Trabajo Final Informática Industrial

AJEDREZ en C++

Memoria del Trabajo

Integrantes:

Jaime Bustos Valera (55156)

Felipe de Gracia Ruiz (55199)

Pablo Núñez Hernández

Víctor Alcolea Medina (55106)

Nikita Zhukov (55525)

Índice de la memoria

1. Introducción

El proyecto tiene como objetivo la creación de un ajedrez con programación orientada a objetos, empleando los conocimientos obtenidos a lo largo del curso de informática Industrial.

La programación orientada a objetos permite una gran eficiencia y organización del código, pudiendo reutilizar código a lo largo del programa, esto permite una estructura modular.

Para la representación de las entidades clave del programa se utilizaran clases (Tablero, Piezas,etc..), mediante el uso de herencia se pueden establecer jerarquías entre los diferentes elementos del juego.

El polimorfismo permitirá implementar métodos comunes pero específicos para cada pieza, pudiendo así determinar por ejemplo como se moverá una pieza dependiendo de su tipo especifico.

En resumen, a lo largo de este trabajo se desarrollará un programa de ajedrez en C++ utilizando la programación orientada a objetos. Se aprovecharán los conceptos de herencia y polimorfismo para establecer una estructura modular y flexible, permitiendo una representación fiel de las reglas del juego.

2. Jugabilidad

Al iniciar el programa, este nos presenta un menú mostrando diferentes elementos:

- Título: “AJEDREZ”
- Fondo de pantalla: Se incluye una imagen muy creativa como fondo de pantalla con las herramientas de GLut.
- Modos de juego e instrucciones: Se indica al jugador que pulse la tecla “T” para empezar a jugar al modo tradicional, la tecla “S” para entrar al modo StarWars, la tecla “E” para el modo ETSIDI, para regresar desde cualquier modo al menú se utiliza la tecla “A”.
- Integrantes del grupo: en la parte inferior del menú se muestran el nombre de todos los participantes de este proyecto.

Figura 2.1. menú de inicio

Al pulsar la tecla “T” en el inicio del programa se abrirá el modo tradicional y como se puede observar en la imagen, en la pantalla contamos con el tablero, en el que se encuentran todas las piezas de ambos colores en sus posiciones iniciales. Una vez activado el modo, podremos comenzar a jugar.


Figura 2.2. Modo de juego Tradicional

El funcionamiento del juego es el siguiente:

- Se trata de un juego por turnos: empieza el jugador que controla las piezas blancas, luego va el jugador que controla las piezas negras, y así sucesivamente.
- Para realizar los movimientos se ha de clicar en una pieza y esta mostrara los movimientos que tiene permitido realizar según su tipo, después se ha de clicar en la casilla de destino para finalizar el movimiento de la pieza.
- Si un jugador intenta hacer una acción inválida, el programa no le permitirá realizar el movimiento y el jugador tendrá que escoger otro que sea fiel a las normas del juego.
- Se considera una acción invalida: aquella en la que se pide controlar una pieza que no es del mismo color que el turno que corresponde, aquella en la cual las coordenadas no estén contenidas en el tablero (coordenadas validas: $0 < x < 9$; 0

< y < 9), cualquier acción en la que se pida que una pieza realice un movimiento que ese determinado tipo de pieza no pueda realizar (por ejemplo, mover una torre en diagonal). Tampoco se permite que implique un movimiento en el que se salte una pieza con otra que no pueda saltar (es decir, intentar saltar una pieza con cualquiera que no sea un caballo), todo movimiento que implique comer a una pieza que no puede ser comida (es decir, una pieza del mismo color, o en caso de ser un peón, de cualquier color si nos movemos hacia adelante y no en horizontal).

- Si un jugador entra en estado de "Jaque", se alertará en la consola.
- Si un jugador entra en estado de "Jaque Mate", se cerrará el programa, dando por terminada la partida y mostrará una imagen final dependiendo del jugador que gano y una música para terminar la sesión.
- En caso de no querer terminar la partida se puede pulsar la tecla "A" para regresar al menú y elegir otro modo de juego.
- Si el modo de juego deseado es el de StarWars el jugador ha de pulsar la tecla "S". A continuación, se muestra una foto del modo de StarWars:

Figura 2.3. Modo de juego StarWars

- Este modo de juego hace referencia a la temática de StarWars, en la que las piezas blancas son representadas por personajes característicos del imperio, mientras que las piezas negras son personajes característicos del lado oscuro.
- De la misma manera de puede regresar al menú para iniciar el modo ETSIDI con la tecla "E":

Figura 2.4. Modo de juego ETSIDI

- Al realizar un movimiento maestro la partida finalizara y mostrara una de las siguientes imágenes dependiendo del vencedor:

Figura 2.5. Imagen que se muestra en pantilla cuando las piezas blancas ganan la partida.

