Cairo University
Faculty of Engineering
Computer Engineering Department
CMP N103 **Spring 2019**

CMPN103
Programming Techniques
Project

Game

Introduction

In this project we are going to build a simple game application that is somehow similar to the Pacman game. The user should be able to build the playing grid as well as play the game itself. The general idea of the game is a player moving in a grid having obstacles, full of enemies that the player tries to escape and friendly items that the player tries to collect. The player has score, health and lives. Moreover, the player can use weapons to defeat enemies. The game is only one level that ends by **losing** if the player lost all its lives or by **winning** if the player is still alive after the game end time. The player target is to collect friendly items and increase his score as much as he can.

Your Task: You are required to write a **C++ code** for a Game. Delivering a working project is NOT enough. You must use ***object-oriented programming*** to implement this application and respect the **responsibilities** of each class as specified in the document. See the evaluation criteria section at the end of the document for more information.

NOTE: The application should be designed so that the types of items and types of operations can be easily extended (*using inheritance*).

Project Schedule

Project Phase	***Deliverables***
Phase 1 [25%]	**Input-Output Classes**
Phase 2 [75%]	**Final Project Delivery**

NOTE: Number of students per team = **3 to 4 students (from same tutorial – time and location).**

The project code must be totally yours. The penalty of cheating from any other source is not ONLY taking ZERO in the project grade but also taking some MINUS Five grades (- 5) from the other class work grades.

Table of contents

Game Description

***Note:** Any **percentage** written below next to any tool operation or game object type is its percentage from **phase 2 grade**. See the evaluation criteria at the end of the document for more details.*

The player is moving in a grid (2-dimentional space) that is divided into a number of cells; a cell is specified by its row and column (0-based) for example cell (0,1) means the cell in the 1st row and 2nd column. Grid cells may contain obstacles, enemies or friendly items.

Each Game has an amount of time, ***GT,*** (integer number of minutes). Timesteps starts from 0 and increments. The duration of each timestep is nearly 100 milliseconds (function Sleep(100) in C++). After this duration, the timestep increments.

A Game object is any type of objects that can occupy a cell in the grid game. Each cell can contain at most one game object. If more than one game object wants to occupy the same cell, this is called "**Collision**" which should be handled differently depending on the type of the colliding game objects.

There are 4 main types of Game Objects: *(you may consider some other objects as game objects too)*

1. **[10%] Player:**
   - The Game is a single player.
   - The player has lives (initially 3), health (initially 100%) and score (initially 0). When health reaches 0, then player loses one live.
   - The player's target in the game is to survive till the end time of the game with gaining as much score as he can.
   - **Win State** → if he survived till the end of the game time.
   - **Lose State** → if he lost all his lives before the end of the game time.
   - The player can move (or not) one cell in each timestep in a certain direction: up, down, left and right by pressing one of the keyboard keys: 'I', 'K', 'J', 'L' respectively (case insensitive). The player cannot penetrate an obstacle (wall).
   - ✓ The player can use **Weapons** to defeat enemies (affect enemies negatively).
   - ✓ Each weapon is available for the player only **twice** throughout the game.
   - ✓ We will describe each type of weapons below.

2. **Enemy:**
   - Enemies' target in the game is to affect the player **negatively** either by reducing his score, health, lives, movement, …etc.
   - We will describe each type of enemies, its movement and effect on the player below.

3. **Friendly Items:**
   - Friendly Items' target in the game is to affect the player **positively** either by increasing his score, health, lives, movement, …etc.
   - Friendly Items only affects player (NOT affect the enemies negatively like Weapons)
   - We will describe each type of friendly items below.

4. **[5%] Obstacles:**
   - Obstacles' target in the game is to block other game objects from movement.
   - You can think of obstacles as the grid walls that NONE of other game object types can penetrate it (move through it).
   - Obstacles cannot move (their position are not changed with time).

There are 4 Types of Enemies [Total: 20%]:

1. **[4%] KillerEnemy**:
   - It decrements the player's lives if the enemy itself collides with the player (i.e. the player and the enemy are in the same cell).
   - It moves one step in a *random* direction every 3 timesteps.

2. **[6%] ShooterEnemy**:
   - It shoots bullets in random directions.
   - Each bullet advance 1 cell each time step in that direction.
   - The bullet will decrement the player's lives if the bullet reaches it (i.e. the player and the bullet are in the same cell). It disappears if met an obstacle.
   - Assume that ShooterEnemy shoots **one** bullet every ***S*** timesteps. (***S*** value differs among different shooter enemies and it will be specified for each)
   - Each one shot has one random direction (direction may vary from a shot to another).
   - ShooterEnemy does NOT move.

3. **[10%] The Other Two Enemy Types are left for your choice.** Choose two types of enemies with the following constraints:
   - Each enemy of your choice has a totally different movement pattern than any other enemy type in the game.
   - Each one of the following aspects must be affected negatively by at least one of your chosen enemies: Player Movement, Player Health and Player Score.

There are 4 Types of Friendly Items [Total: 15%]:

1. **[3%] Gold**:
   - It increments player's score by 1.
   - Gold locations does NOT change throughout the game.
2. **[4%] Live**:
   - It increments player's lives by 1.
   - Each Live friendly item stays in the Grid for only **L** timesteps then disappear. (***L*** value differs among different Live items and it will be specified for each)
   - Live locations does NOT change throughout the game.
3. **[8%] The Other Two Friendly Items Type are left for your choice.** Choose two types of friendly items with the following constraints:
   - One of them moves.
   - Each one of the following aspects must be affected positively by at least one of your chosen friendly items: Player Movement, Player Health and Player Score.

