



TRIBHUVAN UNIVERSITY  
INSTITUTE OF ENGINEERING  
PULCHOWK CAMPUS

A PROJECT REPORT ON  
OBJECT ORIENTED PROGRAMMING WITH C++

## **A Chess Game**

SUBMITTED BY:  
**Prabin Shrestha** (076BCT067)  
**Suban Shrestha** (076BCT082)  
**Yunika Bajracharya** (076BCT095)

SUBMITTED TO:  
**DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING**  
**LALITPUR, NEPAL**

**AUGUST 26, 2021**

## ACKNOWLEDGEMENT

We would like to express our sincere gratitude towards our lecturer, Er. Daya Sagar Baral for his constant guidance, inspiring lectures and precious encouragement.

We would also like to thank the Department of Electronics and Computer Engineering, Institute of Engineering, Pulchowk Campus for providing us the opportunity to develop a project that will enhance our knowledge in Object Oriented Programming with C++ and provide us a new experience of teamwork. We are grateful to our friends for the help and feedback they provided whenever we needed. We would not have done this without the constant support and encouragement from our family, so we owe our deepest gratitude to our family.

Finally, we would like to express our earnest gratitude to all individuals who, through their direct or indirect involvement, granted their helping hands to accomplish our project timely and efficiently.

### **Authors:**

Prabin Shrestha

Suban Shrestha

Yunika Bajracharya

## ABSTRACT

*This project work is a course project submitted to the Department of Electronics and Computer Engineering in the partial fulfillment of the requirements for the course on Object Oriented Programming for our academic session in Bachelors of Computer Engineering, Second Year First Part as prescribed in the syllabus designed by Institute of Engineering, Tribhuvan University.*

*The main aim of this project was to develop a user-friendly program using an Object Oriented Programming language, C++. For this project, we made a classic two-player chess game. The goal of creating this game is also to learn about game development. We have used Simple DirectMedia Layer (SDL) for the graphical user interface along with additional SDL2 libraries (SDL2\_ttf, SDL2\_mixer, and SDL2\_image). The program is being written in Visual Studio Code. For compiler, we used msvc for Windows and g++ for UNIX systems. Finally, we used Cmake for build automation and debugging. The game is fully functioning but still there is a lot of room for improvement. We plan to add more features and improvements to make the game more entertaining in the future.*

**Key Words:** Chess, SDL, OOP, C++

## TABLE OF CONTENTS

<b>TITLE PAGE</b>	<b>i</b>
<b>ACKNOWLEDGEMENT</b>	<b>ii</b>
<b>ABSTRACT</b>	<b>iii</b>
<b>TABLE OF CONTENTS</b>	<b>v</b>
<b>OBJECTIVES</b>	<b>vi</b>
<b>1 INTRODUCTION</b>	<b>1</b>
1.1 Background . . . . .	1
<b>2 APPLICATION</b>	<b>2</b>
<b>3 LITERATURE SURVEY</b>	<b>3</b>
3.1 C++ . . . . .	3
3.1.1 Introduction . . . . .	3
3.1.2 Structure of code . . . . .	3
3.1.3 Features of C++ . . . . .	5
3.2 Simple DirectMedia Layer (SDL) . . . . .	7
3.3 CMake . . . . .	8
3.3.1 Build Process . . . . .	9
<b>4 METHODOLOGY</b>	<b>10</b>
4.1 Planning and Gathering information: . . . . .	10
4.2 System Model and Design: . . . . .	10
4.3 Software Development: . . . . .	10
4.4 Testing and Implementation: . . . . .	10
<b>5 OBJECT ORIENTED APPROACH (OOP)</b>	<b>11</b>
5.1 Classes . . . . .	11
5.2 Abstraction . . . . .	12
5.3 Encapsulation and Data hiding . . . . .	12
5.4 Inheritance . . . . .	13
5.5 Polymorphism . . . . .	15
<b>6 IMPLEMENTATION</b>	<b>19</b>
6.1 BLOCK DIAGRAM . . . . .	19

6.2	Main Menu . . . . .	19
6.3	The board . . . . .	20
6.4	The Pieces . . . . .	20
6.5	FEN Notation . . . . .	21
6.6	Interactivity . . . . .	22
6.7	Game States and Special Moves: . . . . .	22
6.8	End of the game: . . . . .	24
<b>7</b>	<b>RESULTS</b>	<b>26</b>
<b>8</b>	<b>PROBLEMS FACED AND SOLUTIONS</b>	<b>30</b>
8.1	State Management . . . . .	30
8.2	Performing the Perfth Test . . . . .	30
8.2.1	Perfth Test . . . . .	31
8.2.2	Digging Deeper with Stockfish . . . . .	32
8.2.3	Final Results . . . . .	32
8.3	Fixing Memory Leaks . . . . .	34
8.4	Single Player Mode . . . . .	34
8.5	Speed And Optimization . . . . .	34
<b>9</b>	<b>LIMITATIONS AND FUTURE ENHANCEMENTS</b>	<b>36</b>
<b>10</b>	<b>CONCLUSION AND RECOMMENDATIONS</b>	<b>37</b>
<b>11</b>	<b>REFERENCES</b>	<b>38</b>

## OBJECTIVES

The main objectives of our project are as follows:

1. To create a project on Object Oriented Programming and understand its concepts better.
2. To learn the basics of game development
3. To be familiarized with graphics programming and game development using SDL in C++ programming language.
4. To get better acquainted with version control with git and collaboration tools like GitHub
5. To optimize program in terms of time and space up to greatest extent as possible.
6. To build an attractive UI for the users to help them interact easily with our game.
7. To get familiar with document typesetting tool such as LATEX.
8. To learn to work and communicate effectively in a team.
9. To be prepared to work in major projects in the coming years.

## 1. INTRODUCTION

Chess is a widely popular game played in different forms all around the world. It is a two player turn based strategy game involving different pieces. the chess game has a really specific set of rules that should be followed while making a move. We took it as a challenge to include all the rules of chess in our project. Our project CHESS is an mock up of the chess game made by following the rules set by FIDE(the international chess federation).

In our rendition of chess game we have implemented the movement of the pieces, special movements like En-passant,Promotion, castling. This is done by checking the position of the pieces in the board and proving desired output. We have also implemented the chess clock. The chess clock is special type of timer that times both the players separately. If one player's clock runs out, then the other player is awarded victory via timeout. The game can end in three conditions: checkmate, draw , dead position or if a player resigns the game awarding victory to the opponent.

The game was made using C++ with the goal of applying Object oriented programming concepts to create the game of chess. We used free graphics library called (Simple DirectMedia Layer)SDL2 to implement the graphics, SDL2.image to load and display sprites or image and SDL2.ttf for displaying fonts. We also used CMake to make project compilation easy. On the way to develop our project, we took reference to different creators in YouTube like code bullet and jacob, and different books like Game development using SDL.

### 1.1. Background

We, the students of 076 BCT, were assigned to create a project using the concepts of C++ as a requirement for completion of course of Object Oriented programming. Object Oriented Programming(OOP) is a modern programming paradigm where a complex problem is broken down into smaller and simpler terms known as classes and objects.Using this paradigm, complex applications can be created in a more systematic way by focusing only on a particular part of a problem at once. Similarly, the use of other OOP features like inheritance, polymorphism and encapsulation makes the process feel more natural as the classes and objects in the program can be compared with real time objects.

Game development is a vast ocean with limitless possibilities. We have an end goal to develop a fully functioning game using the C++ programming language with the help of different libraries. As fate would have it, we decided to create the chess game for the project.

