Alejandro Rubio Martínez

Ximo Sanz Tornero

Reto 2 - Estructura de datos

Hemos basado nuestro TDA del Tetris en 5 clases, las cuales explicamos por encima a continuación:

Clase Piece

Esta clase es la que representa una pieza como una matriz dinámica que tendrá posiciones vacías (representadas como un 0) y posiciones de un color (representadas como un número del 1 al 7). Además, cada pieza incluye una posición en el plano real (esto es ordenada y abscisa) que será necesaria luego para moverla dentro del tablero (véase board.h). En ningún momento en esta clase se comprueba que estas posiciones sean correctas, esto se hará en otra parte (como en el main o en el tablero).

```
/**
 * @file piece.h
 * @author Alejandro Rubio, Ximo Sanz
 * @brief The file which contains the TDA object Piece
 *
 */
#ifndef PIECE_H
#define PIECE_H

#include <iostream>
#include <fstream>

#define RED = 1
#define GREEN = 2
#define BLUE = 3
#define ORANGE = 4
#define PURPLE = 5
#define YELLOW = 6
#define CYAN = 7

/**
 * @brief The class Piece
 *
 */
class Piece {
private:
    short int** matrix;
    int nRows, nCols;
    int posx, posy;

    /**
     * @brief Allocates memory for the matrix
     *
     * @param r Rows to allocate
     * @param c Columns to allocate
     * @pre @p r and @p c are both greater than 0
     */
    void allocate(int r, int c);
    /**
     * @brief Desallocate the matrix memory in usage
     *
     */
    void deallocate();
    /**
     * @brief Copy an object in another one
     *
     * @param p The object we want to copy
     */
    void copy(const Piece &p);

public:
    /**
     * @brief Construct a new Piece object depending of n
     *
     * @param n If n is 0 it builds a random piece, otherwise build the known
piece
     * @pre @p n must be between the values of 0-7
     */
    Piece(int n = 0);
    /**
     * @brief Construct a new Piece object from a given Piece
     *
     * @param p The piece
     */
    Piece(Piece p);

    /**
     * @brief Destroy the Piece object
     *
     */
    ~Piece();
    /**
     * @brief Get the posx of the object
     *
     * @return int posx
     */
    int getPosx() const;
    /**
     * @brief Get the posy of the object
     *
     * @return int posy
     */
    int getPosy() const;
    /**
     * @brief Get the number of rows of the object
     *
     * @return int nRows
     */
    int getRow() const;
    /**
     * @brief Get the number of columns of the object
     *
     * @return int nCols
     */
    int getCol() const;
    /**
     * @brief Gets the color, (represented by a number), of the matrix
     *
     * @return int The element in the @p r, @p c position
     */
    short int get(int r, int c) const;
    /**
     * @brief Set the Posx of the object
     * @pre @p x must be a possible position
     *
     * @param x The position in the abcissa axis
     *
     */
    void setPosx(int x);
    /**
     * @brief Set the Posy object
     *
     * @param y The position in the y-axis diagram
     */
    void setPosy(int y);
    /**
     * @brief Set the number of rows of the matrix
     *
     * @param r The number of rows
     * @pre @p r must be a valid number
     */
    void setnRows(int r);
    /**
     * @brief Set the number of columns of the matrix
     *
     * @param c The columns
     * @pre @p c must be a valid number
     */
    void setnCols(int c);
    /**
     * @brief Set the number contained inside the number
     *
     * @param color The number of the color
     * @param r The row we want to introduce the number
     * @param c The column where we want to introduce the number
     * @pre @p color must be between 1-7, @p c and @p r must be valids
     */
    void set(int color, int r , int c);
    /**
     * @brief Turns the piece to the right
     *
     */
    void turnRight();
    /**
     * @brief Turns the piece to the left
     *
     */
    void turnLeft();
    /**
     * @brief Move the piece to the right
     *
     */
    void moveRight();
    /**
     * @brief Move the piece to the left
     *
     */
    void moveLeft();
    /**
     * @brief Move the piece down
     *
     */
    void moveDown();
    /**
     * @brief Move the piece up
     *
     */
    void moveUp();
    /**
     * @brief Overload the = operator
     *
     * @param p The piece we want to =
     * @return Piece The new piece
     */
    Piece& operator=(const Piece &p);
    /**
     * @brief Overload the [] operator for vectors of vectors
     *
     * @param i The position we want to return
     * @return short int The first position of the vector
     */
    short int &operator[](int i);
};
/**
 * @brief Overload the >> operator
 *
 * @param stream The stream of datas
 * @param p The piece where we will introduce the data
 * @return std::istream& The stream
 */
std::istream& operator>>(std::istream& stream, Piece &p);
/**
 * @brief Overload the << operator
 *
 * @param stream The stream of datas
 * @param p The piece from which we will show the datas
 * @return std::ostream& The stream
 */
std::ostream& operator<<(std::ostream& stream, const Piece &p);

#endif
```

