70083 13

C++ 3

Submitters

sf23	Shihan Fu

```
Chess Exercise Output: Summary for sf23 of v5
---------------------------------------------

  Execution Comparison:
    Test for invalid inputs:          0 / 1
    Test for static board state:      1 / 1
    Test pawn moves:                  0 / 1
    Test rook moves:                  1 / 1
    Test knight moves:                1 / 1
    Test bishop moves:                1 / 1
    Test queen moves:                 0 / 1
    Test king moves:                  1 / 1
    Test match provided to students:  1 / 1
    Unseen test match:                0 / 1

Git Repo: git@gitlab.doc.ic.ac.uk:lab2324_autumn/msc_lab3_sf23.git
Commit ID: b27d8
```

61/100

Good object oriented design of the board and chess pieces, however there are some major issues with the functionality and design of the code. The main source of the issues is your ‘status’ approach, which seems not to be working properly (see results of failed tests). I am not completely sure what has gone wrong here, and it is very difficult to read and understand the update status function.

A better approach would have been to implement a series of checks to the board where necessary, rather than trying to keep a running status log (which is also somewhat inefficient). E.g. you should actively scan the board and check for potential attacks on the king every time you test for the game being in check. This would have been a more explicit approach and would have been easier to spot errors in

```
Detailed Output for test: Test for invalid inputs
-------------------------------------------------

Test for invalid inputs

  Compiled OK

  Compilation Standard Output:

g++ -Wall -g -c ChessBoard.cpp
g++ -Wall -g -c ChessPiece.cpp
g++ -Wall -g -c Color.cpp
g++ -Wall -g -c ChessMain.cpp
g++ -Wall -g ChessBoard.o ChessPiece.o ChessMain.o -o chess

  Compilation Standard Error:

ChessBoard.cpp: In destructor ???ChessBoard::~ChessBoard()???:
ChessBoard.cpp:29:13: warning: deleting object of abstract class type ???ChessPiece??? which has non-virtual destructor will cause undefined behavior [-Wdelete-non-virtual-dtor]
   29 |             delete cps[i][j];
      |             ^~~~~~~~~~~~~~~~
ChessBoard.cpp: In member function ???void ChessBoard::loadState(std::string)???:
ChessBoard.cpp:627:22: warning: comparison of integer expressions of different signedness: ???int??? and ???std::__cxx11::basic_string<char>::size_type??? {aka ???long unsigned ⤦
int???} [-Wsign-compare]
  627 |         for(int j=0;j<tokens[i].length();j++){
      |                     ~^~~~~~~~~~~~~~~~~~~
ChessBoard.cpp: In member function ???int ChessBoard::castling_move(ChessPiece*, std::string, std::string, ChessPiece* (*)[8], int* (*)[8])???:
ChessBoard.cpp:692:30: warning: comparison of integer expressions of different signedness: ???std::__cxx11::basic_string<char>::size_type??? {aka ???long unsigned int???} and ⤦
???int??? [-Wsign-compare]
  692 |     if(castling.find("piece")==-1){
      |        ~~~~~~~~~~~~~~~~~~~~~~^~~~
ChessBoard.cpp: In member function ???int ChessBoard::submitMoveWithoutComments(std::string, std::string)???:
ChessBoard.cpp:564:1: warning: control reaches end of non-virtual function [-Wreturn-type]
  564 | }
      | ^                                                                                           ✗
ChessBoard.cpp: In member function ???int ChessBoard::submitMove(std::string, std::string)???:
ChessBoard.cpp:616:1: warning: control reaches end of non-void function [-Wreturn-type]
  616 | }
      | ^

  Test failed because Unexpected Exit Code
    Model Answer exit code was 0, Student's exit code was

  Model Output (Left) vs Student's Output (Right):

===========================                                                                 ===========================
Testing the Chess Engine                                                                    Testing the Chess Engine
===========================                                                                 ===========================

loadState("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq"): A new board state is lo    loadState("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq"): A new board state is ⤦
lo

Invalid square references                                                                   Invalid square references
cb.submitMove("", "D5"): Invalid square reference!                                        | cb.submitMove("", "D5"): invalid move input ✓
cb.submitMove("E2", ""): Invalid square reference!                                        | cb.submitMove("E2", ""): invalid move input ✓
cb.submitMove("e2", "e4"): Invalid square reference!                                      | cb.submitMove("e2", "e4"): It is not Black's turn to move! ✗
cb.submitMove("B2", "Z89"): Invalid square reference!                                     | cb.submitMove("B2", "Z89"): invalid move input ✓
cb.submitMove("G2", "Hello"): Invalid square reference!                                   | cb.submitMove("G2", "Hello"): invalid move input ✓
cb.submitMove("W0", "D1"): Invalid square reference!                                      | cb.submitMove("W0", "D1"): It is not Black's turn to move! ✗
cb.submitMove("World", "C3"): Invalid square reference!                                   | cb.submitMove("World", "C3"): invalid move input ✓

```

```
Wrong player tries to move                                            Wrong player tries to move
cb.submitMove("D7", "D6"): It is not Black's turn to move!            cb.submitMove("D7", "D6"): It is not Black's turn to move!

No piece at source square                                             No piece at source square
cb.submitMove("D4", "H6"): There is no piece at position D4!          cb.submitMove("D4", "H6"): There is no piece at position D4!

White tries to capture one of its own pieces                          White tries to capture one of its own pieces
cb.submitMove("D1", "C1"): White's Queen cannot move to C1!           cb.submitMove("D1", "C1"): White's Queen cannot move to C1!

Change player to Black:                                               Change player to Black:
loadState("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR b KQkq"): A new board state is lo   loadState("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR b KQkq"): A new board state is
lo

Black tries to capture one of its own pieces                          Black tries to capture one of its own pieces
cb.submitMove("F8", "E7"): Black's Bishop cannot move to E7!          cb.submitMove("F8", "E7"): Black's Bishop cannot move to E7!

loadState("2bqkbnr/8/8/8/8/8/8/RNBQKBNR w KQkq"): A new board state is loaded!   loadState("2bqkbnr/8/8/8/8/8/8/RNBQKBNR w KQkq"): A new board state is loaded!

Try to move off board                                                 Try to move off board
cb.submitMove("A1", "A9"): Invalid square reference!               <
cb.submitMove("F1", "I4"): Invalid square reference!               <
                                                                   <

  Model Answer (Left) vs Student's Error Ouput (Right):

                                                                   >  /usr/bin/timeout: the monitored command dumped core


Detailed Output for test: Test for static board state
-----------------------------------------------------

Test for static board state

  Compiled OK

  Compilation Standard Output:

g++ -Wall -g -c ChessMain.cpp
g++ -Wall -g ChessBoard.o ChessPiece.o ChessMain.o -o chess

  Test Passed




Detailed Output for test: Test pawn moves
-----------------------------------------

Test pawn moves

  Compiled OK

  Compilation Standard Output:

g++ -Wall -g -c ChessMain.cpp
g++ -Wall -g ChessBoard.o ChessPiece.o ChessMain.o -o chess

  Test failed because Output differs

  Model Output (Left) vs Student's Output (Right):

```

-3

```
===========================                                                                 ===========================
Testing the Pawn Piece                                                                      Testing the Pawn Piece
===========================                                                                 ===========================

loadState("rnbqkbnr/8/8/8/8/8/8/RNBQKBNR w KQkq"): A new board state is loaded!             loadState("rnbqkbnr/8/8/8/8/8/8/RNBQKBNR w KQkq"): A new board state is loaded!

Invalid Moves for Pawns                                                                     Invalid Moves for Pawns
cb.submitMove("A2", "A5"): White’s Pawn cannot move to A5!                                  cb.submitMove("A2", "A5"): White’s Pawn cannot move to A5!
cb.submitMove("C2", "E6"): White’s Pawn cannot move to E6!                                  cb.submitMove("C2", "E6"): White’s Pawn cannot move to E6!

Valid Moves for Pawns                                                                       Valid Moves for Pawns
cb.submitMove("F2", "F3"): White’s Pawn moves from F2 to F3                                 cb.submitMove("F2", "F3"): White’s Pawn moves from F2 to F3
cb.submitMove("B7", "B6"): Black’s Pawn moves from B7 to B6                                 cb.submitMove("B7", "B6"): Black’s Pawn moves from B7 to B6

Invalid Move  (White)                                                                       Invalid Move  (White)
cb.submitMove("F3", "F5"): White’s Pawn cannot move to F5!                                | cb.submitMove("F3", "F5"): White’s Pawn moves from F3 to F5
Still White’s Turn; Valid Move                                                              Still White’s Turn; Valid Move
cb.submitMove("F3", "F4"): White’s Pawn moves from F3 to F4                               | cb.submitMove("F3", "F4"): There is no piece at position F3! ✗

Invalid Moves (Black)                                                                       Invalid Moves (Black)
cb.submitMove("B6", "B4"): Black’s Pawn cannot move to B4!                                | cb.submitMove("B6", "B4"): Black’s Pawn moves from B6 to B4 ✗
cb.submitMove("D7", "A4"): Black’s Pawn cannot move to A4!                                | cb.submitMove("D7", "A4"): It is not Black’s turn to move! ✗


Detailed Output for test: Test rook moves
-----------------------------------------

Test rook moves

  Compiled OK

  Compilation Standard Output:

g++ -Wall -g -c ChessMain.cpp
g++ -Wall -g ChessBoard.o ChessPiece.o ChessMain.o -o chess

  Test Passed




Detailed Output for test: Test knight moves
-------------------------------------------

Test knight moves

  Compiled OK

  Compilation Standard Output:

g++ -Wall -g -c ChessMain.cpp
g++ -Wall -g ChessBoard.o ChessPiece.o ChessMain.o -o chess

  Test Passed




Detailed Output for test: Test bishop moves
-------------------------------------------

