Project Four: Quarto

Out: July 5, 2019; Due: July 19, 2019

I. Motivation

To give you experience in implementing abstract data types (ADTs), using interfaces (abstract base classes), and using interface/implementation inheritance.

II. Introduction

In this project, you will implement a two-player board game called **Quarto** (see this Wikipedia page: https://en.wikipedia.org/wiki/Quarto_(board_game)). It is played on a 4x4 board with 16 unique pieces, which are described by 4 attributes: height (short or tall), color (beige or sepia), shape (circle or square), and top (hollow or solid). See the illustration below.

The game starts with an empty board and a common pool of all the available pieces. At each turn, the players alternatively select one piece in the common pool and the other player must then place it on the board. The winner of the game is the first player who places a piece on the board which forms a horizontal, vertical, or diagonal row of 4 pieces, and all of which have a common attribute (all short, all circle, etc.). The game may end in a draw if all the pieces are placed but no one manages to reach that goal.

To represent this game, we make the following assumptions:

- A square of the board is accessed by its vertical coordinate (A, B, C, or D), followed by its horizontal coordinate (1, 2, 3, or 4). For instance, the top-left square of the board has coordinates A1 and top-right has coordinates A4.
- The values Short and Tall of Height are represented by S and T. The values beige and sepia of Color are represented by B and E. The values Circle and Square of Shape are represented by C and Q. The values Hollow and Solid of Top are represented by H and O.
- A piece is described by a 4-character code for its attributes: Height, Color, Shape and Top. Therefore, the bottom-right piece in the above picture is encoded TBCH.

III. Programming Assignment

Each component of the game will be represented by an ADT: `Piece`, `Pool`, `Square`, `Board`, `Player`, and `GameDriver`, which will be described below. All ADTs are specified in their respective header files whose filenames are the ADT names in lower case. You are not allowed to change the definitions of those ADTs. You will provide an implementation for each of them, except for `Player` for which you will provide two different implementations (see below for more details), in their corresponding cpp files.

To help you, we provide you two files `quarto.h` and `quarto.cpp` that specify some useful global variables, enum types, and their string encodings.

```
const int N = 4;
const int NP = N*N;

enum Height {SHORT, TALL};
enum Color {BEIGE, SEPIA};
enum Shape {CIRCLE, SQUARE};
enum Top {HOLLOW, SOLID};
enum Vaxis {A, B, C, D};
enum Haxis {H1, H2, H3, H4};
```

Moreover, in `exceptions.h` and `exceptions.cpp`, we also provide you the definitions of three exceptions: `SquareException`, `UsedPieceException`, and `InvalidInputException`.

You are also not allowed to change these four files (`quarto.h/cpp, exceptions.h/cpp`) You can check all those related files for more details. We now present the definitions of the classes for each ADT.

1. The Piece ADT

Your first task is to implement the following ADT representing a piece:

```
class Piece{
   // represents a piece of the game
   Height h;
   Color c;
   Shape s;
   Top t;
   bool used; // indicates if it has already been placed on the board

public:
   Piece(Height h=SHORT, Color c=BEIGE, Shape s=CIRCLE, Top t=HOLLOW);
   // EFFECTS: create an unused piece with attributes (h, c, s, t)

   bool compareHeight(const Piece &p) const;
   // EFFECTS: return true if "this" has the same height as "p"
   //          return false otherwise

   bool compareColor(const Piece &p) const;
   // EFFECTS: return true if "this" has the same color as "p"
   //          return false otherwise

   bool compareShape(const Piece &p) const;
   // EFFECTS: return true if "this" has the same shape as "p"
   //          return false otherwise

   bool compareTop(const Piece &p) const;
   // EFFECTS: return true if "this" has the same top as "p"
   //          return false otherwise

   bool isUsed() const;
   // EFFECTS: return true if "this" has been placed on the board
   //          return false otherwise

   void setUsed(bool u);
   // MODIFIES: "used"
   // EFFECTS: set "used" to "u"

   std::string toString() const;
   // EFFECTS: return a 4-char string representing "this"
   // see file "quarto.h/cpp" for the encoding of each attribute
   // e.g., SEQO for a piece that is short, sepia, square, and solid
};
```