Clase Stats

Esta clase es la más simple de todas ya que solo usa datos conocidos por c++ (enteros y strings) que solo servirá para almacenar las estadísticas de la partida. Esta son el tiempo que tienes para introducir un movimiento antes de caer, la cantidad de filas destruidas, la cantidad de piezas usadas y el nivel en el que juegas (más alto en función del tiempo). También encontramos el título del juego.

```cpp
/**
 * @file stats.h
 * @author Alejandro Rubio, Ximo Sanz
 * @brief The file which contains the TDA object Stats
 *
 */
#ifndef STATS_H
#define STATS_H

#include <iostream>
#include <fstream>
#include <string>

/**
 * @brief The class Stats
 *
 */
class Stats {
private:
    int time, destroyedRows, usedPieces, level;
    std::string title;

public:
    /**
     * @brief Construct a new Stats object
     *
     */
    Stats();
    /**
     * @brief Construct a new Stats object with the specified title
     *
     * @param a The title
     */
    Stats(std::string a);
    /**
     * @brief Get the Time of the object
     *
     * @return int time
     */
    int getTime() const;
    /**
     * @brief Get the number of the destroyed rows object
     *
     * @return int destroyedRows
     */
    int getRows() const;
    /**
     * @brief Get the number of  used Pieces of the object
     *
     * @return int usedPieces
     */
    int getPieces() const;
    /**
     * @brief Get the Level object
     *
     * @return int level
     */
    int getLevel() const;
    /**
     * @brief Get the Title of the object
     *
     * @return std::string title
     */
    std::string getTitle() const;
    /**
     * @brief Set the Time of the object
     *
     * @param n The time
     * @pre @p n must be positive
     */
    setTime(int n);
    /**
     * @brief Set the Rows object
     *
     * @param r The destroyed rows
     * @pre r must be positive
     */
    setRows(int r);
    /**
     * @brief Set the Level object
     *
     * @param l The new level
     * @pre @p l must be positive
     */
    setLevel(int l);
    /**
     * @brief Set the usedPieces of the object
     *
     * @param p The number of used pieces
     * @pre @p p must be positive
     */
    setPieces(int p);
    /**
     * @brief Set the Title of the object
     *
     * @param t The string which contains the title
     */
    setTitle(std::string t);
    /**
     * @brief Calculates the level of the game depending of the time
     *
     */
    calculateLevel();


};

/**
 * @brief Overload the >> operator
 *
 * @param stream The stream of datas
 * @param s The stats where we will introduce the data
 * @return std::istream& The stream
 */
std::istream& operator>>(std::istream& stream, Stats &s);
/**
```

```
 * @brief Overload the << operator
 *
 * @param stream The stream of datas
 * @param s The stats from which we will show the datas
 * @return std::ostream& The stream
 */
std::ostream& operator<<(std::ostream& stream, const Stats &s);

#endif
```