- **Note**: The effects or changes made by any of the friendly item are fired when the player and the items collide (i.e. the player and the item are in the same cell).

There are also 4 Types of Weapons [Total: 15%]:

1. **[3%] Bomb:** kills enemies in the region + or - 2 cells in the vertical and horizontal direction from the player's location (5 * 5 Squared Region).
2. **[4%] Slow:** decreases the speed of KillerEnemies by half for 40 timesteps.
3. **[8%] The Other Two Types are left for your choice with the following constraints:**
   - Each one of the following aspects must be affected negatively by at least one of your chosen weapons: Enemies' Movement and Enemies' Life, ShooterEnemy's Bullets.

Main Toolbar Operations

The application supports 2 modes: **Create-Grid Mode** and **Game Mode**. Each mode contains 2 bars: **tool bar** that contains the main operations of the current mode and **status bar** that contains any messages the application will print to the user. The application should support the following operations (actions) in each mode.

[Important Notes]:

- *Each operation in any of the following modes **must** have an **icon** (or more) in the tool bar. The user should click on the operation icon from the tool bar to choose it.*
- **In all the following actions, any dynamically allocated memory MUST be deallocated.**

The main operations supported by the application are the following.

[I] Create-Grid Mode: [15%]

The purpose of this mode is to design the Grid of the game that you will play later. You specify in this mode the location of the obstacles, the number and type of enemies and when they will appear and in which location, …etc. The operations supported by this mode are:

Note: in any of the following actions, if a cell position is needed to be taken from the user, the user clicks on it **AFTER** choosing the action (after clicking on the action icon on toolbar not before it).

1- Add Obstacle: adding an obstacle to one cell of the user choice in the game grid.

2- [3%] Add Enemy: adding an enemy item of a specific type to the game grid.
In this action, the cell and timestep at which this enemy will initially appear in the game should be specified. There should be an icon in the toolbar for each enemy type. There are other parameters needed to be taken from user depending on enemy type (e.g. bullet rate **S** in ShooterEnemy).

3- [3%] Add Friendly Item: adding a friendly item to the game grid.
In this action, the cell and timestep at which this item will appear in the game should be specified. There should be an icon in the toolbar for each friendly item type. There are other parameters needed to be taken from user depending on friendly item type (e.g. **L** in Live items)

4- [1%] Delete Game Object: deleting the game object in the cell that the user chooses.

5- [1%] Set Game Time: setting the duration of the game (number of minutes).

6- [2.5%] Save Grid: saving the information of the designed grid (all the game objects of it and game time) to a file (see "File Format" section). The application must ask the user about the filename to create and save the grid to (overwrite if the file already exists).

7- **[3.5%] Open Grid:** opening a saved grid from a file and re-drawing it (see "File Format").
   - ❑ This operation re-creates the saved game objects and re-draws them.
   - ❑ The application must ask the user about the filename to open and load grid from.
   - ❑ After loading, the user can edit the loaded grid and continue the application normally.
   - ❑ If there is a grid already drawn and the open grid operation is chosen, the application should clear the area and make any needed cleanups then loads the new one.

8- **[0.5%] Switch to Game Mode:** The user can switch to game mode any time even before saving.

9- **[0.5%] Exit:** exiting from the application.
   - ❑ Perform any necessary cleanup (termination housekeeping) before exiting.

[II] Game Mode: [15%]

In this mode, the user chooses a grid to play in then the player starts or resumes the game.
The current player's **health**, number of **lives**, **score** and remaining **usage times** of each weapon type should be displayed in the **Status Bar** all the time. The operations supported by this mode are:

5. **[3%] Choose Playing Grid:** The user has to choose the grid to play in:
   - **Random Grid**: uses a grid that is created by the program randomly.
   - **Current Grid**: uses the current grid just designed in the *create-grid mode (*either was loaded in create-grid mode or just created).
   - Just the user chooses a grid, the game will immediately start.

   This action needs 2 icons in the tool bar one for random and one for current.

6. **[2%] Grab Weapon:** The player chooses a weapon to use from the toolbar. As mentioned before, each weapon is available for the player **twice** throughout the game. There must be an icon in the toolbar for each weapon type.

7. **[2%] Pause Game:** pauses the current game (no movement or advancing in timesteps, …etc.).

8. **[1%] Resume Game:** resumes the current paused game.

9. **[2.5%] Save Game**: saving the information of the current state of the game to a file (see "File Format" section). The application must ask the user about the filename to create and save the game in (overwrite if the file already exists).

10. **[3.5%] Reload Game:** loading a saved game from a file (see "file format").
    - The application must ask the user about the filename to load from.
    - After reloading, the user can continue playing the loaded game normally.
    - If a game is already active, the application should clear the current game and make any needed cleanups then load the new one.

    This action has an icon in the tool bar.

11. **[0.5%] Switch to Create Grid Mode**: At any time, user can switch back to the other mode.
    - An empty grid is opened in the create-grid mode.
    - Don't forget to make any needed cleanups.

12. **[0.5%] Exit:** exiting from the application.
    - Perform any necessary cleanup (termination housekeeping) before exiting.

The remaining 5% of phase 2 is left for code organization and styling. See "Phase 2 Evaluation Criteria" section for more info.

[III] [Bonus] Operations [10%]:

The following operations are bonus and you can get the full mark without supporting them and you will not receive more bonus if you implemented any additional feature other than the features mentioned here in the bonus part. This bonus will be delivered in **Phase 2.**

1- [5%] Rank Players:
- ❑ Each user playing the game should specify its name.
- ❑ Keep track of all players that played the game with the associated scores.
- ❑ When a player finishes playing a game the score together with the player's rank among previous players should be displayed.