2. The Pool ADT

Your second task is to implement the following ADT representing a pool of pieces:

```
class Pool{
   // OVERVIEW: a pool of Quarto pieces
   Piece pieces[NP];

   Piece & getUnusedPiece(int index);
   // EFFECTS: return the reference to a piece "p" given its index
   //          throw UsedPieceException if "p" is placed on the board

public:
   Pool();
   // EFFECTS: creates a pool of the 16 unused Quarto pieces
   // "unused" means "not placed on the board"

   Piece & getUnusedPiece(Height h, Color c, Shape s, Top t);
   // EFFECTS: return the reference to a piece "p" given its
attributes
   //          throw UsedPieceException if "p" is placed on the board

   Piece & getUnusedPiece(const std::string &in);
   // EFFECTS: return the reference to a piece "p" given its encoding
   //          throw UsedPieceException if "p" is placed on the board
   // see file "quarto.h/cpp" for the encoding of each attribute
   // e.g., "SEQO" encodes a piece that is short, sepia, square, and
   // solid

   std::string toString() const;
   // EFFECTS: returns a string that lists all the unused pieces
   // preceded by string "Available:\n" if the list is not empty
   //          returns the empty string otherwise
   // e.g., when the game starts, the returned string would print as
   // follows:
   // Available:
   // SB SB SB SB SE SE SE SE TB TB TB TB TE TE TE TE
   // CH CO QH QO CH CO QH QO CH CO QH QO CH CO QH QO
   //
   // Note that all the 4-character strings have been split over two
   // lines (i.e., SBCH is the first piece, SBCO is the second...)
};
```

When you need to throw a `UsedPieceException`, you will throw it with the piece corresponding to the index, the attribute values, or the string passed as argument to the method of the Pool class.

A piece can be seen as 4 binary digits using the following order over attributes and the encoding of their values provided by the enum types: Height, Color, Shape, and Top. For instance, SBCH corresponds to 0000, while TEQO represents 1111. You will use the corresponding decimal number as the index in the array `pieces`.

3. The Square ADT

Your third task is to implement the ADT to represent a square of a Quarto 4x4 board. Its definition depends on two enum types for representing the position of a square:

```
enum Vaxis {A, B, C, D};
enum Haxis {H1, H2, H3, H4};
```

The Square ADT is given by:

```
class Square{
   // OVERVIEW: a square of Quarto 4x4 board
   Vaxis v;
   Haxis h;
   const Piece *p;

public:
   Square(Vaxis v=A, Haxis h=H1);
   // EFFECTS: create an empty square at position (v, h)

   Vaxis getV() const;
   // EFFECTS: return the vertical coordinate of the square

   void setV(Vaxis v);
   // MODIFIES: "this"
   // EFFECTS: update the vertical coordinate of the square

   Haxis getH() const;
   // EFFECTS: return the horizontal coordinate of the square

   void setH(Haxis h);
   // MODIFIES: "this"
   // EFFECTS: update the horizontal coordinate of the square

   const Piece &getPiece() const;
   // EFFECTS: return a const reference to the piece placed on the
   //          square, throw SquareException if the square is empty

   void setPiece(const Piece *p);
   // MODIFIES: "this"
   // EFFECTS: place "p" on the square

   bool isEmpty() const;
```

```
   // EFFECTS: return true if the square is empty, and false otherwise

   bool isOnFirstDiagonal() const;
   // EFFECTS: return true if the square is on the first diagonal
   // of the board (which goes from top-left to bottom-right)
   //          return false otherwise

   bool isOnSecondDiagonal() const;
   // EFFECTS: return true if the square is on the second diagonal
   // of the board (which goes from top-right to bottom-left)
   //          return false otherwise

   std::string toString() const;
   // EFFECTS: return a string the encodes the position of the square
   // e.g., A1 for the first square on the top-left,
   //       and B1 for the square beneath it
};
```

Here, when you need to throw a `SquareException`, you will throw it with the corresponding square and the string "empty".

