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

SHERLOCK
A STUDY IN PINK - Part 2

Version 1.0

Ho Chi Minh City, 03/2024

ASSIGNMENT SPECIFICATION

Version 1.0

1 Learning outcomes

After completing this assignment, students review and make good use of:

- Function and calling function
- File Input/Output
- Pointer and dynamic memory allocation
- Object-oriented programming
- Singly linked list

2 Introduction

This assignment is based on episode 1 season 1 of a TV series on BBC named Sherlock. This series is a British mystery crime drama television series based on Sir Arthur Conan Doyle's Sherlock Holmes detective stories.

At the end of Part 1, a taxi driver appeared in front of Apartment 221B Baker Street. The driver had invited Sherlock to be a passenger on the next taxi. Sherlock knew that this driver was a criminal, but he still could not understand why the victim had committed suicide after a taxi ride. Sherlock chose to join this dangerous trip. Finally, Sherlock has been taken to a maze with the challenge that he can catch up with the driver.

On the other hand, some time after Sherlock had left the room, Watson reviewed the location signal on the laptop. He had seen that the phone's location was gradually moving away from the apartment. Watson has also taken another bus and has reached the maze. Together, he and Sherlock will hunt down criminals in a maze with many traps.

In this assignment, you are asked to implement classes to describe the crime-chasing process of Sherlock and Watson. Details of the classes that need to be implemented will be detailed in the tasks below.

3 Classes in this program

This assignment takes advantage of Object-oriented programming to best describe the process that Sherlock and Watson chase the criminals. Objects in this assignment are presented through classes and are detailed as below:

3.1 Map's elements

The maze in which Sherlock and Watson chase the criminal is represented by a map of size (nr, nc). This map is a 2-dimensional array of size nr rows and nc columns. Each element of the map is represented by the class **MapElement**. The map has 2 types of elements:

- **Path**: represents a path, objects can move on this element.
- **Wall**: represents a wall, objects cannot move on this element.
- **FakeWall**: represents a fake wall. Because the criminal created the maze, he can recognize the fake wall, and Sherlock, with his observation ability, can detect this fake wall. For Watson, FakeWall will be detected (and moved through) if Watson has EXP greater than the FakeWall's required EXP. All other movable objects cannot be moved on this element.

Besides, all of the objects could just move inside the map.

Define the enum **ElementType** as below:

```
enum ElementType { PATH, WALL, FAKE_WALL };
```

Requirements: **Implement** abstract class **MapElement** with the following description:

1. The protected attribute called **type** has the type of **ElementType** describing the type of the map's element.
2. Constructor method (public) which just has 1 input ElementType parameter. It is used to assign the value for attribute **type**.
   ```
   MapElement(ElementType in_type);
   ```
3. Virtual destructor with access modifier: public.
4. Method **getType** (public) is defined as below:
   ```
   virtual ElementType getType() const;
   ```

Requirements: **Implement** the concrete classes **Path**, **Wall**, **FakeWall inherited from** class **MapElement** with the following information:

1. Class **FakeWall** has got a private attribute called **req_exp** representing the minimum EXP that Watson needs to overcome the wall.
2. Each class has public methods declared as below. The implementation for each constructor method needs to call the constructor of class **MapElement** and assign appropriate values. Class **FakeWall** needs to assign value **in_req_exp** for **req_exp**.

```
// Constructor of class Path
Path();
// Constructor of class Wall
Wall();
// Constructor of class FakeWall
FakeWall(int in_req_exp);
```

3.2 Map

Requirements: **Implement** Class **Map** is used to illustrate the map with the following details:

1. Private attribute **num_rows** and **num_cols** are all in type of int, save the number of rows and cols respectively.
2. Private attribute **map** is used to decrbie a 2D array with each element is a MapElement. Students need to choose appropriate data type for **map** to show the polymorphism property. Recommendation: Consider 2 types: **MapElement**** and **MapElement***.
3. Public constructor with declaration:

```
Map(int num_rows, int num_cols, int num_walls, Position * array_walls,
    int num_fake_walls, Position * array_fake_walls);
```

In which:

- **num_rows**, **num_cols** is the number of rows and cols respectively
- **num_walls** is the number of Wall objects
- **array_walls** is an array of Position elements, describing the location of Wall objects. This array has **num_walls** elements. Details about **Position** class will be descired in the following section.
- **num_fake_walls** is the number of FakeWall objects
- **array_fake_walls** is an array of Position elements, describing the location of FakeWall objects. This array has **num_fake_walls** elements.