2- [5%] Jump: player have an extra move action which is jump where the player can jump one cell in a specified direction (up, down, left and right) avoiding obstacles or enemies in this step.

Main Classes

You are given a ***code framework*** where we have ***partially*** written code of some of the project classes. For the graphical user interface (GUI), we have integrated an open-source ***graphics library*** that you will use to easily handle GUI (e.g. drawing grid on the screen and reading the coordinates of mouse clicks …etc.).

You should **stick to** the **given design** (i.e. hierarchy of classes and the specified job of each class) and complete the given framework by either: extending some classes or inheriting from some classes (or even creating new base classes).

Below is the class diagram then a description for the basic classes.

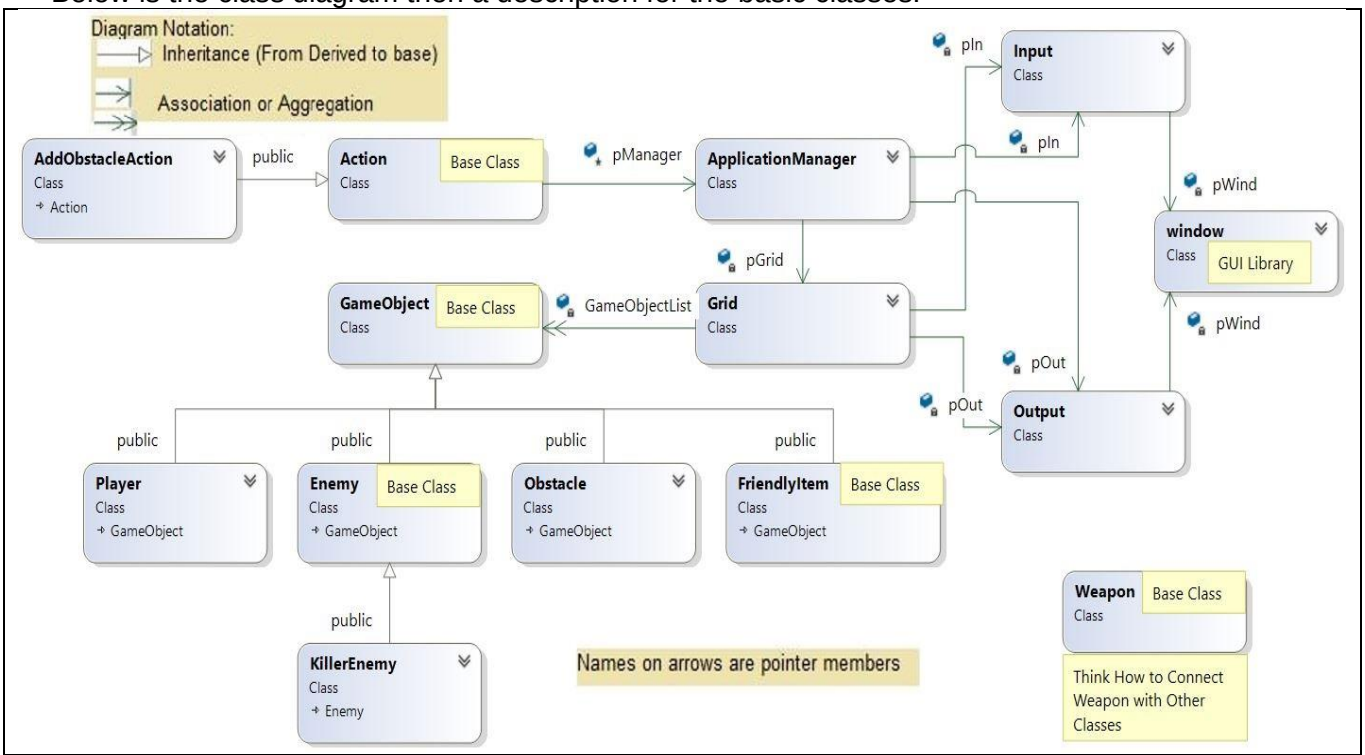

Figure 1 – Class Diagram

Input Class:
ALL user inputs must come through this class. If any other class needs to read any input, it must call a member function of the input class. You should add suitable member functions for different types of inputs.

Output Class:
This class is responsible for **ALL** GUI outputs. It is responsible for toolbar and status bar creation, grid and game objects drawing, and for messages printing to the user. If any other class needs to make any output, it must call a member function of the output class. You should add suitable member functions for different types of outputs.

Notes: - No input or output is done through the console. All must be done through the **GUI window**.
- Input and Output classes are the **ONLY** classes that have **access to GUI library**.

Action Class:
Each operation from the above operations must have a ***corresponding action class***. This is the base class for all types of actions (operations) to be supported by the application. To add a new action, you must **inherit** it from this class. Then you should override virtual functions of class **Action** (ReadActionParameters and Execute functions). Each action may have action parameters. **Action parameters** are the parameters needed to be read from the user, after choosing the action icon, to be able to execute the action. You can also add more details or functions for the class Action itself if needed.

GameObject Class:
This is the **base class** for all types of game objects (i.e. anything that can be added to a cell in the game grid like: player, enemies, friendly items, obstacles, …etc.). Each game object type must **inherit** from this class (GameObject), then you should **override** its virtual functions **(**e.g. Draw, Move, Save, …etc.). You can also add more details or functions for class GameObject itself if needed.

Grid Class:
This class represents the game grid (the vertical and horizontal cells that game objects can move in). It contains a **2-Dimensional Array of Pointers to GameObjects** (called **GameObjectList**). This list keeps track of the grid cells and the game objects occupying each cell. This is the **ONLY** class that can operate directly on the **GameObjectList** which means that NO other classes can Get this List or a copy of it and operate directly on it.

