Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring '2018-2019

Programming Assignment 4

Due date: 11 May 2019, Thursday, 23:55

1 Objectives

In this homework you will get familiar with Inheritance, Polymorphism and Abstract class concepts in Object Oriented programming.

Keywords: *OOP, Inheritance, Abstract class, Polymorphism*

2 Problem Definition

In this homework, you are going to simulate a treasure hunt game. In this game, there are players (abstract class) with different classes (implementations) divided in two teams. The hunt commences on a square grid of size N which is the Board class. The top left square of the grid has the coordinate (0,0) while the bottom right square has coordinates(N-1,N-1). X axis is the horizontal axis and Y axis is the vertical axis. One of the squares contains the treasure, some of them contains the characters and other squares are empty. The classes you are going to define are Player (and its implementations), Board, Game and Input Parser. They are explained below.

3 Class Definitions

3.1 Player

Player is an abstract class. There are 5 types of players, namely, Fighter, Archer, Tank, Priest and Scout. Each player can be in the barbarian team or the knight team. Each type of player has different HP, attack damage, heal power and goal priority (will be explained later in the game class). The class and method definitions of Player is written below. After that there are subsections that explains the sub-classes of Player.

```
class Player{
protected:
  const uint id;
  Coordinate coordinate;
  int HP;
  Team team;
  //DO NOT MODIFY THE UPPER PART
  //ADD YOU OWN PROVATE METHODS/PROPERTIES BELOW

public:

  /**
   * Main constructor
   *
   * @param id id of the player.
   * @param x x-coordinate of the player
   * @param y y-coordinate of the player
   * @param team team of the player, BARBARIAN or KNIGHT
   *
   */

  Player(uint id,int x, int y, Team team);
  virtual ~Player() = default;

  uint getID() const;
  const Coordinate& getCoord() const;

  int getHP() const;

  Team getTeam() const;

  /**
   * @return the board ID of the player, If the ID is single digit add a prefix
   * 0, otherwise just turn it into a string.
   *
   * Example:
   * 3 -> "03"
   * 12 -> "12"
   *
   */
  std::string getBoardID();

  virtual int getAttackDamage() const = 0;

  virtual int getHealPower() const = 0;

  virtual int getMaxHP() const = 0;
```

```
/**
 * For each subclass of Player there are different priority lists defined
 * that controls the next action to take for that Player. It is given in the
 * header of the subclasses.
 *
 * @return the goal priority list for the Player
 */
virtual std::vector<Goal> getGoalPriorityList();
/**
 * @return the class abbreviation of player, if the player is on the barbarian
 * team, the abbreviation will consist of uppercase characters, otherwise it
 * will consist of lowercase characters.
 *
 */

virtual const std::string getClassAbbreviation() const;

/**
 * Attack the given player.
 * Enemy HP should decrease as much as the attack damage of attacker. Print
 * the boardID of the attacker and the attack and the amount of damage as below↩
    .
 * "Player 01 attacked Player 05 (75)"
 *
 * @param enemy player that is attacked.
 * @return true if the opponent is dead false if alive.
 */

bool attack(Player *enemy);

/**
 * Heal the given player by the adding amount of heal power of this character
 * to the HP of ally. Print the boardID of the healer and healed players as
 * below.
 * "Player 01 healed Player 05"
 * Healed player should not have more HP than its max HP.
 */

void heal(Player *ally);
/**
 * @Important The coordinates may not be on the board.
 *
 * @return the coordinates that the unit is able to attack given the position
 * of the unit. Empty vector if the unit cannot attack.
 */

virtual std::vector<Coordinate> getAttackableCoordinates();

/**
 * @Important The coordinates may not be on the board.
 *
 * @return the coordinates the unit is able to move given the position of the
 * unit. Empty vector if the unit cannot move.
 */
virtual std::vector<Coordinate> getMoveableCoordinates();
```

```
  /**
   *
   * @return the coordinates the unit is able to heal allies given the position ↩
       of the
   * unit. Empty vector if none available.
   */
  virtual std::vector<Coordinate> getHealableCoordinates();

  /**
   * Move player to coordinate. Print the boardID of the player and the old and ↩
       new
   * coordinates as below:
   * "Player 01 moved from (0/1) to (0/2)"
   * @Important before calling this method you must verify that this coordinate
   * is valid to move
   */
  void movePlayerToCoordinate(Coordinate c);



  /**
  * Decide whether the player is dead.
  *
  * @return true if the player's HP <= 0, false otherwise.
  */
  bool isDead() const;

};
```