Figura 2.6. Imagen que se muestra en pantalla cuando las piezas negras ganan la partida.

3. Estructura

En este apartado se comentará sobre la jerarquía de clases y la estructura del programa, el código se compone de los siguientes archivos de origen y de encabezado:

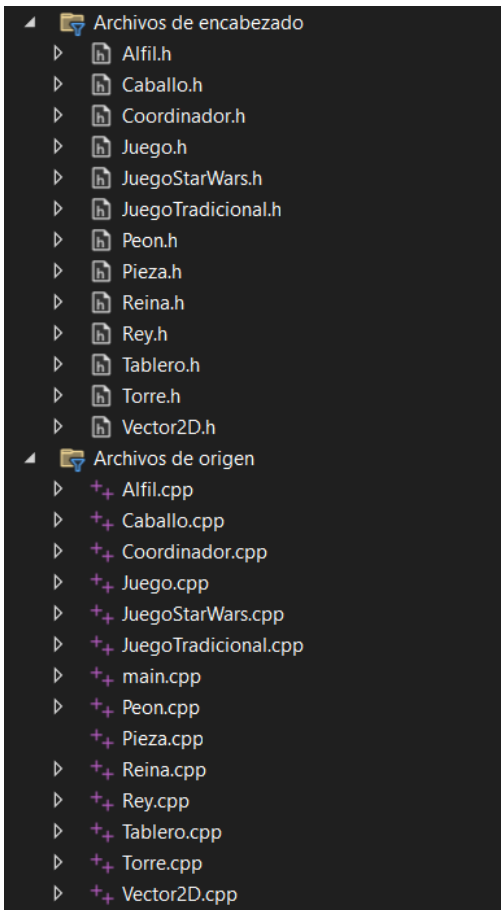

Figura 3.1 Archivos de encabeza de encabeza y de origen.

Como se puede observar el código consta de 13 archivos de encabezado y 14 de origen, estos se desglosan de la siguiente manera.

El control de la partida será administrado por una máquina de estados mediante los archivos de Coordinador.h/.cpp , los cuales mostraran un mensaje al inicio de la partida que permitirá elegir si queremos comenzar a jugar o salir del programa. Esta máquina de estados se encargará de inicializar el sistema cuando el usuario seleccione jugar, para ello necesitará tener control sobre los modos de juego con Juego.h y así decidir qué programa cargar dependiendo si se eligió JuegoStarWars.h/.cpp o JuegoTradicional.h/.cpp. Es en las clases de los modos donde se invoca a Tablero.h/.cpp.

En Tablero.h/.cpp se crea una matriz de punteros a piezas (Pieza.h/.cpp), la cual nos servirá para localizar a las piezas en tableroque y que posteriormente se inicializaran a sus debidos tipos.

En Pieza.h/.cpp, además de inicializar y dibujar el ajedrez (tablero y piezas). La clase Pieza.h/.cpp contiene los atributos (que indican el tipo de pieza, y su color) y los métodos comunes a todas las piezas. Entre esos métodos encontramos algunos métodos virtuales puros (que las clases derivadas heredarán y especificarán según cada caso), tratándose así de una clase abstracta. También habrá métodos que se definirán en la propia clase Pieza, pues su funcionamiento es el mismo independientemente del tipo de pieza. De la clase

Pieza.h/.cpp derivan nuestras piezas específicas, como pueden ser los peones (Peon.h/.cpp), los cuales reciben los atributos y métodos de nuestra clase base. En cada una de ellas se especializan los métodos que describen cómo cada tipo de pieza se muestra en la pantalla, y se define su patrón concreto de movimiento. Por último, la clase Vector2D.h/.cpp sirve para dar un formato (concretamente, en forma de coordenadas x e y) a todas las posiciones que se manejan en el juego, haciendo más fácil su tratamiento y manipulación (a través de métodos y sobrecargas de operadores).

4. Código del programa

Ahora vamos a comentar lo que hace cada una de nuestras clases explicando tanto los atributos como las funciones de cada una de ellas. Para ayudar a la explicación vamos a utilizar las partes del código que creemos imprescindibles para comprenderlas. El resto del código que no utilicemos en la explicación se puede acceder a él mediante nuestro repositorio.