The Grid class responsibility is to maintain the **GameObjectList** (e.g. by providing public functions like AddObject and RemoveObject, …etc.) but **it must NOT make any further logic** in its functions. There may be functions inside the Grid class like SaveAll / MoveAll that loops on the 2D array and **blindly** call the virtual function Save / Move for each GameObject pointer in the array, but the functions of the Grid class only loop and call functions (do NOT make any further logic).

ApplicationManager Class:
This is the ***maestro*** class that controls everything in the application. It creates the Grid, Input and Output objects and have pointers of them as data members. Its job is to create an object of the action class corresponding to the action chosen by the user then executes it. **ApplicationManager** just manages or instructs other classes to do their jobs (**NOT** to do other classes' jobs).

Player Class:
Player is a type of game objects. Each player object contains: lives (initially 3), health (initially100%) and score (initially 0).

Obstacle Class:
This class represents obstacles and their location. Obstacles also are types of game objects.

Enemy Class:
This is a type of game objects and the Enemy class itself is the **base class** for all types of enemies. To create a new enemy type (KillerEnemy class for example), you must **inherit** it from this class. Then you should override virtual functions of class **Enemy.** You can also add more details or functions for the class Enemy itself if needed.

FriendlyItem Class:
This is the **base class** for all types of friendly items. To create a new item type (Gold class for example), you must **inherit** it from this class. Then you should override virtual functions of class **FriendlyItem.** You can also add more details or functions for the class FriendlyItem itself if needed.

Weapon Class:
This is the base class for all types of weapons. To create a new weapon type (Gun class for example), you must **inherit** it from this class. Then you should override virtual functions of class **Weapon.** You can also add more details for the class Weapon itself if needed.

Example Scenarios

The application window in Game mode may look like the window in the following figure. The window is divided to **Tool Bar**, **Grid Area (Playing Area)** and **Status Bar**. The tool bar of any mode should contain icons for all the actions in this mode according to what is mentioned in the Main Operations section above.

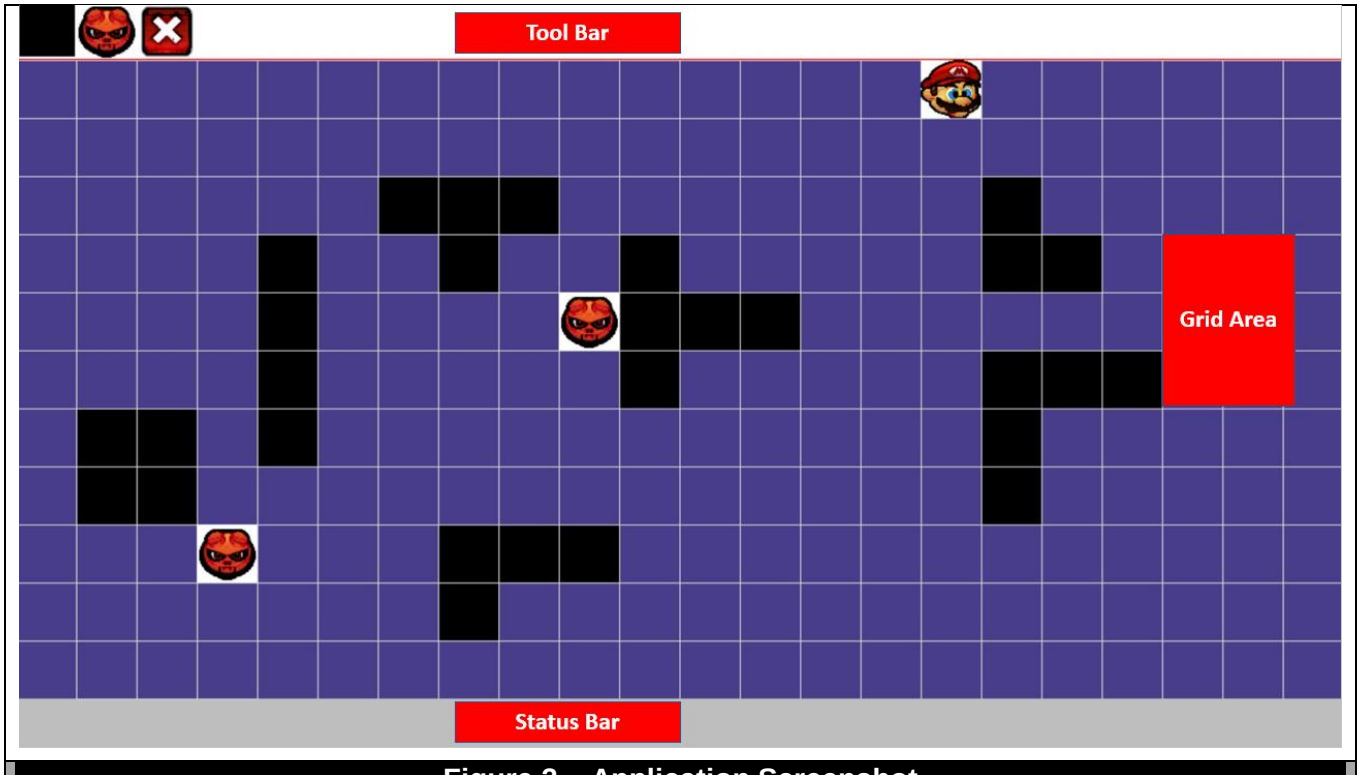

Figure 2 – Application Screenshot

Example Scenario 1: Add Obstacle Action (in Create-Grid Mode)

Here is an example scenario for **adding an obstacle** in the Grid in the Create-Grid Mode. It is performed through the three steps mentioned in '**Appendix A - implementation guidelines**' section. These three steps are in the "main" function of phase 2 code. You **must NOT** change the "main" function of **phase 2**. The 3 steps are as follows:

Step 1: Get user input

1- The **ApplicationManager** calls the **Input** class and waits for user action.
2- The user clicks on the **"Add Obstacle" icon** in the tool bar to add an obstacle.
3- The **Input** class checks the area of the click and recognizes that it is a "add obstacle" operation. It returns **`ADD_OBSTACLE`** (an **"enum"** value representing the **ActionType**) to the manager.