```

```
Test bishop moves

  Compiled OK

  Compilation Standard Output:

g++ -Wall -g -c ChessMain.cpp
g++ -Wall -g ChessBoard.o ChessPiece.o ChessMain.o -o chess

  Test Passed




Detailed Output for test: Test queen moves
------------------------------------------

Test queen moves

  Compiled OK

  Compilation Standard Output:

g++ -Wall -g -c ChessMain.cpp
g++ -Wall -g ChessBoard.o ChessPiece.o ChessMain.o -o chess

  Test failed because Output differs

  Model Output (Left) vs Student's Output (Right):

===========================                                                                 ===========================
Testing the Queen Piece                                                                     Testing the Queen Piece
===========================                                                                 ===========================

loadState("rnbqkbnr/8/8/3p4/8/8/8/RNBQKBNR w KQkq"): A new board state is loaded!           loadState("rnbqkbnr/8/8/3p4/8/8/8/RNBQKBNR w KQkq"): A new board state is loaded!

Testing invalid moves for White's Queen                                                     Testing invalid moves for White's Queen
cb.submitMove("D1", "E3"): White's Queen cannot move to E3!                                 cb.submitMove("D1", "E3"): White's Queen cannot move to E3!
cb.submitMove("D1", "D6"): White's Queen cannot move to D6!                               | cb.submitMove("D1", "D6"): White's Queen moves from D1 to D6

Change player to Black:                                                                     Change player to Black:
loadState("rnbqkbnr/2p5/8/8/8/8/8/RNBQKBNR b KQkq"): A new board state is loaded!           loadState("rnbqkbnr/2p5/8/8/8/8/8/RNBQKBNR b KQkq"): A new board state is loaded!

Testing invalid moves for Black's Queen                                                     Testing invalid moves for Black's Queen
cb.submitMove("D8", "E6"): Black's Queen cannot move to E6!                                 cb.submitMove("D8", "E6"): Black's Queen cannot move to E6!
cb.submitMove("D8", "A5"): Black's Queen cannot move to A5!                               | cb.submitMove("D8", "A5"): Black's Queen moves from D8 to A5
                                                                                          > White is in check


Detailed Output for test: Test king moves
-----------------------------------------

Test king moves

  Compiled OK

  Compilation Standard Output:

g++ -Wall -g -c ChessMain.cpp
g++ -Wall -g ChessBoard.o ChessPiece.o ChessMain.o -o chess

```

```
  Test Passed




Detailed Output for test: Test match provided to students
---------------------------------------------------------

Test match provided to students

  Compiled OK

  Compilation Standard Output:

g++ -Wall -g -c ChessMain.cpp
g++ -Wall -g ChessBoard.o ChessPiece.o ChessMain.o -o chess

  Test Passed




Detailed Output for test: Unseen test match
-------------------------------------------

Unseen test match

  Compiled OK

  Compilation Standard Output:

g++ -Wall -g -c ChessMain.cpp
g++ -Wall -g ChessBoard.o ChessPiece.o ChessMain.o -o chess

  Test failed because Output differs

  Model Output (Left) vs Student’s Output (Right):

===========================                                                                 ===========================
Alekhine vs. Bruce (1938)                                                                   Alekhine vs. Bruce (1938)
===========================                                                                 ===========================

A new board state is loaded!                                                                A new board state is loaded!

White’s Pawn moves from E2 to E4                                                            White’s Pawn moves from E2 to E4
Black’s Pawn moves from C7 to C6                                                            Black’s Pawn moves from C7 to C6

White’s Knight moves from B1 to C3                                                          White’s Knight moves from B1 to C3
Black’s Pawn moves from D7 to D5                                                            Black’s Pawn moves from D7 to D5

White’s Knight moves from G1 to F3                                                          White’s Knight moves from G1 to F3
Black’s Pawn moves from D5 to E4 taking White’s Pawn                                        Black’s Pawn moves from D5 to E4 taking White’s Pawn

White’s Knight moves from C3 to E4 taking Black’s Pawn                                      White’s Knight moves from C3 to E4 taking Black’s Pawn
Black’s Bishop moves from C8 to F5                                                          Black’s Bishop moves from C8 to F5

White’s Knight moves from E4 to G3                                                          White’s Knight moves from E4 to G3
Black’s Bishop moves from F5 to G6                                                          Black’s Bishop moves from F5 to G6

White’s Pawn moves from H2 to H4                                                            White’s Pawn moves from H2 to H4
Black’s Pawn moves from H7 to H6                                                            Black’s Pawn moves from H7 to H6
```

```
  302:
  303: White’s Knight moves from F3 to E5                                                      White’s Knight moves from F3 to E5
  304: Black’s Bishop moves from G6 to H7                                                      Black’s Bishop moves from G6 to H7
  305:
  306: White’s Queen moves from D1 to H5                                                       White’s Queen moves from D1 to H5
  307: Black’s Pawn moves from G7 to G6                                                        Black’s Pawn moves from G7 to G6
  308:
  309: White’s Bishop moves from F1 to C4                                                      White’s Bishop moves from F1 to C4
  310: Black’s Pawn moves from E7 to E6                                                        Black’s Pawn moves from E7 to E6
  311:
  312: White’s Queen moves from H5 to E2                                                       White’s Queen moves from H5 to E2
  313: Black’s Knight moves from G8 to F6                                                      Black’s Knight moves from G8 to F6
  314:
  315: White’s Knight moves from E5 to F7 taking Black’s Pawn                                  White’s Knight moves from E5 to F7 taking Black’s Pawn
  316: Black’s King moves from E8 to F7 taking White’s Knight                                | Black’s King cannot move to F7!  ✗
  317:
  318: White’s Queen moves from E2 to E6 taking Black’s Pawn                                 | It is not White’s turn to move!  ✗
  319: Black is in check                                                                     | It is not White’s turn to move!  ✗
  320: Black’s King moves from F7 to G7                                                      <
  321:
  322: White’s Queen moves from E6 to F7                                                     | Black’s Pawn cannot move to F7!  ✗
  323: Black is in checkmate                                                                 <
  324:
  325: Trying to make a move after the game has ended                                          Trying to make a move after the game has ended
  326: The game is over!                                                                     | Black’s Knight cannot move to E8!
```

```
/*
 * @Author: shihan
 * @Date: 2023-11-22 21:38:52
 * @version: 1.0
 * @description: definition of the ChessBoard
 */
#ifndef CHESSBOARD_H
#define CHESSBOARD_H

#include <iostream>
#include "ChessPiece.h"
#include <string>
#include <vector>
#include "Color.h"

using namespace std;

// defined string split function
vector<string> splitString(const string& input, char delimiter);

class ChessBoard{
    private:
        // each piece on the chess board
        ChessPiece* cps[8][8];
        // the "being attacked" status of each square
        //2: under attack by Black,
        //-1: under attack by White,
        //1: under attack by both Black and White,
        //0: not under attack
        int* status[8][8];
        // whose turn to move the next piece
        char turn;
        // castling status
        string castling;
        // the last "eaten" chess piece
        ChessPiece* temp;

    public:
        // initial constructor
        ChessBoard();
        // deconstructor
        ~ChessBoard();

        // load the state of the chess board
        void loadState(string FEN);
        // update the status of each square
        void updateStatus();

        // the user submit move
        int submitMove(string from, string to);
        // the background logic try to move
        int submitMoveWithoutComments(string from, string to);
        // redo the last move
        void cancelMove(string origin_from, string origin_to);

        // castling move
        int castling_move(ChessPiece* piece, string from, string to, ChessPiece* cb[8][8], int* status[8][8]);

        // whether there is a "in check" status, does not check checkmate
        bool isCheck();
        // whether the check is a checkmate
        bool isCheckmate(int x, int y);
        int checkState();
        // whether there is a "in check" status, also check checkmate
        bool isStatusCheck();
```