4.1 Coordinador

Comenzaremos en la cima de la jerarquía, la primera clase que se muestra es la de Coordinador.h/.cpp:

```
#pragma once

#include"Tablero.h"
#include "Juego.h"

enum Estado { INICIO, JUEGO, MATE_AL_BLANCO, MATE_AL_NEGRO };

class Coordinador
{
	public:
		Coordinador();
		virtual ~Coordinador();

		void teclaEspecial(unsigned char key);
		void Tecla(unsigned char key);
		void seleccionaPosicion(int, int);
		void dibuja();
		virtual void fichero();



	protected:
Juego* juego;
		Estado estado;
		Sprite Mate_al_negro{ "lib/imagenes/MATE_NEGRO_DALL-E.png", 5, 5, 25, 20 };
		Sprite Mate_al_blanco{ "lib/imagenes/MATE_BLANCO_DALL-E.png", 5, 5, 25, 20
};

};
```

Atributos:

- Esta clase consta de dos Sprite que se mostraran al finalizar la partida, un objeto puntero de Juego y los distintos estados.

Métodos:

- El constructor inicializa la máquina de estados con el estado de inicio para que muestre el menú. Y no deja ningún modo de juego asignado.
- El destructor.
- El método **SeleccionaPosicion()**: este método se encarga de llevar la máquina de estados a los estados de MATE en caso de realizarse un movimiento que terminase la partida.
- El método **dibuja()** se encarga de dibujar diferentes entornos dependiendo del estado del programa.
- El método **Tecla()**: modifica el estado del programa en función de la tecla presionada en el teclado.

4.2 Juego

Esta clase es la que enlaza nuestro coordinador con nuestro tablero. Dependiendo de lo que elija el jugador cuando se encuentre en el menú de inicio, la clase juego creara un modo u otro, gracias a que el coordinador le dice que modo de inicializarse.

```
#pragma once

#include "Tablero.h"

class Juego {
 public:

	Juego ();
	~Juego ();
	virtual void dibuja ()=0;
	virtual int mueve (int, int) = 0;

 protected:
	Tablero* tab;

};
```

Esta clase es virtual pura y actúa como interfaz para determinar el tipo de modo de juego que el jugador ha seleccionado.

Ahora vamos a mostrar el código de un modo de ellos ya que al respetarse los conceptos de polimorfismo y de herencia el código es muy parecido y se distinguen cada uno de ellos por pequeños detalles.

A continuación, veremos **JuegoTradicional.**:

```
JuegoTradicional::JuegoTradicional ()
{

	tab = new Tablero(Tradicional);
	std::cout << "\t----------Bienvenid@s a Juego Tradicional----------\t"
<< std::endl;
}
```

Los métodos de la clase son los siguientes:

- El constructor inicializa el tablero con el modo tradicional.
- El método **dibuja ()** se encarga de darle el fondo característico a la escena.
- El método **mueve ()** aplica un sonido característico en caso de clicar fuera de los límites del tablero, también se define la lógica de los turnos y se comprueba el mate, en caso de que el jugador blanco de mate al negro devuelve -1 y si es al revés devolverá 1, en caso de no devolver ninguno de esos dos casos la partida continua.

4.3 Tablero

La clase Tablero está compuesta de una arrays de piezas y se encarga de la lógica principal del programa:

```
#pragma once

#include <iostream>
#include <fstream>
#include "freeglut.h"
#include "Pieza.h"
#include "Peon.h"
#include "Rey.h"
#include "Reina.h"
#include "Torre.h"
#include "Alfil.h"
#include "Caballo.h"
#include"Vector2D.h"

constexpr int filas = 8;
constexpr int columnas = 8;


class Tablero
{

	char numeroALetra[8];
	protected:
		Vector2D* pos_origen, * pos_final, * origen;
		int turno = -1;  //-1=turno blancas; 1= turno negras
		int x_org = -1, y_org = -1; //esto antes era 100 100
		int coger = 1;

	public:

		Pieza* tab[columnas][filas];	//Matriz de punteros a piezas

		Tablero(EstiloModoJuego modo);
		~Tablero();
		void dibuja();


		bool hay_pieza(int, int);
		void quien_soy(Pieza*);

		bool coger_posiciones(int, int, int, int);
		int comprobar_jaque(Pieza*[columnas][filas]);
		int comprobar_mate();
		char convertirPosicion(int) const;

		//// Metodos getters para que esten en protegidos nuestras variables
		std::string print_turno();
		int get_turno();
		int get_coger();
		int get_x();
		int get_y();
		/// Metodos setters para dar valores
		void set_turno(int);
		void set_coger(int);
		void set_x(int);
		void set_y(int);

};
```

Atributos:

- Tres punteros de tipo Vector2D responsables de recoger las posiciones de las casillas del tablero. Los vectores llamados pos_origen y pos_final serán los argumentos que le llegarán a nuestras piezas.
- Variable para seguir el cambio de turno de tipo int
- Posición de la casilla del tablero seleccionada de tipo int
- Variable para seleccionar la pieza de tipo int

