Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring '2017-2018

Programming Assignment 4

Due date: 12 May 2018, Saturday, 23:55

1 Objectives

In this assignment you will practice the Abstraction, Inheritance and Polymorphism concepts of Object Oriented Programming.

Keywords: *OOP, inheritance, polymorphism, (pure) abstract class*

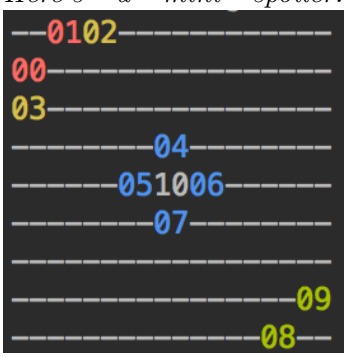
2 Problem Definition

TL;DR: Implement Player and its inherited classes(Berserk, Tracer, Ambusher, Pacifist, Dummy), GameParser, Board, and GameEngine.

Ahh... Another homework, another challenge.. Let's see, is there any way to create a fun homework?
....
Found 1 result(s): Fortnite + King of the Hill challenge.
Ooh, Fortnite on Terminal, and KOTH? I'm interested!
Let's change its name, I don't want to deal with another lawsuit. CengNite? Sounds cool.

Playing Fortnite 7/24...
Pausing Fortnite for a while...
Creating new Player classes...
Pouring concrete on the abstract class... (Get it?)
Overriding already implemented methods... (Why though?)
Inheriting $5M from Nigerian prince...
Buying new Fortnite skins...
Still brawling with C++ Memory Manager... (I missed Swift...)
Preparing test cases...
Ready.

Here's a mini spoiler:



As an aspiring Computer Engineering student, your next task *-if you accept-* is to create a Fortnite Simulation. So, you're going to parse an input file, implement a -pure- abstract class, also few concrete classes, and the algorithms. Basically, it's going to be a Fortnite with a King of the Hill extension, on Terminal.

3 Specifications

3.1 Player

The courageous challengers. It's uniquely identified by **playerID**.
Its attributes are **HP**, **coordinate**, **weapon (damage)** and **armor (damage reduction)**.

Players will act according to their **Priority List**. While taking turns, Players will take turn according to priority. Lower ID == More priority.

```
class Player {
protected:
    const uint id;
    Coordinate coordinate;

    int HP;

    // DO NOT MODIFY THE UPPER PART
    // ADD OWN PROTECTED METHODS/PROPERTIES BELOW
public:
    /**
     * Main Constructor.
     *
     * @param id The ID of the player. [0-100).
     * @param x X-coordinate of the player.
     * @param y Y-coordinate of the player.
     */
    Player(uint id, int x, int y);
    virtual ~Player();

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

    int getHP() const;

    /**
     * Board ID is two-decimal ID for the Player.
     *
     * Player ID = 0, 91
     * Board ID = "00", "91"
     *
     * @return BoardID of the user.
     */
    virtual std::string getBoardID() const;

    virtual Armor getArmor() const = 0;
    virtual Weapon getWeapon() const = 0;
    /**
     * Every player has a different priority move list.
     * It's explained in the Players' header.
     *
     * @return The move priority list for the player.
     */
    virtual std::vector<Move> getPriorityList() const = 0;
    /**
     * Get the full name of the player.
     *
```

```
     * Example (Tracer with ID 92) = "Tracer92"
     * Example (Tracer with ID 1)  = "Tracer01"
     *
     * @return Full name of the player.
     */
    virtual const std::string getFullName() const = 0;
    /**
     * Decide whether the player is dead.
     *
     * @return true if the player's hp <= 0, false otherwise.
     */
    bool isDead() const;
    /**
     * Execute the given move for the player's coordinates.
     *
     * Important note: Priority list does NOT matter here.
     *
     * NOOP and ATTACK are no-op.
     *
     * Do not forget to print the move.
     * "-playerFullName(playerHP)- moved UP/DOWN/LEFT/RIGHT."
     *
     * "Tracer00(100) moved UP."
     *
     * @param move Move to make.
     */
    void executeMove(Move move);
    /**
     * Attack the given player, and decide whether the attacked player is dead.
     *
     * Important note: Priority list does NOT matter here.
     *
     * Formulae : RHS's HP -= max((LHS's damage - RHS's armor), 0)
     *
     * Do not forget to print the attack.
     *
     * "-lhsFullName(lhsHP)- attacked -rhsFullName(rhsHP)-! (-damageDone-)
     *
     * "Tracer00(100) attacked Tracer01(100)! (-5)"
     *
     * @param player Player to be attacked.
     * @return true if attacked player is dead, false otherwise.
     */
    bool attackTo(Player *player);
    /**
     * Return different colors for different Player classes (override!).
     *
     * Note: This method is optional. You may leave this as-is.
     *
     * @return The associated color code with the class.
     */
    virtual Color::Code getColorID() const { return Color::FG_DEFAULT; }

    // DO NOT MODIFY THE UPPER PART
    // ADD OWN PUBLIC METHODS/PROPERTIES BELOW
};
```