Constructor needs to create a 2D array which each element is an appropriate object. If this element is in the **array_walls**, this element is Wall object. Similarly, it is FakeWall object if it exists in **array_fake_walls**. The other elements are Path objects.

4. Public destructor deallocates the memory that has been allocated
5. Some more needed methods are mentioned in other sections

3.3 Position

Class **Position** describes the position in the program. Position is used to store the location of every map elements (such as locations of Wall, FakeWall) or the location of moving objects on the map.

Requirements: **Implement** class **Position** with the following description:

1. Two private attributes called **r** và **c** with the type of int, describe the location with row and column index respectively.
2. Public constructor method with two passing parameter **r** and **c** assign the value for row and column atrributes. These two attributes have the default values 0.

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

3. Public constructor with one parameter **str_pos** demonstrates a string of location. Format of **str_pos** is `"(<r>,<c>)"` with <r> and <c> are the values for row and column respectively.

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

4. 4 get, set methods for 2 row and column attributes with the following declarations:

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

5. Method **str** returns a string representing location information. The return format is the same as that of the string **str_pos** in the Constructor. For example, with $r = 1, c = 15$, **str** returns the value **"(1,15)"**.

```
string str() const
```

6. The **isEqual** method has two parameters **in_r** and **in_c** representing a position. **isEqual** returns **true** if the passed position matches the position of this object. Otherwise, **isEqual** returns **false**.

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

7. Some other methods may be required in the following sections.

3.4 Moving objects

Abstract class **MovingObject** is used to represent moving objects in shows such as Sherlock, Watson, criminals.

Requirements: **Implement** abstract class **MovingObject** with details:

1. Protected attributes:
   - **index**: type *int*, position of the moving object in the array of moving objects, this array will be described later.
   - **pos**: type *Position*, the current position of the moving object.
   - **map**: type *Map **, the map for this object to move in.
   - **name**: type *string*, name of the moving object.
2. Public constructor method with parameters **index**, **pos**, **map**, **name** has the same meaning as the attribute with the same name. The method assigns the value of the parameter to the property of the same name. The **name** parameter has a default value of "".

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

3. Virtual destructor with public access.
4. Pure virtual method **getNextPosition** returns the next Position this object moves to. On the other hand, in case there is no Position for the object to move to, we define a value to return to this method and store it in the variable **npos** of the class **Position**. When there is no Position to move to, the method returns **npos**.

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

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

```
Position getCurrentPosition() const = 0;
```

6. Pure virtual method **move** performs each movement of the object.

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

7. Pure virtual method **str** returns a string representing the object's information.

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

Requirements: Define the variable **npos** (not position) in class **Position** representing that there is no position for the object to move to. The variable **npos** has $r = -1$ and $c = -1$. The variable declaration is as follows:

```
static const Position npos;
```

Requirements: In class **Map**, define a method **isValid** that checks whether the location **pos** is a valid location for the object **mv_obj** to move to. A valid location for a move must depend on what the move object is and the map composition. For example, Sherlock can move on FakeWall but Watson needs to meet the EXP requirement. Students need to read the description in this textbook to implement the method correctly.

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

3.5 Sherlock

Class **Sherlock** performs for the character Sherlock in the program. Class **Sherlock** receives class **MovingObject** as its ancestor class. Therefore, the Sherlock class must implement the pure virtual methods of the MovingObject class.

Requirements: Students implement the Sherlock class with the requirements below. **Students can propose additional attributes, methods or other classes to support the implementation of classes in this assignment.**

1. Constructor (public) is declared as below. This constructor has some additional parameters besides the ones already in **MovingObject**:

```
Sherlock(int index, const string & moving_rule, const Position &
    init_pos, Map * map, int init_hp, int init_exp)
```

   - **moving_rule**: describes how Sherlock moves. This is a string whose characters can only be one of 4 values: 'L' (Left - go left), 'R' (Right - go right), 'U' (Up - go up) , 'D' (Down - go down). An example of **moving_rule** is "LU".
   - **init_hp**: Sherlock's initial HP. HP is within $[0, 500]$. If HP exceeds 500, it is set to 500. If HP is 0, Sherlock is considered to have run out of stamina and cannot continue moving in the maze. If both Sherlock and Watson's HP are 0, then Sherlock and Watson lose the chase with the criminal.
   - **init_exp**: Sherlock's initial EXP. EXP is around $[0, 900]$. If HP exceeds 900, set it to 900. If EXP is 0, Sherlock will not move further in the maze.
   - The **name** parameter of the MovingObject Constructor is passed in the value "Sherlock".
   - Sherlock has additional attributes **hp** and **exp**.