- Variable auxiliar para mostrar la casilla por pantalla
- Matriz de punteros a pieza

Métodos:

- Dentro del constructor se inicializan las piezas blancas y negras en sus respectivas casillas, las posiciones inicial y final de cada pieza y las variables auxiliares.

```
Tablero::Tablero(EstiloModoJuego modo)
{


	for (int i = 0; i < filas; i++)
	{
		for (int j = 0; j < columnas; j++)
		{
			tab[i][j] = nullptr;
		}
	}
	//peones
	for (int j = 0; j < columnas; j++)//EL VECTOR VA DE 0 A 7, POR ESO SE
PONE 6 Y 1 EN LAS COORDENADAS
	{
		tab[1][j] = new Peon(Pieza::BLANCO, modo); // modificar el resto
		tab[6][j] = new Peon(Pieza::NEGRO, modo);
	}
	//creacion de las demas piezas
	 //caballo

	tab[0][1] = new Caballo(Pieza::BLANCO,modo);
	tab[0][6] = new Caballo(Pieza::BLANCO,modo);
	tab[7][1] = new Caballo(Pieza::NEGRO,modo);
	tab[7][6] = new Caballo(Pieza::NEGRO,modo);
	//reyes
	tab[0][4] = new Rey(Pieza::BLANCO,modo);
	tab[7][4] = new Rey(Pieza::NEGRO,modo);
	//reinas
	tab[0][3] = new Reina(Pieza::BLANCO,modo);
	tab[7][3] = new Reina(Pieza::NEGRO,modo);
	//torres
	tab[0][0] = new Torre(Pieza::BLANCO,modo);
	tab[0][7] = new Torre(Pieza::BLANCO,modo);
	tab[7][0] = new Torre(Pieza::NEGRO,modo);
	tab[7][7] = new Torre(Pieza::NEGRO,modo);
	//Alfiles
	tab[0][2] = new Alfil(Pieza::BLANCO,modo);
	tab[0][5] = new Alfil(Pieza::BLANCO,modo);
	tab[7][2] = new Alfil(Pieza::NEGRO,modo);
	tab[7][5] = new Alfil(Pieza::NEGRO,modo);

	pos_origen = new Vector2D(0, 0);
	pos_final = new Vector2D(0, 0);

	numeroALetra[0] = 'A';
	numeroALetra[1] = 'B';
	numeroALetra[2] = 'C';
	numeroALetra[3] = 'D';
	numeroALetra[4] = 'E';
	numeroALetra[5] = 'F';
	numeroALetra[6] = 'G';
	numeroALetra[7] = 'H';
}
```

- En el destructor se libera la memoria reservada que hemos utilizado (new).
- El método **dibuja()**:Esta función se encarga de dibujar el tablero de ajedrez en un contexto gráfico utilizando la biblioteca GLUT. Dentro del bucle, hay una condición que verifica si la casilla actual coincide con la posición seleccionada (x_org y y_org) y si la pieza en esa casilla tiene el mismo color que el turno actual. En ese caso, se resalta la casilla seleccionada con un polígono de color rojo. Además, se realiza un bucle adicional para encontrar las casillas disponibles para mover la pieza seleccionada.
- Método hay_pieza(int x, int y) devuelve si la posición en una casilla concreta del tablero está ocupada
- Método quien_soy(Pieza* tab) devuelve el tipo y color de la pieza que se le pasa como puntero.
- Método coger_posiciones(int x_org, int y_org, int x_dest, int y_dest) recibe las coordenadas de origen y destino. Se verifica si la pieza en las coordenadas de origen coincide con el color del turno actual y si el movimiento desde pos_origen a pos_final es válido utilizando el método validar_mov() de la pieza correspondiente. Además, se verifica si no se genera un jaque utilizando el método comprobar_jaque() del tablero. Si se cumplen todas las condiciones anteriores, se realiza una simulación del movimiento de la pieza en una matriz de piezas copia, copiando el contenido actual del tablero. Si después del movimiento realizado no hay jaque, se imprime la posición de origen y destino en el flujo std::cout y se guarda la misma información en el archivo de salida Partida.txt. Se verifica si el movimiento realizado resulta en un jaque mate utilizando el método comprobar_mate(). Si se cumple la condición, se imprime un mensaje indicando el final de la partida y el ganador correspondiente.

```
Else if ((tab[y_org][x_org]->get_Color()==turno) && (tab[y_org][x_org]
->validar_mov(pos_final, pos_origen, *this))
```

