VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

PROGRAMMING FUNDAMENTALS - CO1027

ASSIGNMENT 2

HOW TO TRAIN YOUR DRAGON
Part 2: The War with the Dragon Lord

Version 1.0

Ho Chi Minh City, 07/2025

ASSIGNMENT SPECIFICATION

Version 1.0

1 Learning Outcomes

After completing this major assignment, students will review and master:

- Functions and function calls.
- File read/write operations.
- Pointers and dynamic memory allocation.
- Object-Oriented Programming.
- Singly linked lists.

2 Introduction

After a challenging training journey, the dragon warriors had come to understand and form deep bonds with their companions. However, during a special flight training session, Toothless - Hiccup's loyal dragon - unexpectedly led him through dense cloud layers and into a place never mentioned on any map: the Giant Dragon's Cave.

Here, Hiccup witnessed a horrifying truth: thousands of dragons from all over had been subjugated and were living as slaves under the control of a giant, cruel, and merciless **Dragon Lord**. This very dragon forced other dragon species to leave their nests, fly out, and plunder food from villages - including Berk. If they failed to bring back food, they would become the next victims, serving as prey for the Dragon Lord.

Although dragons are inherently peaceful and loyal creatures, they were forced to obey in the face of the Dragon Lord's brutal threats. The dragons' peaceful nature was twisted into violence and destruction, causing the relationship between humans and dragons to become strained.

Realizing the truth behind the attacks on the village of Berk and the long years of loss, the young warriors and their dragon companions made a bold decision together: to launch an all-out assault on the Dragon Lord's lair. Their goal was not only to destroy the evil alpha dragon but, more importantly, to free the other dragons from their chains of slavery and restore peace and trust between humans and dragons.

The Dragon Liberation campaign officially began. Each warrior will lead a team, traversing treacherous areas, facing various minion dragons, navigating dense mazes within the lair, and overcoming dangerous traps to advance deep into the heart of the dark base. Only by defeating the Dragon Lord can light shine once again upon the land of Berk and all dragon species.

In this major assignment, you are required to implement classes to simulate the events of this war. The details of the classes to be implemented will be specified in the tasks below.

3 Classes in the Program

This assignment uses Object-Oriented Programming (OOP) to recreate the battle with the Dragon Lord. The objects in the assignment are represented through classes and described as follows.

3.1 Map Elements

The maze on the mysterious island - believed to be the lair of the Dragon Lord - is represented by a map of size (nr, nc). This map is a two-dimensional array of nr rows and nc columns, where each element represents a specific terrain area on the island, described by the class **MapElement**. The map consists of three main types of elements:

- **PATH**: Represents a walkable path where objects can move.
- **OBSTACLE**: Represents an obstacle where objects cannot move.
- **GROUND_OBSTACLE**: Represents underground obstacles such as ancient roots, large rocks, or traces of ancient dragons. These obstacles only affect the Ground Warriors team and do not hinder the Dragon Lord or the Fly Warriors team. Specifically, Ground Warriors can only move through a `GROUND_OBSTACLE` cell if their `damage` exceeds the required threshold for that obstacle. Otherwise, they cannot move in that direction.

Additionally, objects are only allowed to move within the map.

The enum **ElementType** is defined as follows:

```
enum ElementType { PATH, OBSTACLE, GROUND_OBSTACLE };
```

Requirement: **Implement** the class **MapElement** with the following specifications:

1. A private attribute named **type** of type **ElementType**, representing the type of the map element.

2. A private attribute named **req_dmg** of type **int**, representing the required damage to pass a GROUND_OBSTACLE.
3. A constructor (public) with one parameter of type **ElementType**. The constructor assigns the parameter value to the **type** and **req_dmg** attributes.

```
MapElement(ElementType type, int in_req_dmg = 0);
```

4. A virtual destructor with public access.
5. A public method **getType** defined as below:

```
ElementType getType() const;
```

6. A public method **getReqDmg** defined as below:

```
int getReqDmg() const;
```

Requirement: **Implement** the method **getReqDmg** with the following details:

The class **MapElement** must assign the value of **in_req_dmg** to **req_dmg**. The method **getReqDmg** returns the value of the **req_dmg** attribute. The value of **in_req_dmg** is calculated using the formula:

```
in_req_dmg = (r * 257 + c * 139 + 89 ) % 900 + 1
```

where r and c represent the row and column indices of the GROUND_OBSTACLE cell.

3.2 Map

Requirement: **Implement** the class **Map** to represent the map as follows:

1. Private attributes **num_rows** and **num_cols**, both of type **int**, storing the number of rows and columns of the map.
2. A private attribute **map**, representing a 2D array where each element is a **MapElement**. Students should choose a data type suitable for **map** to support polymorphism. Hint: consider using either **MapElement**** or **MapElement***.
3. A public constructor with the following signature:

```
Map(int num_rows, int num_cols, int num_obstacle, Position *
    array_obstacle, int num_ground_obstacle, Position *
    array_ground_obstacle);
```

Where:

- **num_rows**, **num_cols**: the number of rows and columns.