## **2. APPLICATION**

Chess is one of the most popular game in the world. The project in itself is not a feat, Chess has been made many times using different programming language. Still the project has wide range of application in the real world. Even if the project is not in par with the projects made by a mainstream developers, our project of chess can be played with all the rules of chess implemented. As a result, we can play the game of chess between two players following the rule set for chess. In conclusion, we can say that the project provides a reliable place to play the game of chess.



### **3. LITERATURE SURVEY**

#### **3.1. C++**

##### **3.1.1. Introduction**

C++, as we all know is an extension to C language and was developed by Bjarne Stroustrup at Bell Labs. C++ is an intermediate level language, as it comprises a confirmation of both high level and low level language features. C++ is a statically typed, free form, multi-paradigm, compiled general-purpose language.

It is an Object Oriented Programming language but is not purely Object Oriented. Its features like friend and virtual violate some of the very important OOP concepts such as data hiding, so this language can't be called completely Object Oriented.

##### **3.1.2. Structure of code**

A C++ program is structured in a specific and particular manner. In C++, a program is divided into the following four sections:

1. Standard Libraries Section
2. Class Definition Section
3. Functions Definition Section
4. Main Function Section

For example, let us look at the implementation of the Hello World program:

```

1  #include <iostream>
2  #include <string>
3  //using namespace std;
4  class PrintText{;
5      public:
6          PrintText(std::string)
7      }
8  PrintText:: PrintText(std::string txt){
9      std::cout<<txt<<std<<endl;
10 }
11 int main(){
12     PrintText text("Hello World!");
13     return 0;
14 }

```

### Standard Libraries Section

```

1  #include <iostream>
2  #include <string>
3  //using namespace std;

```

1. **#include** is a specific preprocessor command that effectively copies and pastes the entire text of the file, specified between the angle brackets, into the source code. This file is also called header file.
2. The file is merely for input output streams. This header file contains code for console input and output operations. This file is a part of std namespace.
3. The other header file is used for string related functions and is also a part of std namespace.
4. namespace is a prefix that is applied to all the names in a certain set. For example, iostream file is defined in a set what we call std and it defines two names used in this program cout and endl.
5. If using namespace std; is used, the compiler understands that the names like cout and endl of the std namespace are being used.

```

4  class PrintText{;
5      public:
6          PrintText(std::string)
7      }
8  PrintText:: PrintText(std::string txt){
9      std::cout<<txt<<std<<endl;
10 }

```

### Class definition Section

The classes are used to map real world entities into programming. The classes are key building blocks of any C++ program. A C++ program may include several class definitions. This is the section where we define all of our classes. In above program PrintText is the class and text is its object.

C++ allows the programmer to define their own function. A userdefined function groups code to perform a specific task and that group of code is given a name (identifier). When the function is invoked from any part of the program, it executes the codes defined in the body of the function.

In this program the function PrintText is a special member function called constructor of the class PrintText.

### Main Function Section

```

11 int main(){
12     PrintText text("Hello World!");
13     return 0;
14 }

```

main() function is the function called when any C++ program is run. The execution of all C++ programs begins with the main() function and also ends with it, regardless of where the function is actually located within the code.

### 3.1.3. Features of C++

C++ is a general-purpose programming language that was developed as an enhancement of the C language to include object oriented paradigm. It is an imperative and

a compiled language. Some of its main features are:

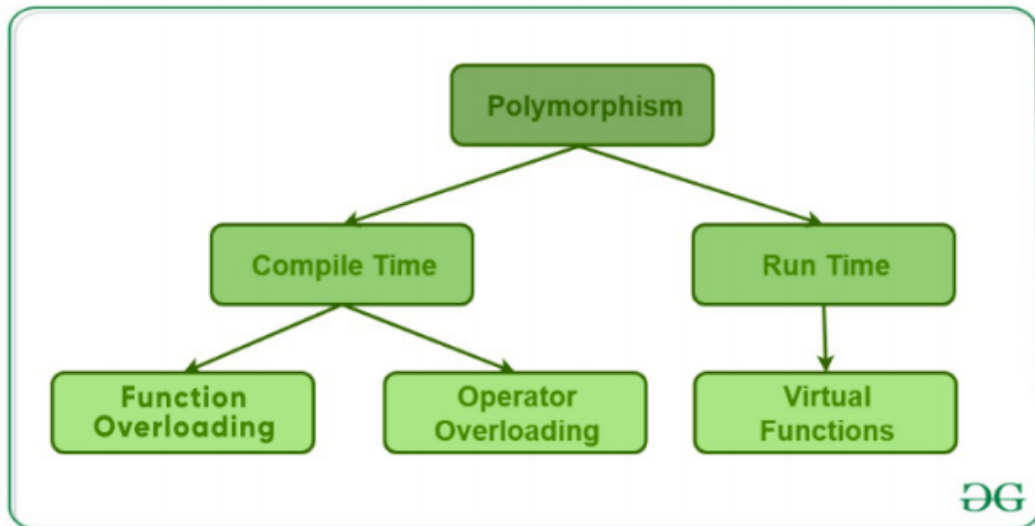
1. **Namespace:** A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when the code base includes multiple libraries.
2. **Inheritance:** Inheritance is a process in which one object acquires some (or all) of the properties and behaviors of its parent object automatically. In such way, one can reuse, extend or modify the attributes and behaviors which are defined in other classes, from existing classes. The class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class. The syntax for derived class is:

```
class derived_class:: visibility-mode base_class
{
    //body of derived class
}
```

3. There are three visibility modes in which a derived class inherits from its base class. They are private, public and protected. Following table clarifies the concept:

Base member Access Specifier	Visibility Mode		
	private	public	protected
private	Not Accessible	Not Accessible	Not Accessible
public	private	public	protected
protected	private	protected	protected

4. **Polymorphism:** The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. Polymorphism is considered as one of the important features of Object Oriented Programming.
5. In C++ polymorphism is mainly divided into two types:
  - (a) Compile time Polymorphism



*fig. 3.1: Types of Polymorphism*

*Source: GeeksForGeeks*

#### (b) Runtime Polymorphism

6. **Templates:** A template is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.
7. C++ adds two new keywords to support templates: 'template' and 'type-name'. The second keyword can always be replaced by keyword 'class'.

### 3.2. Simple DirectMedia Layer (SDL)

Simple DirectMedia Layer is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D. It is used by video playback software, emulators, and popular games including Valve's award winning catalog and many Humble Bundle games.

The latest version of SDL and SDL 2.0, which we will be covering in this book, is still in development. It adds many new features to the existing SDL 1.2 framework. The SDL 2.0 Roadmap ( [wiki.libsdl.org/moin.cgi/Roadmap](http://wiki.libsdl.org/moin.cgi/Roadmap) ) lists these features as:

1. A 3D accelerated, texture-based rendering API

2. Hardware-accelerated 2D graphics
3. Support for render targets
4. Multiple window support
5. API support for clipboard access
6. Multiple input device support
7. Support for 7.1 audio
8. Multiple audio device support
9. Force-feedback API for joysticks
10. Horizontal mouse wheel support
11. Multitouch input API support
12. Audio capture support

While not all of these will be used in our game-programming adventures, some of them are invaluable and make SDL an even better framework to use to develop games. We will be taking advantage of the new hardware-accelerated 2D graphics to make sure our games have excellent performance.

