This Exercise is optional and aims to bind together knowledge and practices about C++.

Chess game

This exercise to implement chess game which can be played between users or user and computer. (to understand the rules, go to wiki: https://en.wikipedia.org/wiki/Chess)
The board use 8X8 interlaced black and white strips. The rows are numbered from 1 to 8 and columns from a to h. This gives us our first class which define position (column, row). First, we identify the set of classes that we will need to implement game.

1. Board class: It will maintain the pieces (King, Queen, Rook, Bishop, Knight and Pawn) on the board.
2. Piece class: This class is the base class for all the pieces.
3. Move class: This class will record the move of piece to a position.
4. Player class: This is the base class for Human player as well as Computer.
5. Position class: This class will define the position on board.

```
class Position {
public:
  Position(int column, int row) ... { ... }
  Position(const Position& p) ... { ... }
  bool operator==(const Position& p) const { ... }
  inline bool operator!= (const Position& p) const
     { return !operator == (p); }
  inline int column (void) const { return c; }
  inline int row(vooid) const {return r;}
  Position left(void) const {…}
  Position right(void) const {…}
  Position up(void) const {…}
  Position down(void) const {…}
  static Position null;
Private:
  Position (void): _column(-1), _row(-1) {}
  int _column;
  int _row;
};
Position Position::null; //Forbid the move outside board
```

1. Complete the class and compile it.

```
class Piece; // pre-decleration (to break declaration cycle)
class Board {
private:
  Board(void) {
     for(int i = 0; i < 8; i++)
       for(int j = 0; j < 8; j++)
          squares[i][j] = NULL;
  }
  Piece *at(Position p) { ... }
  void move(Piece *p, Position p) { ... }
  void remove(Piece *p) { ... }
  void display(void) { ... }
private:
   Piece *squares[8][8];
};
```

2. Complete the class and check if board is created empty or not.

Notice that the squares of the chess board are represented as two-dimensions array of 8 x 8, squares. If there is a piece in the square, it contains a pointer to the piece, else a null pointer, NULL, is stored. The method display() prints the board to standard output but cannot be defined before the declaration of Piece class.

```
class Piece {
public:
  static const int white = 0, black = 1;
  Piece(int color, Position pos): _color(color), _pos(pos) { }
  inline int color(void) const { return _color; }

  Position pos(void) const { return p; }
  void put(Board& board, Position p) { ... }
  void move(Board& board, Position pos) { ... }
  void remove(Board& board) { ... }

  virtual string name(void) = 0;
  virtual string display(void) = 0;
  virtual bool isKing(void) = 0;
  virtual void moves(Board& board, set& poss) = 0;

protected:
  void add(set& poss, Position p) {
    if (p != Position::null) poss.insert(p); }
```

```
private:
  int _color;
  Position _pos;
};
```

3. Complete the class.

The name() gives only a human readable name for the piece.
Display() is used to display the piece by the Board class and should return a character representing the piece.
isKind() returns true of the current piece is king.
moves() allows getting the possible moves for the current piece. They are computed using the passed Board object and stored in the given set (notice it is passed by non-constant reference)

4. Write display() function for board.

```
void Board::display(void) {
…..
}
```

To perform the display, you can use the special strings passed in the useful.h:

- WHITE_SQUARE – set color of white square,
- BLACK_SQUARE – set the color of black square,
- WHITE_PIECE – set color of white piece,
- BLACK_PIECE – set color of black piece.
- NORMAL – back to normal display.

King Class:

```
class King: public Piece {
public:
  King(int color, Position pos): Piece(color, pos) { }

  virtual string name(void) { return "King"; }
  virtual char display(void) { return "K"; }
  virtual bool isKing(void) { return true; }
  virtual void moves(Board& board, set& poss) {
     add(poss, pos().up().left());
     add(poss, pos().up());
     add(poss, pos().up().right());
     add(poss, pos().left());
     add(poss, pos().right());
     add(poss, pos().down().left());
```

```
      add(poss, pos().down());
      add(poss, pos().down().right()); }

private:
   void add(Board& board, set& poss, Position p) {
      if(board.at(p) == NULL || board.at(p)->color != color())
          add(poss, p);
   }
};
```

5. Implement other pieces: Pawn, Rook, Bishop, Queen and Knight.
6. Implement main() function to build Board and display it.

Move class:

```
class Move {

public:
  Move(Piece *piece, const Position& pos): _piece(piece), _pos(pos) { }
  inline Piece *piece(void) const { return _piece; }
  inline const Position& pos(void) const { return _pos; }

private:
  Piece *_piece;
  Position _pos;
};
```

Player class:

```
class Player {

public:

  Player(int color): _color(color) { }
  inline int color(void) const { return _color; }
  inline King *king(void) const { return _king; }
  virtual Move nextMove(Board& board) = 0; // choose move object
  void add(Piece *piece) { ... } // add piece at creation time
  void remove(Piece *piece) { ... } // remove piece when captured
  const vector pieces(void) const { return _pieces; }

private:
  int _color;
```

```
  vector _pieces;
  King *_king;
};
```

7. Complete the class

```
class HumanPlayer:
public Player {

public:
  Player(int color): _color(color) { }
  virtual Move nextMove (Game& game, vector& moves) { ... }
};
```

nextMove ask user to enter move and check whether its legal or not according to initial and final position.

8. Complete the missing part

Finally, the class GAME which takes two instances of Players, create board and pieces and then perform game till end.
It must contain a function run() or launcher to launch the game, that is, a loop that display the current board, tests if the current player has lost (checkmate) else ask for the next move and change the board state. Then it exchanges the current player (_cur) with his opponent (_opp) and do the same. For this, it uses the method underAttack() (to develop) to test if the king of the current player is under attack and computeMoves() to compute the list of possible moves (based on moves of all pieces of the current player).

9. Implement Class Game and write main()

10. Launch the game and play

If you want, you can implement **ComputerPlayer Class**.