- **num_obstacle**: the number of Wall objects.
- **array_obstacle**: an array of **Position** objects representing the positions of Walls. This array has **num_obstacle** elements. The class **Position** will be described later.
- **num_ground_obstacle**: the number of FakeWall objects.
- **array_ground_obstacle**: an array of **Position** objects representing the positions of **GROUND_OBSTACLE** elements. This array has **num_ground_obstacle** elements.

The constructor must create a 2D array where each element is an appropriate object. If the position of an element exists in **array_obstacle**, it is an OBSTACLE. If the position exists in **array_ground_obstacle**, it is a GROUND_OBSTACLE. The remaining elements are PATH.

4. A public destructor that deallocates all dynamically allocated memory.
5. Additional methods will be required as described in later sections.

3.3 Position

The **Position** class represents a position in the program. **Position** can be used to store the position of map elements (such as OBSTACLE or GROUND_OBSTACLE) or the positions of moving objects on the map.

Requirement: **Implement** the **Position** class with the following details:

1. Two private attributes named **r_** and **c_**, both of type **int**, representing the row and column positions.
2. A public constructor with two parameters **r** and **c**, which are assigned to the row and column attributes. These parameters have default values of 0.

```
Position(int r=0, int c=0);
```

3. A public constructor with one parameter **str_pos**, representing a position in string format. The format of **str_pos** is "(<r>,<c>)", where <r> and <c> are the row and column values.

```
Position(const string & str_pos); // Example: str_pos = "(1,15)"
```

4. Four getter and setter methods for the row and column attributes, with the following signatures:

```
int getRow() const ;
int getCol() const ;
void setRow(int r) ;
```

```
void setCol(int c) ;
```

5. The method **str** returns a string representing the position information. The return format is the same as the **str_pos** format in the constructor. For example, with $r = 1, c = 15$, **str** returns **"(1,15)"**.

```
string str() const
```

6. The method **isEqual** takes two parameters **in_r** and **in_c**, representing a position. It returns **true** if the input position matches the object's position, otherwise returns **false**.

```
bool isEqual(int in_r, int in_c) const
```

7. Additional methods may be required in later sections.

3.4 Moving Objects

The abstract class **MovingObject** is used to represent moving objects in the program such as FlyTeam, GroundTeam, and DragonLord.

Requirement: **Implement** the abstract class **MovingObject** with the following specifications:

1. **Protected attributes:**
   - **index**: type *int*, representing the position of the moving object in the array of moving objects, which will be described later.
   - **pos**: type *Position*, representing the current position of the moving object.
   - **map**: type *Map **, the map where this object moves.
   - **name**: type *string*, representing the name of the moving object.
2. A public constructor with parameters **index**, **pos**, **map**, and **name** corresponding to the attributes with the same name. The constructor assigns the parameter values to their respective attributes. The **name** parameter has a default value of **""**.

```
MovingObject(int index, const Position pos, Map * map, const string &
    name="")
```

3. A virtual destructor with public access.
4. A pure virtual method **getNextPosition** that returns the next **Position** where the object will move. If there is no available position to move, we define a special value for this method to return, stored in the **npos** variable of the **Position** class. When no valid position exists, the method returns **npos**.

```
virtual Position getNextPosition() = 0;
```

5. The method **getCurrentPosition** returns the current **Position** of the moving object.

```
Position getCurrentPosition() const;
```

6. The method **move** performs one movement step of the object.

```
virtual void move() = 0;
```

7. A pure virtual method **str** that returns a string representing information about the object.

```
virtual string str() const = 0;
```

Requirement: Define the variable **npos** (not position) in the **Position** class to represent the case where there is no valid position for the object to move. The **npos** has $r = -1$ and $c = -1$. Declare it as:

```
static const Position npos;
```

Requirement: In the **Map** class, define the method **isValid** to check whether a given position **pos** is valid for the **mv_obj** object to move to. A valid position depends on both the moving object and the map element. For example, FlyTeam can move through GROUND_OBSTACLE, but GroundTeam requires sufficient damage (DMG) to move through. Students need to read the detailed description of this assignment to implement this method correctly.

```
bool isValid(const Position & pos, MovingObject * mv_obj) const;
```

3.5 Warrior

The **Warrior** class represents the team of warriors in the program. There are two types of warrior teams in this battle: **FlyTeam** and **GroundTeam**, which will be described in detail later.

The **Warrior** class inherits from the **MovingObject** class. Therefore, **Warrior** must implement all pure virtual methods of **MovingObject**.

