Object-oriented Implementation of Chess Game in C++

Yiran Zhong
Chongqing Foreign Language School, Chongqing, China

Email: 779507224@qq.com

Abstract: In this paper, we implement the standard chess game using C++, a popular object-oriented programming language. Our program is developed and fully tested on Mac OS X system. It can be run in terminal and allows two players to compete together. The object-oriented characteristics of C++ include namely, abstraction, inheritance, encapsulation and polymorphism are highly facilitates the development. At the end of the paper, we also points out some aspects that can be further improved.

1. Introduction

1.1 Background
Chess game belongs to casual games, and it requires two players to proceed on a chessboard of size 8 × 8 [1]. The rules of this game are transparent but have infinite possibilities to move pieces, thus each step is crucial to the player. The chess game is a highly adversarial game that requires players to consolidate step by step and develop their logical thinking ability. Computers, a machine that can perform numerical calculations, logical operations, memory storage, and program operations, are already playing an indispensable role in this digital age. It can replace human thinking and deal with problems that human beings can not solve. Computer provides chess game a brand new carrier, which not only makes it more convenient for people to play recreational games, train their intelligence and improve their chess skills, but also enables people to see the omnipotent compatibility of the computer field and the development vitality of other industries brought by computers. Even more it is helpful to promote research in the field of artificial intelligence [2]. The crucial role the computer plays in the chess game is storing a large number of game records and remnants, acting as assistants in the game, helping players check the game, studying opponents, preparing and analyzing the situation before the game. The methods of analyzing chess games and documenting rules presented by the computer may help solve many practical problems such as economic management, military command and so on in the future.

Our work uses C++ programming language to implement the chess game. With the help of object-oriented [3] properties of such language, designers are able to flexibly apply theoretical knowledge to solve practical problems, improve manipulative ability, and developers are encouraged to profoundly experience the basic thought—detailling the problem and decomposing things into various objects to solve—through the development of chess game. Compared with process-oriented C language, C++ is more suitable for large-scale software projects such as chess which includes multi-task, multi-factor and multi-requirement. By using encapsulation, we can encircle the process and data so that each functional module is independent of each other then we could successfully restrict the modification and maintenance of the program to the class, which ultimately makes things
easier. By applying inheritance, designers could share attributes from parent classes to improve code’s reusability. By making use of polymorphism, it provides a way to abstract the complicated relationships. The scope of computer application is enlarging, and the scale of software is expanding. Designers urgently need to use more advanced design ideas like object-oriented to avoid the second “software crisis” [4].

1.2 Requirements analysis and settings
Our program enables two users to play chess, which means there is no AI involved in our work. The game needs to create different pieces, visualize the chessboard, accept various users input controls, and know when to terminate the game process.

- development environment: Mac OS X
- applicable environment: Mac OS X
- tools: sublime text, terminal

1.3 Roadmap
The rest of the paper is organized as follows. In Section 2, we discuss the object-oriented design for the program, especially the underlying abstraction. In Section 3, we explain the how to interact with our program in the terminal. In Section 4, we discuss several termination situations of the program. Finally, we conclude the paper with a short discussion about possible improvements in Section 5.

2. Game Settings and Abstraction
In chess game, the chessboard and the pieces are the essential elements. Without these two parts, all the functionalities cannot be achieved. Meanwhile, these two parts are closely related because the board must restrict the movement of pieces at any time, and the game cannot run without the pieces. To put all these into the nutshell, the design and initialization of chessboard and pieces should be given priority.

- Before talking about any design of the game, we need first to specify the settings of the game:
  - The chessboard is two dimensional and has size 8 × 8.
  - Players belong to two different camps, say white and black, and the white pieces move first.
  - The chessboard is oriented, white is at the bottom of the board, and black is at the top of the chessboard.
  - There are six types of pieces: king, queen, bishop, knight, rook, and pawn. Each Player starts with one king, one queen, two bishops, two knights, two rooks, and eight pawns.
  - At the beginning of the game, the white pawns are in the second row, and the black pawns are in the seventh row. Setting the rooks at the corners of the board, and the knights are close to the rooks, bishops near the knights, queen near the bishop. The rest square is king’s position.

A standard chessboard should look like Figure 1. Next, we need to declare the movement rules of different pieces:

![Chessboard Image](image-url)

Fig 1: The standard setup of a chessboard [5].
• A king may move in any direction including diagonally, but may only move one square. A king may not move through other pieces. Also, the king cannot move into the square controlled by enemy’s piece, otherwise, count as the foul. The king will be “checked” by other pieces, so we should code the special judgement for king.
• A queen may move any number of spaces in one direction, including diagonally. A queen may not move through other pieces.
• A bishop may move diagonally only, but may move any number of spaces. A bishop may not move through other pieces.
• A knight may move in an L-shape, of length either two-by-one or one-by-two. The knight is the only piece that is not stopped by other pieces in its way (i.e., it can move through other pieces to get to an open square).
• A rook may move any number of squares, but only in a straight line which is not diagonal on the board. A rook may not move through other pieces.
• A pawn can move only forward towards the opponent’s side of the board, but with restrictions. On its first move of the game, a pawn may move forward either one or two squares; on subsequent moves, a pawn may only move forward one square. A pawn may not move through other pieces. Moreover, the pawn may not use this forward move to capture the opponent player’s piece. Instead, the pawn moves one square diagonally-forward to capture a piece and have that piece removed from the board.