- Se realiza el movimiento real en el tablero, reemplazando la pieza en las coordenadas de destino con la pieza en las coordenadas de origen y asignando un puntero nulo a las coordenadas de origen.
- Método comprobar_jaque: lo primero que hace este método es recorrer la matriz de punteros a pieza para localizar a ambos reyes en ella. Una vez localizados se guardan sus coordenadas en dos vectores (reyb y reyn), uno para cada rey. Posteriormente, se comprueban los jaques por separado para el blanco y el negro, devolviendo un -1 si hay jaque al blanco, un 1 si hay jaque al negro y un 0 si no hay jaque. Para ello se recorre la matriz buscando piezas del color correspondiente, y empleando el método validar movimiento, mandándole como posición final el vector de posición del rey. Si la función valida el movimiento significa que el rey está al alcance de esa pieza y hay jaque.
- El método comprobar_mate crea una copia de la matriz de punteros a pieza donde se guardará la posición del juego para poder simular todos los movimientos posibles de las piezas en jaque en la matriz original y comprobar si estas salen de esa situación. Se localizan las piezas que pueden mover, se efectúa el movimiento para cada uno de ellos posible y con la nueva posición se vuelve a comprobar el jaque con el método

anteriormente descrito. Si ya no hay jaque, significa que hay un movimiento posible para salir de jaque, por lo que no es mate y se devuelve la posición del tablero a la original y la función devuelve un 0, mientras que si sigue habiendo jaque se devuelve el tablero a la posición anterior a efectuar el movimiento y se continúa simulando movimientos. Si una vez simulados todos los movimientos posibles de todas las piezas, continua la situación de jaque, significa que hay jaque mate y se devuelve un 1 o un -1 en función del color ganador.

4.4 Vector2D

Esta clase lo único que hace es que guardar las coordenadas que le pasamos cuando pulsamos con el ratón en un vector de dos enteros. Gracias a esta clase nos ahorramos la inicialización de enteros y nos servirá con llamada la clase vector y crear uno de ellos para poder jugar con las posiciones.

```
class Vector2D
{
public:
	int x;
	int y;
	Vector2D(int x, int y);
```

4.5 Pieza

La siguiente clase es una clase genérica de una pieza de ajedrez. Se esta clase se heredan todas las piezas que implementamos en nuestro proyecto. Como se trata de una clase abstracta su archivo de origen está vacío porque todas los métodos son virtuales.

```
#pragma once
#include "ETSIDI.h"
#include "freeglut.h"
#include "Vector2D.h"
#include<iostream>

enum EstiloModoJuego{Tradicional, StarWars,Etsidi};
class Tablero;


class Pieza
{
	public:

		typedef enum {REY = 0, PEON, ALFIL, REINA, CABALLO, TORRE } TipoPieza;
		typedef enum {BLANCO = -1, NEGRO = 1 } Color;
	protected:
		TipoPieza pieza;// ya podemos acceder a cada tipo de las piezas
		Color color;// ya podemos acceder al color de ellas, o blanco o negro
		EstiloModoJuego modo;
	public:

		Pieza(Color c, TipoPieza p, EstiloModoJuego _modo) :color(c), pieza(p),modo(_modo)
{};

		TipoPieza getPieza() { return pieza; }
		Color get_Color() { return color; }

		//Metodo virtuales
	//como le pasamos las coordenadas a piezas desde tablero, 1.creamos  un objeto tablero o se
lo pasamos como arguemnto de	validarmovimientos.
		virtual bool validar_mov(Vector2D*, Vector2D*, Tablero& tablero) = 0;
		virtual void dibuja() = 0;
		virtual std::string getTipo() = 0;
		virtual std::string getColor() = 0;
};


};
```

Atributos:

- Los enumerados TipoPieza y Color son los atributos básicos de tipo enumerados con las que se crearán las piezas a continuación. Son responsables de qué tipo de pieza vamos a crear y de qué color serán. Ambos atributos se definen como protected para que exclusivamente puedan utilizarse dentro de la misma clase u otras clases heredadas de la clase base.
- Dentro de la misma clase Pieza se instancian los atributos mencionados anteriormente.
- Se define otro atributo de tipo enumerado pero esta vez global que sea responsable de cambiar las piezas dependiendo del modo de juego que se utilice EstiloModoJuego. Al igual que antes esta variable se instancia en la propia clase.

Métodos:

- El constructor de pieza inicializa la clase con los atributos previamente mencionados.