Requirement: Students must implement the **Warrior** class as described below. **Students can propose additional attributes, methods, or classes to support the implementation of the classes in this assignment.**

1. A public constructor as defined below. This constructor has additional parameters compared to **MovingObject**:

```
Warrior(int index, const Position & pos, Map * map, const string & name,
    int init_hp, int init_damage);
```

- **init_hp**: The initial HP of the Warrior. HP is in the range $[0, 500]$. If HP exceeds 500, it is set to 500. If HP equals 0, the Warrior is considered exhausted and cannot continue moving in the maze.
  - If both FlyTeam and GroundTeam have HP = 0, the dragon warriors lose the battle against the Dragon Lord.
  - If FlyTeam has HP = 0 but GroundTeam still has HP, the dragon warriors still lose the battle against the Dragon Lord.
  - If FlyTeam has HP $>$ 0 but GroundTeam has HP = 0, the FlyTeam can still continue the fight.
- **init_damage**: The initial damage of the Warrior. Damage is in the range $[0, 900]$. If damage exceeds 900, it is set to 900. If damage equals 0, the Warrior also cannot continue moving in the maze.
- The **name** parameter of the **MovingObject** constructor is passed the name of the warrior team.

2. **Warrior** has additional attributes **hp**, **damage**, and **bag** (of class **BaseBag**).
3. Four getter and setter methods for **hp** and **damage** are defined as:

```
int getHP() const ;
int getDamage() const ;
void setHP(int hp) ;
void setDamage(int damage) ;
```

4. The method **getBag** returns the **BaseBag** instance (this class will be described later).

```
BaseBag* getBag() const;
```

3.6 FlyTeam

The **FlyTeam** class represents the aerial warrior team in the program. The **FlyTeam** class inherits from the **Warrior** class.

Requirement: Students must implement the **FlyTeam** class with the following specifications. **Students may propose additional attributes, methods, or classes to support the implementation of the classes in this project.**

1. A public constructor is declared as follows. This constructor includes additional parameters compared to those in **Warrior**:

```
FlyTeam(int index, const string & moving_rule, const Position & pos, Map
    * map, int init_hp, int init_damage);
```

- **moving_rule**: describes how the FlyTeam moves. It is a string where each character can only be one of four values: 'L' (Left), 'R' (Right), 'U' (Up), 'D' (Down). For example, **moving_rule** = "LU".

2. FlyTeam has additional private attributes **(string) moving_rule** and **(int) moving_index**.
   - **moving_rule**: as described above.
   - **moving_index**: stores the index of the current character in **moving_rule** that is used to determine the next move direction.
3. The public method **getNextPosition** returns the next position where FlyTeam will move. FlyTeam moves based on **moving_rule**. Each time this method is called, the next character is used to determine the movement direction. The first time the method is called, the first character is used. When the last character has been used, it loops back to the beginning. For example, with **moving_rule** = "LR", the order of movement is: 'L', 'R', 'L', 'R', 'L', 'R', ... If the returned Position is not valid for this object to move to, return **npos** from the **Position** class.
4. The public method **move** performs a single movement step of FlyTeam. Inside **move**, **getNextPosition** is called; if the returned value is not **npos**, FlyTeam moves there. Otherwise, FlyTeam stays in place.
5. The public method **attack** performs the attack and finishing move of FlyTeam when facing the Dragon Lord. Only FlyTeam can defeat the Dragon Lord.

```
bool attack(DragonLord *dragon);
```

6. The **str** method returns a string formatted as follows:

   ```
   FlyTeam[index=<index>;pos=<pos>;moving_rule=<moving_rule>]
   ```

   Where:
   - <index> is the value of the *index* attribute.
   - <pos> is the string representation of the *pos* attribute.
   - <moving_rule> is the value of the *moving_rule* attribute.

3.7 GroundTeam

The **GroundTeam** class represents the ground warrior team in the program. The **GroundTeam** class inherits from the **Warrior** class.

Requirement: Students must implement the **GroundTeam** class with the following specifications. **Students may propose additional attributes, methods, or classes to support the implementation of the classes in this project.**

1. A public constructor is declared as follows. This constructor includes additional parameters compared to those in **Warrior**, similar to **FlyTeam**:

```
GroundTeam(int index, const string & moving_rule, const Position & pos,
    Map * map, int init_hp, int init_damage);
```

2. GroundTeam has additional private attributes: **(string) moving_rule**, **(int) moving_index**, and **(int) trap_turns**.
   - **trap_turns**: GroundTeam's primary role is to restrain rather than defeat the Dragon Lord. Upon encountering the Dragon Lord, they will attempt to trap and hold it for `N` turns, creating an opportunity for FlyTeam to strike. The variable `trap_turns` represents the number of turns GroundTeam can hold off the Dragon Lord.
3. The methods **getNextPosition**, **move**, and **str** work similarly to FlyTeam.
4. The public method **trap** performs the restraining action when confronting the Dragon Lord.

```
bool trap(DragonLord *dragon);
```

5. Two getter and setter methods for the `trap_turns` attribute are declared as follows:

```
int getTrapTurns() const;
void setTrapTurns(int turns);
```

3.8 Dragon Lord

The **DragonLord** class represents the Dragon Lord in the program. This class inherits from the **MovingObject** class.