2.1 Board class and Chessgame class
Set up chessboard, create chess pieces, start games, move pieces, analyze chess game, and terminate games—these operations need to be abstracted as behavior of class objects. These functions must be implemented on the chessboard, so we are going to create a Board class to encapsulate all functions. The name of Board class indicates that it can be used in any games that are played on a board and the chess game was just a specific one of them. If we want to write some other board games (e.g., Chinese chess, checkers) in the future, we can directly inherit from the built Board class. This embodies the object-oriented languages’ characteristics such as abstraction, inheritance, encapsulation and polymorphism.

The design of the Board class is as follows, where we only list some of its most important methods and attributes:

class Board {
public:
    /* Create a piece on the board. */
    bool initPiece(int id, Player owner, Position p);
    /* Return a pointer to the piece at the specified position, 
    if the position is valid and occupied, nullptr otherwise. */
    Piece* getPiece(Position p) const;
    /* Perform a move from the start Position to the end Position. 
    The method returns an integer with the status 
    >= 0 is success, < 0 is failure. 
    This method may handle the parts of moving pieces that 
    generalize over different board games. */
    virtual int makeMove(Position start, Position end);
    /* The main gameplay loop. */
    virtual void run();
    /* Return true if the game is over. */
}
virtual bool gameOver() const = 0;

protected:

    unsigned int m_width, m_height;
    /* Current game turn. */
    int m_turn;
    /* Vector containing all the Pieces of the board. */
    std::vector<Piece*> m_pieces;
    /* Functionality for creating a new piece (called by initPiece). */
    Piece* newPiece(int id, Player owner);
};

Successful implementation of above Board class helps us:

* create a new piece on the board by initPiece, which receives the ID of a piece, the holder, the position of the piece, according to certain rules, create the corresponding piece
* retrieve piece information at a specific location by getPiece, which receives the position, and returns the pointer of the corresponding piece
* move a piece on the board by makeMove, which is a very general method to move piece, i.e., put the piece in the ending position and then erasing the starting position of the piece
* start the main loop of the game by run, which shows the menu first, let the player choose to start a new game, and the system will initialize the chessboard and accept player’s input to continue the game (details for interactions will be discussed in Section 3)

After we implement the Board class, we derive the chessGame class from it. It shares the methods and attributes built in the Board class, and then add more functionalities related to chess game. The design is as follows, where we only list some of its most important methods and attributes:

class ChessGame : public Board {
public:

    Position WhiteKingPosition;
    Position BlackKingPosition;
    /* Setup the chess board with its initial pieces. */
    virtual void setupBoard();
    /* Perform a move from the start Position to the end Position.
     * The method returns an integer with the status
     * >= 0 is success, < 0 is failure. */
    virtual int makeMove(Position start, Position end) override;
    void run() override;
    /* Return 0 if White is checked, 1 if Black is checked, -1 if neither. */
    int check();
    /* Return 1 if a stalemate is reached, 0 if not. */
    int Stalemate();
    /* Return 1 if a checkmate is reached, 0 if not. */
    bool checkMate(int status);
/* Display the chessboard in terminal. */
void printBoard();

protected:
  /* 1 if Black is checked, 0 if White is checked, -1 if neither. */
  int m_status;
};

It should be mentioned that we override the general makeMove method defined in Board class since we need not only require the movement of pieces in a general board, but also consider the specific rules that should be followed by each piece, which will be discussed in detail in Section 2.2. Moreover, during each move, we should care about the status of the game, such as whether there is a player that being checked, whether a checkmate or stalemate is reached and so on. All of these termination conditions should be incorporated in the overrode makeMove method of chessGame class and will be discussed in detail in Section 4.