Step 2: Create a suitable action

1- **ApplicationManager::ExecuteAction(ActionType)** is called to create an action object of type **AddObstacleAction** class.

Step 3: Execute the action

1- **ApplicationManager::ExecuteAction(…)** calls **AddObstacleAction::Execute( )**

2- **AddObstacleAction::Execute( )**

a. Calls **AddObstacleAction::ReadActionParameters( )** which calls function **GetCellClicked()** from the **Input** class to get the action parameters of AddObstacleAction (i.e. the cell position to which the obstacle will be added).

Notice that when **AddObstacleAction** wants to print messages to the user on the status bar, it calls some functions from the **Output** class.

b. Creates (allocates) an object of class **Obstacle**

c. Asks the **Grid** to add the created obstacle object to the current list of grid's game objects by calling **Grid::AddObject(…)** function and passing the new obstacle to it.

Function Grid::AddObject(…) will do the following:

1. **Add the passed GameObject** (the new obstacle) **to the GameObjectList**

2. **Call function GameObject::Draw (…) of this object to be drawn in the Interface.**

The **Draw(…)** function of **GameObject** class is a pure virtual function which is overridden in class **Obstacle** to call **Output::DrawObstacle(…)** of class Output that is responsible for Drawing the obstacles in the Interface.

This means that there is a draw function in **Output** class that can draw each GameObject type on the window. The overridden draw function in each **GameObject** class type calls the draw function of class output that can draw this type of object.

Example Scenario 2: Save Grid Action (in Create-Grid Mode)

- **In General, Save/Open** has NO relation to the **Input** or **Output** classes. They save/open grids or games to/from files not the graphical window.

- Here we explain the calling sequence in the execute of '**SaveGridAction'** action as an example. **Note** the responsibility of each class and how each class does only its job or responsibility.

- There is a save function in **Grid** class and in each type of **GameObject** but each function performs a different job:

1. **GameObject :: Save (… , Type)**
   It is a **virtual** function in **GameObject** class. Each class derived from GameObject class should **override** it with its own implementation to save itself because each GameObject type has different information and hence a different way or logic to save itself. The function takes a Type parameter which could be an enum or an integer that represents the GameObject type that should be save. So, each GameObject class type will check the "Type" parameter sent to the function. If it is the same type of the class, it will save its information to the file, otherwise, return without saving.

   This "**Type**" parameter is sent because as shown in the "File Format" section mentioned below, all obstacles should be saved first, then all enemies and so on.

2. **Grid :: SaveAll (… , Type)**
   It is the function responsible for **calling** the **GameObject :: Save(…, Type)** function for each GameObject in the Grid's **GameObjectList** because **Grid class** is the only class that has **GameObjectList** and no one else can access it. Note that it only calls function save of each game object; ONLY calling without making the save logic itself (not the responsibility of **Grid** but the responsibility of each game object class). This note is important and has a huge grade percentage.

   Note: dynamic_cast is NOT needed in this function because polymorphism will automatically call the Save function of the correct object type for each GameObject pointer in the list.

3. **SaveGridAction :: Execute( )**
   It does the following:
   - first reads action parameters (i.e. the filename)
   - then opens the file
   - and calls **Grid::SaveAll (…, ObstaclesType)** to same all obstacles
   - and calls **Grid::SaveAll (…, EnemiesType)** to same all enemies
   - and so on
   - then closes the file

   Note: if any information is available for SaveGridAction class without breaking class responsibilities, it should write it to the file by itself.

   Important Note: Don't abuse the Type enum that you will create for this function. Whenever virtual functions and polymorphism could be applied, apply them. If any use of Type enum in your project will replace virtual functions and break class responsibilities, this will be grade-penalized.

File Format

1- "Grid" File Format:

Your application should be able to **save/open a grid** to/from a simple text file. In this section, the file format is described together with an example.

- **File Format**

```
GameTime(GT)_in_minutes
Number_of_Obstacles(n1)
Obstacle_1_ID      Obstacle_Parameters(cell)
Obstacle_2_ID      Obstacle_Parameters(cell)
……..
Obstacle_n1_ID     Obstacle_Parameters(cell
Number_of_Enemies(n2)
Enemy_1_Type       Enemy_ID     Enemy_Parameters(cell, timestep,…etc)
Enemy_2_Type       Enemy_ID     Enemy_Parameters(cell, timestep,…etc)
……..
Enemy_n2 Type      Enemy_ID     Enemy_Parameters(cell, timestep,…etc)
Number_of_FriendlyItems(n3)
FriendlyItem_1_Type     FriendlyItem_ID     FriendlyItem_Parameters(cell, timestep,…etc)
FriendlyItem_2_Type     FriendlyItem_ID     FriendlyItem_Parameters(cell, timestep,…etc)
……..
FriendlyItem_n3_ Type   FriendlyItem_ID     FriendlyItem_Parameters(cell, timestep,…etc)
```