The Dragon Lord possesses a unique sensing ability - it can detect the "disturbances" in the space caused by the presence of other dragons. Thanks to this, it can locate the warrior teams, especially the dragon riders in the sky.

Because of this, the Dragon Lord's movement pattern differs from the warrior teams: The next position that the Dragon Lord moves to is determined based on the coordinates of the two aerial warrior teams: **FlyTeam1** and **FlyTeam2**.

Let (x1,y1) be the current position of **FlyTeam1**, and (x2,y2) be the current position of **FlyTeam2**.

Note: In each battle, there are always two aerial teams and one ground team participating.

- The x-coordinate of DragonLord = |x1 - x2|
- The y-coordinate of DragonLord = |y1 - y2|
- After determining the coordinates (x,y) of the Dragon Lord, the program must check the Manhattan distance between (x,y) and (x1,y1), and (x,y) and (x2,y2). The Manhattan distance between two points P1 $(x1, y1)$ and P2 $(x2, y2)$ is:

$$|x1 - x2| + |y1 - y2|$$

- Based on these distances:
  - If both distances are greater than 5, keep (x, y) and use it as the next movement position of the Dragon Lord.
  - If one or both distances are less than or equal to 5, swap the coordinates (x, y) to become (y, x). The new position (y, x) will be used as the next move.

Requirement: **Implement** the **DragonLord** class similar to the **Warrior** class with the following differences:

1. The public constructor is declared as follows. Some parameters are similar to those in **Warrior**, with the following differences:

```
DragonLord(int index, const Position & init_pos, Map * map, FlyTeam *
    flyteam1, FlyTeam *flyteam2);
```

   - **flyteam1** and **flyteam2** are pointers to the FlyTeam1 and FlyTeam2 objects. Through these pointers, the current positions of the two teams can be retrieved.
   - The **name** parameter of the **MovingObject** constructor is set to "DragonLord".

2. The **str** method returns a string formatted as:

```
DragonLord[index=<index>;pos=<pos>]
```

3. **Students may propose additional attributes, methods, or classes to support the implementation of this class.**

3.9 Array of Moving Objects

The **ArrayMovingObject** class represents an array of moving objects. During the program execution, this array is iterated from start to end, and the **move** method of each element is called so that each object performs one movement step.

Requirement: Implement the **ArrayMovingObject** class with the following specifications:

1. Private attributes:
   - **arr_mv_objs**: an array of moving objects (MovingObject). Each element in the array must support polymorphism. Students can propose a suitable data type for this variable.
   - **count**: type *int*, the current number of elements in the array.
   - **capacity**: type *int*, the maximum number of elements in the array.
2. Public methods:
   - The constructor **ArrayMovingObject** receives a parameter to initialize the **capacity** attribute. This constructor must also allocate memory accordingly.
   - The destructor of the class must deallocate all dynamically allocated memory.
   - The **isFull** method returns **true** if the array is full; otherwise, it returns **false**. The array is considered full when the current number of elements equals the maximum capacity.

     ```
     bool isFull() const;
     ```
   - The **add** method appends a new moving object to the end of the array if the array is not full, and then returns **true**. Otherwise, it does not add the new object and returns **false**.

     ```
     bool add(MovingObject * mv_obj);
     ```
   - The **str** method returns a string representation of the **ArrayMovingObject**.

     ```
     string str() const;
     ```

     The returned string is formatted as:

     ```
     ArrayMovingObject[count=<count>;capacity=<capacity>;<MovingObject1>;...]
     ```

Where:

- **count**, **capacity**: represent the current number of elements and the maximum capacity of the **ArrayMovingObject**.
- **MovingObject1**, ...: the moving objects stored in the array. Each MovingObject is printed using its respective **str** format.

Example output of the **str** method of **ArrayMovingObject**:

```
ArrayMovingObject[count=3;capacity=10;DragonLord[index=0;pos=(8,9)];
    ↪ FlyTeam1[index=1;pos(1,4);moving_rule=RUU];FlyTeam2[index=2;
    ↪ pos=(2,1);moving_rule=LU];GroundTeam[index=3;pos(8,8);
    ↪ moving_rule=UUL]]
```

3.10 Program Configuration

A configuration file is used to provide settings for the program. The file consists of lines, each of which can follow one of the formats below. The order of the lines can vary.