SDL2 libraries also contains extension to keep SDL as light as possible. Some of the libraries are SDL.image,SDL.net,SDL.mixer,SDL.ttf, true type SDL.ttf.SDL.image is used to load different images,SDL.net is used for cross platform networking , SDL.mixer is an audio mixer library that supports WAV,MP3,MIDI and OGG. SDL.ttf is used to write using fonts in the program and SDL.ttf is Rich Text Format library.

### **3.3. CMake**

CMake is an extensible, open-source system that manages the build process in an operating system and in a compiler independent manner. CMake is designed to be used alongside with native environment. Simple Configuration files placed in each source directory called CMakeLists.txt are used to generate standard build files. Cmake can generate a native buuild environment that will compile the source code, create libraries, generate wrappers and build executables .

### 3.3.1. Build Process

The build of a program or library with CMake is a two-stage process. First, standard build files are created (generated) from configuration files (CMakeLists.txt) which are written in CMake language. Then the platform's native build tools (native toolchain) are used for actual building of programs.

The build files are configured depending on used generator (e.g. Unix Makefiles for Make). Advanced users can also create and incorporate additional makefile generators to support their specific compiler and OS needs. Generated files are typically placed (by using cmake's flag) into a folder outside of the sources one.

Each build project in turn contains a CMakeCache.txt file and CMakeFiles directory in every (sub-)directory of the projects helping to avoid or speed up regeneration stage once it's run over again.

Once the Makefile has been generated, build behavior can be fine-tuned via target properties (since version 3.1) or via CMAKE...-prefixed global variables . The latter is discouraged for target-only configurations because variables are also used to configure CMake itself and to set up initial defaults.

## **4. METHODOLOGY**

The following methods were used for the completion of the project:

### **4.1. Planning and Gathering information:**

We researched online and gathered various information about Chess game and different libraries required for the project. The planning for the work division among the three of us was done.

### **4.2. System Model and Design:**

A basic model of the game was proposed documenting the information about the project was carried out before the coding and implementation during the proposal submission. New testable features were added regularly and validated.

### **4.3. Software Development:**

This project is based on C++ programming language and Object Oriented Programming concept using **SDL 2** (Simple DirectMedia Layer) library for graphics. Additional SDL2 Libraries (SDL2\_ttf, SDL2\_mixer, SDL2\_image) are used for text, sound and image in the program. As compiler, we will be using MSVC for windows and GCC for unix systems. For windows, we have used Cmake for build automation and debugging. And for Linux systems, scripts have been written.

### **4.4. Testing and Implementation:**

The program went through testing to measure usability , functionality and performance. We used git for version control and GitHub for sharing of code among our team members.

Adhering to the style of OOP, different classes were made with suitable access specifiers and data hiding was given a priority by making the members private as much as possible. The concept of code re-usability, data abstraction were implemented in the project. We have created an object for each of the elements of our game and then worked on their interaction with each other.



## 5. OBJECT ORIENTED APPROACH (OOP)

Since the primary objective of the project was to learn object oriented programming concepts by practically implementing them, we have tried our best to implement the features of object oriented programming in our project.

### 5.1. Classes

We have created around 20 classes that specifies various data and their operations. Through some classes, we have tried to abstract real world chess entities. We have created different files for each classes to achieve better readability and easy maintenance of the large program. We have separated header files for class declarations while the member function definitions of a class are stored in separate class implementation files.

The following classes for different operations were used in the program:

1. **TextureManager:** Used in creation of textures.
2. **Texture:** Creating a interface to interact with SDL\_Texture.
3. **Test:** Performs perft tests at a certain depth.
4. **Structures:** Stores various structures.
5. **StateMachine:** Handles the state of the program. Determines which state is to be rendered and updated.
6. **SoundManager:** Manages the audio of the program.
7. **GameMenu:** Renders and updates the main menu.
8. **Game:** Handles states, user input and rendering.
9. **GameState:** Parent class for GameBoard and GameMenu.
10. **GameBoard:** Renders and updates the Chess game.
11. **BoardState:** Stores information about the 8x8 board.
12. **Engine:** Performs various action on the BoardState object.
13. **Piece:** Parent Class for all the pieces.

14. **SlidePiece:** Generates the behaviour of sliding pieces, queen, rook bishop and is a parent class of them.
15. **King:** The King piece, which can move one square each direction and castle.
16. **Queen:** Queen Piece, that can slide along the diagonal or straight.
17. **Bishop:** Bishop Piece that can slide along the diagonal.
18. **Knight:** Knight piece, that moves 2 steps straight and 1 step to the side.
19. **Rook:** Rook piece, slides along a straight line.
20. **Pawn:** Pawn piece, can move forward 1 square ahead, (2 if first move) and attacks diagonally.

## 5.2. Abstraction

Abstraction is a feature of object oriented programming that hides the internal details of how object does its work and only provides the interface to use the service. We can manage complexity through abstraction.

In OOP, classes are used for creating user-defined data for abstraction. When data and its operation are presented together, it is called ADT (Abstract Data Type). Hence, a class is an implementation of abstract data type. So, in OOP, classes are used in creating Abstract Data Type.

For instance, we have created a class **GameMenu** and made it available in the program. Now we can implement the class in creating objects and its manipulation without knowing its implementation.

We created a **Texture** class which abstracts away the task of creating textures and rendering textures. We implemented it such that it can load entire sentences as textures using the method *loadSentence("Some sentence", fontsize, color)* and *loadFromFile("path/to/file")* and *render(posx, posy)* which provided us an interface for interacting with textures. This made it so that we could render textures and text without thinking about SDL2's functions and memory management.

## 5.3. Encapsulation and Data hiding

Combining data and function together into a single unit is called encapsulation. We can say encapsulation is a protective box that prevents the data from being accessed

by other code that is defined outside the box. We can easily achieve abstraction by making use of encapsulation.

We can achieve encapsulation through classes. In classes, each data or function is kept under an access specifier (private, protected or public).

**Public:** it contains data and functions that the external users of the class may know about.

**Private:** this section can only be accessed by code that is a member of a class.

**Protected:** this section is visible to class members, derived class members, friend functions of class, friend classes and friend classes of derived classes.

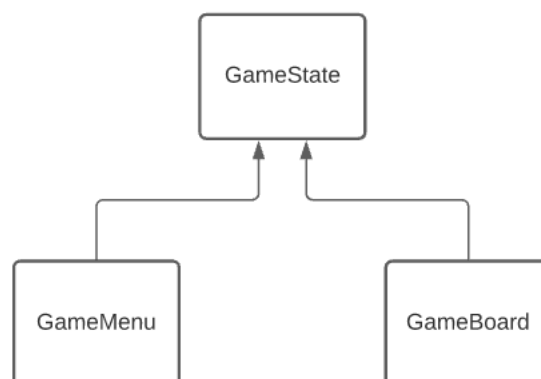
The insulation of data from direct access by the program is called data hiding. Data hiding by making them private or protected makes it safe from accidental alteration. Understanding this, we have tried to make our data private as much as it was possible.

#### 5.4. Inheritance

The main purpose of using inheritance is for:

- the reuse and extension of existing code.
- the elimination of redundant code.

##### The use of Hierarchical Inheritance



Base class: GameState

Derived classes: GameMenu and GameBoard

The GameState class has pure virtual functions, which makes it an abstract base class.