- **Example:** The grid file looks like that (*comments in green are just for explaining the example*)

```
2 //2-minute game (GT = 2)
3  //number of obstacles
1      5    10  // obstacle id(1), cell(5,10)
2      8    9   // obstacle id(2), cell(8,9)
3      10   1   // obstacle id(3), cell(10,1)
2  //number of enemies
ShooterEnemy  4      1  4      2   9   // shooter id(4), cell(1,4), arrival timestamp(2), S = 9
KillerEnemy       5      7  9      4       // killer     id(5), cell(7,9), arrival timestamp(4)
3  //number of friendly items
Gold          6      9  4      6          // Gold id(6), cell(9,4), arrival timestamp(6)
Live          7      3  3      1   20   // Live  id(7), cell((3,3), arrival timestamp(1), L =  20
Live          8      5  9      10  30   // Live  id(8), cell((5,9), arrival timestamp(10), L = 20
```

Notes:

- ❑ You can give any IDs. Just make sure ID is **unique** among all game objects (not a must making them consecutives).
- ❑ You are allowed to modify this file format if necessary **but after instructor approval**.
- ❑ You can use numbers instead of text to simplify the "Open" operation. For example, you can give each enemy type a number. This can be done using ***enum*** in C++.
- ❑ **The "Open" or "Reload" Actions**:
  For lines in the above file, For Example for the enemies: the **OpenAction** first **reads** the enemy type then **creates** (allocates) an object of that type of enemy. Then, it **calls GameObject::Read** virtual function that can be overridden in the class of each enemy type to make the enemy load its data from the opened file by itself (its job). Then, it **calls Grid::AddObject** to add the created enemy object to **GameObjectList**.

2- "Game" File Format:

Your application should be able to **save/reload a game** to/from a simple text file. In this section, the file format is described together with an example and an explanation for that example.

- **File Format**

```
GameTime(GT)_in_minutes
Current_Game_TimeStep
Number_of_Obstacles(n1)
Obstacle_1_ID          Obstacle_Parameters(cell)
Obstacle_2_ID          Obstacle_Parameters(cell)
……..
Obstacle_n1_ID         Obstacle_Parameters(cell)
Number_of_Alive_Enemies(n2)
Enemy_1_Type           Enemy_ID     Enemy_Parameters(cell, timestep,…etc)
Enemy_2_Type           Enemy_ID     Enemy_Parameters(cell, timestep,…etc)
……..
Enemy_n2 Type      Enemy_ID         Enemy_Parameters(cell, timestep,…etc)
Number_of_NotCollected_FriendlyItems(n3)
FriendlyItem_1_Type     FriendlyItem_ID     FriendlyItem_Parameters(cell, timestep,…etc)
FriendlyItem_2_Type     FriendlyItem_ID     FriendlyItem_Parameters(cell, timestep,…etc)
……..
FriendlyItem_n3_ Type   FriendlyItem_ID  FriendlyItem_Parameters(cell, timestep,…etc)
Player_ID          cell              Lives   Health   Score
Number_of_usage_First_Weapon (Bomb for example, …etc.)
Number_of_usage_Second_Weapon
Number_of_usage_Third_Weapon
Number_of_usage_Forth_Weapon
```

- **Example:** The game file looks like that. This is just a dummy example not necessarily generated from the file example of the Grid File (*comments in green are just for explaining*)

```
2   // 2-minute game (GT = 2)
30 // current timestep of the game
3  //number of obstacles
1       5     10  // obstacle id(1), cell(5,10)
2       8     9    // obstacle id(2), cell(8,9)
3       10    1    // obstacle id(3), cell(10,1)
2  //number of alive enemies ( this also includes the enemies that haven't arrived yet )
ShooterEnemy   4       1   4       2    9  // shooter enemy id(4), cell(1,4), arrival timestamp(2),S=9
KillerEnemy      5       7   9      40       // killer enemy      id(5), cell(7,9), arrival timestamp(40)
3  //number of not-collected friendly items (includes not-arrived items too)
Live             8       5   9      10     30   // Live  id(8), cell((5,9), arrival timestamp(10), L = 30
Gold             6       9   4      6    // Gold id(6), cell(9,4), arrival timestamp(6)
Live             7       3   3      1      20   // Live  id(7), cell((3,3), arrival timestamp(1), L = 20
9       2  9      2      80     23 // Player id(9), cell(2,9), remaining Lives(2), health(80%), score(23)
1   // usage of the first weapon type – here used once and can be used one more
2   // usage of the second weapon type – here used twice so cannot use it anymore
0   // usage of the third weapon type – here used 0 times so can use it 2 times
0   // usage of the forth weapon type – here used 0 times so can use it 2 times
```

- **Notes of the Game File Format:**

  - The alive enemies that we write in the Game File includes the enemies that are arrived and still alive. In addition, it includes the enemies that didn't enter the Game yet (its arrival time step didn't come)
  - if (enemy's arrival timestep <= current game time step), then this enemy should be drawn at the same time of loading the file.
  - If (enemy's arrival timestep > current game timestep), then this enemy should be drawn later when its arrival timestep comes (not directly after loading the file).
  - The previous notes are applied in the same way for **FriendlyItems**.

Project Phases

A partially-implemented code frameworks for Phase 1 and for Phase 2 are given to you to complete them.

For **fast navigation** in the given code in **Visual Studio**, you may need the following:

- **F12 (go to definition):** to go to definition of functions (code body), variables, …etc.
- **"Ctrl" then "Minus":** to return to the previous location of the cursor.

1- Phase 1 (Input / Output Classes) [25% of total project grade]

In this phase, you will implement the **Input** and the **Output** classes as they do not depend on any other classes. The **Input** and **Output** classes should be **finalized** and ready to run and test. Any expected user interaction (input/output) that will be needed by phase 2, should be implemented in phase 1.