4. The Board ADT

Your fourth task is to implement an ADT to represent the Quarto board.

```
class Board{
   // OVERVIEW: a Quarto 4x4 board
   Square grid[N][N];

public:
   Board();
   // EFFECTS: create a 4x4 board of empty squares

   Square &getSquare(Vaxis v, Haxis h);
   // EFFECTS: return a reference to the square at position (v, h)

   Square &getEmptySquare(Vaxis v, Haxis h);
   // EFFECTS: return a reference to the square at position (v, h)
   //          throw SquareException if the square is not empty

   Square &getEmptySquare(const std::string &s);
   // EFFECTS: return a reference to the square at a position encoded
   //          in "s" (e.g., "B2").
   //          throw SquareException if the square is not empty

   void place(Piece &p, Square &sq);
   // MODIFIES: "p" and "sq"
   // EFFECTS: place piece "p" on square "sq"

   bool isWinning(const Piece &p, const Square &sq);
```

```
    // REQUIRES: if "p" is used, then it is already placed on "sq".
    //           Otherwise, "sq" is empty.
    // EFFECTS: return true if piece "p" on square "sq" yields a
    // winning position (i.e., 4 pieces with at least one common
    // attribute, which are aligned horizontally, vertically, or
    // diagonally
    // REMARK: "p" may or may not have been placed on "sq"
    // (Please think about why), which may be empty.

    std::string toString() const;
    // EFFECTS: return a string that represents the board
    // e.g., at the beginning of the game, the returned string
    //       prints as follows:
    //     1    2    3    4
    //   +----+----+----+----+
    // a |    |    |    |    |
    //   |    |    |    |    |
    //   +----+----+----+----+
    // b |    |    |    |    |
    //   |    |    |    |    |
    //   +----+----+----+----+
    // c |    |    |    |    |
    //   |    |    |    |    |
    //   +----+----+----+----+
    // d |    |    |    |    |
    //   |    |    |    |    |
    //   +----+----+----+----+
    //       after placing, SBCH on A1 and TBQO on C2, we get:
    //     1    2    3    4
    //   +----+----+----+----+
    // a | SB |    |    |    |
    //   | CH |    |    |    |
    //   +----+----+----+----+
    // b |    |    |    |    |
    //   |    |    |    |    |
    //   +----+----+----+----+
    // c |    | TB |    |    |
    //   |    | QO |    |    |
    //   +----+----+----+----+
    // d |    |    |    |    |
    //   |    |    |    |    |
    //   +----+----+----+----+
};
```

Here, when you need to throw a `SquareException`, you will throw it with the corresponding square and the string `"not empty"`.

5. The Player ADT

Your next task is to implement two different Quarto players. You will provide their implementations in `Player.cpp`. The interface for a Player is:

```
class Player{
   // OVERVIEW: A base class for the player interface
protected:
   Board *board;
   Pool *pool;

public:
   Player(Board *board, Pool *pool): board(board), pool(pool){}
   // EFFECTS: create a player with access to "board" and "pool"

   virtual Piece & selectPiece() = 0;
   // REQUIRES: there is an unused piece to select
   // EFFECTS: return an unused piece for the next player to place

   virtual Square & selectSquare(const Piece &p) = 0;
   // REQUIRES: there is an empty square to place the piece "p"
   // EFFECTS: return an empty square to place the piece "p".
};
```

The first derived class is `HumanPlayer`, which asks the standard input to specify the piece to select or the square to place a piece.

The method `selectPiece` repeatedly prints `"Enter a piece:"` and reads a 4-character string until it corresponds to an **unused** piece. In such a case, it returns a reference to that piece. If the string does not correspond to a piece, it prints the string followed by `" is an invalid input."` For example, if the string is `"ABCD"`, it prints:

```
ABCD is an invalid input.
```

If the string corresponds to a used piece (say `SBCH`), it prints:

```
SBCH is already used.
```

The method `selectSquare` repeatedly prints `"Enter a position:"` and reads a 2-character string until it corresponds to an empty square. In such a case, it returns a reference to that square. If the string corresponds to a non-empty square, it prints the string followed by `" is not empty."` For example, if the square at position `"B4"` is occupied, it prints:

```
B4 is not empty.
```

If the string does not correspond to a valid position, it prints the string followed by `" is an invalid input."` For example, if the string is `"C5D"`, it prints:

```
C5D is an invalid input.
```

The second derived class is `MyopicPlayer`. For `selectPiece`, the player will choose randomly (see below) among the pieces that are not considered bad. A bad piece is one that can directly lead to the victory of the opponent (if the latter places it correctly on the board). If all the pieces are bad, the player randomly selects one among them.