1. MAP_NUM_ROWS=<nr>
2. MAP_NUM_COLS=<nc>
3. MAX_NUM_MOVING_OBJECTS=<mnmo>
4. ARRAY_OBSTACLE=<ao>
5. ARRAY_GROUND_OBSTACLE=<ago>
6. FLYTEAM1_MOVING_RULE=<f1mr>
7. FLYTEAM1_INIT_POS=<f1ip>
8. FLYTEAM1_INIT_HP=<f1ih>
9. FLYTEAM1_INIT_DAMAGE=<f1id>
10. FLYTEAM2_MOVING_RULE=<f2mr>
11. FLYTEAM2_INIT_POS=<f2ip>
12. FLYTEAM2_INIT_HP=<f2ih>
13. FLYTEAM2_INIT_DAMAGE=<f2id>
14. GROUNDTEAM_MOVING_RULE=<gmr>
15. GROUNDTEAM_INIT_POS=<gip>
16. GROUNDTEAM_INIT_HP=<gih>
17. GROUNDTEAM_INIT_DAMAGE=<gid>
18. DRAGONLORD_INIT_POS=<dip>
19. NUM_STEPS=<ns>

Where:

1. <nr> is the number of rows of the map, for example:
   `MAP_NUM_ROWS=10`
2. <nc> is the number of columns of the map, for example:
   `MAP_NUM_COLS=10`
3. <mnmo> is the maximum number of moving objects, for example:
   `MAX_NUM_MOVING_OBJECTS=10`
4. <ao> is the list of OBSTACLE positions. This list is enclosed in square brackets "[]", and positions are separated by a semicolon ';'. Each position is a pair of values enclosed in parentheses "()", where row and column values are separated by a comma. For example:
   `ARRAY_OBSTACLE=[(1,2);(2,3);(3,4)]`
5. <ago> is the list of GROUND_OBSTACLE positions, with the same format as <ao>. For example:
   `ARRAY_GROUND_OBSTACLE=[(4,5)]`
6. <f1mr> is the **moving_rule** string of FlyTeam1. Example:
   `FLYTEAM1_MOVING_RULE=RUU`
7. <f1ip> is the initial position of FlyTeam1. Example:
   `FLYTEAM1_INIT_POS=(1,3)`
8. <f2mr> is the **moving_rule** string of FlyTeam2. Example:
   `FLYTEAM2_MOVING_RULE=LLD`
9. <f2ip> is the initial position of FlyTeam2. Example:
   `FLYTEAM2_INIT_POS=(4,8)`
10. <gmr> is the **moving_rule** string of GroundTeam. Example:
    `GROUNDTEAM_MOVING_RULE=LU`
11. <gip> is the initial position of GroundTeam. Example:
    `GROUNDTEAM_INIT_POS=(2,1)`
12. <dip> is the initial position of the Dragon Lord. Example:
    `DRAGONLORD_INIT_POS=(7,9)`
13. <ns> is the number of simulation steps. In each step, the program iterates through all MovingObjects and lets them perform one movement step. Example:
    `NUM_STEPS=100`
14. <f1ih>, <f1id> are the initial HP and DMG of FlyTeam1.
15. <f2ih>, <f2id> are the initial HP and DMG of FlyTeam2.
16. <gih>, <gid> are the initial HP and DMG of GroundTeam.

Requirement: Implement the **Configuration** class that represents a configuration of

the program by reading the configuration file. The **Configuration** class has the following specifications:

1. Private attributes:
   - **map_num_rows**, **map_num_cols**: number of rows and columns of the map.
   - **max_num_moving_objects**: type *int*, corresponding to `<mnmo>`.
   - **num_obstacles**: type *int*, number of OBSTACLE objects.
   - **arr_obstacles**: type *Position**, corresponding to `<ao>`.
   - **num_ground_obstacles**: type *int*, number of GROUND_OBSTACLE objects.
   - **arr_ground_obstacles**: type *Position**, corresponding to `<ago>`.
   - **flyteam1_moving_rule**: type *string*, corresponding to `<f1mr>`.
   - **flyteam1_init_pos**: type *Position*, corresponding to `<f1ip>`.
   - **flyteam1_init_hp**: type *int*, corresponding to `<f1ih>`.
   - **flyteam1_init_damage**: type *int*, corresponding to `<f1id>`.
   - **flyteam2_moving_rule**: type *string*, corresponding to `<f2mr>`.
   - **flyteam2_init_pos**: type *Position*, corresponding to `<f2ip>`.
   - **flyteam2_init_hp**: type *int*, corresponding to `<f2ih>`.
   - **flyteam2_init_damage**: type *int*, corresponding to `<f2id>`.
   - **groundteam_moving_rule**: type *string*, corresponding to `<gmr>`.
   - **groundteam_init_pos**: type *Position*, corresponding to `<gip>`.
   - **groundteam_init_hp**: type *int*, corresponding to `<gih>`.
   - **groundteam_init_damage**: type *int*, corresponding to `<gid>`.
   - **dragonlord_init_pos**: type *Position*, corresponding to `<dip>`.
   - **num_steps**: type *int*, corresponding to `<ns>`.
2. The constructor **Configuration** is declared as follows. It accepts **filepath**, a string representing the path to the configuration file. The constructor initializes the attributes accordingly.

```
Configuration(const string & filepath);
```