2.2 Piece class and its derivations
Naturally, we also need to create a Piece class that abstracts the properties that an individual piece should have. The design is straightforward:

class Piece {
public:
  /* Return the owner of the piece. */
  Player owner() const;
  /* Return the id of the piece. */
  int id() const;
  /* Return an integer representing move validity, >= 0 valid, < 0 invalid.
   * This method implements generic logic to check for a valid move. */
  virtual int validMove(Position start, Position end, const Board& board) const;
  /* Return 1 if the move is blocked by other pieces, 0 if not. */
  int handleBlock(Position start, Position end, const Board& board) const;

...}

protected:
  /* Owner of a piece. */
  Player m_owner;
  /* Id of a piece. */
  int m_id;
  /* Constructs a piece with a specified owner. */
  Piece(Player owner, int id) : m_owner(owner), m_id(id) {}
Note that Piece::validMove is used to deal with the generic logic to check for a valid move, which applies to all types of pieces, including determine whether the player moves the piece of the other side, whether the starting and ending positions of the piece are same, and whether the player capture the piece of its own side. If any one of them is true, an error will be returned.

In addition, all the pieces may be exposed to the problem that be blocked by other piece. Method handleBlock will help us analyze the movement from the horizontal, vertical, and diagonal directions. An error will be returned if the piece moves in these three directions and encounters other piece.

Since different types of pieces have different movement rules, the validMove is better to be overrode for different types of pieces. Therefore, we further derive the Piece class for all types and then override the validMove method in the derived classes. Following is the derived Pawn class from Piece class, and all the other types of pieces should be derived in a similar way:

```cpp
class Pawn : public Piece {
    protected:
        Pawn(Player owner, int id) : Piece(owner, id) {} 
    public:
        /* Implement piece-specific logic for checking valid moves. */
        int validMove(Position start, Position end, const Board& board) const override;
};
```

According to the rules specified before for each piece, the implementation for validMove in each derived class is not difficult:

- Pawn can only walk correctly in three situations: move forward one square, move forward two squares only at its starting position, move diagonally-forward one square to capture a piece of the other side. If none of these happens, then an error will be returned.
- For rook, after receiving the starting position and the ending position, we will check whether the horizontal and vertical coordinates of the starting position and the ending position are the same or not. If no one is the same, it indicates that the moving track of rook is not a straight line, and then an error will be returned.
- Knights are L-shaped moves, so we only need to consider two cases: 1: move one grid in horizontal way and then walk two grid vertically 2: move two grids horizontally and then walk one grid vertically. Return an error if player violates the rules above.
- Bishop can only move diagonally, so we only need to ensure that the number of squares that move horizontally after the movement equals to the number of squares that move vertically. If the player moves in other ways, an error will be returned.
- Queen can move in a straight line or in a diagonal line. This is just an combination of rook’s rule and bishop’s rule.
- The moving direction of king is the same as that of the queen, but king can only move one grid each time, so we need to ensure that the distance between the ending position and the starting position in each direction is one square. Otherwise, an error will be returned.

3. Interactions in Terminal
Throughout the game, column coordinates are specified with letters, and row coordinates are 1- based with numbers assigned starting from the bottom left. This is a standard chess notation, as is shown in Figure 1.

When starting the program, we will enter into a new game, the pieces will be set up automatically on the chessboard by appropriate constructor. And the turn counter will be initialized to 1. The turn
counter increases by 1 at every move, starting with turn 1 for white, moving on to turn 2 for black, then 3 for white, etc.

Table 1. (a) turn board mode on (b) white player moves one of pawns two squares forward (c) black player moves its- knight

| row/column | a   | b   | c   | d   | e   | f   | g   | h   |
|------------|-----|-----|-----|-----|-----|-----|-----|-----|
| 1          | Rook | Knight | Bishop | Queen | King | Bishop | Knight | Rook |
| 2          | Pawn | Pawn | Pawn | Pawn | Pawn | Pawn | Pawn | Pawn |
| 3          |      |      |      |      |      |      |      |      |
| 4          |      |      |      |      |      |      |      | Pawn |
| 5          |      |      |      |      |      |      |      |      |
| 6          |      | Pawn |      |      |      |      |      | Knight |
| 7          | Pawn | Pawn | Pawn | Pawn | Pawn | Pawn | Pawn | Pawn |
| 8          | Rook | Knight | Bishop | Queen | King | Bishop | Knight | Rook |

(a) The arrangement of position

| row/column | a   | b   | c   | d   | e   | f   | g   | h   |
|------------|-----|-----|-----|-----|-----|-----|-----|-----|
| 1          | Rook | Knight | Bishop | Queen | King | Bishop | Knight | Rook |
| 2          | Pawn | Pawn | Pawn | Pawn | Pawn | Pawn | Pawn | Pawn |
| 3          |      |      |      |      |      |      |      |      |
| 4          |      |      |      |      |      |      |      | Pawn |
| 5          |      |      |      |      |      |      |      |      |
| 6          |      | Pawn |      |      |      |      |      | Knight |
| 7          | Pawn | Pawn | Pawn | Pawn | Pawn | Pawn | Pawn | Pawn |
| 8          | Rook | Knight | Bishop | Queen | King | Bishop | Knight | Rook |