Clase Piecelist

Esta clase estará formada por un vector de piezas que estará capado por un máximo (en nuestro caso 4) y que solo podremos incluir al final y extraer del principio.

```
/**
 * @file Piecelist.h
 * @author Alejandro Rubio, Ximo Sanz
 * @brief The file which contains the TDA object Piecelist, a vector of pieces
 */

#ifndef PIECELIST_H
#define PIECELIST_H

#include "piece.h"

#define MAX_PIECES = 4

/**
 * @brief The Piecelist class
 *
 */
class Piecelist {
private:
    Piece* currentPieces;
    int nPieces;            //Must be always between 0 and MAX_PIECES

    /**
     * @brief Allocates memory for the matrix
     *
     * @param n Rows to allocate
     * @pre @p n is greater than 0
     */
    void allocate(int n);
    /**
     * @brief Desallocate the vector memory in usage
     *
     */
    void deallocate();
    /**
     * @brief Copy an object in another one
     *
     * @param pl The object we want to copy
     */
    void copy(const Piecelist &pl);
    /**
     * @brief Fill the list with pieces
     *
     * @param n The number of pieces we want to fill
     * @pre nPieces + @p n must be less or equal to MAX_PIECES
     */
    void fill(int n);
public:
    /**
     * @brief Construct a new Piecelist object
     *
     * @param n The number of pieces in the list, by default MAX_PIECES
     */
    Piecelist(int n = MAX_PIECES);
    /**
     * @brief Copy constructor of the class
     *
     * @param pl The object we want to copy
     */
    Piecelist(const Piecelist &pl);
    /**
     * @brief Destroy the Piecelist object
     *
     */
    ~Piecelist();
    /**
     * @brief Get the Size of the object
     *
     * @return int nPieces
     */
    int getSize() const;
    /**
     * @brief Extract one piece from the list
     *
     * @return Piece The extracted piece
     * @post After a piece has been extraxted is called fill(1);
     */
    Piece extract();
    /**
     * @brief Remove a piece from the beggining
     *
     */
    void pop();
    /**
     * @brief Add a piece to the end of the list
     *
     * @param p The piece we want to introduce
     * @pre nPieces must be < MAX_PIECES
     */
    void push(const Piece &p);
    /**
     * @brief Overload the += operator
     *
     * @param p The piece we want to add
     * @return Piecelist& The piecelist with the piece added
     */
    Piecelist& operator+=(const Piece &p);
    /**
     * @brief Overload the [] operator for vectors of pieces
     *
     * @param i The position we want to return
     * @return Piece The piece
     */
    Piece &operator[](int i);
};

/**
 * @brief Overload the >> operator
 *
 * @param stream The stream of datas
 * @param pl The stats where we will introduce the data
 * @return std::istream& The stream
 */
std::istream& operator>>(std::istream& stream, Piecelist &pl);
/**
 * @brief Overload the << operator
 *
 * @param stream The stream of datas
 * @param pl The stats from which we will show the datas
 * @return std::ostream& The stream
 */
std::ostream& operator<<(std::ostream& stream, const Piecelist &pl);
```