There will be 5 different types of Players:

{Berserk, Tracer, Ambusher, Pacifist, Dummy}

3.1.1 Berserk:

Now Playing: Eminem - Not Afraid... (Wait, why not Berzerk?)
Berserk is not afraid to kill, not afraid to die. A necessity for every CengNite round.

HP = **100**, Weapon = **PISTOL**, Armor = **WOODEN**

PriorityList = {**ATK / UP / LEFT / DOWN / RIGHT**}.

Name = Berserk, ColorID = FG_RED.

3.1.2 Tracer

Now Playing: Iron Maiden - Run To The Hills...
Tracer will run to the hill first. If there are obstacles, blood will be shed..

HP = **100**, Weapon = **SHOVEL**, Armor = **BRICK**

PriorityList = {**UP / LEFT / DOWN / RIGHT / ATK**}.

Name = Tracer, ColorID = FG_YELLOW.

3.1.3 Ambusher

Now Playing: Sepultura - Ambush...
*Ambusher doesn't care about the *** hill at all. So join them and make every other player leave this land!*

HP = **100**, Weapon = **SEMIAUTO**, Armor = **NOARMOR**

PriorityList = {**ATK**}.

Name = Ambusher, ColorID = FG_BLUE.

3.1.4 Pacifist

Now Playing: John Lennon - Imagine...
Pacifist believes that there is nothing to kill or die for. Well, this is CengNite, everything is real here! Especially the STORM. Run Pacifist, Run!

HP = **100**, Weapon = **NOWEAPON**, Armor = **METAL**

PriorityList = {**UP / LEFT / DOWN / RIGHT**}.

Name = Pacifist, ColorID = FG_GREEN.

3.1.5 Dummy

Now Playing: Toby Fox - Dummy...
*... (Dummy cannot speak! It cannot attack, it cannot defend, what the *** is he doing in here?)*
... (Wait, what? 1000 HP? You must be kidding.)
... (Can he start on the hill? Gosh...)

HP = **1000**, Weapon = **NOWEAPON**, Armor = **NOARMOR**

PriorityList = {**NOOP**}.

Name = Dummy, ColorID = FG_DEFAULT.

3.2 GameParser

GameParser is responsible for parsing the given input file, and creating Players accordingly.

```
class GameParser {
public:
    /**
     * Parse the file with given name and create players accordingly.
     *
     * GameParser DOES NOT have any responsibility over these Players.
     *
     * Note: The file will always exists, and there will be no erroneous input.
     *
     * @param filename The name of the file to be parsed.
     * @return  pair.first: Board size.
     *          pair.second: The vector of the constructed players.
     */
    static std::pair<int, std::vector<Player *> *>parseFileWithName(const std::↩
        string& filename);
};
```

This part is left blank intentionally.

Well, not really, I couldn't fit anything here.

