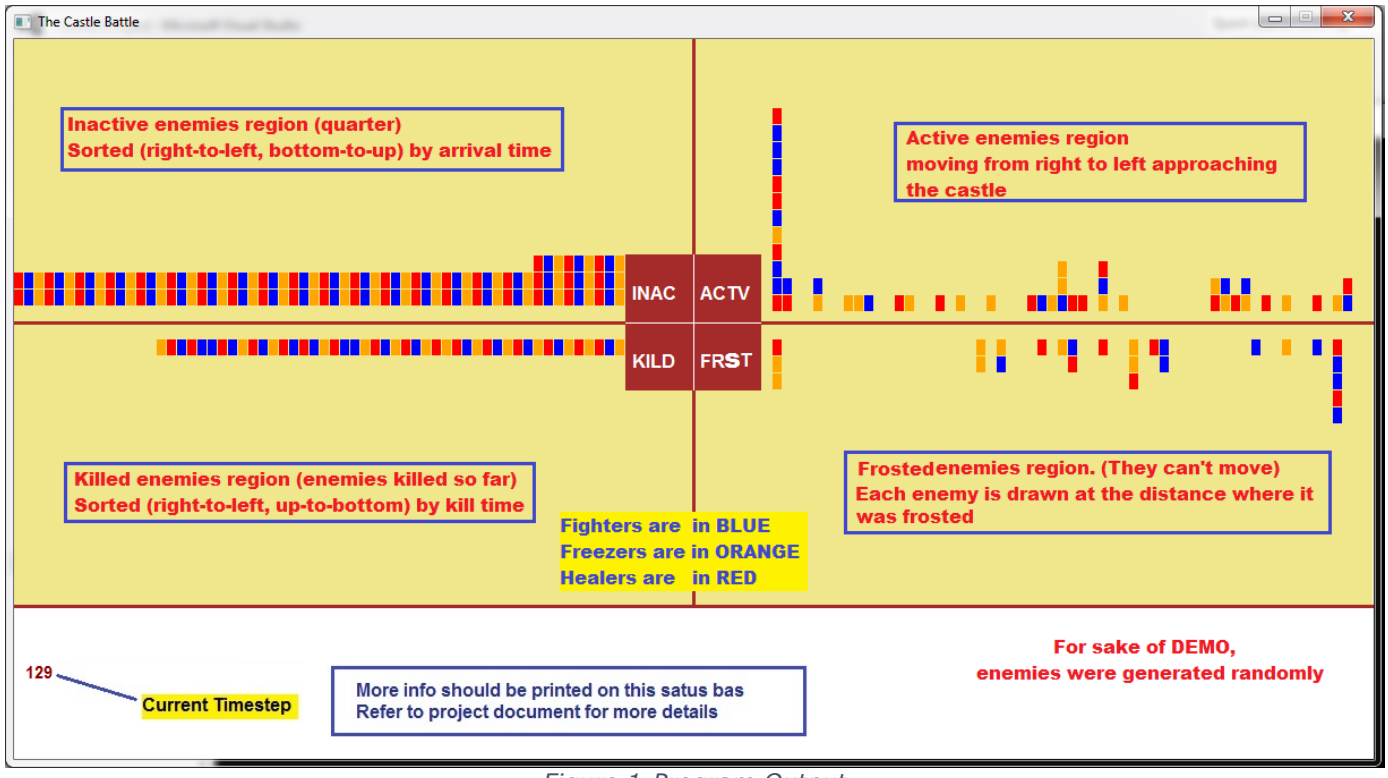

Figure 1-Program Output

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.(see Appendix A)

Files Formats

Your program should receive information about scenario 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: **CH N CP**
  CH is the starting health of the castle; **N** is the maximum number of enemies a castle can attack at any time step and **CP** is the castle power.
- Then the total number of enemies **M** is written in a separate line
- Then the input file contains **M** many lines (one line for each enemy) of the format

ID	TYP	AT	H	POW	RLD	SPD
Enemy ID	Enemy type	Arrival Time	Enemy Health	Enemy Power	Reload Period	Enemy Speed

- The input file lines are **sorted by arrival time** in ascending order.