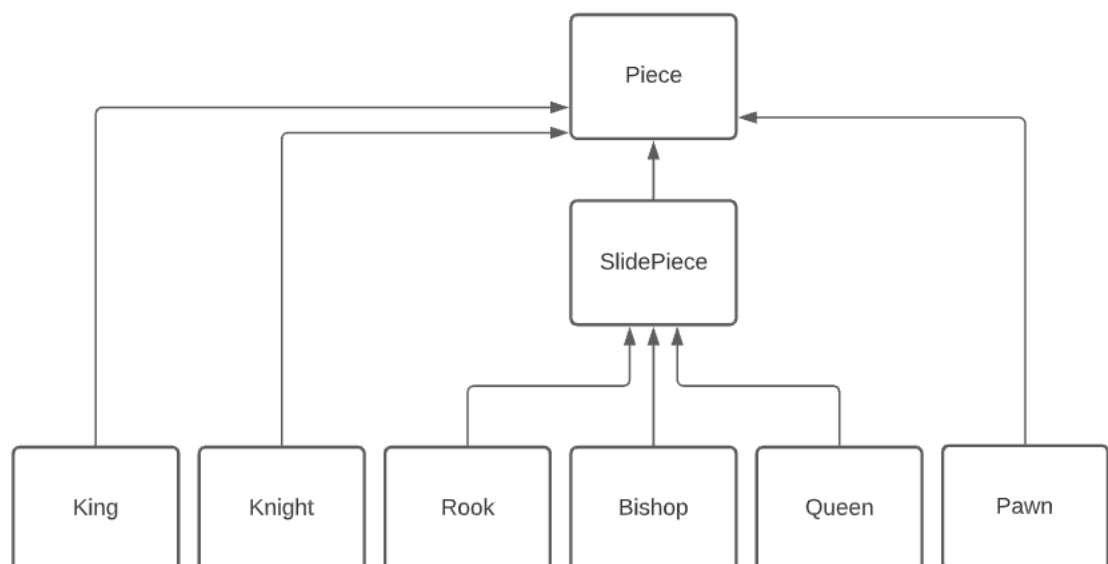
```
#pragma once
#include <SDL2/SDL.h>

class GameState {
public:
    virtual ~GameState() {}

    virtual void init() = 0;
    virtual void handleInput(SDL_Event &event) = 0;
    virtual void update() = 0;
    virtual void render() = 0;
};
```

We have derived classes GameMenu and GameBoard from the base class GameState. The derived classes have their own different function definitions. We do this inheritance as we need to store both GameMenu and GameState in a stack. We cannot store two objects of different types in a container and because they share similar functionalities. We make them derived classes as every derived class is a sub-type of base class. In this way, we can store them in a stack and render the game menu and game board as per need.

### The use of Hierarchical and Multilevel Inheritance



Base class: SlidePiece

Derived classes: Rook, Bishop, Queen

While creating the logic for movement of different pieces, we realized that the movement of **Rook**, **Bishop** and **Queen** are very similar as all three of them slide at a certain direction. So to eliminate code redundancy, we decided to implement their movements in one place and created a class **SlidePiece**. In this class, we have implemented movement of sliding in different directions. And the classes **Rook**, **Bishop** and **Queen** inherit this functionality with the only difference being the direction in which they slide.

Base class: Piece

Derived classes: SlidePiece, King, Knight, Pawn Each of the derived class generates a set of moves and stores the moves in a `std::vector`. These moves are known as **Pesudo Legal** moves as the moves generated do not take *checks* into account. After each derived class generates all the moves, the Base class is used to filter out all the illegal moves, giving us an `std::vector` of legal moves.

## 5.5. Polymorphism

Polymorphism is another important feature of OOP. It allows different objects to respond to same operation in different ways. The different ways of using same function or operator depending on what they are operating on is called polymorphism.

In c++, polymorphism is mainly divided into two types:

### 1. Compile time polymorphism

- Function Overloading
- Operator Overloading

### 2. Run time polymorphism

- Virtual functions

### Operator Overloading:

In our project, we have overloaded equal to, not equal to and addition operators so that they can operate on coordinates.

### Pure Virtual Functions:

Pure virtual functions is a virtual function that only has a declaration but doesn't have a definition. Since they have no definition, these functions cannot be called

```

bool operator==(Coordinate c) {
    if (c.i == i && c.j == j) {
        return true;
    } else {
        return false;
    }
}

bool operator!=(Coordinate c) {
    if (c.i != i || c.j != j) {
        return true;
    } else {
        return false;
    }
}

```

Figure 5.1: Relational operator overloading

```

Coordinate operator+(Coordinate c) {
    Coordinate temp;
    temp.i = i + c.i;
    temp.j = j + c.j;
    return temp;
}

Coordinate operator+=(Coordinate c) {
    i += c.i;
    j += c.j;
    return *this;
}

```

Figure 5.2: Binary operator overloading

and object consisting of pure virtual function cannot be created. It's usefulness comes from the fact that any class that derives from a base class consisting of a pure virtual function must implement the function for the derived class.

```

class Parent {
public:
    virtual void doSomething() = 0;
};

```

Since the function "doSomething()" is a pure virtual function, it makes the class an **Abstract Class** and an object of the class "Parent" cannot be created. But the function can be overridden in a Child class as

```

class Child: public Parent {
public:
    void dosomething() {
        // Does something
    }
};

```

Now we can create the class "Child" and call the method "doSomething()" Pure virtual functions were used extensively in our project and the example can be seen in the next section.

### Abstract Base Class:

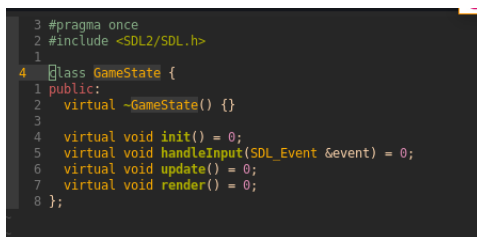
A class that has a pure virtual function is an **Abstract Base Class**. These classes cannot be used to instantiate an object but serve the following function.

- \* Deter from creation of the base class.
- \* Act as a base class from any derived classs and allow for easy polymorphism.

Example of abstract base class

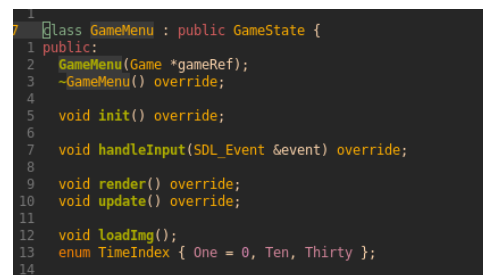
```
class Parent {
public:
    virtual void doSomething() = 0;
};
```

In our project we used this concept to create the concept of abstract classes and used to create two classes "GameState" and "Piece". These act as Abstract base class for other derived classes as standalone they do not have much meaning or functionality. We used the concept of virtual classes extensively which can be seen in the code snippet below.



```
3 #pragma once
2 #include <SDL2/SDL.h>
1
4 class GameState {
1 public:
2     virtual ~GameState() {}
3
4     virtual void init() = 0;
5     virtual void handleInput(SDL_Event &event) = 0;
6     virtual void update() = 0;
7     virtual void render() = 0;
8 };
```

Figure 5.3: Parent Class



```
1
2 class GameMenu : public GameState {
1 public:
2     GameMenu(Game *gameRef);
3     ~GameMenu() override;
4
5     void init() override;
6
7     void handleInput(SDL_Event &event) override;
8
9     void render() override;
10    void update() override;
11
12    void loadImg();
13    enum TimeIndex { One = 0, Ten, Thirty };
14
```