3. The destructor must deallocate all dynamically allocated memory.
4. The **str** method returns a string representation of Configuration.

```
string str() const;
```

   The format of the returned string is:

```
Configuration[
<attribute_name1>=<attribute_value1>...
]
```

Example output of the **str** method of **Configuration**:

```
Configuration[
MAP_NUM_ROWS=10
MAP_NUM_COLS=10
MAX_NUM_MOVING_OBJECTS=10
NUM_OBSTACLE=3
ARRAY_OBSTACLE=[(1,2);(2,3);(3,4)]
NUM_GROUND_OBSTACLE=1
ARRAY_GROUND_OBSTACLE=[(4,5)]
FLYTEAM1_MOVING_RULE=RUU
FLYTEAM1_INIT_POS=(1,3)
FLYTEAM1_INIT_HP=250
FLYTEAM1_INIT_DMG=500
FLYTEAM2_MOVING_RULE=LLD
FLYTEAM2_INIT_POS=(1,3)
FLYTEAM2_INIT_HP=350
FLYTEAM2_INIT_DMG=400
GROUNDTEAM_MOVING_RULE=LU
GROUNDTEAM_INIT_POS=(2,1)
GROUNDTEAM_INIT_HP=300
GROUNDTEAM_INIT_DMG=350
DRAGONLORD_INIT_POS=(7,9)
NUM_STEPS=100
]
```

Each attribute and its corresponding value are printed in the order listed in the "Private attributes" section of this class.

3.11 SmartDragon

During the movement of the Dragon Lord, after every **5 steps** that the Dragon Lord takes, a Smart Dragon will be created. Note that if the **getNextPosition** method of **DragonLord** does not return a valid position to move to, the **move** method will not execute a movement step, and this will not be counted as a step. Each Smart Dragon type inherits from the **MovingObject** class. Each Smart Dragon can move on **PATH** or **GROUND_OBSTACLE**, but cannot move on **OBSTACLE**. Once created, a Smart Dragon will be added to the array of moving objects (**ArrayMovingObject**) via the **add** method of the **ArrayMovingObject** class. If the array has reached its maximum capacity, the Smart Dragon will not be created.

After every **5 steps** of the Dragon Lord, a Smart Dragon will be created at the **previous position where the Dragon Lord stood**. The Smart Dragon types and their creation conditions are as follows:

- If this is the first Smart Dragon created on the map, it will be of type **SD1**. Otherwise, consider the distance from the Smart Dragon to FlyTeam1, FlyTeam2, and GroundTeam:
- If the distance to FlyTeam1 is the smallest: Create a **SD1** Smart Dragon.
- If the distance to FlyTeam2 is the smallest: Create a **SD2** Smart Dragon.
- If the distance to GroundTeam is the smallest: Create a **SD3** Smart Dragon.
- If there is a tie between teams, prioritize creating the type of Smart Dragon that currently has the least number on the map (the first one found with the minimum count).

The Smart Dragon types are defined in the **DragonType** enum as follows:

```
enum DragonType { SD1, SD2, SD3 };
```

Requirement: Students must implement all necessary classes related to Smart Dragons according to the following rules (with similarities to other characters but some changes):

1. All Smart Dragon types share the following attributes:
   - **smartdragon_type**: of type **DragonType**, representing the type of Smart Dragon.
   - **damage**: of type **int**, representing the damage the Smart Dragon can inflict.
   - **item**: of type **BaseItem ***, representing the item that the Smart Dragon drops upon defeat. The details of items will be described in a later section.
   - **target**: of type **MovingObject ***, a pointer to the object that the Smart Dragon is targeting:
     - **SD1**: The **target** points to FlyTeam1.

  - **SD2**: The **target** points to FlyTeam2.
  - **SD3**: The **target** points to GroundTeam.

2. The public constructor of each Smart Dragon is defined as follows:

```
SD1: SmartDragon(int index, const Position & init_pos, Map * map,
DragonType type, MovingObject * flyteam1, int damage);

SD2: SmartDragon(int index, const Position & init_pos, Map * map,
DragonType type, MovingObject * flyteam2, int damage);

SD3: SmartDragon(int index, const Position & init_pos, Map * map,
DragonType type, MovingObject * groundteam, int damage);

```

3. The **getNextPosition** method (public): Each Smart Dragon moves according to its own rule:
   - **SD1:** Moves 1 step closer to FlyTeam1 based on Manhattan distance. Movement priority: UP, RIGHT, DOWN, LEFT (clockwise).
   - **SD2:** Moves 1 step closer to FlyTeam2 based on Manhattan distance. Movement priority: UP, RIGHT, DOWN, LEFT (clockwise).
   - **SD3:** Does not move (stays in place).
4. The **move** method (public): If the next position is not **npos**, move to that position.
5. The **str** method (public): The returned string format is:

```
smartdragon[pos=<pos>;type=<dragon_type>;tg=<target>]
␣␣␣␣
```

   Where:
   - <**pos**>: Current position of the Smart Dragon.
   - <**dragon_type**>: Either SD1, SD2, or SD3.
   - <**target**>: The object (team) the Smart Dragon is currently targeting.