Here are some lyrics for you:

Never gonna give you up

Never gonna let you down

Never gonna run around and desert you

Never gonna make you cry

Never gonna say goodbye

Never gonna tell a lie and hurt you

3.3 Board

Board class is the backbone of any CengNite games. It's responsible for Coordinates, the Players' visibility areas, creating the hill and the STORM, and such. The Board will get angry when a Player tries to go out of bounds, or on top of another Player. Don't do that.

```
class Board {
private:
    const uint boardSize;
    std::vector<Player *> *players;

    // DO NOT MODIFY THE UPPER PART
    // ADD OWN PRIVATE METHODS/PROPERTIES BELOW
public:
    Board(uint boardSize, std::vector<Player *> *players);
    ~Board();

    uint getSize() const;
    /**
     * Decide whether the coordinate is in the board limits.
     *
     * @param coord Coordinate to search.
     * @return true if coord is in limits, false otherwise.
     */
    bool isCoordInBoard(const Coordinate& coord) const;
    /**
     * Decide whether the given coordinate is in storm.
     *
     * @param coord Coordinate to search.
     * @return true if covered in storm, false otherwise.
     */
    bool isStormInCoord(const Coordinate &coord) const;
    /**
     * Decide whether the given coordinate is the hill.
     *
     * @param coord Coordinate to search.
     * @return true if the coord is at the very center of the board, false ↩
        otherwise.
     */
    bool isCoordHill(const Coordinate& coord) const;
    /**
     * Indexing.
     *
     * Find the player in coordinate.
     *
     * nullptr if player does not exists in given coordinates, or !isCoordInBoard
     *
     * @param coord  The coordinate to search.
     * @return The player in coordinate.
     */
    Player *operator[](const Coordinate& coord) const;
    /**
     * Calculate the new coordinate with the given move and coordinate.
     *
     * NOOP and ATTACK are no-op, return coord.
     *
     * The new coordinate cannot be outside of the borders.
```

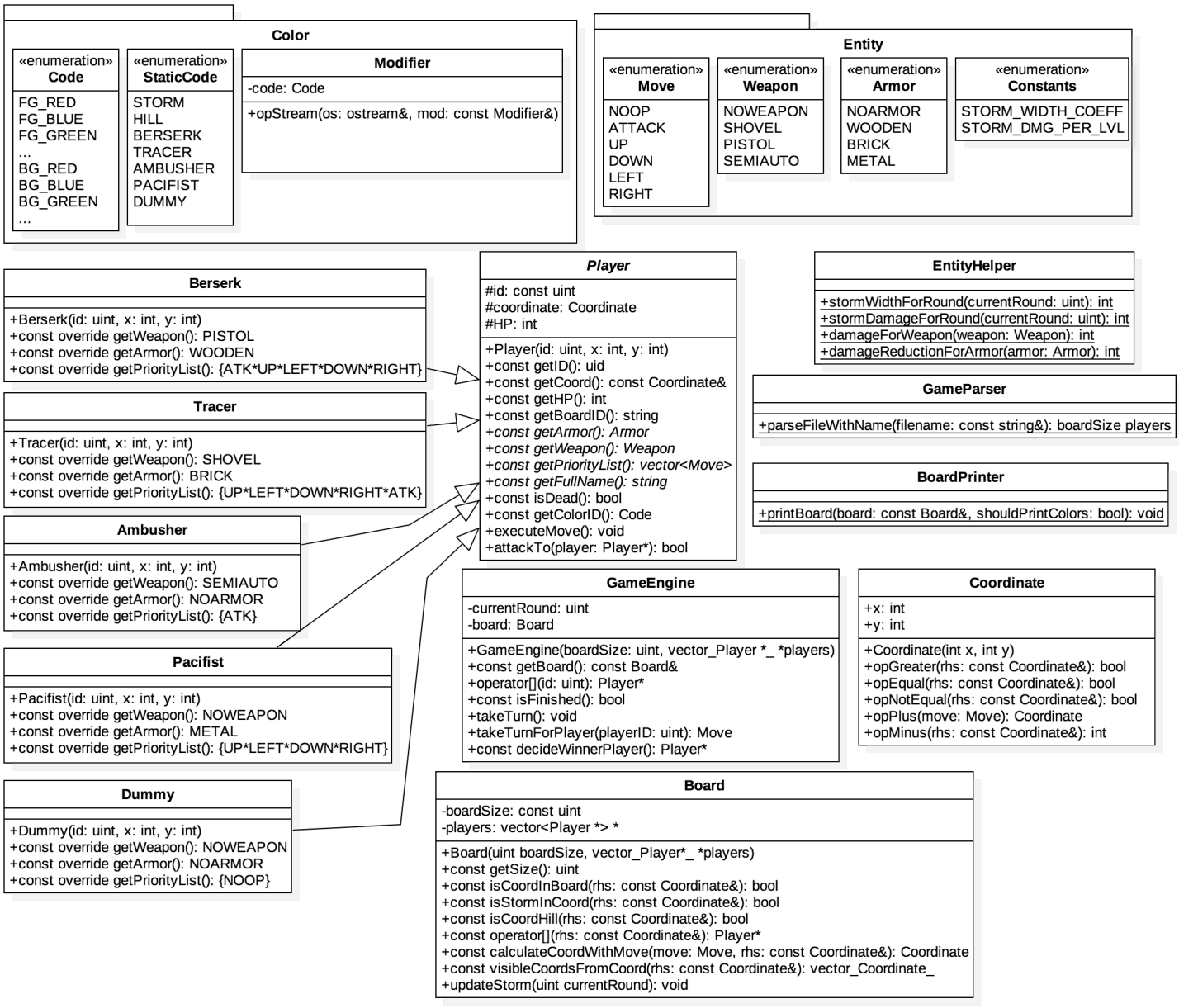
```
     * If it's the case, return coord.
     *
     * Also, if there's another player in the new coordinate,
     * return coord.
     *
     * @param move Move to be made.
     * @param coord The coordinate to be moved.
     * @return Calculated coordinate after the move.
     */
    Coordinate calculateCoordWithMove(Move move, const Coordinate &coord) const;
    /**
     * Find the visible coordinates from given coordinate.
     *
     * Explanation: The immediate UP/DOWN/LEFT/RIGHT tiles must be calculated.
     *
     * There could be max of 4 tiles, and min of 2 tiles (on corners).
     * Every found coordinate must be in the board limits.
     *
     * If the given coordinate is not in board, return a vector with size = 0. ↩
       Order does NOT matter.
     *
     * Example:
     *
     * 01----
     * 02HH--
     * ------
     *
     * visibleCoordsFromCoord(Coordinate(0, 0)) == { (1, 0), (0, 1) }
     * visibleCoordsFromCoord(Coordinate(1, 1)) == { (1, 0), (2, 1), (1, 2), (0, ↩
       1) }
     * visibleCoordsFromCoord(Coordinate(-1, 0)) == { }
     *
     * @param coord The coordinate to search.
     * @return All coordinates visible.
     */
    std::vector<Coordinate> visibleCoordsFromCoord(const Coordinate &coord) const↩
       ;
    /**
     * Calculate the storm according to the currentRound.
     *
     * @param currentRound The current round being played.
     */
    void updateStorm(uint currentRound);

    // DO NOT MODIFY THE UPPER PART
    // ADD OWN PUBLIC METHODS/PROPERTIES BELOW
};
```

