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.
- Use data structures to solve a real life problem.
- Write a complete procedural C++ program that performs a non-trivial task.

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 d ay 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 will simulate the defensing scheme of the towers during killing the enemies then calculate some statistics from this simulation.

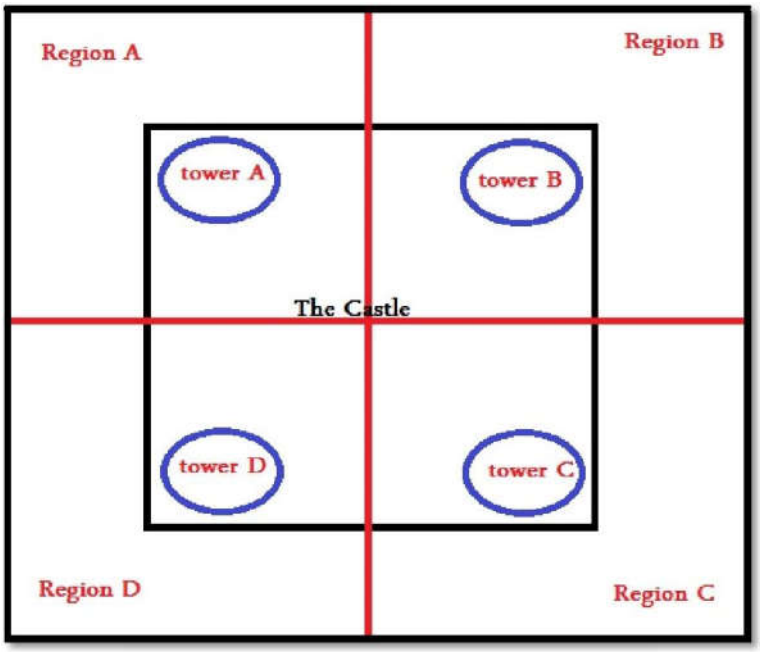

Figure 1 the castle

Problem description

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

- **Time stamp:** When the enemy will appear (enemy arrival time).
- **Health:** The start health of the enemy.
- **Fire Power:** The shot hit power of the enemy.
- **Hit Period:** The delay between two successive shots from the enemy.
- **Type:** Three types of enemies (paver, fighter and shielded fighter)
- **Region:** The attack region of the enemy.

Each tower will have a starting health and have the ability to shoot at most N enemies at each time step. Each tower guards one region and can only shoot enemies in its region.

Simulation Approach & Assumptions

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

Some Definitions

- **Active Enemy**
  We define an ***active*** enemy as an enemy who arrives at time less than or equals to current time step and its health is greater than zero. At each time step, each tower should choose ***N*** active enemies to fight. (N is given in the input file). Other enemies are either ***killed*** or ***inactive***.

- **Enemy distance**
  The horizontal distance between the enemy and the tower.

- **Damage to the tower by certain enemy**
  We define this damage by the following formula

$$\textbf{Damage (Enemy → Tower)} = \frac{1}{Enemy\_distance} * Enemy\_fire\_power$$

Note: If enemy is not allowed to file at current step, it will not cause any damage to the tower.

- **Damage to a certain enemy by the tower**
  We define this damage by the following formula

$$\textbf{Damage (Tower → Enemy)} = \frac{1}{Enemy\_distance} * Tower\_fire\_power * \frac{1}{k}$$

Use k=2 for shielded enemies and k=1 for other enemies

- **Enemy Priority**
  For the tower to determine the next enemy to kill the TAs proposed the following formula which will be applied by your program:

$$\textbf{Priority (Enemy)} = \left(\frac{1}{Enemy\_distance} * Enemy\_fire\_power * \frac{1}{Enemy\_remaming\_time\_to\_shoot+1} * C1\right) + Enemy\_health * C2 + Enemy\_Type * C3$$

Note: C1, C2 and C3 are constants defined by your system. (You can read them from the input file as well, check file formats section)

- **Paver enemies**
  All enemies can approach to 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 road to the castle are not paved and only the paver enemies can enter the unpaved distance to pave it so that enemies of other types can enter this paved distance in the **next** time steps.
  Note:
  A paver does not shoot the towers and its "Fire Power" represents the number of meters it can pave at each ***allowed*** attack time step. Hence, Formula Damage(Enemy Tower) described above is not applicable for pavers. However, Formula Damage(Tower Enemy) is applicable.