Input and Output Classes Code and Test Code

You are given a code for phase 1 (separate from the code of the whole project) that contains both the input and output classes partially implemented. Each team should complete such classes as follows:

1- *Output Class*:

- **Draw the Full Tool Bars**:
  Output class should create 2 FULL tool bars: one for the **Create-Grid mode** and one for the **Game mode**. Each contains icons for every action in this mode.
  [Notes]:
  - Some operations need **more than one icon**, for example, "Add Enemy" in Create-Grid mode should have an icon for each enemy type.
  - **ALL** icons of the project must be added in phase 1.

- **Implement the Following Functions:**
  - **ClearGridArea ( ):** it draws an empty grid. It is partially implemented. You are required to implement the code that adds vertical and horizontal lines between the cells of the grid.
  - **DrawObstacle (Cell):** it draws an obstacle in the passed cell. You MUST use function DrawRectangle of the library to draw the obstacle with the background color of obstacles.
  - **DrawGameObject (Image, Cell):** it draws the passed image in the passed cell. It will be used in drawing any game objects other than obstacles.
  - **ClearCell (Cell):** it clears the passed cell from any drawing.

2- Input Class:

- **Implement the Following Functions:**
  - **GetUserAction (…):** it is where the input class should detect all possible actions of any of the 2 modes according to the coordinates clicked by the user. It is partially implemented.
  - **TakeMoveDirection (Cell):** it takes a keyboard key press from the user that represents the movement direction of the player. It checks on the pressed key and returns the corresponding direction. Letter 'i' means UP, 'j' means Left, 'k' means Down and 'l' means Right. The function should detect the direction either the pressed key is small or capital letter.
  - **GetCellClicked ( ) :** it takes a mouse click from the user and returns the cell position on which the user clicks. If the click is NOT on a cell, cell (-1,-1) is returned.

3- *DEFS.h and UI_Info.h Files*: as you implement the Input and Output class, you will find some parts you need to add in DEFS.h and UI_Info.h files.

4- *Test Code:* this is not part of the input or the output classes; it is just a test code (the **main**).

- Complete the code given in **TestCode.cpp** file to test both Input and Output classes.
- Do NOT re-write the main function from scratch, just complete the required parts.

Note: In Phase 1 code, Follow the any **///TODO** comments to know what should be done in each file.

Notes on The Project Graphics Library*:*

- The origin of the library's window **(0, 0)** is at the **upper left** corner of the window.
- The direction of **increasing** the **x coordinate** is to the **right**.
- The direction of **increasing** the **y coordinate** is **down**.
- The only image extension that the library accepts is "**jpg**".
- You can use an alternative graphics library if you want, but with the condition of keeping the same interface (function prototypes) of the **Input** and **Output** classes and any other classes of phase 2 (same interface but the function body will be using your new library). However, we as TAs do not guarantee the support if you have any problem with your new library. It is your responsibility to deal with it.

Phase 1 Deliverables*:*

On the discussion day, each team should deliver an email that contains IDs.txt (team number, member names, IDs, email) and phase 1 code that has Input and Output classes, DEFS.h, UI_Info.h and test program completed.

2- Phase 2 (Project Delivery) [75% of total project grade]

In this phase, the completed **I/O classes**, **DEFS.h** and **UI_Info.h** (without phase 1 test code) should be added to the project **framework code** (given for phase 2) and the remaining classes should be implemented. Start by implementing the base classes then move to derived classes.

In the given code of phase 2, AddObstacleAction of Create-Grid mode is fully implemented. In addition, some base classes are partially implemented with some guiding comments to follow.

You are required to:

- **Complete the Implementation of functions marked with ///TODO comment** as it is described in the comments. These are some useful functions that you can use in implementation of phase 2.

- ❑ **Add all the classes mentioned in the "Main Classes Section"** with full implementation of their functions and finalize the project to perform all the operations mentioned in the document in the 2 modes.
- ❑ **You may need to add more classes to make the code more organized and object-oriented** but you are NOT allowed to change the initial classes' design, hierarchy and responsibilities we mentioned in the "Main Classes Section".

Phase 2 Deliverables*:*

(1) **Workload division**: a **printed page** containing team information and a table that contains members' names and the actions each member has implemented.
(2) **Innovation document:** a **printed document** describing the full description of your newly-chosen types of enemies, weapons and friendly items.
(3) An email that contains the following:
    a. ID.txt file. (Information about the team: names, IDs, team email)
    b. **Workload division** + **Innovation document** (described in the previous 2 points)
    c. The project code and resources files (images, saved files, …etc.).
    d. Sample grid files: at least three different grids. For each grid, provide:
        i. Grid text file (created by save operation)
        ii. Grid screenshot for the grid generated by your program
        iii. Screenshot of one action of *create grid* mode
    e. Sample game files: at least three different games. For each game, provide:
        i. Game text file (created by save operation)
        ii. Game screenshot for the game generated by your program
        iii. Screenshot of one action of *game* mode

Note that Each project phase should be sent first **by mail** (same time for all groups). **No modifications are allowed after the mail delivery.** After that the face-to-face **discussion** of the project will be held. The week day of sending each project phase by mail and the discussion schedule of each phase will be announced later.

Phase 2 Evaluation Criteria

Percentages are added according to the **difficulty** of the task, so to divide the project load **equally** on team members, each member should take actions that their percentages add up to about **25%** of phase 2 (if 4 students in the team).

Toolbar Operations [30%]

- ❑ 15% for Create-Grid Mode + 15% for Game Mode
- ❑ Each operation percentage is mentioned beside its description.

