# CSE 212 Final Project: Chess Game

Rebecca Oehmen
roehmen@nd.edu

Cecilia Hopkins
chopkins@nd.edu

Ryan Lichtenwalter
rlichten@nd.edu

**Abstract**

This project implements a classic version of Chess with a Graphical User Interface. The Chess game follows the basic rules of chess, and all the chess pieces only move according to valid moves for that piece. Our implementation of Chess is for two players (no Artificial Intelligence). It is played on an 8x8 checkered board, with a dark square in each player's lower left corner. Initially developing a text-based version, we then used .NET forms to implemented it in a GUI. Although we were not able to design a working artificial intelligence for the chess game in the time allowed, we researched the past implementations. We successfully created a GUI using inheritance and templates, as specified. Despite several unusual bugs in the GUI, our Chess program is a great, user-friendly game for two players.

**Keywords**
Chess, hierarchy, artificial intelligence, templates, GUI, inheritance, C++

## 1 Introduction

This project implements a classic version of Chess using C++ and a Graphical User Interface. The Chess game follows the basic rules of chess, and all the chess pieces only move according to valid moves for that piece. Our implementation of Chess is for two players (no Artificial Intelligence). It is played on an 8x8 checkered board, with a dark square in each player's lower left corner.

### 1.1 Previous Work

Much previous research has been conducted in Chess artificial intelligence and creating a more realistic/intelligent Chess match. Although we were not able to design a working artificial intelligence for the chess game in the time allowed, we researched the past implementations. In [1], Russell presented the minimax algorithm and the game tree developed by it. Russell discussed applying an evaluation function to the leaves of the tree, that judges the value of certain moves from a given position. Another method he mentioned is to cutoff the search by setting a limit to its depth. Russell evaluated a particular technique called alpha-beta pruning to remove branches of a tree that will not influence the final decision. Although we did not use this advanced of a method, we stored the possible moves for a piece one level down. Bratko [2] discusses the method of chess players to "chunk" together familiar chess patterns, and using this to reduce the complexity for AI when considering a position. However, this technique is in its early stages, and requires that multiple assumptions and a complicated detection process. Berliner [3] recognized that two similar positions can be very different, and sought to present a taxonomy of positions in chess that require special knowledge. However, this kind of research is essentially never complete.

## 2 Main

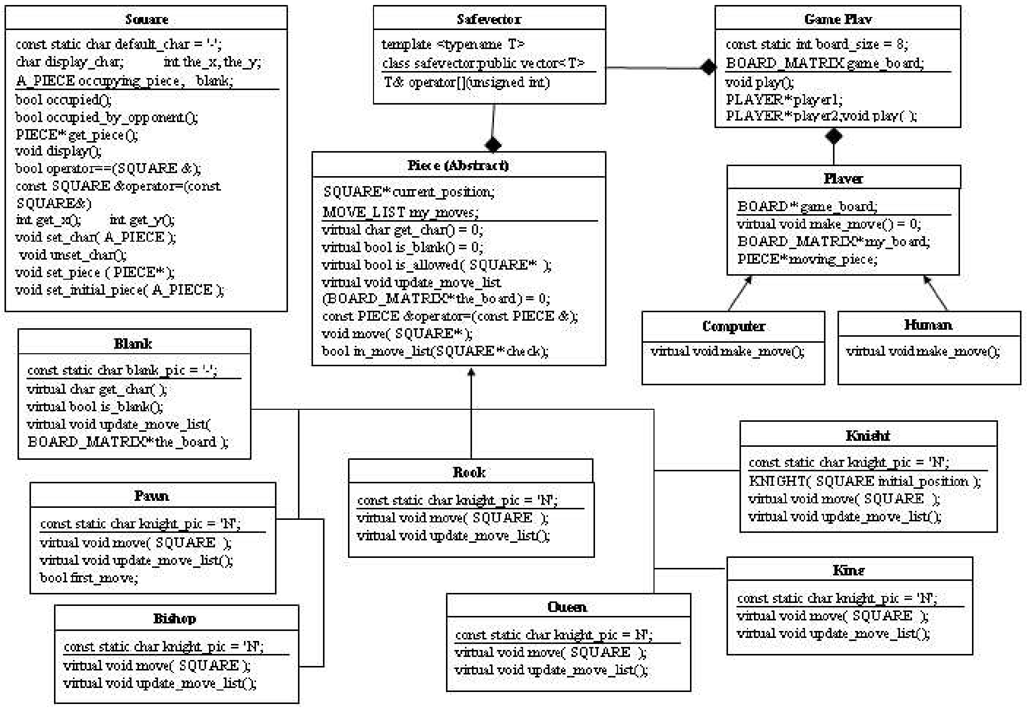
In this section we describe our approach to the problem. We discuss the inheritance hierarchy for our chess game, and briefly review the text-based version of the chess game. Then we explain how we designed the GUI for the Chess game from the text-based version. We also discuss the problems encountered.