Would be better to use an enum to store this attack info instead of int -1

```
        // try every move to see if the check can be removed
        bool tryRemoveCheck(Color color, string FEN);

        // print the status of each square
        void printStatus();
        // print each piece of the chess board
        void printBoard();
        // transfer the chess board to the FEN format
        string transferFEN();
        // transfer the loaction to FEN format
        string getLocation(int x, int y);


};
#endif
```

Good use of Color enum

```cpp
/*
 * @Author: shihan
 * @Date: 2023-12-02 18:30:46
 * @version: 1.0
 * @description: definition of the ChessPiece
 */
#ifndef CHESSPIECE_H
#define CHESSPIECE_H

#include <iostream>
#include <string>
#include "Color.h"

using namespace std;

class ChessPiece{
    protected:
        Color color;   // the color of the piece
        char symbol;    // the symbol of the piece
        bool first_move;    // whether it is the first move of a piece


    public:
        ChessPiece();   // initial constructor
        ~ChessPiece();  // deconstructor
        ChessPiece(Color _color);  // constructor qith color

        // virtual function of move
        virtual int move(int *from_index, int *to_index, ChessPiece* cb[8][8], int* status[8][8]);
        // virtual function of get the tyoe of a chess piece
        virtual string getType() const = 0;

        // setter and getter fucntion of ChessPiece
        void setSymbol(char _symbol);
        void setColor(Color _color);
        void setFirstMove(bool _firstMove);
        Color getColor();
        char getSymbol();
        bool getFirstMove();

        // define the cout of each chess piece
        friend ostream &operator<<(ostream &out, const ChessPiece &object) {
            out << object.symbol;
            return out;
        }

};

class King: public ChessPiece{
    private:
        int move(int *from_index, int *to_index, ChessPiece* cb[8][8], int* status[8][8]) override;
    public:
        King(Color _color);
        string getType() const override;
};

class Rook: public ChessPiece{
    private:
        int move(int *from_index, int *to_index, ChessPiece* cb[8][8], int* status[8][8]) override;
    public:
        Rook(Color _color);
        string getType() const override;
};
```

```cpp

class Bishop: public ChessPiece{
    private:
        int move(int *from_index, int *to_index, ChessPiece* cb[8][8], int* status[8][8]) override;
    public:
        Bishop(Color _color);
        string getType() const override;
};

class Queen: public ChessPiece{
    private:
        int move(int *from_index, int *to_index, ChessPiece* cb[8][8], int* status[8][8]) override;
    public:
        Queen(Color _color);
        string getType() const override;
};

class Knight: public ChessPiece{
    private:
        int move(int *from_index, int *to_index, ChessPiece* cb[8][8], int* status[8][8]) override;

    public:
        Knight(Color _color);
        string getType() const override;
};

class Pawn: public ChessPiece{
    private:
        int move(int *from_index, int *to_index, ChessPiece* cb[8][8], int* status[8][8]) override;

    public:
        Pawn(Color _color);
        string getType() const override;
};
#endif
```

```
#ifndef COLORS_H
#define COLORS_H
#include <string>
using namespace std;
// Declare the enum in the header file
enum Color {
    Black,
    White
};

string colorToString(Color c);

#endif
```

It is completely unclear what this function is attempting to do, and in any case it looks very duplicative and inefficient. Try to split your functions out in to sub functions with meaningful names rather than having a giant set of nested if statements -5

Having now read through the rest of your code, I assume that this function is updating the status of every square on the board after every move. This is quite inefficient in terms of time and memory. It is time inefficient because you are updating the whole board rather than just checking the necessary pieces at each turn, and it is memory inefficient because you are storing and passing around two chessboards instead of one. It also appears to have caused incorrect functionality (see the test results above), but it is very difficult to know where something has gone wrong due to the structure -10

```
/*
 * @Author: shihan
 * @Date: 2023-11-22 21:40:28
 * @version: 1.0
 * @description: implementation of ChessBoard
 */
#include "ChessBoard.h"
#include "ChessPiece.h"
#include <iostream>
#include <string>
#include <vector>
#include "Color.h"

using namespace std;

// initial constructor
ChessBoard::ChessBoard(){
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 8; ++j) {
            cps[i][j] = nullptr;
        }
    }
}

// deconstructor
ChessBoard::~ChessBoard(){
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 8; ++j) {
            delete cps[i][j];
        }
    }
}

// update the status of each square
void ChessBoard::updateStatus(){
    // Initialize the array with some values (for demonstration purposes)
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 8; ++j) {
            status[i][j] = new int(0);
        }
    }
    //update the status
    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            if(cps[i][j]!=nullptr){
                if(cps[i][j]->getSymbol()=='p'){
                    // for Black's Pawn
                    // the next one to pawn is being attacked by Black
                    if(i+1<8){
                        if(*status[i+1][j]==-1 ||*status[i+1][j]==0){*status[i+1][j]+=2;}
                    }
                    // if the next squae is empty, the second next is also attcked
                    if( i+2 <8 && cps[i+1][j]==nullptr){
                        if(*status[i+2][j]==-1 ||*status[i+2][j]==0){*status[i+2][j]+=2;}
                    }

                }
                if(cps[i][j]->getSymbol()=='k'){
                    // for Black's king
                    // every square next to it is attacked
                    if(i-1>=0){ // each left square
                        if(*status[i-1][j]==-1 || *status[i-1][j] ==0){*status[i-1][j]+=2;}
```

Your code lines are too wide. In general, code should not be more than 80 characters wide (industry standard). -3 for readability

```
                    }
                    if(i+1<8){  // each right square
                        if(*status[i+1][j]==-1 ||*status[i+1][j]==0){*status[i+1][j]+=2;}
                    }
                    if(j-1>=0){ // each upper square
                        if(i-1>=0){
                            if(*status[i-1][j-1]==-1 || *status[i-1][j-1]==0){*status[i-1][j-1]+=2;}
                        }
                        if(i+1<8){
                            if(*status[i+1][j-1]==-1 || *status[i+1][j-1]==0){*status[i+1][j-1]+=2;}
                        }
                        if(*status[i][j-1]==-1||*status[i][j-1]==0){*status[i][j-1]+=2;}
                    }
                    if(j+1<8){  // each lower square
                        if(i-1>=0){
                            if(*status[i-1][j+1]==-1 || *status[i-1][j+1]==0){*status[i-1][j+1]+=2;}
                        }
                        if(i+1<8){
                            if(*status[i+1][j+1]==-1 || *status[i+1][j+1]==0){*status[i+1][j+1]+=2;}
                        }
                        if(*status[i][j+1]==-1||*status[i][j+1]==0){*status[i][j+1]+=2;}

                    }

                }
                if(cps[i][j]->getSymbol()=='r'||cps[i][j]->getSymbol()=='q'){
                    // for black's Rook and Queen
                    int ii=i-1;
                    while(ii>=0 && cps[ii][j]==nullptr){     // for each left square
                        if(*status[ii][j]==-1||*status[ii][j]==0){*status[ii][j]+=2;}
                        ii--;
                    }
                    if(ii>=0){if(*status[ii][j]==-1||*status[ii][j]==0){*status[ii][j]+=2;}}
                    ii=i+1;
                    while(ii<8 && cps[ii][j]==nullptr){      // for each right square
                        if(*status[ii][j]==-1||*status[ii][j]==0){*status[ii][j]+=2;}
                        ii++;
                    }
                    if(ii<8){if(*status[ii][j]==-1||*status[ii][j]==0){*status[ii][j]+=2;}}

                    int jj = j-1;
                    while(jj>=0 && cps[i][jj]==nullptr){     // for each upper square
                        if(*status[i][jj]==-1||*status[i][jj]==0){*status[i][jj]+=2;}
                        jj--;

                    }
                    if(jj>=0){if(*status[i][jj]==-1||*status[i][jj]==0){*status[i][jj]+=2;}}

                    jj=j+1;
                    while(jj<8 && cps[i][jj]==nullptr){      // for each lower square
```

```
                        if(*status[i][jj]==-1||*status[i][jj]==0){*status[i][jj]+=2;}
                        jj++;
                    }
                    if(jj<=8){if(*status[i][jj]==-1||*status[i][jj]==0){*status[i][jj]+=2;}}

                }
                if(cps[i][j]->getSymbol()=='b'||cps[i][j]->getSymbol()=='q'){
                    // for black's Bishop and Queen
                    int ii=i-1, jj = j-1;
                    while(ii>=0 && jj>=0 && cps[ii][jj]==nullptr){
                        if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}
                        ii--;
                        jj--;
                    }
                    if(ii>=0&&jj>=0){if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}}
                    ii=i-1;
                    jj=j+1;
                    while(ii>=0 && jj<8 && cps[ii][jj]==nullptr){
                        if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}
                        ii--;
                        jj++;
                    }
                    if(ii>=0&&jj<8){if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}}
                    ii=i+1;
                    jj=j-1;
                    while(ii<8 && jj>=0 && cps[ii][jj]==nullptr){
                        if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}
                        ii++;
                        jj--;
                    }
                    if(ii<8&&jj>=0){if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}}
                    ii=i+1;
                    jj=j+1;
                    while(ii<8 && jj<8 && cps[ii][jj]==nullptr){
                        if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}
                        ii++;
                        jj++;
                    }
                    if(ii<8&&jj<8){if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}}
                }
                if(cps[i][j]->getSymbol()=='n'){
                    // for Black's Knight
                    int ii=i+1;
                    int jj = j+2;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8 ){
                        if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}
                    }

                    jj = j-2;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8 ){
                        if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}
                    }