2. The **getNextPosition** (public) method returns the position of Sherlock's next move. Sherlock moves according to **moving_rule**. Each time the method is called, the next character is used as the direction of movement. The first time the method is called, the first character will be used. When the last character is used, it will return to starting the process from the first character. For example, with **moving_rule** = "LR", the order of characters used is: 'L', 'R', 'L', 'R', 'L', 'R',. .. If the returned Position is not a valid position for this object to move then return **npos** of class **Position**.
3. The **move** (public) method performs a Sherlock move. In **move** there is a call to **getNextMovePosition**, if the return value other than **npos** is a valid move then Sherlock will move there. If it wasn't a legal move, Sherlock would stand still.
4. The **str** method returns a string in the following format:

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

   In which:

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

3.6 Watson

Class **Watson** represents the character Watson on the show. Class **Watson** receives class **MovingObject** as its ancestor class.

Requirements: Students implement class **Watson** similar to class **Sherlock**. However, class **Watson** has the following change:

1. The **name** parameter of the MovingObject Constructor is passed in the value "Watson".
2. The **str** method returns a string in the following format:

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

   These attributes and methods have the same meaning as in class **Sherlock**.

3.7 Criminal

Class **Criminal** represents the criminal character in the show. Class **Criminal** receives class **MovingObject** as its ancestor class. The criminal has cameras monitoring both Sherlock and Watson in this maze. Therefore, unlike the detective couple's way of moving, the criminal will choose the next moving location as the valid location with the greatest total distance to Sherlock and Watson. In this assignment, when talking about distance, we are using the **Manhattan** distance type. The Manhattan distance type between two points P1 with coordinates $(x1, y1)$ and P2 with coordinates $(x2, y2)$ is:

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

In case there is more than 1 location with the greatest total distance to Sherlock and Waton, priority is given to choosing the location in the order of directions 'U', 'L', 'D', 'R'.

Requirements: **Implement** class **Criminal** is similar to class **Sherlock** with the following changes:

1. Public constructor is declared as below. Some parameters have the same meaning as class **Sherlock**. Some differences are:

```
Criminal(int index, const Position & init_pos, Map * map, Sherlock *
    sherlock, Watson * watson);
```

   - **sherlock**, **watson** are pointers to Sherlock and Watson objects, respectively. Through 2 pointers, we can get the current position of these two characters.
   - The **name** parameter of the MovingObject constructor is passed in the value "Criminal".

2. The **str** method returns a string in the following format:

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

3.8 Array of Moving Objects

Class **ArrayMovingObject** represents an array of moving objects. When the program runs, this array is traversed from beginning to end and the **move** method of each element is called so that each object makes one move.

Requirements: Implement class **ArrayMovingObject** with the following requirements:

1. Private attributes:
   - **arr_mv_objs**: array of moving objects (MovingObject). Each element in the array needs to demonstrate polymorphism. Students themselves suggest data types for variables.
   - **count**: type *int*, the current number of elements in the array.
   - **capacity**: type *int*, maximum number of elements in the array.
   - Constructor of **ArrayMovingObject** takes a parameter to initialize the **capacity** property. At the same time, the method needs to be allocated appropriately.
   - The class's destructor needs to reclaim dynamically allocated memory.
   - The **isFull** method returns **true** if the array is full, otherwise it returns **false**. The array is full if the current number of elements is equal to the maximum number of elements in the array.

```
    bool isFull() const;

```

   - The **add** method adds a new move object to the end of the array if the array is not full, then returns **true**. Otherwise, the method will not add a new object and returns **false**.

```
    bool add(MovingObject * mv_obj)
```

   - The **str** method returns a string representing information for **ArrayMovingObject**.

```
string str() const
```

     The format of the returned string is:

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

     In which:
     - count, capacity: The number and maximum number of elements of the object **ArrayMovingObject**
     - MovingObject1,...: These are the MovingObjects in the array, respectively. Each MovingObject is printed in the corresponding format of that object type.

     Example about return string of **str** method of **ArrayMovingObject**:

```
ArrayMovingObject[count=3;capacity=10;Criminal[index=0;pos=(8,9)];
    ↪ Sherlock[index=1;pos(1,4);moving_rule=RUU];Watson[index=2;pos
    ↪ =(2,1);moving_rule=LU]]
```

3.9 Program Configuration

A file used to contain configuration for a program. The file consists of lines, each line can be one of the following formats. Note that the order of lines may vary.