## 2.1 Our Approach

We started by planning the hierarchy of our chess game, and constructing a UML of the inheritance. This organized our plan for coding and setting up the class hierarchies. Next we began coding a text-based version of the Chess game. Initially we worked together on developing the header files so that we knew what functions and logic we should use. Then we divided the group's responsibility: Cecilia mainly worked on the text-based version of the game, Ryan focused on the GUI, and Rebecca wrote the reports.

## 2.2 Inheritance Design

The hierarchy of the Chess game is displayed in the UML below (fig. 1). The different types of chess pieces naturally form an "is-a" relationship with the abstract class Piece. The derived classes are Pawn, Knight, Bishop, Rook, Queen, King, and Blank. The Blank class is the placeholder for a square on the chess board that is empty. The pieces hold a pointer to the square they are on. The Square class was a means of communication between all of the pieces and the board as a whole. Gameplay drove the creation of all the necessary objects (e.g. the squares, pieces, and players) and monitored critical game events, such as checkmate. The abstract class Player gave rise to two sub-classes: Human and Computer. Though the AI has not yet been implemented, these two derived classes allow for an easy integration of an AI in the future.



```
Square
const static char default_char = '-';
char display_char;          int the_x, the_y;
A_PIECE occupying_piece,   blank;
bool occupied();
bool occupied_by_opponent();
PIECE* get_piece();
void display();
bool operator==(SQUARE &);
const SQUARE &operator=(const
SQUARE&)
int get_x();       int get_y();
void set_char( A_PIECE );
 void unset_char();
void set_piece ( PIECE* );
void set_initial_piece( A_PIECE );
```

```
Safevector
template <typename T>
class safevector:public vector<T>
T& operator[](unsigned int)
```

```
Game Play
const static int board_size = 8;
BOARD_MATRIX game_board;
void play();
PLAYER*player1;
PLAYER*player2,void play( );
```

```
Piece (Abstract)
SQUARE* current_position;
MOVE_LIST my_moves;
virtual char get_char() = 0;
virtual bool is_blank() = 0;
virtual bool is_allowed( SQUARE* );
virtual void update_move_list
(BOARD_MATRIX*the_board) = 0;
const PIECE &operator=(const PIECE &);
void move( SQUARE* );
bool in_move_list(SQUARE* check);
```

```
Player
BOARD* game_board;
virtual void make_move() = 0;
BOARD_MATRIX*my_board;
PIECE*moving_piece;
```

```
Computer
virtual void make_move();
```

```
Human
virtual void make_move();
```

```
Blank
const static char blank_pic = '-';
virtual char get_char( );
virtual bool is_blank();
virtual void update_move_list(
BOARD_MATRIX*the_board );
```

```
Knight
const static char knight_pic = 'N';
KNIGHT( SQUARE initial_position );
virtual void move( SQUARE  );
virtual void update_move_list();
```

```
Rook
const static char knight_pic = 'N';
virtual void move( SQUARE  );
virtual void update_move_list();
```

```
Pawn
const static char knight_pic = 'N';
virtual void move( SQUARE  );
virtual void update_move_list();
bool first_move;
```

```
King
const static char knight_pic = 'N';
virtual void move( SQUARE  );
virtual void update_move_list();
```

```
Queen
const static char knight_pic = N';
virtual void move( SQUARE  );
virtual void update_move_list();
```

```
Bishop
const static char knight_pic = 'N';
virtual void move( SQUARE );
virtual void update_move_list();
```