The Output File

The first line should indicate whether game is **WIN, LOSS, or DRAWN**
The output file you are required to produce should contain **M** output line of the format
KTS ID FD KD LT
which means that the enemy identified by sequence number **ID** 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.

Then a line indicating **Castle total damage** caused by the attacking enemies.

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
50
1  0    1    10   2    4    2
2  1    4    12   5    4    5
3  0    4    15   5    3    1
4  2    7    21   3    2    3
.....................................................................
```

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

- An enemy number 1 of **type=0 (fighter)** arrived at time step 1 with Health =10, power = 2 and Reload_period= 4 and speed = 2.
- An enemy number 2 of **type=1 (healer)** arrived at time step 4 with Health =12, power = 5 and Reload_period= 4 and speed = 5.
- An enemy number 3 of **type=0 (fighter)** arrived at time step 4 with Health =15, power = 5 and Reload_period= 3 and speed = 1.
- An enemy number 4 of **type=2 (freezer)** arrived at time step 7 with Health =21, power = 3 and Reload_period= 2 and speed = 3.

Sample Output File

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

```
Game is WIN
KTS   ID    FD  KD   LT
5     1     0   5    5
10    2     4   4    8
15    3     5   2    7
.....
.....
Castle Total Damage      = 33.5
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 ID 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 **object-oriented** code with **Templates** 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 (see Appendix A).

Before explaining the requirement of each phase, all the following is 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. ***This is a data structures course where you should build data structures by yourself from scratch.***
- You need to get instructor approval before making any **custom (new)** data structure. **Note:** No approval is needed to use the common data structures.
- ***Do NOT allocate the same enemy twice***. Allocate it once and make whatever data structures you chose reference it (use pointers). Then, when another list needs an access to the same enemy, DON'T create a new copy of the same order. Just **share** it by making the new list point to it or **move** it from current list to the new one.

  "*SHARE, MOVE, DON NOT COPY* "
- You are not allowed to use **global variables** in your implemented part of the project.
- You need to get instructor approval before using **friendships.**

Phase 1

In this phase you should finish all parts 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 its subclasses), Castle, and Battle Classes.
2- **Full implementation for all data structures** that you will use to represent **the lists of enemies**.

ALLOWED Data Structures to implement lists:
You are allowed to use the following lists types only: Stacks, Queues, Priority Queue, Trees, or Heaps. **General Lists are not allowed.**

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:
1- Do you need a **separate** list for each enemy type? Why?
2- Do you need to store **killed** enemies? In separate lists or in one list? When should you

get rid of them to save memory?

3- **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 about 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.

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.

3- **File loading function.** The function that reads input file to:
   a. Load Castle data
   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 deal with enemies in different lists. This function should:
   a. Perform any needed initializations
   b. Call file loading function
   c. At ***each time step*** do the following:
      i. Activate enemies who have arrived.
      ii. **Move** all active enemies of all types according to their speeds and movement pattern.
      iii. For **each enemy type**, pick two active enemies that are on top of list and freeze them.
      iv. Pick two previously frosted enemies and return them back to their normal status.
      v. Kill one active enemy and one frosted enemy.
      vi. Draw all lists at the end of the time step.
      vii. Print on the status bar:
         1. Current Time Step
         2. Current health of the castle and whether it is frosted or not
         3. Number of active fighters, healers and freezers and their total
         4. Number of frosted fighters, healers and freezers and their total
         5. Number of killed fighters, healers and freezers and their total

Notes about phase 1:
- No output files should be produced at this phase.
- 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 submit a folder 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).
- **Phase1 Report (shown below)**

University of Science and Technology
Communication and Information Engineering Program
CIE 205: Fundamentals of Computer Programming

Spring 2021

Data Structures and Algorithms
Project Phase1 Report

Team Name: **Number of members:**
Team Email:

Members Info:

Member Name	ID	Email

Project Data Structures

List Name	Chosen DS	Justification
e.g. Inactive enemies List	Write the DS you have chosen e.g. Queue/Stack/ Pri-Q/ Tree .. etc	Write here the operations you need to perform on this list and the complexity of each operation Then justify why you have chosen this DS to implement that list

➔ **Repeat he above for each list in the project**

- Compress all the above in one file named after your team number (e.g. Phase1_team3.zip)

Phase 2

In this phase, you should build the full application that

1- implements the data structures you have chosen in phase1.
2- Handles project logic and operation rules.
3- Load scenarios from input file and produces the final output file.
4- Supports the different operation modes described in “Program Interface” section.