1. MAP_NUM_ROWS=<nr>
2. MAP_NUM_COLS=<nc>
3. MAX_NUM_MOVING_OBJECTS=<mnmo>
4. ARRAY_WALLS=<aw>
5. ARRAY_FAKE_WALLS=<afw>
6. SHERLOCK_MOVING_RULE=<smr>
7. SHERLOCK_INIT_POS=<sip>
8. WATSON_MOVING_RULE=<wmr>
9. WATSON_INIT_POS=<wip>
10. CRIMINAL_INIT_POS=<cip>
11. NUM_STEPS=<ns>

In which:

1. <nr> is the number of rows in 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 elements of the array of moving objects, for example:
   MAX_NUM_MOVING_OBJECTS=10
4. <aw> is a list of Wall locations. The list is enclosed in a pair of "[]" quotes, consisting of positions separated by a ';'. Each position is a pair of opening and closing brackets "()", inside is the row number and column number separated by a comma. For example:
   ARRAY_WALLS=[(1,2);(2,3);(3,4)]
5. <afw> is a list of locations of FakeWalls. This list has the same format as <aw>. For example:
   ARRAY_WALLS=[(4,5)]
6. <smr> is the string representation **moving_rule** of Sherlock. For example:
   SHERLOCK_MOVING_RULE=RUU
7. <sip> is Sherlock's original location. For example:
   SHERLOCK_INIT_POS=(1,3)

8. <wmr> is Watson's string representation **moving_rule**. For example:
   WATSON_MOVING_RULE=LU
9. <wip> is Watson's initial position. For example:
   WATSON_INIT_POS=(2,1)
10. <cip> is the criminal's initial location. For example:
    WATSON_INIT_POS=(7,9)
11. <ns> is the number of iterations the program will perform. In each loop, the program will browse through all MovingObjects and let these objects perform one move. For example:
    NUM_STEPS=100

Requirements: Implement class **Configuration** represents a program's configuration through reading the configuration file. Class **configuration** has the following descriptions:

1. Private attributes:
   - **map_num_rows**, **map_num_cols** are the number of rows and columns of the map, respectively.
   - **max_num_moving_objects**: type *int*, corresponding to <mnmo>.
   - **num_walls**: type *int*, number of Wall objects.
   - **arr_walls**: type *Position**, corresponding to <aw>.
   - **num_fake_walls**: type *int*, number of FakeWall objects.
   - **arr_fake_walls**: type *Position**, corresponding to <afw>.
   - **sherlock_moving_rule**: type *string*, corresponding to <smr>.
   - **sherlock_init_pos**: type *Position*, corresponding to <sip>.
   - **watson_moving_rule**: type *string*, corresponding to <wmr>.
   - **watson_init_pos**: type *Position*, corresponding to <wip>.
   - **criminal_init_pos**: type *Position*, corresponding to <cip>.
   - **num_steps**: type *int*, corresponding to <ns>.
2. Constructor **Configuration** is declared as below. The constructor accepts **filepath** which is a string containing the path to the configuration file. The constructor initializes the attributes in accordance with the above descriptions.

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

3. Destructor needs to reclaim dynamically allocated memory areas.
4. The **str** method returns the string representation of Configuration.

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

   The format of this string is:

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

Example of a string returned to the **str** method of **Configuration**:

```
Configuration[
MAP_NUM_ROWS=10
MAP_NUM_COLS=10
MAX_NUM_MOVING_OBJECTS=10
NUM_WALLS=3
ARRAY_WALLS=[(1,2);(2,3);(3,4)]
NUM_FAKE_WALLS=1
ARRAY_FAKE_WALLS=[(4,5)]
SHERLOCK_MOVING_RULE=RUU
SHERLOCK_INIT_POS=(1,3)
SHERLOCK_INIT_HP=250
SHERLOCK_INIT_EXP=500
WATSON_MOVING_RULE=LU
WATSON_INIT_POS=(2,1)
WATSON_INIT_HP=300
WATSON_INIT_EXP=350
CRIMINAL_INIT_POS=(7,9)
NUM_STEPS=100
]
```

In which each attribute and corresponding attribute value will be printed in order as listed in the "Private attributes" section of this class.

3.10 Robot

During the criminal's movement, for every **3 steps** that the criminal moves, a robot will be created. Note that when the **getNextPosition** method of **Criminal** does not return a valid move position, the **move** method does not perform the move, which is not counted as a move. move. Each type of robot will be a **MovingObject** (receiving **MovingObject** as the ancestor class). Each type of robot can move on **Path** or **FakeWall**, but cannot move on **Wall**. Once created, the robot will be added to the array of moving objects (**ArrayMovingObject**) via

the **add** method of the **ArrayMovingObject** class. In case the quantity of this array is full, the robot is not created.