Figure 5.4: Child Class

### Virtual Destructors:

Since a derived class may have many heap allocation, it is important for it to have its own destructor. For this we make the desctuctor of the base class virtual and override it in our derived class. We used Virtual Distructors as such

```
class GameState {
public:
    virtual ~GameState() {}

    virtual void init() = 0;
    virtual void handleInput(SDL_Event &event) = 0;
    virtual void update() = 0;
    virtual void render() = 0;
};
```

```
class GameBoard : public GameState {  
public:  
    ~GameMenu() override;  
    ...  
};  
  
Gameboard::~Gameboard() { SoundManager::clean(); }
```



## 6. IMPLEMENTATION

The code for this project is accessed through the GitHub repository of the project:  
<https://github.com/Yunika-Bajracharya/Chess>

### 6.1. BLOCK DIAGRAM

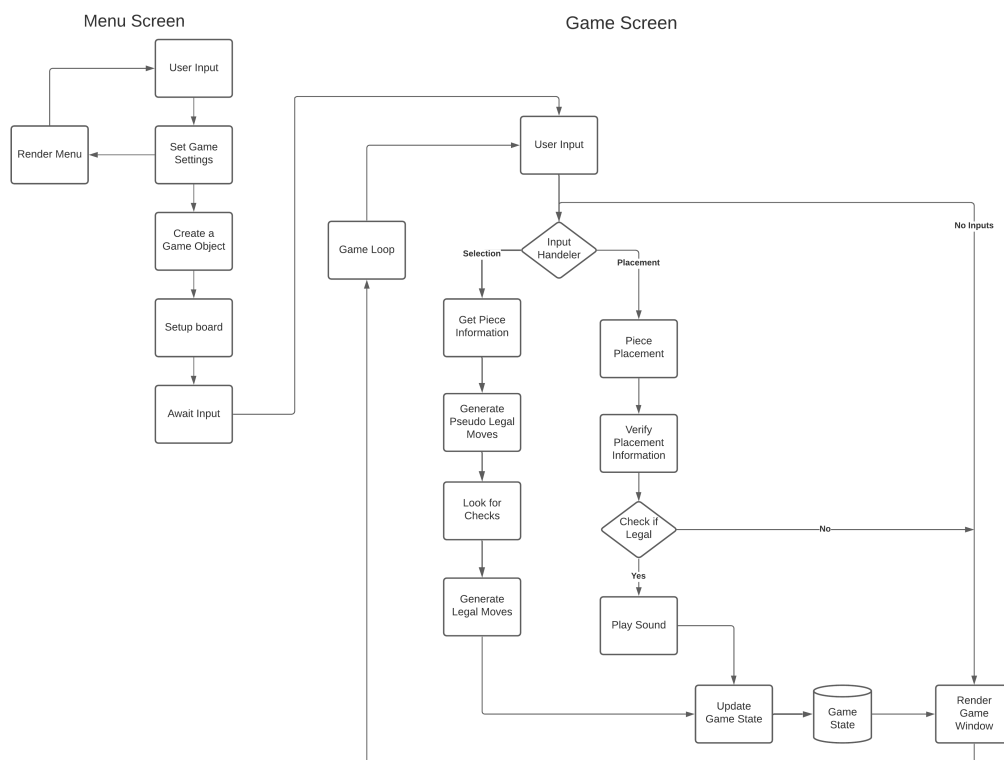


Figure 6.1: Block Diagram

### 6.2. Main Menu

The game starts with the main menu that consists of **player name input**, **Play button** and **exit button**.

- **Player Name Input:** This takes input of players' names.
- **Single Player:** User can play with the AI
- **Two Player:** User can play in two player mode

- **Time Control:** User has the option to choose time: 1min, 10 mins or 30 mins.
- **Exit:** This allows the user to close the program.

### 6.3. The board

A 8 \* 8 game board was created. The top left corner of the board was made the origin. On adding the index row and columns, we can see the pattern that every other square in the board is even/odd.

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14

Here, looping over each square, if  $(\text{row} + \text{column}) \% 2$  is equal to 0, it is a white square, and

if  $(\text{row} + \text{column}) \% 2$  is equal to 1, it is a black square.

For instance, let's see the highlighted square at row = 8 and column = 2,  
 $(8 + 2) \% 2 = 0$  so it should be white square.

Here, instead of white and black, we have used cream color and green color for the ease of eyes, but we will be referring them as white and black respectively for convenience in explaining.

### 6.4. The Pieces

We have six types of pieces in total: King, Queen, Bishop, Knight, Rook and Pawn. We have 1 King, 1 Queen, 2 Bishops, 2 Knights, 2 Rooks and 8 pawns in black as well as white.

For pieces, we have created an abstract base class called **Pieces**, and six derived

classes: **King, Queen, Bishop, Rook, Knight, Pawn**. Hierarchical inheritance is being used. The pseudo legal moves have been defined separately in all derived classes while the legal moves for all are checked in the base class.

## 6.5. FEN Notation

FEN notation is a way of representing a chess board. It is :

```
rnbqkbnr / pppppppp / 8 / 8 / 8 / 8 / PPPPPPPP / RNBQKBNR w KQkq - 01
```

### Field 1 :

The first field contains letters and numbers that represents all pieces and their locations in the board. It begins from the top row of the chess board and the slash represents new row. The letters represent pieces as follows:

r : rook

n : knight

b : bishop

q : queen

k : king

p : pawn

Lowercase letters denote black pieces and uppercase letters denote white ones. The numbers represent the amount of blank spaces. For eg. If number is 8 means that it has 8 blank spaces i.e. 8/8/8/8 : means four empty rows.

### Field 2 :

It tells whose turn it currently is. Here W is for white and b is for black.

### Field 3 :

This field represents castling availability, where k,q is for black and K,Q is for white).

If there is a k, king side castling is available.

If there is a q, queen side castling is available.

### Field 4 :

This field is the square that can be moved onto if enpassant. If there is no square, which means the last move is not the pawn moving two squares, then there is just a dash.

The last two fields are related to how many moves have been made.

## 6.6. Interactivity

Here, we check for mouse presses and check where the mouse is clicked on. and then drag the piece to the desired location while making sure the players aren't able to move the opponent's pieces.

### Pseudo-legal moves:

A pseudo-legal move is any move that can be made on the board based on how the pieces move. Pseudo legal moves ignore checks and checkmates. Pseudo-legal moves are:

- King : moves one square in any direction
- Queen : combines the power of a rook and bishop and can move any number of squares along a rank, file, or diagonal, but cannot leap over other pieces.
- Bishop : can move any number of squares diagonally, but cannot leap over other pieces.
- Rook : can move any number of squares along a rank or file, but cannot leap over other pieces.
- Knight : moves in an "L" shape: two squares vertically and one square horizontally, or two squares horizontally and one square vertically.) The knight is the only piece that can leap over other pieces.
- Pawn : can move forward to the unoccupied square immediately in front of it on the same file, or on its first move it can advance two squares along the same file, provided both squares are unoccupied (black dots in the diagram). A pawn can capture an opponent's piece on a square diagonally in front of it by moving to that square (black crosses). A pawn has two special moves: the en passant capture and promotion.