```

This looks like it would be super hard to debug or update

```
                    ii=i+2;
                    jj = j+1;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8){
                        if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}
                    }
                    jj = j-1;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8 ){
                        if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}
                    }

                    ii=i-1;
                    jj=j+2;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8 ){
                        if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}
                    }
                    jj=j-2;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8 ){
                        if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}
                    }

                    ii=i-2;
                    jj=j+1;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8 ){
                        if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}
                    }
                    jj=j-1;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8 ){
                        if(*status[ii][jj]==-1||*status[ii][jj]==0){*status[ii][jj]+=2;}
                    }
                }

                // white piece
                if(cps[i][j]->getSymbol()==’P’){
                    // for White’s Pawn
                    // the next one to pawn is being attacked by White
                    int ii = i-1;
                    if(ii>=0){
                        if(*status[ii][j]==2 ||*status[ii][j]==0){*status[ii][j]+=-1;}
                    }
                    // if the next squae is empty, the second next is also attcked
                    if( ii-1>=0 && cps[ii][j]==nullptr){
                        if(*status[ii-1][j]==2 ||*status[ii-1][j]==0){*status[ii-1][j]+=-1;}
                    }

                }
                if(cps[i][j]->getSymbol()==’K’){
                    // for white’s’s king
                    // every square next to it is attacked
                    if(i-1>=0){
                        if(*status[i-1][j]==2 || *status[i-1][j] ==0){*status[i-1][j]+=-1;}
                    }
                    if(i+1<8){
                        if(*status[i+1][j]==2 ||*status[i+1][j]==0){*status[i+1][j]+=-1;}
                    }
                    if(j-1>=0){
```

```
                        if(i-1>=0){
                            if(*status[i-1][j-1]==2 || *status[i-1][j-1] ==0){*status[i-1][j-1]+=-1;}
                        }
                        if(i+1<8){
                            if(*status[i+1][j-1]==2 || *status[i+1][j-1]==0){*status[i+1][j-1]+=-1;}
                        }
                        if(*status[i][j-1]==2||*status[i][j-1]==0){*status[i][j-1]+=-1;}
                    }
                    if(j+1<8){
                        if(i-1>=0){
                            if(*status[i-1][j+1]==2 || *status[i-1][j+1] ==0){*status[i-1][j+1]+=-1;}
                        }
                        if(i+1<8){
                            if(*status[i+1][j+1]==2 || *status[i+1][j+1]==0){*status[i+1][j+1]+=-1;}
                        }
                        if(*status[i][j+1]==2||*status[i][j+1]==0){*status[i][j+1]+=-1;}
                    }
                }

                if(cps[i][j]->getSymbol()==’R’||cps[i][j]->getSymbol()==’Q’){
                    // for white’s Rook and Queen
                    int ii=i-1;

                    while(ii>=0 && cps[ii][j]==nullptr){
                        if(*status[ii][j]==2||*status[ii][j]==0){*status[ii][j]+=-1;}
                        ii--;
                    }
                    if(ii>=0){if(*status[ii][j]==2||*status[ii][j]==0){*status[ii][j]+=-1;}}
                    ii=i+1;
                    while(ii<8 && cps[ii][j]==nullptr){
                        if(*status[ii][j]==2||*status[ii][j]==0){*status[ii][j]+=-1;}
                        ii++;
                    }
                    if(ii<8){if(*status[ii][j]==2||*status[ii][j]==0){*status[ii][j]+=-1;}}

                    int jj = j-1;
                    while(jj>=0 && cps[i][jj]==nullptr){
                        if(*status[i][jj]==2||*status[i][jj]==0){*status[i][jj]+=-1;}
                        jj--;
                    }
                    if(jj>=0){if(*status[i][jj]==2||*status[i][jj]==0){*status[i][jj]+=-1;}}
                    jj=j+1;
                    while(jj<8 && cps[i][jj]==nullptr){
                        if(*status[i][jj]==2||*status[i][jj]==0){*status[i][jj]+=-1;}
                        jj++;
                    }
                    if(jj<8){if(*status[i][jj]==2||*status[i][jj]==0){*status[i][jj]+=-1;}}

                }
                if(cps[i][j]->getSymbol()==’B’||cps[i][j]->getSymbol()==’Q’){
                    // for white’s Bishop and Queen
                    int ii=i-1, jj = j-1;
                    while(ii>=0 && jj>=0 && cps[ii][jj]==nullptr){
                        if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}
                        ii--;
                        jj--;
                    }
                    if(ii>=0&&jj>=0){if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}}
                    ii=i-1;
                    jj=j+1;
                    while(ii>=0 && jj<8 && cps[ii][jj]==nullptr){
                        if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}
                        ii--;
                        jj++;
                    }
                    if(ii>=0&&jj<8){if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}}
                    ii=i+1;
                    jj=j-1;
                    while(ii<8 && jj>=0 && cps[ii][jj]==nullptr){
                        if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}
                        ii++;
                        jj--;
                    }
                    if(ii<8&&jj>=0){if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}}
                    ii=i+1;
                    jj=j+1;
                    while(ii<8 && jj<8 && cps[ii][jj]==nullptr){
                        if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}
                        ii++;
                        jj++;
                    }
                    if(ii<8&&jj<8){if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}}
                }
                if(cps[i][j]->getSymbol()=='N'){
                    // for Black's Knight
                    int ii=i+1;
                    int jj = j+2;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8){
                        if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}
                    }
                    jj = j-2;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8){
                        if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}
                    }

                    ii=i+2;
                    jj = j+1;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8){
                        if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}
                    }
                    jj = j-1;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8){
                        if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}
                    }

                    ii=i-1;
```

```
                    jj=j+2;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8 ){
                        if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}
                    }
                    jj=j-2;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8 ){
                        if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}
                    }

                    ii=i-2;
                    jj=j+1;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8){
                        if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}
                    }
                    jj=j-1;
                    if(ii>=0 && ii<8 && jj>=0 && jj<8){
                        if(*status[ii][jj]==2||*status[ii][jj]==0){*status[ii][jj]+=-1;}
                    }
                }
            }
        }    // end for loop of j
    }    // end for loop of i
}

// redo the last move
void ChessBoard::cancelMove(string origin_from, string origin_to){
    // if only the origin move is success, can we do the cancel move
    // since this is done by force, no need to check the rule
    int from_index[2]={'8'-origin_from[1],origin_from[0]-'A'};
    int to_index[2]={'8'-origin_to[1],origin_to[0]-'A'};
    cps[from_index[0]][from_index[1]]=cps[to_index[0]][to_index[1]];
    cps[to_index[0]][to_index[1]]=temp;
    temp=nullptr;
}

// whether there is a "in check" status, does not check checkmate
bool ChessBoard::isCheck(){
    // only check if the king is in check
    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            if(cps[i][j]!=nullptr){
                if(cps[i][j]->getSymbol()=='k'){
                    if(*status[i][j]==-1 || *status[i][j]==1){
                        return true;
                    }
                }
                if(cps[i][j]->getSymbol()=='K'){
                    if(*status[i][j]==2 || *status[i][j]==1){
                        return true;
                    }
                }
            }
        }    // end for loop of j
    }    // end for loop of i
    return false;
}

// whether there is a "in check" status, also check checkmate
bool ChessBoard::isStatusCheck(){
    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            if(cps[i][j]!=nullptr){
```