After every **3 steps** of the criminal, a robot will be created at that location. Types of robots and corresponding generating conditions:

- If it is the first robot created on the map, it will be the robot type **RobotC**. If not:
- Closer distance to Sherlock's location: Create a robot **RobotS**
- Closer distance to Watson's location: Creating a robot **RobotW**
- The distance to Sherlock and Watson's location is equal: Create a robot **RobotSW**

Robot types are defined as enum **RobotType** as follows:

```
enum RobotType { C, S, W, SW} // C stands for type RobotC,...
```

Requirements: Students implement necessary classes related to robot objects according to the same requirements as other characters, with some changes as follows:

1. Robots with similar attributes include:
   - The attribute **robot_type** has the value of the robot type of type **RobotType**
   - The attribute **item** has type **BaseItem *** whose value is the type of attribute the character will receive if he wins. Details about the items will be presented in the following section.
2. Attibutes for each robot types:
   - **RobotC**: The attribute **Criminal* criminal** is a pointer to the criminal
   - **RobotS**: The attribute **Criminal* crimminal** is like **RobotC** and the attribute **Sherlock* sherlock** is a pointer to Sherlock
   - **RobotW**: The attribute **Criminal* criminal** is like **RobotC** and the attribute **Watson* watson** is a pointer to Watson
   - **RobotSW**: Attribute **Criminal* criminal** like **RobotC** and 2 attributes **Sherlock* sherlock** and **Watson* watson**
3. Public constructor is passed parameters related to **MovingObject** and parameters specific to each robot type. Specifically, the constructor of each type is called as follows:

```
    RobotC(int index, const Position & init_pos, Map * map,
RobotType robot_type, Criminal* criminal);

    RobotS(int index, const Position & init_pos, Map * map,
RobotType robot_type,   Criminal* criminal, Sherlock* Sherlock);
```

```
4
5       RobotW(int index, const Position & init_pos, Map * map,
    RobotType robot_type, Criminal* criminal, Watson* watson);
6
7       RobotSW(int index, const Position & init_pos, Map * map,
    RobotType robot_type, Criminal* criminal, Sherlock* sherlock, Watson
    * watson);
8
```

In which:

- **robot_type** is passed to the attribute **robot_type**
- **criminal** is a pointer to the corresponding Criminal object that can get the current location of the criminal.
- **Sherlock* sherlock** is a pointer to Sherlock. From there you can get Sherlock's current location.
- **Watson* watson** is a pointer to Watson. From there, Watson's current location can be obtained.
- The remaining parameters have the same meaning as the previous sections.

4. Method **getNextPosition** (public): For each type of robot, the moving rules are also different, specifically:

   - **RobotC:** Moves to the next location in the same location as the criminal
   - **RobotS:** Move to the next location 1 unit away from the original and closest to Sherlock's next location. Note that when we say the location is 1 unit away from the original location, the distance referred to is the Manhattan distance. If there are multiple nearest locations, the order of selection ***clockwise rotation***, starting from the ***upwards*** direction, and selecting the first location .
   - **RobotW:** Moves to the next location 1 unit away from the original and closest to Watson's next location. If there are multiple suitable positions, the order of selection is as if in **RobotS**.
   - **RobotSW:** Move to the next location that is 2 units away from the original and has the closest total distance to both Sherlock and Watson. If there are multiple suitable positions, the order of selection is as if in **RobotS**.
   - If the position is invalid, returns the **npos** value of **Position**

5. **move** (public) method: If the next position is not **npos**, move to this position.
6. Method **getDistance** (public): For robot types S, W or SW, this method returns the corresponding distance value of that robot object to Sherlock, Watson or the sum of both.

Particularly, type C robots provide two methods to calculate the distance to Sherlock or to Watson based on an input parameter: pointer **Sherlock* sherlock** or **Watson* watson**.

7. **str** (public) method: Format the returned string as follows:

```
Robot[pos=<pos>;type=<robot_type>;dist=<dist>]
```

<**pos**> prints the current position of the Robot
robot_type prints a value that can be C, S, W or SW
dist prints the distance to Sherlock, Watson or the sum of both depending on whether the robot is S, W or SW. If the robot type is RobotC, print an empty string.

3.11 Item

Class **BaseItem** is an abstract class that represents information for an item. Students define their own class **Character** to represent a character such as Sherlock, Watson or a criminal. **Character** needs to meet the following conditions: **Character** recognizes **MovingObject** as the ancestor class, and **Criminal**, **Sherlock**, **Watson** receives **Character** as ancestor class. Students define their own class **Robot** to represent a certain type of Robot. **Robot** needs to satisfy the following conditions: **Robot** receives **MovingObject** as ancestor class; and **RobotC**, **RobotS**, **RobotW**, **RobotSW** receive **Robot** as the ancestor class. The class has two public pure virtual methods:

- **canUse**

```
virtual bool canUse(Character* obj, Robot * robot) = 0;
```

Method returns `true` if Sherlock or Watson can use this item, otherwise returns `false`.