3.4 GameEngine

```
class GameEngine {
private:
    uint currentRound;
    Board board;

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

public:
    /**
     * Constructor.
     *
     * GameEngine "owns" these players.
     * GameEngine also "owns" the vector, too.
     *
     * @param boardSize The side length of the board.
     * @param players All players to participate in the game.
     */
    GameEngine(uint boardSize, std::vector<Player *> *players);
    ~GameEngine();

    const Board& getBoard() const;
    /**
     * Indexing.
     *
     * Find the player with given ID.
     *
     * nullptr if not exists.
     *
     * @param id ID of the player.
     * @return The player with given ID.
     */
    Player* operator[](uint id) const;
    /**
     * Decide whether the game is finished.
     *
     * @return true if there is only 1 player (alive), on top of the hill; or ↩
        there are 0 players. False otherwise.
     */
    bool isFinished() const;
    /**
     * Take turn for every player.
     *
     * How-to:
     * - Announce turn start (cout).
     * Example: -- ROUND 1 START --
     * - board.updateStorm(currentRound)
     * - For every player (sorted according to their IDs) that isn't dead (HP <= ↩
        0):
     *       - takeTurnForPlayer(player).
     * - Announce turn end (cout).
     * Example: -- ROUND 1 END --
     */
    void takeTurn();
    /**
     * The most important (algorithm-wise) method.
```

```
     *
     * How-to:
     * - Get player with ID. Return NOOP if not exists.
     * - Get player's priority list.
     * - Get player's visibility from the board (visibleCoordsFromCoord).
     *
     * - If the player is in the storm (isStormInCoord), announce the damage and ↩
        give player stormDamage.
     * - Example: Tracer01(10) is STORMED! (-10)
     *
     * - If dead, announce the death, remove player from the board/list/anywhere,↩
         and return NOOP.
     * - Example: Tracer01(0) DIED.
     *
     * - For MOVE in player's priority list:
     *         - If the MOVE is NOOP:
     *             - return NOOP.
     *         - Else if the MOVE is ATTACK:
     *             - Get all players that this player can attack (board[coord] ↩
        for each in visibilityCoords).
     *             - If none, continue.
     *             - Else:
     *                 - Pick the one with most priority (lowest ID).
     *                 - isPlayerDead = player.attackTo(thatPlayer).
     *                 - if isPlayerIsDead:
     *                     - announce the death.
     *                     - remove thatPlayer from the board/list/anywhere.
     *                 - return ATTACK.
     *         - Else (UP/DOWN/LEFT/RIGHT):
     *             - calculateCoordWithMove(move).
     *             - If the new coordinate is different than the player's (↩
        meaning it's able to do that move)
     *             AND the player is getting closer to the hill;
     *                 - player.executeMove(MOVE).
     *                 - return MOVE.
     *             - Else:
     *                 - continue.
     *
     * // If the priority list is exhausted;
     * return NOOP.
     *
     * @param player Player ID to move.
     * @return move Decided move.
     */
    Move takeTurnForPlayer(uint playerID);
    /**
     * Find the winner player.
     *
     * nullptr if there are 0 players left, or the game isn't finished yet.
     *
     * @return The winner player.
     */
    Player *getWinnerPlayer() const;

    // DO NOT MODIFY THE UPPER PART
    // ADD OWN PUBLIC METHODS/PROPERTIES BELOW
};
```