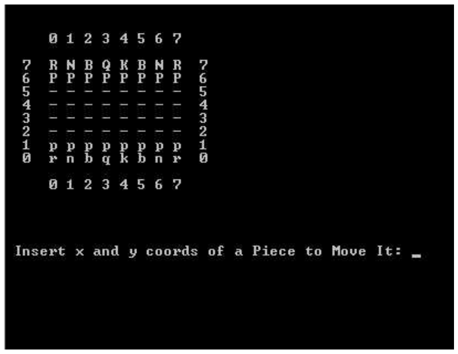
Figure 1: Inheritance Diagram

## 2.3 Template

Our chess implementation uses templates to implement a vector that has bounds checking. We designed a templated class named Safevector that publicly inherited from the Vector class. If the index is out of bounds, a specific error is outputted to the screen, and an error message is thrown. This class was extremely helpful in debugging our text-based version of chess, since we could more easily identify what caused most errors in the code.

## 2.4 Text-Based Chess Game

The text-based game (fig. 2) used upper and lowercase letters for the chess pieces for players 1 and 2 (resp.) The blank pieces are indicated with a '-' and there is a number grid provided for eaier input of the coordinates of squares. The player is prompted to input the coordinates of the piece to be moved, and where to move it. We decided that because chess is a complicated game with many rules, we would not be able to implement all its functionality. So, we do not allow for certain moves, such as castling or en passant. Starting with testing the simpler functions and gradually working up to playing chess matches, we tested the text-based version for functionality. Designing and Debugging the text-based version was challenging, but because we are familiar with C++ there were not very many problems. An example of a problem encountered was that we had circular include statements due to the set up of our inherited classes. Once we discovered this it was easy to fix. After we had a working text-based version of the Chess game, Ryan created a GUI chess board with buttons and added the functionality of the text-based chess game.



```
   0 1 2 3 4 5 6 7

7  R N B Q K B N R  7
6  P P P P P P P P  6
5  - - - - - - - -  5
4  - - - - - - - -  4
3  - - - - - - - -  3
2  - - - - - - - -  2
1  p p p p p p p p  1
0  r n b q k b n r  0

   0 1 2 3 4 5 6 7


Insert x and y coords of a Piece to Move It: _
```

Figure 2: Text-based version of our chess game.

## 2.5 Graphical User Interface

For the Graphical User Interface(GUI) we decided to use .NET Forms. This GUI program was very easy to use to design the interface, including buttons for the squares of the chess board and a message box. The message box was useful in keeping the player apprised of whether a move was valid and various other pertinent information. Many of the button functions and properties made it a very logical choice for an implementation of chess. Some examples of the helpful properties are background color and foreground image. They allowed for easy specification of square color and served as a container for the piece bitmaps. Functions such as set_Image() allowed for rapid and easy modification of these properties and were invaluable. Using buttons was the most logical choice for the user interface, as the vast majority of chess GUIs use clickable zones for input. One of the largest challenges was integrating the text-based game code with the GUI. Once everything compiled, we had issues with move and other functions that worked in the text-based version, but that didnt make a perfect transition to forms. Several major changes in implementation were required due to our lack of experience with designing for GUIs. For example, we had to change SQUARE to a pointer to SQUARE, which meant changes in every .cpp file. We had to do major debugging before the buttons would produce a change in the board. Some of the errors we encountered were circular includes, the board calling every move invalid, or the board saying that it was always the other player's turn and not allowing a move. After these problems were rectified, however, the button code was not difficult. It drew upon functions already defined for the text based game, passing the button procedures that connected the GUI to the chess logic. Functional coding for the buttons became much simpler when we created a function called handleButtonPress rather than copying and pasting the same logic into each button. Despite the difficulties, the GUI version of the Chess game was complete.