- **use**

```
virtual void use(Character* obj, Robot * robot) = 0;
```

Method acts on Sherlock or Watson to change their parameters to suit the item's effects. The parameter **robot** represents the character encountering a Robot and will use the item accordingly. If the character does not encounter a robot, the parameter **robot** is equal to **NULL**.

Every time Sherlock or Watson encounters a certain type of Robot, each person will have 2 times to search for **Item** in **Bag** (described in the following Section). The first time of use is when they first meet **Robot**, they will look to see if there are any items in their bag that

will help overcome the Robot's challenges. Then, the **robot** parameter needs to be passed to the **Robot** object. At first, only **ExcemptionCard** or **PassingCard** can be used. The second time to use is at the end of meeting the Robot (after performing the challenge if the challenge occurs), Sherlock or Watson will look in the test bag for any items that can restore **hp** or **exp** or not. Then, the parameter **robot** has the value **NULL**. The second time can only use **MagicBook**, **EnergyDrink** or **FirstAid**

Classes MagicBook, EnergyDrink, FirstAid, ExcemptionCard, PassingCard respectively represent items: Energy drink, **HP** first aid kit, obstacle immunity card, execution exemption card challenge. Particularly for challenge exemption cards, there is a card type attribute to represent the type of challenge that the card can exempt. These classes inherit from the BaseItem class and need to redefine the two methods canUse and use accordingly.

The effects of each type of item are described as follows:

Item	Effect	Using conditions
MagicBook	Contains ancient magical knowledge that helps Sherlock and Watson increase their knowledge, experience and experience quickly, so it easily recovers **exp** by 25% when used.	**exp** $\leq$ 350
EnergyDrink	Energy Drink, when used, will help the target recover **hp** by 20% when used.	**hp** $\leq$ 100
FirstAid	First Aid Kit, when used, will help the character recover **hp** by 50% when used.	**hp** $\leq$ 100 or **exp** $\leq$ 350
ExcemptionCard	Immunity Card helps the character to be immune to **hp, exp** when not overcoming challenges at a destination.	Only Sherlock can use this card and when Sherlock's **hp** is an odd number
PassingCard	When using the PassingCard to perform a challenge, the character does not need to perform the challenge at a destination location. The tag has an attribute **challenge** (type string) which is the name of a challenge (for example type RobotS is a tag to pass the challenge posed by RobotS without doing it). If the tag's type is **all** the tag can be used for any type regardless of the challenge encountered. Otherwise, when using the **use** method to perform an action that affects Sherlock and Watson, it is necessary to check whether the card type matches the type of challenging action the character encounters. In this case, if the card type does not match, the character's **exp** will be reduced by **50 EXP** even though the effect will still be performed.	Only Watson can use this card and when Watson's **hp** is an even number

These items are contained within the robots. The conditions created are as follows:

Call the location where the robot is created with coordinates (i,j) where i is the row index, j is the column index.

With $p = i*j$. Call s the cardinal number of p . We define the cardinal number of a number as the sum value of the digits, until the sum value is a 1-digit number. We have:

- If s is in the range [0, 1], it will create **MagicBook**
- If s is in the segment [2, 3], it will generate **EnergyDrink**
- If s is in the segment [4, 5], it will generate **FirstAid**
- If s is in the segment [6,7], it will generate **ExcemptionCard**
- If s is in segment [8, 9], **PassingCard** will be generated. Let $t = (i * 11 + j)\%4$. The **challenge** property of **PassingCard** is initialized as follows:
  - $t = 0$: **challenge** = "RobotS"
  - $t = 1$: **challenge** = "RobotC"
  - $t = 2$: **challenge** = "RobotSW"
  - $t = 3$: **challenge** = "all"

Requirements: Students implement the classes described above.

3.12 Bag

Sherlock and Watson are each equipped with a bag to store items collected along the way. The inventory is implemented as a singly linked list. Actions performed with the bag include:

- Add an item to the inventory (method **insert)**): Add the item to the beginning of the list
- Use any item (**get** method): Find the item in the bag that can be used and is closest to the top of the bag. This item will be flipped with the first item in the list and then removed from the list.
- Use a specific item (method **get(ItemType)**: If in the inventory there is an item of the type to be used, the character can use this item by reversing its position with first item (if it is not the first item) and remove it from the list. If there are multiple items, the item closest to the top of the bag will be done as above.