3.12 Items

The **BaseItem** class is an abstract class representing an item. It has two public pure virtual methods:

- **canUse**

```
virtual bool canUse(Warrior* w) = 0;
```

This method returns `true` if the Warrior can use this item; otherwise, it returns `false`.

- **use**

```
virtual void use(Warrior* w) = 0;
```

This method applies the effects of the item to the Warrior, adjusting their attributes as needed. Some items have usage conditions. If the condition is not satisfied when the item is obtained, the item is temporarily stored in the team's bag. When the condition is later met, the Warrior checks the bag and uses the item if applicable.

The effects of each item type are as follows:

Item	**Effect**	**Usage Condition**
DragonScale	Contains magical power of Smart Dragons, boosting Warrior's strength by increasing **DMG** by 25% of its current value.	**exp** $\leq$ 400
HealingHerb	A healing herb that restores **HP** by 20%.	**hp** $\leq$ 100
TrapEnhancer	A tool that enhances GroundTeam's traps, increasing the trap duration by 1 turn.	None

Items are held by Smart Dragons:

- **SD1**: Carries DragonScale.
- **SD2**: Carries HealingHerb.
- **SD3**: Carries TrapEnhancer. However, TrapEnhancer only drops if SD3 is defeated by GroundTeam. If defeated by FlyTeam1 or FlyTeam2, it drops HealingHerb or DragonScale respectively.

Requirement: Students must implement the classes DragonScale, HealingHerb, and TrapEnhancer as described above.

3.13 Bag

Each warrior team is equipped with a bag to store collected items. The bag is implemented as a singly linked list. Actions include:

- Adding an item to the bag (method **insert**): Insert the item at the head of the list.
- Using any usable item (method **get**): Find the first usable item closest to the head. This item will be swapped with the first item in the list and then removed.
- Using a specific item (method **get(ItemType)**): If the bag contains the requested item type, it is used by swapping it with the first item (if not already at the head) and then removing it from the list. If multiple such items exist, the one closest to the head is chosen.

Each team has a maximum number of items it can carry: FlyTeam can hold up to **5 items**, and GroundTeam can hold up to **7 items**.

Requirement: Students must implement the bag class as follows:

The **BaseBag** class represents the bag. It has an attribute **warrior** of type **Warrior***, pointing to the team owning the bag. Its public methods include:

```
virtual bool insert (BaseItem* item); // returns true if inserted
    successfully
virtual BaseItem* get(); // returns the item as described above, NULL if not
     found
virtual BaseItem* get(ItemType itemType); // returns the item as described
    above, NULL if not found
virtual string str() const;
```

The output format of **str()** is:

```
Bag[count=<c>;<list_items>]
```

Where:

- <c>: current number of items in the bag.
- <list_items>: a string listing all items from head to tail, each represented by its class name, separated by commas.

Requirement: Students must implement a class that inherits from BaseBag to represent the different types of team bags, named **TeamBag**. Its constructor has only one parameter: **Warrior * warrior**.

3.14 DragonWarriorsProgram

If during movement the warrior teams encounter Smart Dragons or the Dragon Lord, the following events occur:

1. When warriors encounter a Smart Dragon:
   - SD1: If a team faces SD1, they fight. If the team's damage > SD1's damage, they win and receive the item SD1 carries. Otherwise, the team loses 100 HP.
   - SD2: Similar to SD1.
   - SD3: Similar to SD1. However, if GroundTeam wins, they receive TrapEnhancer.
2. When warriors encounter the Dragon Lord:
   - FlyTeam: Fights and can defeat the Dragon Lord, ending the game.
   - GroundTeam: Cannot directly fight the Dragon Lord. They use traps to hold the Dragon Lord for N turns. If FlyTeam reaches the Dragon Lord within N turns, they can defeat it. Otherwise, the Dragon Lord teleports the GroundTeam to position (x,y) → (y,x) and reduces their HP by 200.

Requirement: Students must design and implement the **DragonWarriorsProgram** class as follows:

(a) Attributes include: config (**Configuration***), flyteam1 (**FlyTeam***), flyteam2 (**FlyTeam***), groundteam (**GroundTeam***), dragonlord (**DragonLord***), map (**Map***), arr_mv_objs (**ArrayMovingObject**).

(b) Constructor takes one parameter, the path to the configuration file, and initializes all attributes. After initialization, **arr_mv_objs** must add *dragonlord*, *flyteam1*, *flyteam2*, and *groundteam* using **add**.

```
DragonWarriorsProgram(const string & config_file_path);
```

(c) The **isStop** method returns **true** if the program stops, otherwise **false**. The program stops when one of these conditions is met: both FlyTeams' HP = 1; all teams' HP = 1; or FlyTeam defeats the Dragon Lord.

```
bool isStop() const;
```

(d) The **printResult** and **printStep** methods print program information. These methods are provided and must not be modified.