For `selectSquare`, `MyopicPlayer` first searches for a square that would let the player win directly if such a square exists. In that case, it will return the square with the lowest Vaxis index and in case of equality with the lowest Haxis index. Otherwise it selects an empty square randomly.

When the player needs to make a random choice among `n` candidates (we assume that the `n` pieces are ordered such that their indices are increasing), the player would choose a random number by calling `rand()%n` which would return the index of the candidate to select.

You should provide access to an instance of those two implementations of `Player` via the following two functions:

```
extern Player *getHumanPlayer(Board *b, Pool *p);

extern Player *getMyopicPlayer(Board *b, Pool *p, unsigned int seed);
```

They return a pointer to static global instances declared in your `Player.cpp` file. Those instances should update their pointers to the board and the pool passed in the arguments. To initialize the pseudo-random sequence generator, you will call `srand(seed)` in `getMyopicPlayer`.

4. The Driver program

Finally, you should implement a driver program that can be used to play Quarto given the implementation of the ADTs described above.

You are asked to put your implementation of this driver program in a file named `game.cpp`.

The driver program, when run, takes at least two arguments, with an additional optional third one:

```
    <first player> <second player> [seed]
```

The first and second arguments are either `"h"` or `"m"` indicating whether the first and second players are controlled by `HumanPlayer` or `MyopicPlayer`. The third argument is the seed used in `getMyopicPlayer` to generate the random numbers in the game.

For example, suppose that you program is called `game`. It may be invoked by typing in a terminal:

```
./game h m 2019
```

Then your program would start the game with the first player controlled by `HumanPlayer`, the second controlled by `MyopicPlayer`, and the seed is set to 2019.

The driver first builds a 4x4 board, a pool of the 16 Quarto pieces, an instance of `HumanPlayer` and an instance of `MyopicPlayer`. Depending on the arguments to the program, it uses the correct instances of Player to control the two players. Note that there is no issue letting the two players be controlled by the same instance (if the first two arguments were `"h h"` or `"m m"`).

For any game turn, the driver prints the board and the available pieces. For instance, at the start of the game, the output would be:

```
    1    2    3    4
  +----+----+----+----+
a |    |    |    |    |
  |    |    |    |    |
  +----+----+----+----+
b |    |    |    |    |
  |    |    |    |    |
  +----+----+----+----+
c |    |    |    |    |
  |    |    |    |    |
  +----+----+----+----+
d |    |    |    |    |
  |    |    |    |    |
  +----+----+----+----+

Available:
SB SB SB SB SE SE SE SE TB TB TB TB TE TE TE TE
CH CO QH QO CH CO QH QO CH CO QH QO CH CO QH QO
```

Then the driver prints:

```
Player <i>'s turn to select a piece:
```

where `<i>` is either `1` or `2` when it is the i-th player's turn to select a piece. Method `selectPiece` of the correct instance is then called.

Then the driver prints the code of the selected piece (even if the player was `HumanPlayer`):

```
WXYZ selected.
```

where `WXYZ` is the code representing the selected piece.

Then the driver prints:

```
Player <j>'s turn to select a square:
```

where `<j>` is `1` if `<i>` was `2`, and it is `2` if `<i>` was `1`. Method `selectSquare` of the correct instance is then called.

Then the driver prints the code of the selected position (even if the player was `HumanPlayer`):

```
XY selected.
```

where `X` is `A, B, C,` or `D`, and `Y` is `1, 2, 3` or `4`.

Then the driver prints the updated board and pool of unused pieces.

If j-th player won with his/her last move, then the driver prints the following message and exits:

```
Player <j> has won!
```

Otherwise a new turn starts with the role of players `i` and `j` swapped.

If the game ends with a draw, the driver prints the following message and exits:

```
It is a draw.
```

IV. Implementation Requirements and Restrictions

- You may `#include <iostream>`, `<iomanip>`, `<string>`, `<cstdlib>`, `<cstring>`, `<ctime>` and `<cassert>`. No other system header files may be included, and you may not make any call to any function in any other library.
- Output should only be done where it is specified.
- You may not use the `goto` command.
- You may not have any global variables in the driver. You may use global state in the class implementations, but it must be static and (except for the two players) const.