Phase 2 Deliverables:

Each team is required to submit:

- 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 <u>Comprehensive</u>** sample input files (test cases) and their corresponding output files. The test samples must cover different cases ranging from **weak-enemy-weak-castle case** to **strong-enemy-strong-castle case**.
- **A project document** describing workload distribution among team members.

- Compress all the above in one file named after your team number (e.g. Phase2_team3.zip)

Project Evaluation

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 and each DS has its class.
    - 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.
  - **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** *(if not a template class).*
  - **Class Responsibilities:** No class is performing the job of another class.

- **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 modes described in the document.

- **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.

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.

Bonus Requirements (maximum 10%)

- **[3%] Freezing amount and melting rates:**

  When an enemy is hit by fire ice, it is not frosted until certain amount of ice accumulates on that enemy. However enemy speed should be affected according to the amount of ice accumulated so far. Also at each time step the ice melts by certain rate that depends on both enemy health and power.

  Think about suitable formulas for (1) ice freezing threshold, (2) ice effect on enemy speed, and (3) ice melting rate.

- **[7%] Super soldiers (SS):**

  Castle has **three** super soldiers (SS) that it uses only when castle health goes behind certain threshold. SS are very powerful. They can kill up to 3 enemies in every gunshot regardless of enemies' health. They affect all enemies within a distance of 2 meters in both forward and backward directions. They are not affected by freezers but are affected by fighters.

  The castle should place the SS at the distance that has maximum number of enemies. To reach its fighting distance a SS starts moving away from the castle with speed of 3 meters per time step. Once it reaches its fighting place it starts fighting (shooting and get shot). It doesn't move anymore and keep fighting in its position until it is killed or the castle wins the fight.

  If castle health goes behind certain threshold at any time step, it starts sending the first SS to its place and waits until it reaches its place then waits for 5 more time steps then checks castle health again. If health is still behind the threshold, it repeats the process with the 2nd SS and so on.

  Think about a formula for fighter's effect on the SS.

  The SS should be shown on the interface in a color different than the enemies.

Appendix A–Guidelines on the Provided Framework

The main classes of the game are Enemy, Castle, and Battle classes.

Enemy class is the base class for each type of enemies. **You should inherit a class for each enemy type** from Enemy class. Enemy class should later be an abstract class that contains some pure virtual functions (e.g. Act, …etc.).

Castle class represents the castle that should attack and got attacked by enemies.

Battle class is ***the controller*** of the game. It contains an object of the castle and a queue of all inactive enemies. It should also contain list(s) for active and killed enemies. It should contain the functions that perform the fight logic and collect required statistics**.**

For the sake of demo, class Battle has a list called **DemoList** which is an array of pointers to ALL enemy types. This array should be deleted in next phases and other lists of your choice should be used to store different enemy types.

GUI class contains the input and output functions that you should use to show game progress. The class contains list of items to be drawn, **DrawingList**. If you want to add an enemy to that list to be drawn on the interface, you should call function **GUI::AddToDrawingList(Enemy *)**

At the end of each time step, you should call the following function in order:

1. Function **GUI::AddToDrawingList** for all enemies (active/inactive) to add them to the DrawingList.
2. Function **GUI::UpdateInterface** This function will draw all enemies on the screen.
3. Function **GUI::UpdateStausBar** to print required information on the status bar. Currently, this function just prints the time step. You should extend this function and add code that prints all required information as described in the document above.
4. Function **GUI::ResetDrawingList** to clear drawing list to be ready for items of the next time step.

.

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