Métodos virtuales puros:

- `dibuja()`: Este método virtual puro se encarga de dibujar la pieza en la interfaz gráfica.
- `std::string getTipo()`: Este método virtual puro devuelve una cadena de texto que representa el tipo de la pieza.
- `std::string getColor()`: Este método virtual puro devuelve una cadena de texto que representa el color de la pieza.
- `TipoPieza getPieza()`: función getter que devuelve el tipo de pieza.
- `Color get_Color()`: función getter que devuelve el color de la pieza.
- "validar_mov()": Este método virtual puro se encarga de validar si un movimiento de la pieza es válido. Recibe un puntero a la posición final, un puntero a la posición inicial y una referencia al objeto "Tablero". La implementación de este método debe verificar si el movimiento es válido según las reglas específicas de cada tipo de pieza.

4.5.1 Peón

La declaración de la clase comienza con la línea class Peon : public Pieza, lo cual indica que peón es una clase derivada de la clase base Pieza. Los métodos implementados son sobrecargados de la clase base Pieza.

Vamos a explicar todos los atributos y métodos que son comunes en todas las piezas para que quede claro su funcionamiento:

```
#pragma once
#include "Pieza.h"
#include<iostream>
using ETSIDI::Sprite;

class Peon :public Pieza
{

public:
	Peon(Color c,EstiloModoJuego modo)
:Pieza(c, PEON,modo) {};
	void dibuja();
	//bool validar_mov(int, int, int, int);
	bool validar_mov(Vector2D*, Vector2D*,
Tablero&);
	void getTipoPieza();
	std::string getTipo();
	std::string getColor();

};
```

Atributos:

Los mismos atributos que se han declarado en clase *Pieza* se usan en la clase *Peon*.

Métodos:

- Se define la función getTipo() que devuelve la cadena de texto "PEON". Esta función es una implementación de la función virtual pura declarada en la clase base "Pieza".
- Se define la función getColor() que utiliza una instrucción switch para determinar el color de la pieza y devolver la cadena de texto correspondiente ("BLANCO" o "NEGRO"). También se maneja un caso por defecto para devolver "NULL" si el color no es reconocido.
- Se define la función validar_mov() que toma dos punteros Vector2D y una referencia a un objeto Tablero como parámetros. Esta función se encarga de validar el movimiento del peón en el tablero de ajedrez. La implementación de la función verifica las condiciones de movimiento recto y movimiento diagonal, y comprueba si hay piezas en el camino o si se puede capturar una pieza enemiga.

```
        //Movimiento de avance recto
        if (abs(pos_origen->x - pos_final->x) == 0) {
            //movimiento recto de peones blancos
            if (color == -1 && (pos_final->y - pos_origen->y == 1
|| ((pos_final->y - pos_origen->y == 2) && (pos_origen->y == 1))))
{
                while (y != pos_final->y) {
                    //avanza a la siguiente casilla
                    y += dy;

                    //Si la casilla que analiza no apunta a puntero
nulo; hay pieza
                    if (tablero.tab[y][x] != nullptr)
                        return false;
                }
                return true;
            }
```

Por ejemplo, en esta parte del código, es como se hace el movimiento hacia adelante del peón blanco.

- Se define la función getTipoPieza() que escribe en un archivo de salida el tipo de pieza ("Peon blanco" o "Peon negro") dependiendo del color de la pieza. El archivo de salida es "Partida.txt" ubicado en la carpeta "lib/ficheros".

4.5.2 Caballo

Todas las clases van a tener el mismo cuerpo en el archivo de encabezado por lo que esa parte de código la vamos a omitir y vamos a explicar la función validar movimiento de cada pieza y ha explicar el funcionamiento de estos.

Como todo sabemos el movimiento del cabello es único y se caracteriza por ser no lineal. El caballo se mueve en forma de “L” sobre el tablero, es decir, un salto de dos casillas en una dirección (horizontal o vertical) y luego un salto de una casilla en una dirección perpendicular a la anterior (esta combinación puede ser en cualquier orden).

Encima esta pieza puede saltar las demás piezas, por lo que significa aún más nuestro código ya que no tenemos que comprobar las piezas que hay en su trayectoria.

En el código lo que hemos realizado que si la diferencia absoluta entre las coordenadas x e y del punto de inicio y el punto de destino es (2,1) o (1,2) el movimiento es válida.

```
bool Caballo::validar_mov(Vector2D* posfinal, Vector2D* posini, Tablero&
tablero)
{

    if ((abs(posfinal->x - posini->x) == 2 && abs(posfinal->y - posini->y) ==
1)|| (abs(posfinal->x - posini->x) == 1 && abs(posfinal->y - posini->y) == 2))
    {
        if (tablero.tab[posfinal->y][posfinal->x] == nullptr)//si no hay pieza
            return true;
        else    if    (tablero.tab[posfinal->y][posfinal->x]->get_Color()    !=
tablero.tab[posini->y][posini->x]->get_Color())//si son del  mismo color
            return true;
        else {
            return false;
        }
    }


    else {
       /* std::cout << "movimiento no valido\n";*/
        return false;
    }
}
```

4.5.3 Alfil

El alfil es una de las piezas que se mueve en forma diagonal sobre el tablero (como por ejemplo el rey y la reina). El alfil puede moverse cualquier número de casillas en diagonal en cualquier dirección (arriba, abajo, izquierda o derecha), siempre que no haya obstáculo en su camino.