- You may assume that functions are called consistently with their advertised specifications. This means you need not perform error checking. However, when testing your code in concert, you may use the `assert()` macro to program defensively.

V. Source Code Files and Compiling

There are seven header files (`quarto.h`, `piece.h`, `pool.h`, `square.h`, `board.h`, `player.h`, and `exceptions.h`) and two C++ source files (`quarto.cpp` and `exceptions.cpp`) located in the `Project-Four-Related-Files.zip`:

You should copy these files into your working directory. **DO NOT modify them!**

You need to write six other C++ source files: `piece.cpp`, `pool.cpp`, `square.cpp`, `board.cpp`, `player.cpp`, and `game.cpp`. They are discussed above and summarized below:

`piece.cpp`:	your Piece ADT implementation
`pool.cpp`:	your Pool ADT implementation
`square.cpp`:	your Square ADT implementation
`board.cpp`:	your Board ADT implementation
`player.cpp`:	your two player ADT implementations
`game.cpp`:	your game driver

After you have written these files, you can type the following command in the terminal to compile the program:

```
g++ -Wall -std=c++11 –o game game.cpp quarto.cpp exceptions.cpp
piece.cpp pool.cpp square.cpp board.cpp player.cpp
```

This will generate a program called `game` in your working directory. In order to guarantee that the TAs compile your program successfully, you should name you source code files exactly like how they are specified above. For this project, the penalty for code that does not compile will be **severe**, regardless of the reason.

VI. Testing

For this project, you should write individual, focused test cases for all the ADT implementations. For these ADTs, determine the behaviors required of the implementation. Then, for each of these behaviors:

- Determine the **specific** behavior that the implementation must exhibit.

- Write a program that, when linked against the implementation of the ADT, tests for the presence/absence of that behavior.

For example, if you identify two behaviors in the Piece implementation, you would have two files, each testing one behavior. You can name them as follows:

```
piece.case.1.cpp
piece.case.2.cpp
```

Your test cases for this project are considered "acceptance tests". The tests for your Pool/Board ADT (each of which includes a `main()` function) should be linked against a correct `piece.cpp` and a possibly incorrect `pool.cpp`/`board.cpp` when you compile your program. The tests for your Player ADT (each of which includes a `main()` function) should be linked against a correct `piece.cpp`, a correct `pool.cpp`, and a possibly incorrect `player.cpp` when you compile your program.

Your test case must decide, based on the results from calls to Pool/Board/Player methods, whether the Pool/Board/Player ADT is correct or incorrect. If your case believes the Pool/Board/Player to be correct, it should return 0 from `main()`. If your case believes the Board/Player to be incorrect, it should return any value other than zero (the value -1 is commonly used to denote failure). Do not compare the output of your test cases against correct/incorrect implementations. Instead, look at the return value of your program when it is run in Linux to see if you return the right value based upon whether your test finds an error in the implementation of the ADT you are testing.

In Linux you can check the return value of your program by typing

```
echo $?
```

immediately after running your program. You also may find it helpful to add error messages to your output.

Here is an example of code that tests a hypothetical "integer add" function (declared in `addInts.h`) with an "expected" test case:

```
// Tests the addInts function
#include "addInts.h"
int main()
{
    int x = 3;
    int y = 4;
```

```
    int answer = 7;
    int candidate = addInts(x, y);
    if (candidate == answer) {
        return 0;
    } else {
        return -1;
    }
}
```

You should write a collection of Pool/Board/Player implementations with different, specific bugs, and make tests to identify the incorrect code.

VII. Submitting and Due Date

You should submit six source code files `piece.cpp`, `pool.cpp`, `square.cpp`, `board.cpp`, `player.cpp`, and `game.cpp` via the online judgment system. The due time is 11:59 pm on July 19, 2019.

VIII. Grading

Your program will be graded along three criteria:

1. Functional Correctness
2. Implementation Constraints
3. General Style

Functional Correctness is determined by running a variety of test cases against your program, checking against our reference solution. We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. For example, significant code duplication will lead to General Style deductions.