3.1.1 Archer

```
class Archer : public Player{
  /**
   * Attack damage 50
   * Heal power 0
   * Max HP 200
   * Goal Priorities -> {ATTACK}
   * Class abbreviation -> "ar" or "AR"
   * Not able to move at all.
   * Can attack to a range of 2 squares directly up, down, left or right, from
   * its coordinate.
   *
   */
};
```

3.1.2 Fighter

```
class Fighter : public Player{
  /**
   * Attack damage 100
   * Heal power 0
   * Max HP 400
   * Goal Priorities -> {ATTACK,TO_ENEMY,CHEST} in decreasing order
   * Class abbreviation -> "fi" or "FI"
   * Can move to adjacent up, down, left or right square
   * Can attack to adjacent up, down, left or right square
   *
   */
};
```

3.1.3 Priest

```
class Priest : public Player{
  /**
   * Attack damage 0
   * Heal power 50
   * Max HP 150
   * Goal Priorities -> {HEAL,TO_ALLY,CHEST} in decreasing order
   * Class abbreviation -> "pr" or "PR"
   * Can move to all adjacent squares, including diagonals.
   * Can heal all adjacent squares, including diagonals.
   *
   */
};
```

3.1.4 Scout

```
class Scout : public Player{
  /**
   * Attack damage 25
   * Heal power 0
   * Max HP 125
   * Goal Priorities -> {CHEST,TO_ALLY,ATTACK} in decreasing order
   * Class abbreviation -> "sc" or "SC"
   * Can move to all adjacent squares, including diagonals.
   * Can attack all adjacent squares, including diagonals.
   *
   */
};
```

3.1.5 Tank

```
class Tank : public Player{
  /**
   * Attack damage 25
   * Heal power 0
   * Max HP 1000
   * Goal Priorities -> {TO_ENEMY,ATTACK,CHEST} in decreasing order
   * Class abbreviation -> "ta" or "TA"
   * Can move to adjacent up, down, left or right square
   * Can attack to adjacent up, down, left or right square
   *
   */
};
```

3.2 Board

Board holds size of the board, players and position of the chest. Its methods are concerned with printing the contents of the field and returning the properties of coordinates. The class header is given below.

```
class Board{

private:
  uint size;
  std::vector<Player*>* players;
  Coordinate chest;

  //DO NOT MODIFY THE UPPER PART
  //ADD YOU OWN PROVATE METHODS/PROPERTIES BELOW

public:
```

```
  Board(uint _size, std::vector<Player*>* _players, Coordinate chest);
  ~Board();
  /**
   * @return true if the coordinate is in the board limits, false otherwise.
   */
  bool isCoordinateInBoard(const Coordinate& c);
  /**
   * @return true if there is a player on the given coordinate, false otherwise.
   */
  bool isPlayerOnCoordinate(const Coordinate& c);
  /**
   * @return pointer to the player at the given coordinate. Return NULL if no
   * player is there.
   */
  Player *operator [](const Coordinate& c);

  /**
   * @return the chest coordinate
   */
  Coordinate getChestCoordinates();
   /**
    * Print the board with character ID's.
    * For each empty square print two underscore characters.
    * For the squares with a player on it, print the board id of the player.
    * For the square with the chest, print the string "Ch".
    * If a character is on the square with the chest, only print the ID of the
    * character.
    * For each row print a new line, for each column print a space character.
    * Example:
    * __ __ 01 __
    * __ 02 __ 05
    * Ch __ __ 03
    * __ __ __ __
    */
  void printBoardwithID();
  /**
   * For each empty square print two underscore characters.
   * For the squares with a player on it, print the class abbreviation of the
   * player.
   * For the square with the chest, print the string "Ch".
   * If a character is on the square with the chest, only print the abbreviation
   * of the character.
   * To separate each row print a new line, to separate each column print a
   * space character.
   * Example:
   * __ __ PR __
   * __ ar __ TA
   * Ch __ __ fi
   * __ __ __ __
   */
  void printBoardwithClass();
};
```