Lo que hemos realizado nosotros es un bucle que recorre la trayectoria diagonal desde la posición inicial hasta la posición final. En cada iteración se verifica si hay una pieza en la casilla que se está analizando. Si se encuentra una pieza y es diferente color al del alfil, en ese caso se considera movimiento valido y se devuelve true.

Además, se realiza una verificación adicional al final de bucle para asegurarse de que el movimiento del alfil sea una trayectoria diagonal válida.

En resumen, si la diferencia absoluta entre las coordenadas x e y de las posiciones inicial y final no es la misma, se considera un movimiento inválido y se devuelve false.

```
bool Alfil::validar_mov(Vector2D* posfinal, Vector2D* posini,
Tablero& tablero)
{
    int no_valid = -1;

    //Establece hacia donde se mueve el alfil, 1=arriba o
derecha; -1=abajo o izq
    int dx = (posfinal->x > posini->x) ? 1 : -1;
    int dy = (posfinal->y > posini->y) ? 1 : -1;
    //Empieza en el origen
    int x = posini->x;//no es necesario
    int y = posini->y;

    //Recorre toda la trayectoria diagonal
    while (x != posfinal->x && y != posfinal->y) {
        //avanza a la siguiente casilla
        x += dx;
        y += dy;

        //Si la casilla que analiza no apunta a puntero nulo; hay
pieza
        if (tablero.tab[y][x] != nullptr) {

            //Si la pieza en la trayectoria es en la casilla
destino y es de otro color la come
            if (x == posfinal->x && y == posfinal->y &&
tablero.tab[posfinal->y][posfinal->x]->get_Color() !=
tablero.tab[posini->y][posini->x]->get_Color()) {

                return true;
            }

            else {
                // Hay una pieza en el camino
                no_valid = 1;
                return false;
            }
        }
    }

    //Si no ha encontrado piezas, o puede comer, efectua el
movimiento
    if (no_valid != 1)
        if (abs(posfinal->x - posini->x) != abs(posfinal->y -
posini->y)) {
           /* std::cout << "movimiento no valido\n";*/
            return false;
        }
        else
            return true;

}
```

4.5.4 Reina

La reina es una pieza importante que puede moverse en todas las direcciones (horizontal, vertical y diagonal). Lo que hemos combinado son las características de la torre y el alfil.

Lo que hemos realizado nosotros para determinar la dirección del movimiento es mediante las diferencias de las coordenadas x e y entre las posiciones finales e inicial.

Si el movimiento es en línea recta en esa dirección (horizontal o vertical) la diferencia será 0, mientras que si la diferencia no es la misma entre las coordenadas entonces el movimiento es diagonal.

Siempre se verifica si hay alguna pieza en el camino ya que no se pueda saltar como si lo hacia nuestro caballo. Además, se comprueba si la posición final contiene una pieza ya que si es diferente al color de la reina de la que se está moviendo se devolverá true.

```
bool Reina::validar_mov(Vector2D* posfinal, Vector2D*
posini, Tablero& tablero)
{
   /* std::cout << "soy reina" << std::endl;*/
    int dx = (posfinal->x > posini->x) ? 1 : -1;
    int dy = (posfinal->y > posini->y) ? 1 : -1;
    if (posfinal->x == posini->x)dx = 0;
    if (posfinal->y == posini->y)dy = 0;

    if (abs(posfinal->x - posini->x) != abs(posfinal->y -
posini->y)) {//si no es diagonal
        if (dx != 0 && dy != 0){//si no es recto
           /* std::cout << "movimiento no valido\n";*/
            return false;
        }

    }
    int x = posini->x;
    int y = posini->y;
    int full = 0;
    while (y != posfinal->y || x != posfinal->x)
    {
        x += dx;
        y += dy;
        if (tablero.tab[y][x])//si hay una pieza
            full++;
    }
    if (tablero.tab[posfinal->y][posfinal->x] !=
nullptr)//si en la ultima casiilla hay pieza
    {
        if (tablero.tab[posfinal->y][posfinal->x]-
>get_Color() != tablero.tab[posini->y][posini->x]-
>get_Color())
        {//si el objetivo es de otro color
            full--;
        }
    }

    if (full == 0)
        return true;
    else
        return false;
}
```

4.5.5 Rey

El rey es la pieza central y más importante del juego. Para explicar rápidamente su principal objetivo es evitar ser captura y mantenerse a salvo, por ello se puede mover en cualquier posición.