Each character will have a maximum number of items that can be stored in their bag. Sherlock's bag can hold up to **13 items**, while Watson can hold up to **15 items**.

In case Sherlock and Watson meet, Sherlock gives Watson a **PassingCard** card if any and vice versa, Watson will give a **ExcepmtionCard** card if any. If there is more than one card in their inventory, they give them all to their opponent. The act of giving means removing those

tags from your inventory and adding them to the top of the other person's inventory. This exchange action will take place in the order of Sherlock first and then Watson. In case either one does not have or both do not have that type of item, the exchange action will not take place. The order of characters giving items is Sherlock gives Watson first, then Watson gives Sherlock. The order in which a character gives items is that the character searches the bag from beginning to end via the **get** method. Each time a giftable item is found, the character removes (delete operation as described in *using an item*) the item from his/her inventory and adds (via the **m**ethod insert) into the other character's bag.

Requirements: Students need to implement the information presented in the bag including:

Class **BaseBag** is a class that represents a shopping bag. The class has an attribute **obj** of type **Character*** representing the character that owns the bag. In addition, the class also has basic public methods as described, including:

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

For the str() method, the returned string format is:

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

In which:

- <c>: is the current number of items that the inventory has.
- <list_items>: is a string representing the items from beginning to end of a linked list, each item is represented by the item's type name, the items are separated by a comma . The type names of the items are the same as the class names described above.

Requirements: Students proposed classes that inherit from the BaseBag class and demonstrated different types of bags for Sherlock and Watson. The names of these two classes are SherlockBag and WatsonBag respectively. The constructors of these two classes only have 1 parameter passed in, respectively **Sherlock *** (representing Sherlock) and **Watson *** (representing Watson).

3.13 StudyInPinkProgram

If in the process of moving Sherlock or Watson encounter robots or criminals, problems will occur. Cases include:

1. If Sherlock meets:
    - RobotS: Sherlock needs to solve a problem to win against RobotS. If Sherlock's **EXP** is now **greater than 400**, Sherlock will solve the problem and receive the item this robot holds. Otherwise, Sherlock's EXP will be lost by 10%.
    - RobotW: Sherlock will pass and receive the item without having to fight.
    - RobotSW: Sherlock can only win against RobotSW when **EXP** of Sherlock **is greater than 300** and **HP** of Sherlock **greater than 335**. If he wins, Sherlock receives the item this robot holds. Otherwise, Sherlock will lose 15% HP and 15% EXP.
    - RobotC: Sherlock meeting RobotC means meeting the location next to the criminal. At this time, if Sherlock's **EXP** is greater than **500**, Sherlock will defeat the robot and capture the criminal (no items this robot holds). On the contrary, Sherlock will let criminals escape. However, you will still be able to destroy the robot and receive the items this robot holds.
    - FakeWall: As described above
2. If Watson encounters:
    - RobotS: Watson will not perform any actions with the robot and will not receive items held by this robot.
    - RobotW: Watson needs to confront this Robot and only win when it has **HP greater than 350**. If Watson wins, Watson will receive the item held by the robot. If he loses, Watson's HP will be reduced by 5%.
    - RobotSW: Watson can only win against RobotSW when Watson's **EXP is greater than 600** and Watson's **HP is greater than 165**. If he wins, Watson receives the item that this robot holds. Otherwise, Watson will lose 15% HP and 15% EXP.
    - RobotC: Watson meets RobotC, which means he has met the location adjacent to the criminal. Watson could not catch the criminal because he was held back by RobotC. However, Watson will still destroy the robot and receive the item this robot holds.
    - FakeWall: As described above

    Before and after each encounter event, the character will examine and use the possible items in his bag

Requirements: Students propose and implement class **StudyInPinkProgram** according to the following requirements:

(a) Class has the following attributes: config (type **Configuration***), sherlock (type **Sherlock***), watson (type **Watson***), criminal (type **Criminal***) , map (type **Map***), arr_mv_objs (type **ArrayMovingObject**.

(b) Constructor receives 1 parameter which is the path to the configuration file for the program. The constructor needs to initialize the necessary information for the class. As for **arr_mv_objs**, after initialization, *criminal*, *sherlock*, *watson* are added sequentially using the **add** method.

```
StudyPinkProgram(const string & config_file_path)

```

(c) The **isStop** method returns **true** if the program stops, otherwise returns **false**. The program stops when one of the following conditions occurs: Sherlock's **hp** is 0; Watson's **hp** is 0; Sherlock catches the criminal; Watson caught the criminal.

```
bool isStop() const;

```

(d) The **printResult** and **printStep** methods print program information. These methods are already implemented, students do not change these methods.