You should have a chessboard attribute keeping track of the kings position instead of looping over the board repeatedly to look for it, which is inefficient -2

Also here your use of ints for status instead of enum makes the code very unclear - another developer looking at this would have a hard time understanding it

```
                if(cps[i][j]->getSymbol()=='k'){
                    if(*status[i][j]==-1 || *status[i][j]==1){
                        string str = isCheckmate(i,j)?"mate":"";
                        cout << "Black is in check" << str << endl;
                        return true;
                    }
                }
                if(cps[i][j]->getSymbol()=='K'){
                    if(*status[i][j]==2 || *status[i][j]==1){
                        string str = isCheckmate(i,j)?"mate":"";
                        cout << "White is in check" << str << endl;
                        return true;
                    }
                }
            }
        }   // end for loop pf j
    }   // end for loop for i
    return false;
}

// transfer the loaction to FEN format
string ChessBoard::getLocation(int x, int y){
    string str="  ";
    str[0]=(char)(y+'A');
    str[1]=(char)('8'-x);
    return str;
}

// try every move to see if the check can be removed
bool ChessBoard::tryRemoveCheck(Color color, string FEN){
    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            if(cps[i][j]!=nullptr && cps[i][j]->getColor()==color){
                // try to move to every square
                char file = 'A';
                char rank = '1';
                for(int x=0;x<8;x++){
                    for (int y=0;y<8;y++){
                        string str_to = "  ";
                        str_to[0]=char(file+x);
                        str_to[1]=char(rank+y);
                        string str_from = getLocation(i,j);
                        if(submitMoveWithoutComments(str_from,str_to)==0){
                            if(isCheck()==false){   // after move, we can reduce the check
                                cancelMove(str_from,str_to);
                                return true; // remove the check
                            }
                            else{
                                cancelMove(str_from,str_to);    // cancel the move
                            }
                        }
                    }   // end for loop of y
                }// end for loop of x
            }// end for loop of j
        }// end for loop of i
    }
    return false;   // cannot remove the check
}

// transfer the chess board to the FEN format
string ChessBoard::transferFEN(){
    string str = "";
    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
```

Instead of using strings here you should use some kind of struct for position (e.g. struct pos {int rank; int file;}) or at least separate ints, which would be more readable -2

```
            if(cps[i][j]==nullptr){
                str+=".";
            }
            else{
                str+=cps[i][j]->getSymbol();
            }
        }
        if(i!=7){
            str+="/";
        }

    }
    string str2 = " ";
    str2[0]=turn;
    str+= " "+str2+" "+castling;
    return str;
}

// whether the check is a checkmate
bool ChessBoard::isCheckmate(int x, int y){
    int checkmate[2] = {(cps[x][y]->getSymbol()=='K')?2:-1,1};
    // check every square around the king
    //upper row
    if(x-1>=0){
        if(y-1>=0 && *status[x-1][y-1]!=checkmate[0]&&*status[x-1][y-1]!=checkmate[1]&&cps[x-1][y-1]==nullptr){
            return false;
        }
        if(*status[x-1][y]!=checkmate[0]&&*status[x-1][y]!=checkmate[1]&&cps[x-1][y]==nullptr){
                return false;
        }
        if(y+1<8 && *status[x-1][y+1]!=checkmate[0]&&*status[x-1][y+1]!=checkmate[1]&&cps[x-1][y+1]==nullptr){
            return false;
        }
    }
    // this row
    if(y-1>=0 && *status[x][y-1]!=checkmate[0]&&*status[x][y-1]!=checkmate[1]&&cps[x][y-1]==nullptr){
            return false;
    }
    if(y+1<8 && *status[x][y+1]!=checkmate[0]&&*status[x][y+1]!=checkmate[1]&&cps[x][y+1]==nullptr){
            return false;
    }
    //downer row
    if(x+1<8){
        if(y-1>=0 && *status[x+1][y-1]!=checkmate[0]&&*status[x+1][y-1]!=checkmate[1]&&cps[x+1][y-1]==nullptr){
            return false;
        }
        if(*status[x+1][y]!=checkmate[0]&&*status[x+1][y]!=checkmate[1]&&cps[x+1][y]==nullptr){
                return false;
        }
        if(y+1<8 && *status[x+1][y+1]!=checkmate[0]&&*status[x+1][y+1]!=checkmate[1]&&cps[x+1][y+1]==nullptr){
            return false;
        }
    }
    // whether move a piece can change the status
    if(tryRemoveCheck(cps[x][y]->getColor(),transferFEN())){
        // can remove the check, return false ( is not a checkmate )
        return false;
    }
    // cannot remove the check, return true ( is a checkmate )
    return true;
}

// print the status of each square
void ChessBoard::printStatus(){
    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            cout << *status[i][j] << "\t";
        }
        cout << endl;
    }
}

// the background logic try to move
int ChessBoard::submitMoveWithoutComments(string from, string to){
    // 1. check input valid
    if(from.length()!=2 || to.length()!=2){
        return -1;
    }

    int from_index[2]={'8'-from[1],from[0]-'A'};
    int to_index[2]={'8'-to[1],to[0]-'A'};

    // 2. check if from has a chesspiece
    if(cps[from_index[0]][from_index[1]]==nullptr){
        return -1;
    }

    // 4. check if has piece and with the same side
    if(cps[to_index[0]][to_index[1]]!=nullptr && cps[from_index[0]][from_index[1]]->getColor()==cps[to_index[0]][to_index[1]]->getColor())
{
        return -1;
    }

    // has no chesspiece, check the move logic
    // has chesspiece, both check the move logic and if the chesspiece can be eaten
    if(cps[from_index[0]][from_index[1]]->move(from_index,to_index,cps,status)==-1){
        return -1;
    }

    // if success
    if(cps[to_index[0]][to_index[1]]!=nullptr){
        temp = cps[to_index[0]][to_index[1]];
    }
    else{
        temp = nullptr;
    }

    cps[to_index[0]][to_index[1]]=cps[from_index[0]][from_index[1]];

    cps[from_index[0]][from_index[1]]=nullptr;
    // no need to switch turn
    updateStatus();
}

// the user submit move
int ChessBoard::submitMove(string from, string to){
    // 1. check input valid
```

```
    if(from.length()!=2 || to.length()!=2){
        cout << "invalid move input" << endl;
        return -1;
    }
    int from_index[2]={'8'-from[1],from[0]-'A'};
    int to_index[2]={'8'-to[1],to[0]-'A'};

    // 2. check if from has a chesspiece
    if(cps[from_index[0]][from_index[1]]==nullptr){
        cout << "There is no piece at position " << from <<"!" << endl;
        return -1;
    }
    // 3. check turn
    Color turn_color = (turn=='w')?White:Black;
    if((char)tolower(cps[from_index[0]][from_index[1]]->getColor())!=turn_color){
        string turn_color = turn=='w'?"Black":"White";
        cout <<"It is not "<< turn_color <<"'s turn to move!" << endl;
        return -1;
    }

    string taking="";
    // 4. check if has piece and with the same side
    if(cps[to_index[0]][to_index[1]]!=nullptr && cps[from_index[0]][from_index[1]]->getColor()==cps[to_index[0]][to_index[1]]->getColor())
{
        cout << colorToString(cps[from_index[0]][from_index[1]]->getColor()) << "'s " << cps[from_index[0]][from_index[1]]->getType() << " cannot move to " << to << "!" << endl;
        return -1;
    }

    if(cps[to_index[0]][to_index[1]]!=nullptr){
        taking = " taking "+colorToString(cps[to_index[0]][to_index[1]]->getColor()) + "'s " +cps[to_index[0]][to_index[1]]->getType();
    }

    // has no chesspiece, check the move logic
    // has chesspiece, both check the move logic and if the chesspiece can be eaten
    if(cps[from_index[0]][from_index[1]]->move(from_index,to_index,cps,status)==-1){
        cout << colorToString(cps[from_index[0]][from_index[1]]->getColor()) << "'s " << cps[from_index[0]][from_index[1]]->getType() << " cannot move to " << to << "!" << endl;
        return -1;
    }

    // if success
    cps[to_index[0]][to_index[1]]=cps[from_index[0]][from_index[1]];
    cps[from_index[0]][from_index[1]]=nullptr;

    cout << colorToString(cps[to_index[0]][to_index[1]]->getColor()) << "'s " << cps[to_index[0]][to_index[1]]->getType() << " moves from " << from << " to " << to << taking << endl;
    turn = (turn=='w')?'b':'w'; // switch turn
    updateStatus();
    isStatusCheck();
}

// load the state of the chess board
void ChessBoard::loadState(string FEN){
    // sample: "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq"
    string FEN1 = FEN.substr(0,FEN.find(' '));
    string FEN2 = FEN.substr(FEN.find(' ')+1, FEN.length());
```