(b) The arrangement of position
At the start of every turn, the program prints out a summary of the current state in the form of

| row/column | a | b | c | d | e | f | g | h |
|------------|---|---|---|---|---|---|---|---|
| 1          | Pawn |   |   |   |   |   |   |   |
| 2          |   |   |   |   |   |   |   |   |
| 3          |   |   |   |   |   |   |   |   |
| 4          | Pawn | Pawn | Pawn | Pawn | Pawn | Pawn | Pawn | Pawn |
| 5          | Rook | Knight | Bishop | Queen | King | Bishop | Knight | Rook |

Fig 2: Illustration of some interactions in the terminal.
9

And waits for the user input. The program accepts a regular movement in a form of

\texttt{<letter><number> <letter><number>}

where the first letter/number combination stands for the starting position and the second letter/number combination stands for the ending position. For example, at the start of the game, it will print \texttt{White turn 1:}, and if the white player wants to move the leftmost pawn forward by two squares, we can type \texttt{a2 a4}. Certainly, for invalid positions, an error will be returned and the program continues to wait for a legit user input.

Besides the movement, our program also accepts some other non-move inputs:

• \texttt{q}: exit the game.
• \texttt{forfeit}: the current player forfeits the game so that the other player wins.
• \texttt{board}: toggles display of the board in the terminal. By default, the board is off. Entering board after being in no-board mode immediately displays the relevant board for the current turn. The board will now be shown after every turn, until board is entered again to turn the board off.

In order to be user-friendly, all of the above user inputs are case insensitive. And to display the board, we just produce a board similar to the standard chessboard using different color settings in terminal. An illustration of some interactions can be seen in Figure 2.

4. Termination Situations

This section focuses on several termination conditions of our program. The user-entered commands \texttt{q} and forfeit mentioned in previous section will exit the game, while there are also some other situations require the program to make judgment whether the game should be terminated.

If a king is ever threatened by any piece of the opposing player in such a way that a single move by the opponent could result in the king’s immediate capture, this is considered a checked position. After each step, we have to decide whether this step will cause the opponent to be checked. The implementation is quite simple. After one step, use the enumeration method to find all of pieces of the current player, and check if there is any piece could capture the other player’s king after moving one more step. If such a piece and corresponding pace exist, then it means that the opponent is in a checked position.

With the help of the ability to decide whether the opponent is being checked, we have two more termination situations of the game: checkmate and stalemate.

Checkmate means a player is checked but cannot get rid of this dangerous situation. Then the checking player is declared to be the winner of the game and the game should be terminated. We use a similar enumeration method to determine if a checkmate happens: traverse all the pieces of the checked side on the board to see whether there exists a pace that releases this checked position. If there exists such pace, then the checked player is not checkmated, and the game continues; if such a pace does not exist, then the game is over, and the checking player wins. It should be noticed that once a player is checked, the highest priority is to get rid of this position. And we must ensure that players would not voluntary to let king be in check by using enumeration.

Stalemate means a deadlock happens and each of the players is not able to take a step to checkmate the other. This can also be implemented by enumeration. When no one is in check, traverse the all the pieces of the player who has the right to move, and see whether the player has a legal movement. If there exists any legal movement, then this is not a stalemate and game continues; otherwise, the game is over and a stalemate message will be returned.

It should be mentioned that all of the above implementations involve using enumeration methods, where we will consider the movement of each piece one step in advance. However, this movement is not a real behavior of the piece. It only happens in our computation since we need to see its
consequence. Therefore, after each of such computations, we need to recover the board state to the one that before any further movements.

5. Conclusion

In this paper, we demonstrate our work of implementing the standard chess game with two players using C++. The properties of object-oriented language facilitate the development of this project. According to the standard rules, we make some high level abstractions of the game and then implement the details for each functionality. Meanwhile, various interaction ways in terminal are provided to make our program easier to used.

Of course, our work has many parts that can be improved. For example, add a scaler to record the number of pieces on both sides to make the whole situation clearer, or using drawing software to create the more refined chessboard. Also, the enumeration method used in determining termination conditions seems not to be optimal and we may use more advanced algorithms to enhance the efficiency.

This development experience is priceless, and it will help to lay a solid foundation for future learning.

References

[1] Chisholm, Hugh, ed. “Chess”. (1911). Encyclopædia Britannica, 11th edition, Cambridge University Press (pp. 93–106).
[2] Russell, Stuart J. (2009). Norvig, Peter. Artificial Intelligence: A Modern Approach. Upper Saddle River, New Jersey: Prentice Hall.
[3] Bjarne Stroustrup. (1936).The C++ Programming Language. Addison-Wesley Professional, (4th ed).
[4] Edsger W. Dijkstra. (1972). The Humble Programmer. ACM Turing Lecture.
[5] https://www.chess.com/article/view/how-to-set-up-a-chessboard