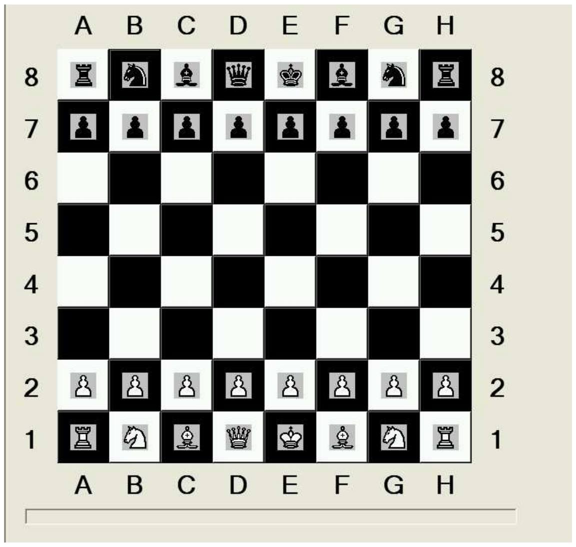
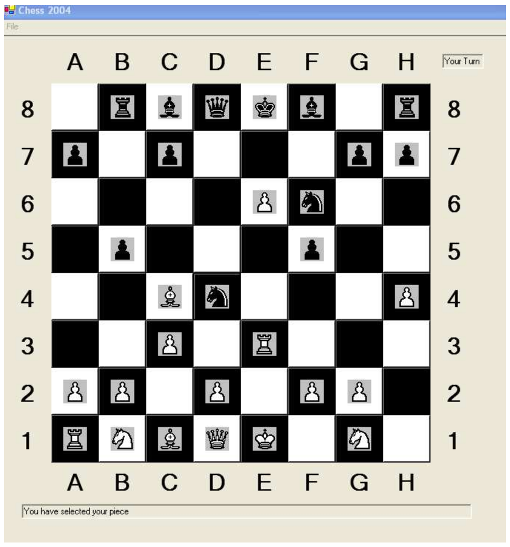


Figure 3: Chess board Graphical User Interface

## 2.6 Playing Chess

After the GUI window appears upon execution, the first player (white chess pieces) clicks on the piece that (s)he would like to move and then clicks on a valid position for this piece (including capturing another piece, which then disappears from the board). Invalid moves are not allowed, and the message box notifies the player of the problem. A player is also not allowed to move another player's pieces. The message box tells the player when (s)he is in check, and the player must move the King to get out of check. We do not currently have the logic such that the player in check may move another piece to get out of check. We played many games of Chess, and in a few rare instances, an exception occurred. The majority of the time, however, our Chess game worked well. Figure 4 below is a screen shot of the Chess game after several different moves and captures have taken place on the board.



Your Turn

You have selected your piece

Figure 4: Several moves into a Chess match.

## 3 Summary

In this paper we presented the classic version of Chess implemented as a GUI. We successfully created a game using inheritance, templates, and a GUI (and learned how to use Latex in the process). Although we encountered problems using .NET forms, we were able to design a working, user-friendly graphical interface for our Chess game.

## 4 Future Work

In future versions of the Chess game, we would implement logic for a simple Artificial Intelligence. In addition, we would make certain there were no errors through rigorous testing and improve the user interface.

## References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Upper Saddle River, New Jersey 07458: Prentice Hall, 1995.

[2] Bratko, P. Tancig, and S. Tancig, "Detection of positional patterns in chess," in *Advances in Computer Chess 4.* New York, NY: Pergamon Press, Apr. 1984, pp. 113–126.

[3] H. Berliner, D. Kopec, and E. Northam, "A taxonomy of concepts for evaluating chess strength," in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing.* Washington, DC: IEEE Computer Society, Nov. 1990, pp. 336–343.

## 5 Biographies

### 5.1 Rebecca Oehmen

Figure 5: Rebecca playing with her pet Nemo :)

Rebecca is a sophomore majoring in Computer Science. She has a twin sister and an older sister, as well as two cats. Rebecca is a brown belt in the Notre Dame Martial Arts Institute and loves hanging out with her roommate Debbie and her friends. This summer she will be returning to Argonne National Laboratory to intern at the Advanced Photon Source for her 3rd year.

## 5.2 Cecilia Hopkins

Figure 6: Cecilia with Amy Lee :)

Once upon a time at Notre Dame, there lived a girl named Cecilia. Now this was no ordinary girl– she could sing off tune and loved computers. One day, instead of working on C++, she decided to learn how to play ”Inspector Gadget” on the guitar. Now she is studying frantically in order to do well in C++ class.

## 5.3 Ryan Lichtenwalter

Figure 7: Ryan on top of the world!

Ryan is a CSE undergraduate at the University of Notre Dame. Although he has worked with computers for years, he also greatly enjoys music, nature, and the arts. He often combines computers and music in his endeavors to compose, and also advances his interest in photography with his skill in digital imaging. Ryan would like to investigate a career where he can culminate his computer and musical knowledge, or research artificial intelligence.