There is no check for whether the characters in from/to are valid (i.e. within the board) -3

```
    vector<string> tokens = splitString(FEN1, '/');

    //load the chessboard
    for(int i=0;i<8;i++){
        for(int j=0;j<tokens[i].length();j++){
            // if 8, go to the next line
            if(tokens[i][j]=='8'){
                break;
            }
            else if(tokens[i][j]==' ' || tokens[i][j]=='.'){
                // cout << "leave this space" << endl;
            }
            else{
                Color color = (islower(tokens[i][j])?Black:White);
                switch((char)tolower(tokens[i][j])){
                    case 'p':
                        cps[i][j] = new Pawn(color);
                        cps[i][j]->setSymbol(tokens[i][j]);
                        break;
                    case 'n':
                        cps[i][j] = new Knight(color);
                        cps[i][j]->setSymbol(tokens[i][j]);
                        break;
                    case 'b':
                        cps[i][j] = new Bishop(color);
                        cps[i][j]->setSymbol(tokens[i][j]);
                        break;
                    case 'r':
                        cps[i][j] = new Rook(color);
                        cps[i][j]->setSymbol(tokens[i][j]);
                        break;
                    case 'q':
                        cps[i][j] = new Queen(color);
                        cps[i][j]->setSymbol(tokens[i][j]);
                        break;
                    case 'k':
                        cps[i][j] = new King(color);
                        cps[i][j]->setSymbol(tokens[i][j]);
                        break;
                    default:
                        break;
                }   // end switch
            }   // end if-else
        }   // end for loop of j
    }   // end for loop of i
    cout << "A new board state is loaded!\n";

    //load the turn and castling info
    turn = FEN2[0];
    castling=FEN2.substr(FEN2.find(' ')+1,FEN2.length());

    updateStatus();
}

// print each piece of the chess board
void ChessBoard::printBoard(){
    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            if(cps[i][j]!=nullptr)
                cout << *cps[i][j];
            else{
                cout << " ";
            }
        }
        cout  << endl;
```

```
    }
}

int ChessBoard::castling_move(ChessPiece* piece, string from, string to, ChessPiece* cb[8][8], int* status[8][8]){
    if(castling.find("piece")==-1){
        // invalid castling move
        return -1;
    }
    // if the king/queen or the rook has been moved
    if(!piece->getFirstMove()){
        return -1;
    }

    // the king/queen is in check
    if(from.length()!=2 || to.length()!=2){
        cout << "invalid move input" << endl;
        return -1;
    }
    int from_index[2]={'8'-from[1],from[0]-'A'};
    int to_index[2]={'8'-to[1],to[0]-'A'};
    int attacked[2]={0,0};
    attacked[0]=(piece->getColor()==Black)?2:-1;
    attacked[1]=1;
    if(*status[from_index[0]][from_index[1]]==attacked[0] || *status[from_index[0]][from_index[1]]==attacked[1]){
        return -1;
    }

    // if after castling, the king will be attacked
    if(*status[to_index[0]][to_index[1]]==attacked[0] || *status[to_index[0]][to_index[1]]==attacked[1]){
        return -1;
    }

    // if there is a piece between the king/queen and the rook
    if(*from_index == *to_index){
        int step= (*(from_index+1)< *(to_index+1))?1:-1;
        int y = *(from_index+1) +step;
        while(y<8 && y>=0 && y!=*(to_index+1)){
            if(cb[*from_index][y]!=nullptr){
                // if there is piece, cannot move
                return -1;
            }
            y+=step;
        }
    }

    // castling
    return 0;
}

// defined string split function
vector<string> splitString(const string& input, char delimiter) {
    vector<string> result;
    size_t start = 0;
    size_t end = input.find(delimiter);

    while (end != string::npos) {
        result.push_back(input.substr(start, end - start));
        start = end + 1;
        end = input.find(delimiter, start);
    }

    // Add the last substring after the last delimiter
    result.push_back(input.substr(start));
```

```cpp

    return result;
}

string colorToString(Color c){
    switch (c) {
        case Black:
            return "Black";
        case White:
            return "White";
        default:
            return "UNKNOWN";
    }
}

```

Would have been better to overload the << operator -2

```cpp
#include <iostream>
#include "ChessPiece.h"
#include <string>
#include "Color.h"

/*
 * @Author: shihan
 * @Date: 2023-12-02 18:30:46
 * @version: 1.0
 * @description: implementation of chess piece
 */

using namespace std;

// ChessPiece class
ChessPiece::ChessPiece(){
    symbol = ' ';
    first_move = true;
}

ChessPiece::~ChessPiece(){}

ChessPiece::ChessPiece(Color _color):color(_color){
    first_move = true;
}

int ChessPiece::move(int *from_index, int *to_index, ChessPiece* cb[8][8], int* status[8][8]){
    cout << "general chess piece move\n";
    return 0;
}

void ChessPiece::setSymbol(char _symbol){
    symbol = _symbol;
}

void ChessPiece::setColor(Color _color){
    this->color=_color;
}

void ChessPiece::setFirstMove(bool _firstMove){
    this->first_move = _firstMove;
}

Color ChessPiece::getColor(){
    return color;
}

char ChessPiece::getSymbol(){
    return symbol;
}

bool ChessPiece::getFirstMove(){
    return first_move;
}


// King class
int King::move(int *from_index, int *to_index, ChessPiece* cb[8][8], int* status[8][8]){
    int diff = (*from_index)*10 + (*(from_index+1)) - (*to_index)*10 - (*(to_index+1));
    if(abs(diff)!=1 && abs(diff)!=10){
        // it is not a square
        return -1;
    }
```

It is not good to pass the chessboard raw pointers to the chesspiece. Access should be done via getters that return necessary information - passing raw pointers around is dangerous because it means the chess pieces can edit the board, which we don’t want! -3

```
    else{
        // if the move will make the king be attacked
        int attacked = (symbol=='k')?-1:2;
        if(*status[*to_index][*(to_index+1)]==1||*status[*to_index][*(to_index+1)]==attacked){
            return -1;
        }
        return 0;
    }

}

King::King(Color _color) : ChessPiece(_color) {}

string King::getType() const{
    return "King";
}


// Rook class
int Rook::move(int *from_index, int *to_index,ChessPiece* cb[8][8], int* status[8][8]){
    // 1. check if the move obeys the rule
    if(*from_index!=*to_index && *(from_index+1)!=*(to_index+1)){
        return -1;
    }
    // 2. check if there is pieces between the from and the to
    if(*from_index == *to_index){
        int step= (*(from_index+1)< *(to_index+1))?1:-1;
        int y = *(from_index+1) +step;
        while(y<8 && y>=0 && y!=*(to_index+1)){
            if(cb[*from_index][y]!=nullptr){
                // if there is piece, cannot move
                return -1;
            }
            y+=step;
        }
    }
    if(*(from_index+1) == *(to_index+1)){
        int step= (*(from_index)< *(to_index))?1:-1;
        int x = *(from_index) +step;
        while(x<8 && x>=0 && x!=*(to_index)){
            if(cb[x][*(from_index+1)]!=nullptr){
                return -1;
            }
            x+=step;
        }
    }

    //valid move
    return 0;
}

Rook::Rook(Color _color):ChessPiece(_color){}

string Rook::getType() const{
    return "Rook";
}


// Bishop class
int Bishop::move(int *from_index, int *to_index, ChessPiece* cb[8][8], int* status[8][8]){
    // 1. check if the input obeys the rule
    if(abs(*from_index - *to_index)!=abs(*(from_index+1) - *(to_index+1))){
        return -1;
```