Game Logic [65%]

- ❑ Any percentage mentioned next to a Game Object means: handing all the Logic of this object in the game

Code Organization & Style [5%]

- ❑ Every class in .h and .cpp files
- ❑ Variable naming
- ❑ Indentation & Comments

Bonus [10%]

- ❑ Each operation percentage is mentioned beside its description.

General Evaluation Criteria for any Operation:

1. **Compilation Errors → MINUS 50%** of Operation Grade
   - ❑ The remaining 50% will be on logic and object-oriented concepts (see **point no. 3**)

2. **Not Running** (runtime error in its **basic functionality**) → **MINUS 40%** of Operation Grade
   - ❑ The remaining 60% will be on logic and object-oriented concepts (see **point no. 3**)
   - ❑ If we found runtime errors but in corner (not basic) cases, that's will be part of the grade but not the whole 40%.

3. **Missing Object-Oriented Concepts → MINUS 30%** of Operation Grade
   - ❑ **Separate class** for each item and action
   - ❑ Each class does its **job**. No class is performing the job of another class.
   - ❑ **Polymorphism:** use of pointers and virtual functions
   - ❑ **See the "Implementation Guidelines" in the Appendix which contains all the common mistakes that violates object-oriented concepts.**

4. **For each corner case** that is not working → **MINUS 10% to 20%** of the Operation Grade according to instruction evaluation.

Notes:

- ❑ The code of any operation does NOT compensate for the absence of any other operation.

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

- ✓ The grade of **each student** will be divided as follows:
  - **[70%]** of the student grade is on his individual work (the project part he was responsible for).
  - **[25%]** of the student grade is on integrating his work with the work of ALL students who Finished or nearly finished their project part.
  - **[5%]** of the student grade is on cooperation with other team members and helping them if they have any problems implementing their parts (helping does NOT mean implementing their parts).

- ✓ If one or more members didn't make their work, the other members will NOT be affected **as long as** the students who finished their part integrated all their parts together AND their part can be tested to see the output.
- ✓ If a student couldn't finish his part, try to still integrate it with the other parts to at least take the integration grade.

You should **inform the TAs** before the deadline **with a sufficient time (some weeks before)** if any problems happened between team members to be able to take grading action appropriately.

APPENDIX A

[I] Implementation Guidelines

- **Any user operation is performed in 3 steps:**
  - Get user action type.
  - Create suitable action object for that action type.
  - Execute the action (i.e. function **Action::Execute()** which first calls **ReadActionParameters()** then executes the action).

- **Use of Pointers/References**: Nearly all the parameters passed/returned to/from the functions should be pointers/references to be able to exploit **polymorphism** and **virtual** functions. **GameObjectList** should be an array of **Base Pointers (GameObject)** to be able to point to any game object type. Many class members should be pointers for the same reason

- **Classes' responsibilities:** Each class must perform tasks that are related to its responsibilities only. No class performs the tasks of another class. For example, when class **Obstacle** needs to draw itself on the GUI, it calls function ***Output::DrawObstacle*** because dealing with the GUI window is the responsibly of class **Output**. Similarly, class **Grid** must not contain any logic. It only should call functions. Read the "main classes" section to know the responsibility of each class.

- **Abusing Getters:** Don't use getters to get data members of a class to make its job inside another class. This breaks the classes' responsibilities rule. For example, do NOT add in **Grid** class function **GetGameObjectList()** that gets the 2D array of game objects to the other classes to loop on it and use it there. **GameObjectList** and looping on it are the responsibility of the Grid class only.

- **Virtual Functions:** In general, when you find some functionality (e.g. saving) that has different implementation based on each type, you should make it virtual function in the base class and override it in each derived class with its own implementation.
  - A common mistake here is the **abuse** of **dynamic_cast** (or similar implementations like **type** data member) by checking the object type outside the class and perform this class's job there (this job should be inside that class in a virtual member function in it).
  - This does not mean you should never use **dynamic_cast** but do NOT use it in a way that breaks the constraint of class responsibilities or replaces the use of virtual functions.

- **Not all the actions** need to add a corresponding function inside **Grid**. This will make **Grid** perform the responsibility of these actions. However, some actions need to loop on **GameObjectList** (e.g. ***SaveGridAction*** …etc.). In this case only (looping on GameObjectList), you can add functions for them in **Grid** that loop on the list and just call functions without making any further logic.

- You are not allowed to use **global variables** in your implemented part of the project, use passing variables as function parameters instead. That is better from software engineering point of view.

- You need to get instructor approval before using **friendships.**

[II] Workload Division Guidelines

Workload must be distributed among team members. A first question to the team at the project discussion and evaluation is "who is responsible for what?" An answer like" we all worked together" is a **failure** and will be penalized.

Here is a recommended way to divide the work based on **Actions**

- Divide workload by assigning some actions to each team member. Each member takes an action, should make any needed changes in any class involved in that action then run and try this action and see if it performs its operation correctly then move to another action.

- For example, the member who takes action **'SaveGrid'** should create **'SaveGridAction'** and write the code related to **SaveGridAction** inside **'Grid'** and **'GameObject'** derived classes. Then run and check if the game objects are successfully saved. Don't wait for the whole project to finish to run and test your implemented action.

- It is recommended to give similar actions to the same member because they have similar implementation.

- After finishing and trying few related actions, it's recommended to integrate them with the last integrated version and any subsequent divided action should increment on this project version and so on. We call this ***'Incremental Implementation'***.

- It's recommended to first divide the actions that other actions depend on (e.g. adding and creating grids) then integrate before dividing the rest of the actions.

- In the game logic, you may divide by enemy type, …etc. Each member takes an enemy type and totally handles it.