- **Fight Delay (FD)**
  The time elapsed until an enemy is first shot by a tower
  $$FD = T_{first\_shot} - T_{arrival}$$

- **Kill Delay (KD)**
  The time elapsed between first time a tower shoots an enemy and its kill time
  $$KD = T_{enemy\_killed} - T_{first\_shot}$$

- **Fight Time (FT)**
  The total time an enemy stays alive until being killed
  $$FT = FD + KD = T_{enemy\_killed} - T_{arrival}$$

Assumptions

- Every tower can **only** attack the enemies in its region.
- Every enemy can **only** attack the tower in its region.
- All enemies start at **60 meters** distance from the castle.
- Every tower can attack at most **N enemies** at each time step. N is read from the input file.
- The minimum possible distance for any enemy to approach is **2 meters**.
- The enemies can approach to the castle **one paved meter** at every **time step**.
- The game is "win" if all enemies are killed
- The game is "loss" if the all towers are destroyed.
- 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)

Program Interface

The program should read an integer from standard input (*cin*). If this integer is 1, the program runs in **interactive mode**. If it is 2, the program runs in **step-by-step mode**. Otherwise the program runs in **silent mode**.
Interactive mode allows you to monitor the attacking of enemies to the castle and active enemies 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 and pause for a user input until instructed to continue. At the bottom of the screen, the following information should be shown:

- Simulation Time Step
- For each region print:
  - total number of current enemies
  - number of enemies killed at last time step
    - total number of killed enemies from the beginning of simulation
    - unpaved distance to castle in this region

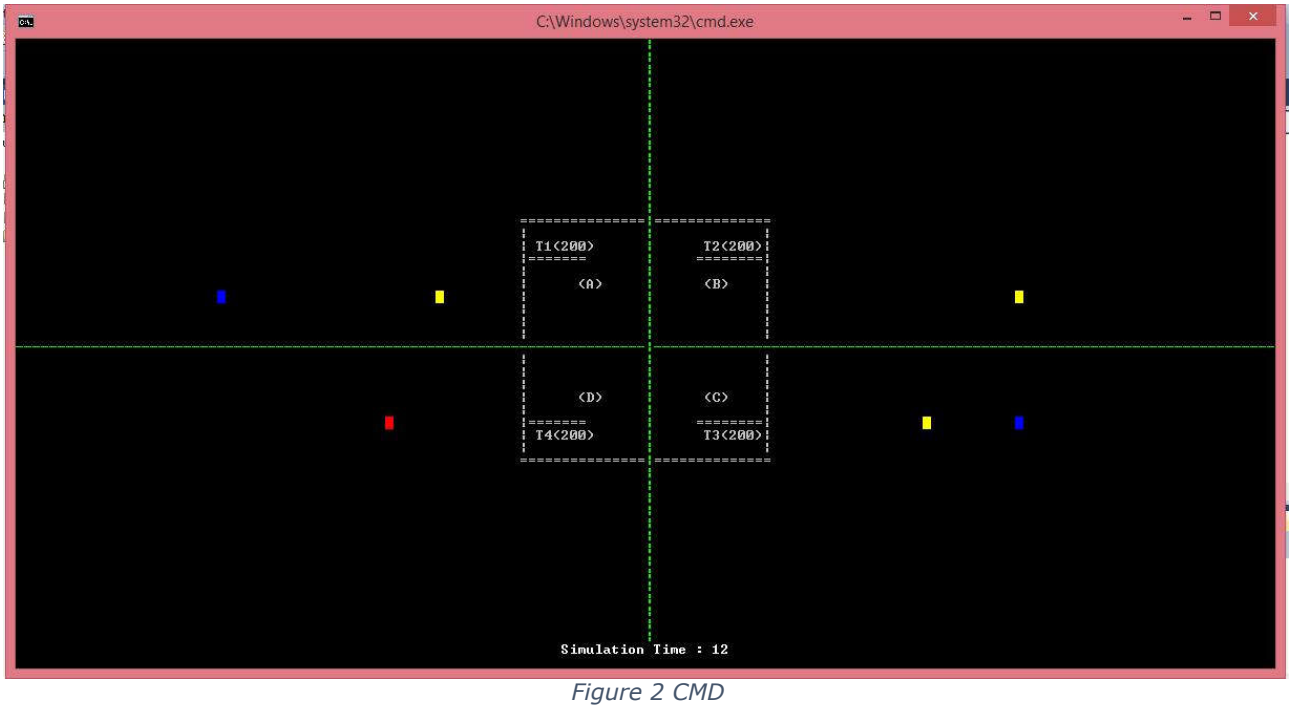

Figure 2 CMD

Step-by-step mode is identical to interactive mode except that the program pauses for one second (instead of pausing for user input) then resumes automatically.

In **silent mode**, the program produces only an output file. It neither pauses nor provides screen output.

You will be given some functions for drawing the above interface and you **should** integrate it with your system.