4 Class Diagram



5 Input Format

The input will be in this format:

```
Board Size:  <s>
Player Count:  <n>
Player<0>ID::Player<0>Type::Player<0>xCoord::Player<0>yCoord
Player<1>ID::Player<1>Type::Player<1>xCoord::Player<1>yCoord
...
Player<n-1>ID::Player<n-1>Type::Player<n-1>xCoord::Player<n-1>yCoord
```

5.1 Input Example

```
1 Board Size: 3
2 Player Count: 2
3 0::Tracer::2::0
4 1::Berserk::2::2
```

The starting for the given input:

```
-- -- 00
-- HH --
-- -- 01
```

6 Extras

6.1 Input

1. There will be no erroneous input.
2. Input generator is also shared with you. Every input file will be created by that generator.
3. Player count will be $n > 0$ && $n < 100$.
4. Board size will be $n > 2$, $n < 28$, $n\%2 == 1$

6.2 Output Notes

1. The grades will be determined with black-box testing, hence you need to be careful about spaces, typos, etc.
2. Output Format is specified in the source code, and there will be plenty of I/O examples provided to you.

6.3 Helpers

There are `Color.h, Entity.h, Coordinate.h, BoardPrinter.h` files provided to you as-is, for your convenience. Do not forget to check these files before implementing!

6.4 Memory

- GameParser and Board is **NOT** responsible for the Players at any time.
- **Only** GameParser **CAN** create Players, and it **MUST** be GameEngine's responsibility to remove/delete these Players.
- A player must be removed (from Board, from GameEngine, from memory) if it's dead, **immediately**.
- The winner (if there is) must be removed **after** GameEngine is deallocated.

7 Grading

For your convenience, there are multiple partial tests already given to you in **StudentPack**. The rest of the points can be earned with proper memory management and proper outputs. The example I/O will be shared on COW, and a simple one is in **StudentPack**.

8 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. Class-wise memory management is explained to you in source-code. 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 newsgroup (news.ceng.metu.edu.tr) for discussions and possible updates on a daily basis.