3.3 Game

Game class is responsible for running the game and managing the memory. The most important method here is the playTurnForPlayer method which computes the moves for the players given their different goal

priorities.

```
class Game{

private:
  Board board;
  uint turnNumber;
  uint maxTurnNumber;
  std::vector<Player*> players;

  //DO NOT MODIFY THE UPPER PART
  //ADD YOU OWN PROVATE METHODS/PROPERTIES BELOW

public:
  /**
   * Costructor for Game class.
   * Game manages the memory allocated for future contents the vector (added players).
   * Pass a pointer to the players vector to board constructor so that the board will
   * not miss the addition of players to the game.
   * @param maxTurnNumber turn number to end the game
   * @param boardSize size of the board
   * @param chest coordinate of the chest
   */
  Game(uint maxTurnNumber, uint boardSize, Coordinate chest);
  ~Game();

  /**
   * Add a new player to the game. Add a pointer to the new player to the this->players vector.
   * Do not forget that Game will manage the memory allocated for the players.
   * @param id ID of the new player.
   * @param x x coordinate of the new player.
   * @param y y coordinate of the new player.
   * @param team team of the new player.
   *
   */
  void addPlayer(int id, int x, int y, Team team );

  /**
   * The game ends when either of these happens:
   * - All barbarians die (knight victory)
   * - All knights die (barbarian victory)
   * - A barbarian gets to the square containing the chest (barbarian victory)
   * - maxTurnNumber of turns played (knight victory)
   *
   * If the game ends announce it py printing the reason, turn number and the victor
   * as in the following examples:
   *
   * Game ended at turn 13. All barbarians dead. Knight victory.
   * Game ended at turn 121. All knights dead. Barbarian victory.
   * Game ended at turn 52. Chest captured. Barbarian victory.
   * Game ended at turn 215. Maximum turn number reached. Knight victory.
   *
   * @return true if any of the above is satisfied, false otherwise
   *
```

```
 */
bool isGameEnded();

/**
 * Play a turn for each player.
 * Actions are taken in the order of ID numbers of players (player with
 * smaller ID acts first).
 * At the start of the turn it announces that the turn has started by printing
 * to stdout. Turn numbers starts with 1.
 * Ex: "Turn 13 has started."
 * Call playTurnForPlayer for every player.
 *
 */
void playTurn();
/**
 * Play a turn for the player with the given ID.
 * If the player is dead announce its death by printing the boardID of the ↩
    player
 * as in "Player 07 died.". Remove that player from the board and release its ↩
    resources.
 *
 * Each player has a goal list sorted by its priority for that player.
 * When a player plays a turn it iterates over its goal list and tries to take
 * an action. Valid actions are attack, move and heal. A player can take only
 * one action in a turn, and if there is no action it can take it stops and ↩
    does nothing.
 * Before moving a player you must check if the coordinate to move is valid.
 * Meaning that, the coordinate is in the bounds of the board and there are no
 * players there.
 *
 * IMPORTANT NOTE: every usage of the word nearest is referencing smallest the ↩
    Manhattan
 * distance, which is formulated as (abs(x_1-x_2) + abs(y_1-y_2)). operator-
 * overloaded in Coordinate.h computes exactly that, so you can use that method↩
     to
 * calculate the distance between two coordinates.
 *
 * Below are the explanations for goals:
 *
 * ATTACK:
 *   - If there are any enemies in the attack range of the player attack to it.
 *     If there are more than 1 enemy in the range attack to the one with
 *     lowest ID. If there is no one to attack try the next goal.
 *
 * CHEST:
 *   - Move to the direction of the chest, if both vertical and horizontal ↩
    moves
 *     are available, pick the horizontal one. If the horizontal move is ↩
    blocked
 *     but the vertical move is not, move vertically. If all directions towards
 *     the chest is blocked try the next goal.
 *
 * TO_ENEMY:
 *   - Move towards the nearest enemy. If there are more than one enemies with ↩
    the same distance
 *     move towards the one with the smallest ID. If both vertical and ↩
    horizontal moves
```

```
     *     are available, pick the horizontal one. If an enemy is in the squares
     *     that the player can move or every move that brings the player closer to
     *     the selected enemy is blocked, try the next goal.
     *
     * TO_ALLY:
     *   - Move towards the nearest ally. If there are more than one allies with the same distance
     *     move towards the one with the smallest ID. If both vertical and horizontal moves
     *     are available, pick the horizontal one. If an ally is in the squares
     *     that the player can move or every move that brings the player closer to
     *     the selected ally is blocked, try the next goal.
     *
     * HEAL:
     *   - If there are any allies in the healing range heal all of them. if there
     *     is no one to heal try the next goal.
     *
     *
     * @return the goal that the action was taken upon. NO_GOAL if no action was taken.
     */
    Goal playTurnForPlayer(Player* player);
};
```