Checking for the presence of pieces between the source and destination is a common process shared by all of the pieces. This logic should be abstracted into the parent piece class, with the individual piece providing information on which squares it will pass during its move. The parent piece class should query the chess board to find out if the squares are unoccupied rather than accessing the raw pointers -5

```
    }
    // 2. check if there is pieces between the from and the to
    int steps = abs(*from_index - *to_index);
    int each_x = (*to_index - *from_index)/steps;
    int each_y = (*(to_index+1) - *(from_index+1))/steps;
    int index_x = *from_index;
    int index_y = *(from_index+1);
    for(int i=0;i<steps-1;i++){
        index_x+=each_x;
        index_y+=each_y;
        if(cb[index_x][index_y]!=nullptr){
            return -1;
        }
    }
    // valid move
    return 0;

}

Bishop::Bishop(Color _color):ChessPiece(_color){}

string Bishop::getType() const {
    return "Bishop";
}


// Queen class
Queen::Queen(Color _color):ChessPiece(_color){}

string Queen::getType() const{
    return "Queen";
}

int Queen::move(int *from_index, int *to_index, ChessPiece* cb[8][8], int* status[8][8]){
    // 1. check if it is a rook's rule or a bishop's rule
    // for a rook:
    if(*from_index == *to_index){
        int step= (*(from_index+1)< *(to_index+1))?1:-1;
        int y = *(from_index+1) +step;
        while(y<8 && y>=0 && y!=*(to_index+1)){
            if(cb[*from_index][y]!=nullptr){
                return -1;
            }
            y+=step;
        }
        return 0;
    }
    else if(*(from_index+1) == *(to_index+1)){
        int step= (*(from_index)< *(to_index))?1:-1;
        int x = *(from_index) +step;
        while(x<8 && x>=0 && x!=*(to_index)){
            if(cb[x][*(from_index+1)]!=nullptr){
                return -1;
            }
            x+=step;
        }
        return 0;
    }
    // for a bishop
    else if(abs(*from_index - *to_index)==abs(*(from_index+1) - *(to_index+1))){
        int steps = abs(*from_index - *to_index);
        int each_x = (*to_index - *from_index)/steps;
        int each_y = (*(to_index+1) - *(from_index+1))/steps;
        int index_x = *from_index;
        int index_y = *(from_index+1);
```

This is clear duplication with the rook and bishop class - should have been abstracted into a function -3

```
        for(int i=0;i<steps−1;i++){
            index_x+=each_x;
            index_y+=each_y;
            if(cb[index_x][index_y]!=nullptr){
                return −1;
            }
        }
        return 0;
    }

    // else, the logic is wrong and no valid move
    return −1;
}


// Knight class
Knight::Knight(Color _color):ChessPiece(_color){}

string Knight::getType() const {
    return "Knight";
}

int Knight::move(int *from_index, int *to_index, ChessPiece* cb[8][8], int* status[8][8]){
    int diff_x = abs(*from_index − *to_index);
    int diff_y = abs(*(from_index+1)− *(to_index+1));
    // check whether the move follows the rule
    if(diff_x + diff_y !=3){
        return −1;
    }
    if(diff_x ==2 && diff_y ==1){
        return 0;
    }
    if(diff_x ==1 && diff_y ==2){
        return 0;
    }
    return −1;
}


// Pawn class
Pawn::Pawn(Color _color):ChessPiece(_color){}

string Pawn::getType() const{
    return "Pawn";
}

int Pawn::move(int *from_index, int *to_index, ChessPiece* cb[8][8], int* status[8][8]){
    // for Black’s Pawn
    if(symbol == ’p’){
        // one step is okay
        if(*(from_index+1)==*(to_index+1) && (*from_index +1 == *to_index) && cb[*to_index][*(to_index+1)]==nullptr ){
            return 0;
        }
        // two steps need to check if overleap, if it is the first move
        if(first_move && *(from_index+1)==*(to_index+1) && (*from_index +2 == *to_index) && cb[*from_index +1][*(from_index+1)]==nullptr && cb[*to_index][*(to_index+1)]==nullptr ){
            first_move = false;
            return 0;
        }
        // diagonally in front of it and have piece
        if((*from_index +1 == *to_index)){
            if(abs(*(from_index+1) − *(to_index+1))==1 && cb[*(to_index)][*(to_index+1)]!=nullptr && cb[*(to_index)][*(to_index+1)]−>getColor()!= cb[*(from_index)][*(from_index+1)]−>getColor()){
                return 0;
            }
        }

    }
```

```

    // for White’s Pawn
    else if(symbol == ’P’){
        // one step is okay
        if(*(from_index+1)==*(to_index+1) && (*from_index −1 == *to_index) && cb[*to_index][*(to_index+1)]==nullptr){
            return 0;
        }
        // two steps need to check if overleap, if it is the first move
        if(first_move && *(from_index+1)==*(to_index+1) && (*from_index −2 == *to_index) && cb[*from_index −1][*(from_index+1)]==nullptr && cb[*to_index][*(to_index+1)]==nullptr){
            first_move = false;
            return 0;
        }
        // diagonally in front of it and have piece
        if((*from_index −1 == *to_index)){
            if(abs(*(from_index+1) − *(to_index+1))==1 && cb[*(to_index)][*(to_index+1)]!=nullptr && cb[*(to_index)][*(to_index+1)]−>getColor()!= cb[*(from_index)][*(from_index+1)]−>getColor()){
                return 0;
            }
        }

    }

    // else, the move is invalid
    return −1;
}

```

```
/*
 * @Author: shihan
 * @Date: 2023-11-22 21:38:19
 * @version: 1.0
 * @description:
 */
#include"ChessBoard.h"

#include<iostream>

using std::cout;

void location();
void testQueen();
void testKnight();
void testPawn();
void testCancelMove();
void testtransferFEN();

int main() {

  // test
  // 1. test location transformation
  // location();
  // 2. test piece move
  // testQueen();
  // testKnight();
  // testPawn();
  // testCancelMove();
  // 3. test helper function
  // testtransferFEN();



  // cout << "========================\n";
  // cout << "Testing the Chess Engine\n";
  // cout << "========================\n\n";

  ChessBoard cb;
  // cb.loadState("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq");
  // cb.loadState("qqqqqqqq/8/8/8/8/8/RNBQKBNR w KQkq");
  cout << '\n';
  // A new board state is loaded!

  // // It is not Blackâ\200\231s turn to move!
  // cb.submitMove("D7", "D6");
  // cout << '\n';

  // // There is no piece at position D4!
  // cb.submitMove("D4", "H6");
  // cout << '\n';

  // // Whiteâ\200\231s Pawn moves from D2 to D4
  // cb.submitMove("D2", "D4");
  // cout << '\n';

  // // cb.submitMove("A8", "B4");
  // // Black's rook cannot move to B4


  // // Blackâ\200\231s Bishop cannot move to B4!
  // cb.submitMove("F8", "B4");
  // cout << '\n';

  // cout << "=========================\n";
  // cout << "Alekhine vs. Vasic (1931)\n";
```

```
  // cout << "=========================\n\n";

  // cb.loadState("rnbqk.nr/ppp..ppp/....p.../...p..../...PP.../..bB..../PPP..PPP/R.BQK.NR w KQkq");
  // cb.loadState("....b.../8/8/8/8/8/8/8 w KQkq");
  //  cb.printBoard();
  // cb.printStatus();


  // // A new board state is loaded!
  cb.loadState("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq");
  cout << '\n';

  cb.submitMove("E2", "E4");
  cb.submitMove("E7", "E6");
  // cb.printBoard();
  cout << '\n';

  cb.submitMove("D2", "D4");
  cb.submitMove("D7", "D5");
  // cb.printBoard();
  cout << '\n';

  cb.submitMove("B1", "C3");
  cb.submitMove("F8", "B4");
  // cb.printBoard();
  cout << '\n';

  cb.submitMove("F1", "D3");
  cb.submitMove("B4", "C3");
  // // Blackâ\200\231s Bishop moves from B4 to C3 taking Whiteâ\200\231s Knight
  // // White is in check

  cout << '\n';



  // Whiteâ\200\231s Pawn moves from B2 to C3 taking Blackâ\200\231s Bishop
  cb.submitMove("B2", "C3");
  cb.submitMove("H7", "H6");
  // cb.printBoard();
  cout << '\n';

  cb.submitMove("C1", "A3");
  cb.submitMove("B8", "D7");
  // cb.printBoard();
  cout << '\n';

  cb.submitMove("D1", "E2");
  // Blackâ\200\231s Pawn moves from D5 to E4 taking Whiteâ\200\231s Pawn
  cb.submitMove("D5", "E4");
  // cb.printBoard();
  cout << '\n';

  // Whiteâ\200\231s Bishop moves from D3 to E4 taking Blackâ\200\231s Pawn
  cb.submitMove("D3", "E4");
  cb.submitMove("G8", "F6");
  // cb.printBoard();
  cout << '\n';

  cb.submitMove("E4", "D3");
  cb.submitMove("B7", "B6");
  cout << '\n';

  cb.submitMove("E2", "E6");
  // Whiteâ\200\231s Queen moves from E2 to E6 taking Blackâ\200\231s Pawn
  // Black is in check
  // cb.printBoard();
  // cb.printStatus();

  //Blackâ\200\231s Pawn moves from F7 to E6 taking Whiteâ\200\231s Queen
  cb.submitMove("F7", "E6");
  cout << ’\n’;

  // Whiteâ\200\231s Bishop moves from D3 to G6
  // Black is in checkmate
  cb.submitMove("D3", "G6");

  // cb.submitMove("E8","E7");
  // cb.printBoard();
  // cb.printStatus();
  cout << ’\n’;

  return 0;
}

void location(){
    int x=3,y=3;
    string str;
    cout << (char)(y+’A’) << endl;
    cout << (char)(’8’-x) << endl;
    str[0]=(char)(y+’A’);
    str[1]=(char)(’8’-x);
    cout << str << endl;
    // string str = (char)(y+’A’)+""+(char)(’8’-x);
    // cout << str << endl;
}

void testQueen(){
    ChessBoard cb;

  // cb.loadState("8/8/8/8/...Q..../8/8/8 w KQkq");
    // cb.submitMove("D4","D3"); // ok
    // cb.submitMove("D4","D5"); // ok
    // cb.submitMove("D4","B4"); // ok
    // cb.submitMove("D4","H4"); // ok
    // cb.submitMove("D4","B6"); // ok
    // cb.submitMove("D4","H8"); // ok
    // cb.submitMove("D4","A1"); // ok
    // cb.submitMove("D4","G1"); // ok


    cb.loadState("8/8/8/.....r../...Q..../8/8/8 w KQkq");
    // cb.submitMove("D4","H4"); // not ok
    // cb.submitMove("D4","E6"); // not ok
    cb.submitMove("D4","A3"); // not ok


    // cb.printStatus();
}

void testKnight(){
    ChessBoard cb;
    cb.loadState("8/8/8/.....r../...N.r../8/8/8 w KQkq");
    // cb.submitMove("D4","F5"); // ok
    // cb.submitMove("D4","E6"); // ok
    // cb.submitMove("D4","B3"); // ok
    // cb.submitMove("D4","F4"); // not ok
    cb.submitMove("D4","F5"); // ok

}
```

```

void testPawn(){
    ChessBoard cb;
    // cb.loadState("8/..p...../8/8/8/8/8/8 b KQkq");
    // cb.submitMove("C7","C6"); // ok
    // cb.submitMove("C7","C5"); // ok
    cb.submitMove("C7","C8"); // not ok

    // cb.loadState("8/..p...../..R...../8/8/8/8/8 b KQkq");
    // cb.submitMove("C7","C5"); // not ok
    // cb.submitMove("C7","C6"); // ok

    cb.loadState("8/..p...../.R....../8/8/8/8/8 b KQkq");
    cb.submitMove("C7","D6"); // not ok
    // cb.submitMove("C7","B6"); // ok

}

void testCancelMove(){
    ChessBoard cb;
    cb.loadState("8/..p...../.R....../8/8/8/8/8 b KQkq");
    cb.submitMove("C7","B6"); // ok
    cb.printBoard();
}

void testtransferFEN(){
    ChessBoard cb;
    cb.loadState(
"......../..p...../.R....../......../......../......../......../........ b KQkq ");
    cout << cb.transferFEN() << endl;
}