## 6.7. Game States and Special Moves:

### Check :

When a king is under immediate attack, it is said to be in check. A move in response to a check is legal only if it results in a position where the king is no longer in check. This can involve capturing the checking piece; interposing a piece between the checking piece and the king (which is possible only if the attacking piece is a

queen, rook, or bishop and there is a square between it and the king); or moving the king to a square where it is not under attack. Castling is not a permissible response to a check

### **Checkmate :**

The aim of the game is to checkmate the opponent; this occurs when the opponent's king is in check, and there is no legal way to get it out of check. It is never legal for a player to make a move that puts or leaves the player's own king in check.

### **Pawn Promotion :**

The new piece replaces the pawn on its square on the same move. The choice of the new piece is not limited to pieces previously captured, thus promotion can result in a player owning, for example, two or more queens despite starting the game with one.

### **En passant :**

It is a special pawn capture that can only occur immediately after a pawn makes a move of two squares from its starting square, and it could have been captured by an enemy pawn had it advanced only one square. The opponent captures the just-moved pawn "as it passes" through the first square. The result is the same as if the pawn had advanced only one square and the enemy pawn had captured it normally. The en passant capture must be made on the very next turn or the right to do so is lost.

### **Castling :**

Castling is a special move in chess that uses both a rook and the king. In castling, the king is moved two squares toward the rook, and the rook moves past the king to the square right next to where the king has moved. Castling takes one move, and is the only way for a player to move two of his own pieces on the same move. Castling can be done on either side of the board. Castling can either be done on the king-side (also known as castling short) or on the queen-side (also known as castling long).

### Rules of Castling :

Castling is only possible if each of the following things are true:

- Neither the king nor the rook being used to castle have moved in the game.

- The king is not in check, and is not moving into check or through check.
- There are no pieces between the king and the rook.

Once a piece is clicked, the possible moves and captures are highlighted. Similarly, the last position and the current position are also highlighted.

### **Legal moves:**

For legal move, we go through every pseudo-legal move and check if king is being in check. If it isn't, then we add it to the legal moves.

## **6.8. End of the game:**

**Win :** textbfCheckmate: The king is in check and the player has no legal move.

**Resignation:** A player may resign, conceding the game to the opponent. Most tournament players consider it good etiquette to resign in a hopeless position.

**Win on time:** In games with a time control, a player wins if the opponent runs out of time, even if the opponent has a superior position, as long as the player has a theoretical possibility to checkmate the opponent were the game to continue.

### **Draw :**

**Stalemate:** If the player to move has no legal move, but is not in check, the position is a stalemate, and the game is drawn.

**Dead position:** If neither player is able to checkmate the other by any legal sequence of moves, the game is drawn. For example, if only the kings are on the board, all other pieces having been captured, checkmate is impossible, and the game is drawn by this rule. On the other hand, if both players still have a knight, there is a highly unlikely yet theoretical possibility of checkmate, so this rule does not apply. The dead position rule supersedes the previous rule which referred to "insufficient material", extending it to include other positions where checkmate is impossible, such as blocked pawn endings where the pawns cannot be attacked.

**Time control**

If a player's time runs out before the game is completed, the game is automatically lost (provided the opponent has enough pieces left to deliver checkmate). The duration of a game ranges from long (or "classical") games, which can take up to seven hours (even longer if adjournments are permitted), to bullet chess (under 3 minutes per player for the entire game). Intermediate between these are rapid chess games, lasting between one and two hours per game, a popular time control in amateur weekend tournaments.

Time is controlled using a chess clock that has two displays, one for each player's remaining time. In our project, we have used digital clocks.

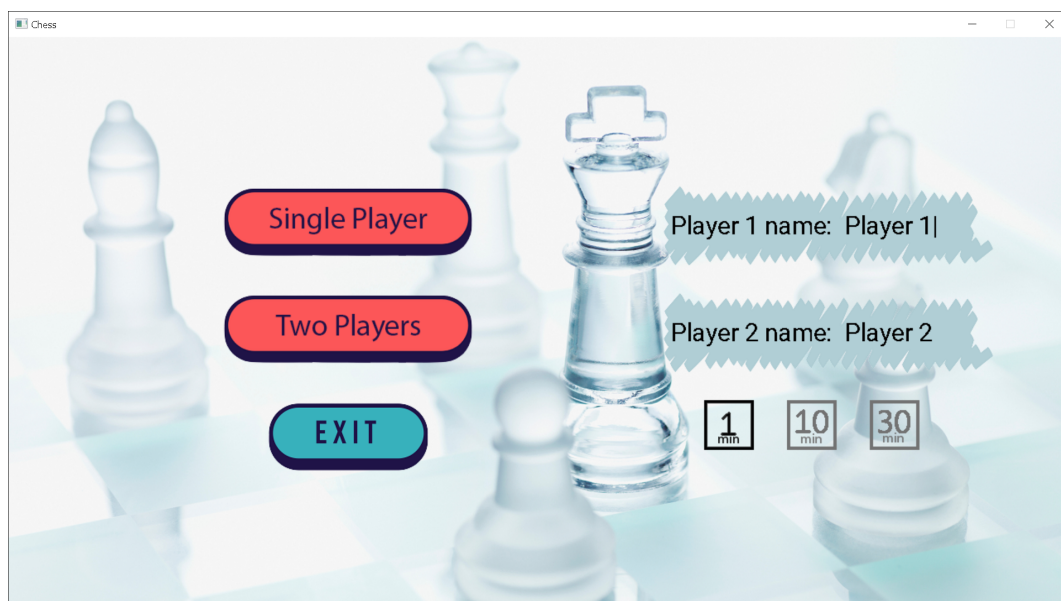
## 7. RESULTS

We were able to achieve the objects of the project and a chess game was designed with all the rules and the necessary features. The development of the game is fulfilled but still some additional features can be added in the future.

From this project, we understood the concepts of Object Oriented Programming better, learnt the basics of game development and graphics programming using SDL2 library. We got better acquainted with version control with git and collaboration with GitHub. Finally, we learnt about how to work and communicate effectively in a team which has prepared us to work in major projects in the coming years.

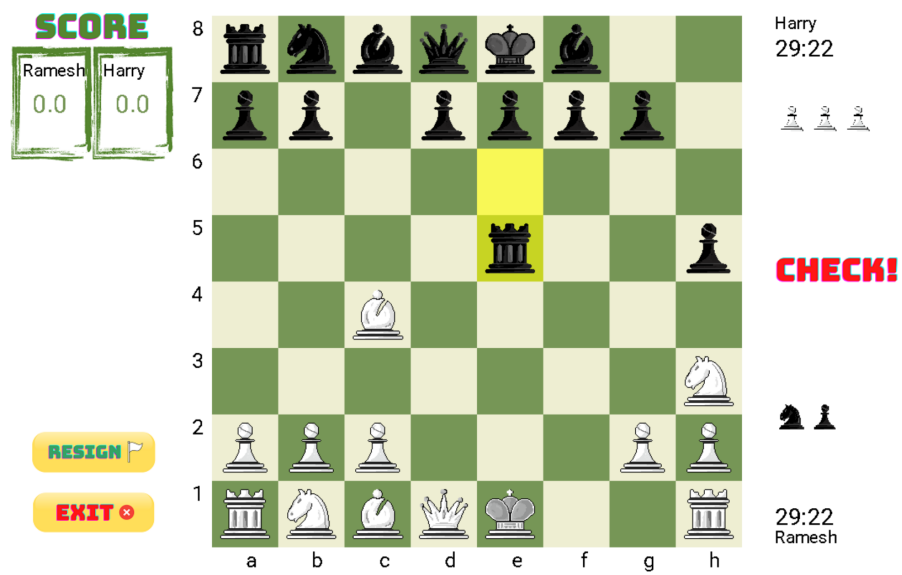
The following screenshots from different states throughout the game illustrate the final result of the project:

### 1. Main menu

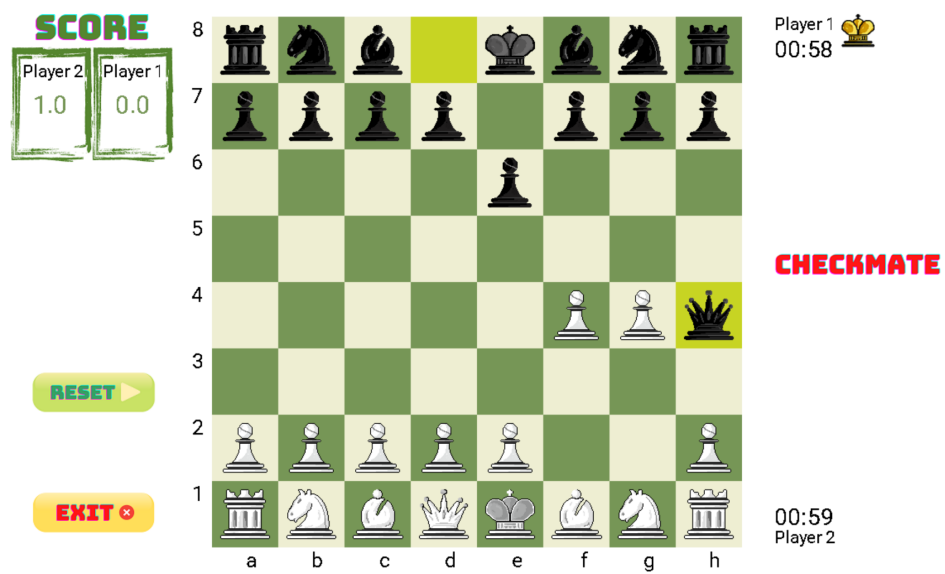




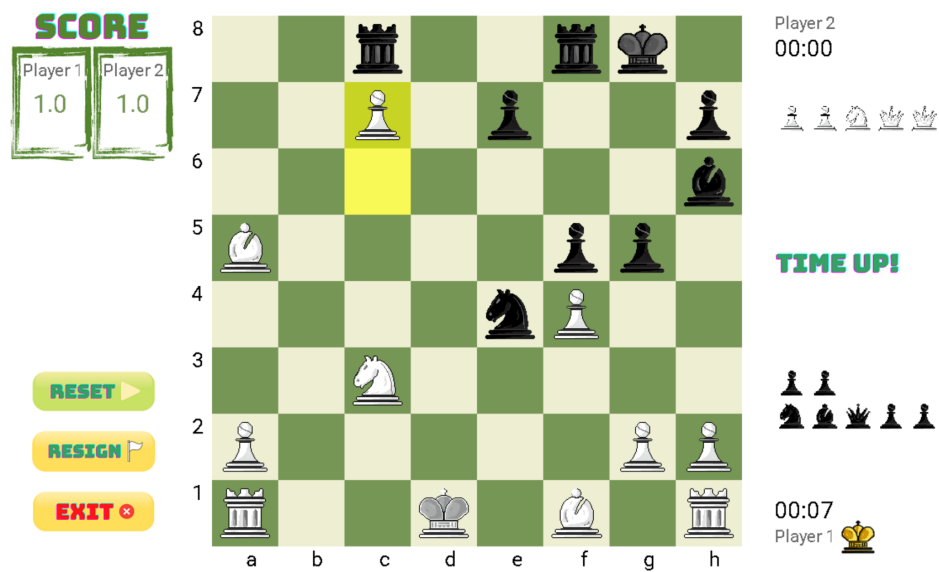
## 2. Check



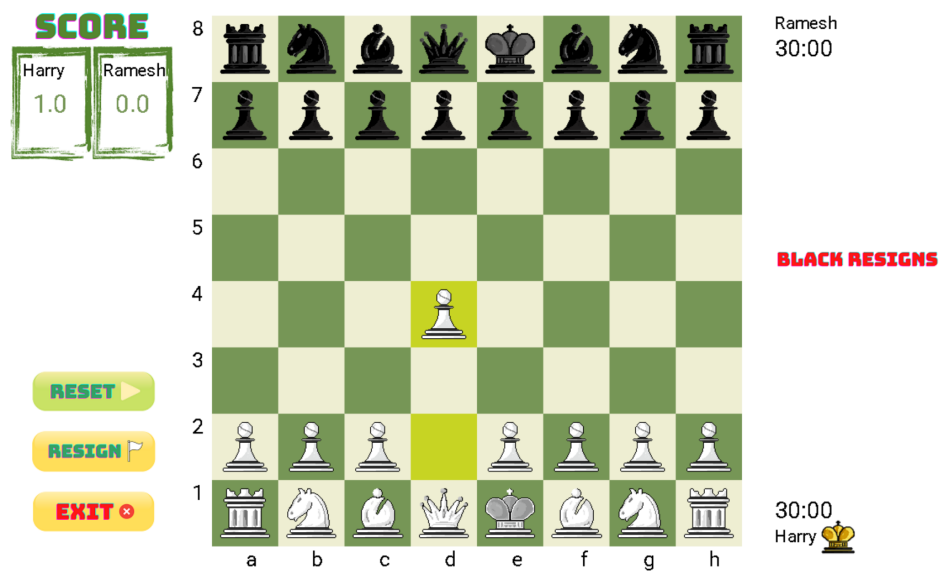
## 3. Checkmate



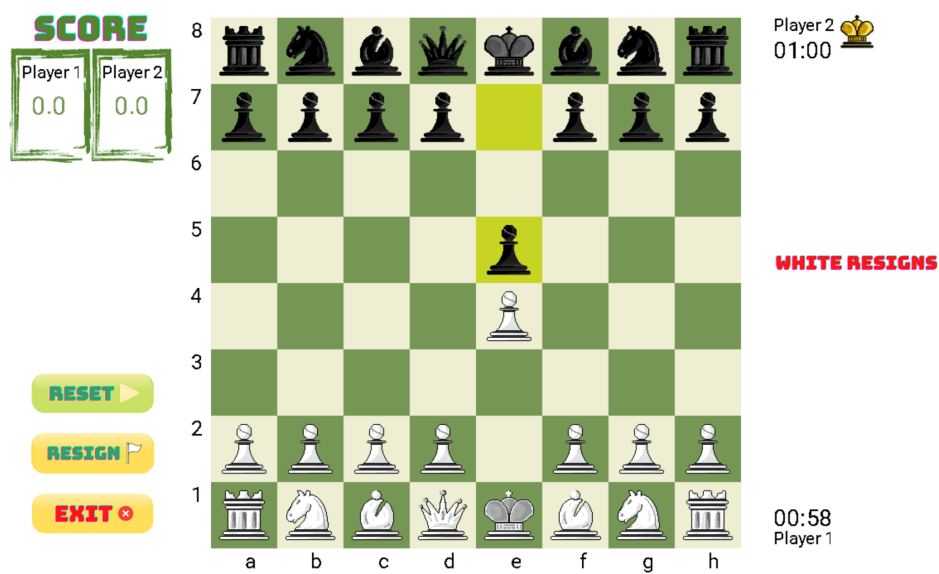
## 4. Time up



## 5. Black Resigns



## 7. White Resigns



## 8. PROBLEMS FACED AND SOLUTIONS

We faced a number of problems in our journey in creating a chess game.