(e) The **run** method takes a parameter **verbose**. If **verbose = true**, print the information of each MovingObject in ArrayMovingObject after each step, along with updates (e.g., FlyTeam1 encountering a Smart Dragon). This method runs for up to **num_steps** (from the config file). After each step, check for stop conditions, handle encounters, and create Smart Dragons if needed.

```
void run(bool verbose);
```

3.15 TestDragonWar

TestDragonWar is a class used to verify the attributes and behaviors of all classes in this assignment. Students must declare **TestDragonWar** as a friend class when defining all the classes above.

4 Requirements

To complete this assignment, students must:

(a) Read this entire description file.

(b) Download the file `initial.zip` and extract it. After extraction, students will obtain the files: `main.cpp`, `main.h`, `dragons.h`, `dragons.cpp`, and sample data files. Students must submit only two files: `dragons.h` and `dragons.cpp`, and must not modify `main.h` when testing the program.

(c) Use the following command to compile:
g++ -o main main.cpp dragons.cpp -I . -std=c++11
Use the following command to run the program:
./main
These commands are used in the command prompt/terminal to compile and run the program. If students use an IDE to run the program, they must ensure all necessary files are added to the project/workspace of the IDE and adjust the compilation command accordingly. IDEs usually provide buttons for building (Build) and running (Run). When clicking Build, the IDE typically compiles only `main.cpp`. Students must configure the IDE to include `dragons.cpp` and add the `-std=c++11` and `-I .` options.

(d) The program will be evaluated on a Unix-based platform. The student's local environment and compiler might differ from the grading environment. The submission platform (BKeL) is set up to match the grading environment. Students must test their program on the submission platform and fix all errors there to ensure correct results during grading.

(e) Modify only `dragons.h` and `dragons.cpp` to complete this assignment while ensuring:

- There must be exactly one **#include** statement in `dragons.h`, which is **#include "main.h"**, and one **#include** in `dragons.cpp`, which is **#include "dragons.h"**. No other **#include** statements are allowed in these files.
- Implement all the functions described in the tasks of this assignment.

(f) Students are encouraged to write additional classes and functions to complete this assignment.

(g) Students must provide implementations for all classes and methods listed in this assignment. Otherwise, they will receive a score of 0. If a class or method cannot be implemented, students must provide an empty implementation with only braces {}.

5 Submission

Students must submit only two files: `dragons.h` and `dragons.cpp`, before the deadline provided in the "Assignment 2 - Submission" section. A few simple test cases are used to check whether the student's submission compiles and runs. Students may submit as many times as they want, but only the latest submission will be graded. As the system may be overloaded near the deadline, students are advised to submit as early as possible. Submissions made too close to the deadline (within 1 hour) are at the student's own risk. After the deadline, the system will be closed and no submissions will be accepted. Submissions through other means will not be accepted.

6 Additional Rules

- Students must complete this assignment on their own and prevent others from copying their work. Violators will be subject to academic misconduct penalties according to school regulations.
- All decisions of the instructor in charge of the assignment are final.
- Students will not be provided with test cases after grading, but they will receive a score distribution of the assignment.
- The content of this assignment will be harmonized with a question on the final exam.

7 Assignment Harmony

The final exam will include questions related to the assignment. Suppose the student's assignment score is **a** (out of 10), and the total score for Harmony questions on the exam is **b** (out of 5). Let **x** be the final assignment score after Harmony. The Harmony score is calculated as follows:

- If $a = 0$ or $b = 0$, then $x = 0$.

- If both a and b are non-zero, then

$$x = \frac{a}{2} + HARM\left(\frac{a}{2}, b\right)$$

Where:

$$HARM(x, y) = \frac{2xy}{x + y}$$

Students must complete the assignment themselves. If a student cheats, they will not be able to answer the Harmony questions and will receive a score of 0 for the assignment. Students **must** complete the Harmony questions in the final exam. Failure to do so will result in a score of 0 for the assignment and failure of the course. **No exceptions or explanations will be accepted.**

8 Cheating

The assignment must be completed **individually**. A student is considered to have cheated if:

- There is unusual similarity between submissions. In such cases, **all** submissions involved will be considered cheating. Therefore, students must protect their own code.
- A student's code is submitted by another student.
- A student does not understand the code they wrote (except for the starter code). Students may refer to any materials but must understand all the code they write. If they do not understand code from a reference source, they are strongly warned not to use it.
- A student submits another student's work under their own account.
- A student uses auto-generated code from tools without understanding its meaning.

If cheating is confirmed, the student will receive a score of 0 for the entire course (not just the assignment).

NO EXPLANATIONS OR EXCEPTIONS WILL BE ACCEPTED!

After submissions, some students may be randomly called for interviews to verify that the submitted work is their own.

9 Changes Compared to Previous Version

...