9 Submission

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

- Player.h
- Player.cpp
- GameParser.h
- GameParser.cpp
- Board.h
- Board.cpp
- GameEngine.h
- GameEngine.cpp
- {Berserk/Tracer/Ambusher/Pacifist/Dummy} .h / .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. In Makefile, there's a command "zipper" that will automatically zip the requested files. *Caution: it requires filesToUpload.txt, do not delete that.*

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
```

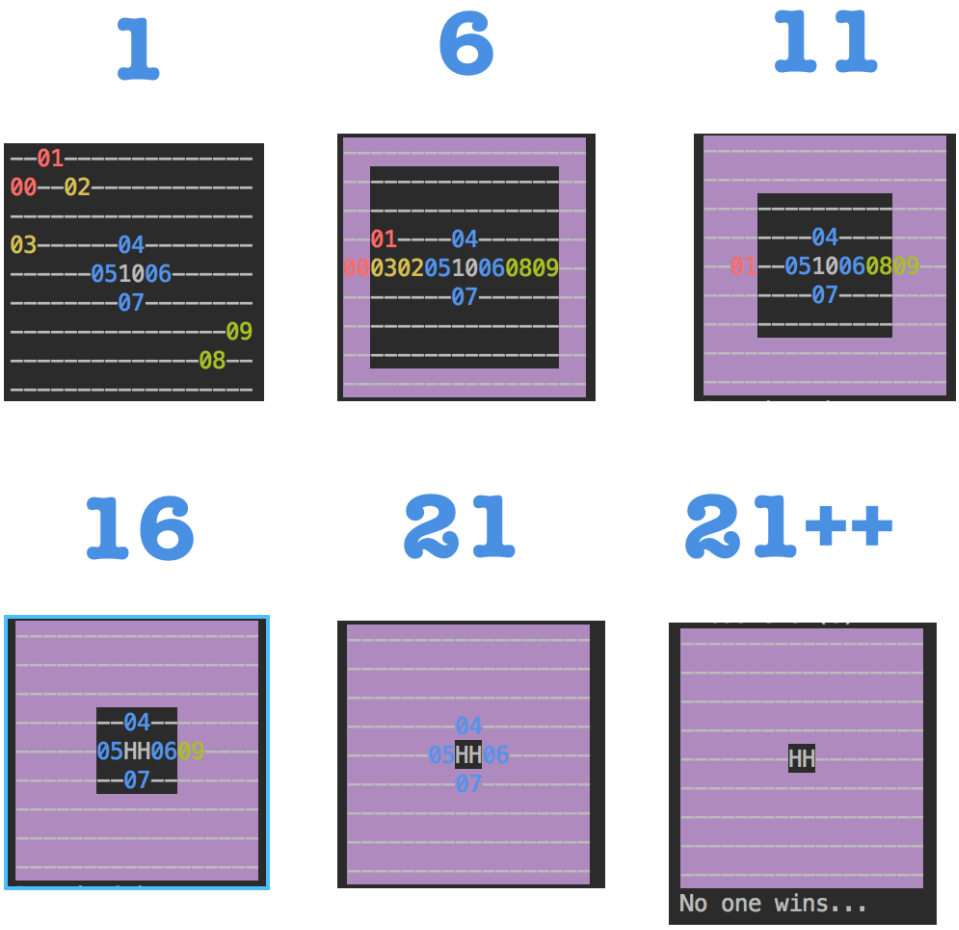
10 Addendum

10.1 Storm

So, how does the Storm work? There are 2 methods given to you for calculations, one is `stormWidthForRound`, and one is `stormDamageForRound`. They will calculate the storm damage and the width according to the currentRound given.

You may noticed that hill cannot be stormed, but the width can be much more than the board size itself. It's intended, and the damage is also intended, so you must use these methods accordingly (hill should not be covered in storm in any case).

Here's an image which shows how the storm is built up:



Hint: Don't think too complex, if your calculations seems to have too many edge cases, there is most probably a better way to do that. Think about square(s).

10.2 Players, What NOT to do

You **MUST NOT** store any info regarding the type of the Player. You **SHOULD NOT** act according to the type of the Player (you may check the name of the player instead of storing a type, don't do that.)

You **SHOULD NOT** have any code like:

```
Armor Player::getArmor() const {
  if(this->type == BERSERK) {
    return WOODEN
  }
  .
  .
  .

  or

  if(this->fullName() -contains- BERSERK) {
    return WOODEN
  }
  .
  .
  .
}
```

Every class must implement/override necessary methods. There will be white-box tests which will check for this structures, and if you write such code on the upper part, your homework **WILL NOT** be graded at all.

Good luck, have fun.

- engin