### 8.1. State Management

Since our program consists of multiple states, it was quite difficult to incorporate them. Thus we decided to use a `std::stack` for dealing with states. We created a wrapper class to interact with the stack and push and pop states from the stack. The wrapper class is called "State Machine".

### 8.2. Performing the Perf Test

Since chess is a game with rigid rules, it was a priority to be able to incorporate all the rules of chess without any sort of undesired behaviour. Playing hundreds of games by hand to find bugs, was our first idea but pretty soon we realized it wasn't going to cut it. Playing hundreds of games by hand and hoping to catch any sort of undesired behaviour is just being too optimistic and there was no way we were going to be able to catch all the undesired behaviour. So to tackle this, we designed a **Test** class which will be responsible for performing the tests. The class was used to perform **Perft Test** on a given starting position.

**Perft** (performance test, move path enumeration) a debugging function to walk the move generation tree of strictly legal moves to count all the leaf nodes of a certain depth, which can be compared to predetermined values and used to isolate bugs. It is a recursive algorithm that figures out the total number of leaves at a certain depth. In simpler terms, it is a function that can give us the total number of positions that can be reached starting from a chess board position up to certain depth. We used the **divide** method to list the number of moves generated after each move.

A much more in depth view of perft can be obtained from the chess programming wiki: <https://www.chessprogramming.org/Perft>

It is at this point, we found out that our chess move generation has tremendous speed issues when going at larger depths. So we limited our testing depth of 4 which took around 2 mins to calculate. We shall discuss more about the performance issue in the next section. We also skipped some perft optimization such as hashing and

Depth	Nodes
0	1
1	20
2	400
3	8,902
4	197,281
5	4,865,609
6	119,060,324
7	3,195,901,860
8	84,998,978,956
9	2,439,530,234,167

Figure 8.1: Perft Results Example

bulk-counting.

### 8.2.1. Perft Test

After writing the test class, we performed perft test up to a depth of 5. The number of moves obtained were compared with the information provided in the wiki: [https://www.chessprogramming.org/Perft\\_Results](https://www.chessprogramming.org/Perft_Results) and we were able to get a bird's eye view of all the bugs. After finding differences in the results of our program and data on the website, it was necessary for us to dig deeper and find out bugs.



bqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1 We tried more

Depth	Nodes	
0	1	Depth: 1 Number of Positions: 20 Time taken: 1 ms
1	20	Depth: 2 Number of Positions: 400 Time taken: 19 ms
2	400	Depth: 3 Number of Positions: 8902 Time taken: 352 ms
3	8,902	Depth: 4 Number of Positions: 197281 Time taken: 7923 ms
4	197,281	Depth: 5 Number of Positions: 4865609 Time taken: 203318 ms
5	4,865,609	
6	<del>110,866,024</del>	
7	<del>3,105,881,000</del>	

Figure 8.5: Perf Test Results

intricate fen strings, that have been known to find bugs and obtained similar results.

Depth	Nodes	
1	14	Depth: 1 Number of Positions: 14 Time taken: 0 ms
2	191	Depth: 2 Number of Positions: 191 Time taken: 5 ms
3	2812	Depth: 3 Number of Positions: 2812 Time taken: 77 ms
4	43238	Depth: 4 Number of Positions: 43238 Time taken: 1177 ms
5	[6] 674624	Depth: 5 Number of Positions: 674624 Time taken: 18096 ms

Figure 8.6: On FEN: 8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - -

Depth	Nodes	
1	44	Depth: 1 Number of Positions: 44 Time taken: 2 ms
2	1,486	Depth: 2 Number of Positions: 1486 Time taken: 66 ms
3	62,379	Depth: 3 Number of Positions: 62379 Time taken: 2824 ms
4	2,103,487	Depth: 4 Number of Positions: 2103487 Time taken: 91041 ms
5	89,941,194	Depth: 5 Number of Positions: 89941194 Time taken: 5662793 ms

Figure 8.7: On FEN: rnbq1k1r/pp1Pbppp/2p5/8/2B5/8/PPP1NnPP/RNBQK2R w KQ - 1 8

### 8.3. Fixing Memory Leaks

Since, c++ requires us to manage our memory ourselves, untended memory leaks were bound to occur at some point because of our inexperience. These memory leaks included, forgetting to delete `SDL_textures`, `Mix_chunks`, and other objects that were allocated on the heap. **Solutions we implemented**

1. We implemented a Wrapper object called **Texture** which helped us ensure that all the textures allocated on the heap were destroyed.
2. We used clang to detect memory leaks.

### 8.4. Single Player Mode

As much as we wanted to implement single player mode in our project, it proved quite difficult because of the slow move generation. Even with some optimization, we realized that our structure was not equipped in working at higher depths. Thus we decided, that for the time being, the single player mode will use a random ai, that will select a move at random.

### 8.5. Speed And Optimization

In the later half of our project, we realized how crucial speed and optimizations are. But it was too late to overhaul our entire structure and try a different approach.

The problem with our method, was that all the processes were very brute force in nature and we were copying too many things.

1. We checked for checks in a brute force manner, that is, we ran a check for each move by the opponent to see if they could attack the king.
2. Filtering legal moves from pseudo illegal moves was also quite inefficient as it would look for checks even when there was no possibility of given one, and a lot of the checks were redundant.
3. While performing the perf test, checking for `*checks*` and ai testing , we were cloning a structure that would result in cloning/ copying a lot of variables which was a big performance hit.



In hindsight, we realized the structure in which we stored the boardState should have been modified with a stack for move tracking, which would mean we could just pop and push to the stack for performing changes instead of cloning the entire structure for performing changes on the chess board.

## 9. LIMITATIONS AND FUTURE ENHANCEMENTS

Even though a lot of work was put onto it, to make it as good as possible, there still are some limitations to the game. Some of these limitation include:

1. No move take back.
2. No online multiplayer, which would make it really fun to play with other friends.
3. Single Player move severely lacking in usability.
4. The AI in single player mode is completely random and does not pose much challenge.
5. Dynamic screen size, making the program work in multiple screen sizes.

The possible future enhancements are as follows.

1. A multiplayer mode, in which two people from any place can play with each other.
2. An overhaul in the way we store the state of the game with the help of stack to track moves. This would allow for move take backs and will make it so that we don't need to copy the state of the board again and again to make changes.
3. A better AI for single player, using minimax and other techniques.
4. Use of bitboard to represent the board

## 10. CONCLUSION AND RECOMMENDATIONS

In this way, we completed our project. The project was a great learning experience for our group. We learned about the library SDL2, features of c++ and most importantly the object oriented paradigm. We learned how powerful the object oriented paradigm is, with the help of key concepts like inheritance, polymorphism, encapsulation and abstraction. We were able to create interfaces that made the coding experience much cleaner.

We also learned a great deal about collaboration and team work from the project. We learned to use various industry tools such as *Notion*, to help with collaboration and workflow, *Github* for version control of our project, *LATEX* for typesetting documents, *Canva* for making presentation and *Lucidcharts* for making flowcharts.

The project taught us a great deal about the development cycle, including planning, analysis, development, testing and debugging. We learned the power of Object Oriented Programming paradigm in making efficient software.

## 11. REFERENCES

- Chess Programming Wiki
- Daya Sagar Baral and Diwakar Baral, “The Secrets of Object oriented Programming”, Bhundipuran Prakasan
- SDL Tutorials by Lazy Foo’ Productions
- SDL Game Development