Clase Board

Esta clase representa al tablero donde jugaremos. Está representada como una matriz dinámica sobre la que iremos moviendo las piezas y que al final iremos llenando poco a poco.

```
/**
 * @file Board.h
 * @author Alejandro Rubio, Ximo Sanz
 * @brief The file which contains the TDA object Board
 */

#ifndef BOARD_H
#define BOARD_H

#include "piece.h"

/**
 * @class Board
 * @brief Represents the board in the game
 *
 */
class Board {
private:
    short int** matrix;
    int rows, columns;

    /**
     * @brief Allocates memory for the matrix
     *
     * @param r Rows to allocate
     * @param c Columns to allocate
     * @pre @p r and @p c are both greater than 0
     */
    void allocate(int r, int c);
    /**
     * @brief Desallocate the matrix memory in usage
     *
     */
    void deallocate();
    /**
     * @brief Copy an object in another one
     *
     * @param t The object we want to copy
     */
    void copy(const Board &t);

public:
    /**
     * @brief Construct a new Board object
     *
     */
    Board();
    /**
     * @brief Construct a new Board object from parameters
     *
     * @param r The rows
     * @param c The columns
     * @pre @p r and @p c are both greater than 0
     */
    Board(int r, int c);
    /**
     * @brief Copy constructor of a new object T
     *
     * @param t The Board we want to copy
     */
    Board(const Board &t);
    /**
     * @brief Destroy the Board object
     *
     */
    ~Board();
    /**
     * @brief Return a value in matrix at the given position
     *
     * @param r The row where the value is
     * @param c The column where the value is
     * @pre @p r and @p c must be in the right values
     * @return int The value
     */
    int get(int r, int c) const;
    /**
     * @brief Return the number of rows
     *
     * @return int The number of rows
     */
    int getRows() const;
    /**
     * @brief Return the number of columns
     *
     * @return int The number of columns
     */
    int getColumns() const;
    /**
     * @brief Set the value of a given position
     *
     * @param r The row where the value is
     * @param c The column where the value is
     * @param v The value we want to introduce between 1 and 7
     * @pre @p r and @p c must be in the right values
     */
    void set(int r, int c, int v);
    /**
     * @brief Introudce a piece onto the top of the matrix
     *
     * @param p The piece we want to introduce
     */
    void add(const Piece &p);
    /**
     * @brief Function that clears a row and moves the upper rows one position
below
     *        and the top becomes empty
     *
     * @param r The row to clear
     */
    void clearRow(int r);
    /**
     * @brief Check if the given row is full
     *
     * @param r The row we want to check
     * @return true if the row is full
     * @return false if the row has any empty positions
     */
    bool fullRow(int r) const;
    /**
     * @brief Check if a given piece can continue falling
     *
     * @param p The piece to check
     * @return true If the piece can continue falling
     * @return false if not
     */
    bool canFall(const Piece &p) const;
    /**
     * @brief Check if a piece can move into a given direction
     *
     * @param p The piece we want to move
     * @param direction True to the right, false to the left
     * @return true If the piece can move
     * @return false If the piece can not move
     */
    bool canMove(const Piece &p, bool direction) const;
    /**
     * @brief Check if a piece can turn into a given direction
     *
     * @param p The piece we wanto to turn
     * @param direction True to right, false to left
     * @return true If it can be turned
     * @return false If it can not be turned
     */
    bool canTurn(const Piece &p, bool direction) const;
    /**
```

```
     * @brief Fix a piece in the board when cant continue moving
     *
     * @param p The piece to fix
     */
    void fix(const Piece &p);
    /**
     * @brief Check if the next piece is out of the board on the top
     *
     * @return true If the next piece can be introduced
     * @return false If not
     */
    bool finish();
    /**
     * @brief Overload the = operator
     *
     * @param b The board we want to =
     * @return Board The new board
     */
    Board& operator=(const Board &b);
    /**
     * @brief Overload the [] operator for vectors of vectors
     *
     * @param i The position we want to return
     * @return short int The first position of the vector
     */
    short int &operator[](int i);

};

/**
 * @brief Overload the >> operator
 *
 * @param stream The stream of datas
 * @param b The board where we will introduce the data
 * @return std::istream& The stream
 */
std::istream& operator>>(std::istream& stream, Board &b);
/**
 * @brief Overload the << operator
 *
 * @param stream The stream of datas
 * @param b The board from which we will show the datas
 * @return std::ostream& The stream
 */
std::ostream& operator<<(std::ostream& stream, const Board &b);

#endif
```

Clase Game

Esta clase solo sirve para enlazar las otras clases y que sea más fácil trabajar con ellas en un futuro main. Esta clase es extensible a la parte de visualización por pantalla.

```
/**
 * @file game.h
 * @author Alejandro Rubio, Ximo Sanz
 * @brief The file which contains the TDA object Game
 *
 */
#ifndef GAME_H
#define GAME_H

#include "board.h"
#include "piecelist.h"
#include "stats.h"
/**
 * @brief The Game class
 *
 */
class Game {
private:
    Stats stat;
    Board board;
    Piecelist pieces;
    /**
     * @brief Copy an object in another one
     *
     * @param g The object we want to copy
     */
    void copy(const Game &g);
public:
    /**
     * @brief Construct a new Game object
     *
     */
    Game();
    /**
     * @brief Construct a new Game object from another one
     *
     * @param match The game we want to copy
     */
    Game(const Game& match);
    /**
     * @brief Destroy the Game object
     *
     */
    ~Game();
    /**
     * @brief Plays a determinated movement
     *
     * @param movement The movement we want to do
     * @post If there is not a defined movement to that string it doesn't do
anything
     */
    void play(std::string movement);
    /**
     * @brief Overload the = operator
     *
     * @param g The game we want to =
     * @return Game The new game
     */
    Game& operator=(const Game &g);
};

/**
```

```
 * @brief Overload the >> operator
 *
 * @param stream The stream of datas
 * @param g The game where we will introduce the data
 * @return std::istream& The stream
 */
std::istream& operator>>(std::istream& stream, Game &g);
/**
 * @brief Overload the << operator
 *
 * @param stream The stream of datas
 * @param g The game from which we will show the datas
 * @return std::ostream& The stream
 */
std::ostream& operator<<(std::ostream& stream, const Game &g);

#endif
```