File Formats

The Input File

- First line will contain two integers **THNETP**
  TH is the starting health of all towers, **NE** is the number of enemies the tower can attack at every time step and **TP** is the tower fire power.
- Second line will contain **C1 C2 C3** the constants of the equation **Priority (Enemy)**
- Then the input file will contain N input lines of the format
  S T H P Pr TY R
  where **S** is a sequence number that identifies the enemy, **T** is the enemy arrival time, **H** is the enemy health, **P** is the enemy power, **Pr** is the enemy hit period, **TY** is the enemy type 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.
  The input file name must be "input.txt"

The Output File (No matter what mode of operation your program is running in, the output file should be produced)

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

A line in 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_Distance R2_Distance R3_Distance R4_Distance

Another line for string of **"win" or "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 "Fight Delay" and Average "Kill Delay"
2- In case of game "loss"
   a. Number of killed enemies
   b. Number of alive enemies
   c. Average "Fight Delay" and Average "Kill Delay" f or killed enemies only

Sample Input File

```
200 3   14
4   3   1
1   1   10   0   2   1   A
2   3   15   5   4   2   A
3   4   15   2   3   3   B
-1
```

The above file initialize the towers with health 200, the tower can attack 3 enemies at every time step and the tower fire power is 14.
The second line means that constants C1=4, C2=3 and C3=1
Then enemies details

- An enemy of type=1 (paver) arrived at time step 1 in region A with Health =10 and Hit_period = 2. (Hit_power is not applicable for this enemy type)
- An enemy of type=2 (fighter) arrived at time step 3 in region A with Health=15 and Hit_power = 5 and Hit_period=4
- An enemy of type=3 (shielded fighter) arrived at time step 4 in region B with Health=15 and Hit_power = 2 and Hit_period=3.

Sample Output File

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

```
5     1     0   5   5
10    2     4   4   8
15    3     5   2   7
.
.
.
33.5      12.5   55    200
30        30     25    2
WIN
50
4.5
12.36
```

The above file indicates enemy with sequence number 1 killed at time step=5 and it took FD=0 And KD= **5**

The seventh line indicates the total damage for each tower
The eighth line indicates the remaining unpaved distance at each region
Last four lines indicate you won the game, total enemies=50, average fight delay = 4.5, and average kill delay=**12.36**

You are advised to divide your work into two phases

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- Full declarations of Enemy, Tower, and Castle "***structs***".
2- The data structure that you will use to represent the lists of enemies (active and inactive enemy lists). Think about the following:
    a. Which list type is much suitable to represent the lists?
    b. Will you use one list for all regions or a separate list for each region and why?
    c. How will you store the high priority enemies?
3- File loading function. The function that reads input file and creates and populate lists accordingly.
4- Simple Simulator function. This function should
    a. Load Towers data from the input file
    b. Load enemies data and populate the inactive enemies list
    c. At each clock tick do the following
        i. Move active enemies from inactive list to active list
        ii. Kill any enemy that has arrived 10 ticks ago (or earlier)
        iii. Remove killed enemies from the list.
        iv. **For each region**, print current active enemies with information of each enemy and total number of enemies killed so far.

Note:
No output files should be produced at this phase.

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.

When you are using the priority queue as binary heap you must work in the template given and complete the function prototypes in the attached file binheap.cpp

Copying source code from another team or from the internet will penalize you and you will lose all the project grades.

Project Evaluation

Evaluation Criteria

- **Successful Compilation:** Your program should compile successfully with 0 errors and 0 warnings. If you find a warning that you cannot get rid off, post a question on the project discussion forum.

- **Data Structure & Algorithm**: After all, this is what the course is about. You should be able to provide a concise description of the data structure(s) and algorithm(s) you use to solve the problem. The logic of your program should be correct.

- **Coding style**: How elegant and **consistent** is your coding style (indentation, naming convention ...etc)? How useful and sufficient are your comments?

- **Modularity**: How modular is your code? A modular code does not mix several program features within the same function. 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.

- **Understanding**: How much does each team member understand the project? This means that you should not only be able to describe existing code but also be able to describe the modifications that have to be made in order to change the program behavior in a certain way.
  Note: Each group member will be evaluated individually.

Bonus Criteria

- **Efficiency:** Design a data structure that allows determining the next enemy to kill as well as adding a new active enemy in $O(lg\ N)$ time.
- **More modularity:** Divide program modules among multiple files. Each module should be implemented in at least one *.cpp* file and one *.h* file.
- **Enemies speed:** Handling enemies with different speed as input to your simulation.

Deliverables

Each group is required to deliver the following a **CD** that contains:

- Program source code (.cpp and .h files, project file(s), workspace/solution) [Do not include executable files].
- Three Sample input files that you used to test your program
- A text file called ID.txt containing group members' names, section(s) and bench numbers.
- Write your group number on the back of the CD cover.
- A project document with 2 or more pages describing your solution method, any clever or innovative alternatives you followed in implementing the solution

Group Size

A group should consist of 4 students.