La validación movimiento del rey es bastante sencilla, ya que se comprueba la diferencia absoluta entre las coordenadas x e y de la posición final e inicial y tiene que ser menor que 2. Si esto se cumple es que el rey se mueve una casilla por lo que retornara true la función.

Como todas las piezas comprueba que si la posición destina contiene una pieza y es de distinto color del rey que se va a mover, te dirá que el movimiento es válido.

```
bool Rey::validar_mov(Vector2D* posfinal, Vector2D*
posini, Tablero& tablero)
{

    if ((abs(posfinal->x - posini->x) < 2) &&
(abs(posfinal->y - posini->y) < 2))
    {
        if (tablero.tab[posfinal->y][posfinal->x] ==
nullptr)//si la casilla está vacía, no hay que
comprobar color de pieza
            return true;
        else if (tablero.tab[posfinal->y][posfinal-
>x]->get_Color() != tablero.tab[posini->y][posini-
>x]->get_Color())
            return true;
        else
            return false;
    }
    else
        return false;
}
```

4.5.6 Torre

El movimiento de la torre es en línea recta, es decir, horizontal o vertical siempre que no haya ninguna pieza en su propia trayectoria.

Para implementar esta condición, como hemos hecho en las demás piezas, calculamos la diferencia absoluta entre las coordenadas x e y de las posiciones finales e iniciales. Si la diferencia es 0 es que el movimiento es recto. Como se ha explicado antes, la torre no puede saltar a ninguna pieza por lo que se comprueba en cada iteración si hay piezas en las múltiples trayectorias.

```
bool Torre::validar_mov(Vector2D* posfinal, Vector2D* posini,
Tablero& tablero)
{
    int dx = (posfinal->x > posini->x) ? 1 : -1;
    int dy = (posfinal->y > posini->y) ? 1 : -1;
    if (posfinal->x == posini->x)dx = 0;
    if (posfinal->y == posini->y)dy = 0;

    if (dx != 0 && dy != 0)//hay incrmento enlos dos ejes=mal
        return false;

    int x = posini->x;
    int y = posini->y;
    int full = 0;
    while (y!=posfinal->y || x!=posfinal->x)
    {
        x += dx;
        y += dy;
        if (tablero.tab[y][x])//si hay una pieza
            full++;
    }
    if (tablero.tab[posfinal->y][posfinal->x]!=nullptr)//si en la
ultima casiilla hay pieza
    {
        if (tablero.tab[posfinal->y][posfinal->x]->get_Color() !=
tablero.tab[posini->y][posini->x]->get_Color())
        {//si el objetivo es de otro color
            full--;
        }
    }

    if (full == 0)
        return true;
    else
        return false;

}
```

5. Conclusión

En este trabajo de la programación de un ajedrez en C++, hemos aplicado y afianzado los conceptos claves y necesarios de la programación orientada a objetos, como es el polimorfismo, la herencia y el encapsulamiento.

El polimorfismo lo hemos utilizado en nuestro código a través de las clases y métodos virtuales. En este caso tenemos la clase Pieza que va a ser nuestra clase base ya que de ella van a heredar las distintas piezas. Gracias a esto se permite tratar a todos los tipos de pieza como objetos de la clase base y así poder crear luego el array de punteros en el Tablero. Además, con los métodos virtuales permiten que los comportamientos de cada pieza sean sobrescritos y así conseguir este fenómeno de polimorfismo.

En cuanto a la herencia, como se ha comentado antes, ha sido imprescindible para implementación de la clase Pieza como también para la clase juego. Gracias a ella se ha ahorrado la reutilización de código en distintas partes de nuestro trabajo y facilita la agregación de por ejemplo de un modo distinto.

Por último, el grado de encapsulamiento que hemos agregado a nuestras clases ha sido bastante tedioso ya que para ello hemos ocultado los atributos de la clase para que no se puedan manipular y acceder a ellos de manera no controlada.

En resumen, este trabajo es un claro ejemplo de lo que es la programación orientada objetos ya que se han afianzado los conceptos representativos que son polimorfismo, herencia y encapsulamiento. Gracias a estos conceptos se ha facilitado la implementación de reglas especificas de cada pieza y el desarrollo completo de nuestro ajedrez.