```

```
/*
 * @Author: shihan
 * @Date: 2023-12-03 12:31:51
 * @version: 1.0
 * @description:
 */
#include <iostream>
#include <string>
#include <cstdlib> // For abs function (integer)
#include "ChessBoard.h"
using namespace std;

void move(string from, string to){
    cout << from[0]-’A’<< endl;
    int from_index[2]={’8’-from[1],from[0]-’A’};
    cout<< from_index[0]<< from_index[1]<< endl;

}

void int_array_to_int(int *from, int *to){
    int diff = (*from)*10 + (*(from+1)) - (*to)*10 - (*(to+1));
    cout << abs(diff) << endl;
    if(abs(diff)!=1 && abs(diff)!=10){
        cout << "break the king’s rule"<< endl;
    }
    else{
        cout<<"okay for king"<< endl;
    }
    // cout << *to << endl;
    // cout << *(to+1) << endl;
    // cout << (*to)*10 + (*to+1)<< endl;
}

void check_line(int *from, int *to){
    if(abs(*from - *to)==abs(*(from+1) - *(to+1))){
        cout << "ok" << endl;
    }
    else{
        cout << "not ok" << endl;
    }

    int steps = abs(*from - *to);
    int each_x = (*to - *from)/steps;
    int each_y = (*(to+1) - *(from+1))/steps;
    cout << each_x;
    cout << " " << each_y << endl;

    int index_x = *from;
    int index_y = *(from+1);
    for(int i=0;i<steps;i++){
        index_x+=each_x;
        index_y+=each_y;
        cout << index_x << index_y << endl;
    }

}
int main(){
    ChessBoard cb;
  cb.loadState("8/8/......Q./8/8/8/8/8 w KQkq");
    cb.printStatus();

    return 0;
}
```

```
/*
 * @Author: shihan
 * @Date: 2023-12-02 18:46:21
 * @version: 1.0
 * @description:
 */
#include <iostream>
#include <vector>
#include <string>
#include"ChessBoard.h"

using namespace std;


// vector<string> splitString(const string& input, char delimiter) {
//     vector<string> result;
//     size_t start = 0;
//     size_t end = input.find(delimiter);

//     while (end != string::npos) {
//         result.push_back(input.substr(start, end - start));
//         start = end + 1;
//         end = input.find(delimiter, start);
//     }

//     // Add the last substring after the last delimiter
//     result.push_back(input.substr(start));

//     return result;
// }

class Test{
    public:
        Test(){cout << "test\n";}
};

void testChessBoard();
void testCheck();
int main() {

    // testChessBoard();
    testCheck();
    // cout << islower(’8’)<< endl;


    // for (const auto& token : tokens) {
    //     cout << token << endl;
    // }
    // delete cb;
    return 0;
}

void testSubstr(){
    string s="hello world";
    char delimiter = ’ ’;
    string token = s.substr(s.find(delimiter)+1, s.length()); // token is "scott"
    cout << token;
}

void testCheck(){
    ChessBoard cb;
    cb.loadState("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq");
  cout << ’\n’;

  cb.submitMove("E2", "E4");
  cb.submitMove("E7", "E6");
```

```cpp
  cout << '\n';

  cb.submitMove("D2", "D4");
  cb.submitMove("D7", "D5");
  cout << '\n';

  cb.submitMove("B1", "C3");
  cb.submitMove("F8", "B4");
  cout << '\n';

  cb.submitMove("F1", "D3");
  cb.submitMove("B4", "C3");
    cb.printBoard();
}
void testChessBoard(){
    ChessBoard cb;
  cb.loadState("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq");
  // cb.submitMove("D7", "D6");
  // cb.submitMove("D4", "H6");
  cb.submitMove("D2", "D4");
    cb.printBoard();


}
void testTypeid(){
    Test t;
    cout << typeid(t).name()<< endl;;
}
void testSplitString(){
    string input = "Hello,World,This,Is,C++";
    vector<string> tokens = splitString(input, ',');

    cout <<  tokens[0][0] << endl;
    for(int i=0;i<2;i++){
        cout << tokens[i].length() << endl;
        for(int j=0;j<tokens[i].length();j++){
            cout<< tokens[i][j] << endl;
        }
    }
}
```

```cpp
// File: colors.cpp

#include "Color.h"

#include <string>
using namespace std;
// Implementation of colorToString function

string colorToString(Color c){
    switch (c) {
        case Black:
            return "Black";
        case White:
            return "White";
        default:
            return "UNKNOWN";
    }
}
```

```
chess: ChessBoard.o ChessPiece.o Color.o ChessMain.o
	g++ -Wall -g ChessBoard.o ChessPiece.o ChessMain.o -o chess

Color.o : Color.cpp Color.h
	g++ -Wall -g -c Color.cpp

ChessPiece.o: ChessPiece.cpp ChessPiece.h
	g++ -Wall -g -c ChessPiece.cpp

ChessBoard.o: ChessBoard.cpp ChessBoard.h
	g++ -Wall -g -c ChessBoard.cpp

ChessMain.o: ChessMain.cpp ChessBoard.h ChessPiece.h Color.h
	g++ -Wall -g -c ChessMain.cpp

clean:
	rm -rf *.o ChessMain

```