3.4 InputParser

Reads the standard input and creates a Game object. Details are explained in the header.

```
class InputParser{

public:

    /**
     * Parse the initial parameters of the game from stdin.
     * The input will be as follows.
     * First line contains the size of the board.
     * Second line contains the coordinates of the chest.
     * Third line contains the number of players, P.
     * Each of the next P lines contains a description for a player as follows.
     * ID of the player, class of the player, team of the player, x coordinate, y coordinate, .
     * Call the addPlayer method of the Game class to add the players.
     * Example input:
     * 6
     * 3 3
     * 2
     * 12 ARCHER BARBARIAN 3 5
     * 11 FIGHTER KNIGHT 1 1
     *
     * @returns Pointer to the Dynamically allocated Game object
     */
    static Game* parseGame();
};
```

4 Extras

The enumerations you need are defined in the Constants.h header. You can use them from there.

The Game class owns every player and vector of players in terms of memory management and is responsible for the destruction of these.

In order to get full grade from each part your code should not have any memory leak. This will be checked with valgrind. While grading your classes will be used with the correct implementations, therefore they are expected to work as commented in the code.

5 Regulations

- **Programming Language:** You must code your program in C++ (11). Your submission will be compiled with `g++` with `-std=c++11` flag on department lab machines.
- **Allowed Libraries:** You may include and use C++ Standard Library. Use of any other library (especially the external libraries found on the internet) is forbidden.
- **Memory Management:** When an instance of a class is destructed, the instance must free all of its owned/used heap memory. Any heap block, which is not freed at the end of the program will result in grade deduction. Please check your codes using valgrind –leak-check=full for memory-leaks.
- **Late Submission:** You have a total of 10 days for late submission. You can spend this credit for any of the assignments or distribute it for all. For each assignment, you can use at most 3 days-late.
- **Cheating:** In case of cheating, the university regulations will be applied.
- **Newsgroup:** It's your responsibility to follow the cengclass forums for discussions and possible updates on a daily basis.

6 Submission

Submission will be done via CengClass. Create a zip file named `hw4.zip` that contains:

- Archer.h
- Archer.cpp
- Board.h
- Board.cpp
- Fighter.h
- Fighter.cpp
- Game.h
- Game.cpp
- InputParser.h
- InputParser.cpp
- Player.h
- Player.cpp
- Priest.h

- Priest.cpp
- Scout.h
- Scout.cpp
- Tank.h
- Tank.cpp

Do not submit a file that contains a `main` function. Such a file will be provided and your code will be compiled with it. Also, do not submit a Makefile.

Note: The submitted zip file should not contain any directories! The following command sequence is expected to run your program on a Linux system:

```
$ unzip hw4.zip
$ make clean
$ make all
$ make run
$ -optional- make valgrind
```