(e) The **run** method has 1 parameter, **verbose**. If **verbose** is equal to **true** then it is necessary to print out the information of each MovingObject in the ArrayMovingObject after performing a move and subsequent updates if any (e.g. Watson encounters a Robot and performs challenge with Robot). Students refer to the initial code about the location of the **printStep** function in **run**. The **run** method runs up to **num_steps** (taken from the configuration file). After each **step**, if the program meets the stopping condition stated in the **isStop** method, the program will stop. Each **step**, the program will run an ArrayMovingObject from start to finish and call **move**. Then perform tasks such as when two objects meet and create a new robot.

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

3.14 TestStudyInPink

TestStudyInPink is a class used to test the properties of classes in this assignment. Students must declare **TestStudyInPink** as a friend class when defining all of the above classes.

4 Requirements

To complete this assignment, students must:

1. Read entire this description file.
2. Download the initial.zip file and extract it. After extracting, students will receive files including: main.cpp, main.h, study_in_pink2.h, study_in_pink2.cpp, and example file inputs. Students will only submit 2 files, study_in_pink2.h and study_in_pink2.cpp. Therefore, you are not allowed to modify the main.h file when testing the program.
3. Students use the following command to compile:
   g++ -o main main.cpp study_in_pink2.cpp -I . -std=c++11
   Students use the following command to run the program:
   ./main
   The above command is used in the command prompt/terminal to compile the program. If students use an IDE to run the program, students should pay attention: add all the files to the IDE's project/workspace; change the IDE's compile command accordingly. IDEs usually provide buttons for compiling (Build) and running the program (Run). When you click Build, the IDE will run a corresponding compile statement, normally, only main.cpp should be compiled. Students need to find a way to configure the IDE to change the compilation command, namely: add the file study_in_pink2.cpp, add the option -std=c++11, -I .
4. The program will be graded on the Unix platform. Students' backgrounds and compilers may differ from the actual grading place. The submission place on LMS is set up to be the same as the actual grading place. Students must test the program on the submission site and must correct all the errors that occur at the LMS submission site in order to get the correct results when final grading.
5. Modify the files study_in_pink1.h, study_in_pink1.cpp to complete this assignment and ensure the following two requirements:
   - There is only one **include** directive in the study_in_pink2.h file which is **#include "main.h"** and one directive in the study_in_pink2.cpp file which is **#study_in_pink2** . Apart from the above directives, no other **#include** is allowed in these files.
   - Implement the functions described in the Missions in this Assignment.
6. Students are encouraged to write additional functions to complete this assignment.

5 Submission

Students submit only 2 files: study_in_pink2.h và study_in_pink2.cpp, before the deadline given in the link "Assignment 2 - Submission". There are a number of simple test cases used to check student work to ensure that student results are compilable and runnable. Students can submit as many times as they want, but only the final submission will be graded. Since the system cannot bear the load when too many students' submissions at once, students should submit their work as soon as possible. Students do so at their own risk if they submit assignments by the deadline. When the submission deadline is over, the system will close so students will not be able to submit any more. Submissions through other means will not be accepted.

6 Other regulations

- Students must complete this assignment on their own and must prevent others from stealing their results. Otherwise, the student treat as cheating according to the regulations of the school for cheating.
- Any decision made by the teachers in charge of this assignment is the final decision.
- Students are not provided with test cases after grading, students will be provided with the assignment's score distribution.
- Assignment contents will be harmonized with a question in exam with similar content.

7 Cheating treatment

Assignment must be done by students themselves. Students will be considered cheating if:

- There is an unusual similarity between the source code of the submissions. In this case, ALL submissions will be considered fraudulent. Therefore, students must protect their source code.
- Student's work is submitted by another student.
- Students do not understand the source code they write, except for the code parts provided in the initialization program. Students can refer to any source, but must ensure that they clearly understand the meaning of all the commands they write. In case they do not clearly understand the source code from the place they refer to, students are specifically warned NOT to use this source code; Instead, they should use what they have learned to write programs.
- Wrongly submitted another student's work to the other personal account.

- Using source code from tools capable of generating source code without understanding the meaning.

In case of being concluded as cheating, the student will receive a score of 0 for the entire subject (not just this assgingment).

DO NOT ACCEPT ANY INTERPRETATION AND NO EXCEPTIONS!

After each assignment is submitted, a number of students will be randomly called for an interview to prove that the assignment just submitted was done by themselves.

8 Changelog