**Summary**: Our project was a digital two-player chess game with a timer as well as changing it to the variant atomic chess. The timer is a standard chess timer which gives each player a certain amount of time to make all their moves. If a player’s time runs out, they lose the game. Atomic chess has all the same rules as regular chess, except that when a piece is taken, it explodes all the adjacent pieces, excluding pawns. Also, in atomic chess, the game ends by blowing up the king, rather than checkmate.

**Inputs**:

The inputs for the timer function is an unsigned long which represents in minutes how much time each player has to start out with. When running the game, the program will ask the user for 2 pairs of a char and an int. The first pair of char and int represents the column letter and row number of the piece you want to move. The second pair of char and int represents the column letter and row number of the square you want to move to. Also, when a pawn promotes by calling the method Board::Promote\_Pawns(), the program will ask for a character which represents which piece you want the piece to promote into.

**Outputs**:

The main output of the program comes from displaying the game board. This is done by calling Board::Display\_Board() and shows the position of all the pieces on the board. The program also outputs the amount of time each player has left and outputs when the game is over, either through a player running out of time, or through a king being captured.

**Relationships Between Files**:

The main file first creates two instances of the timer class which is defined in timer.cpp which keeps track of how much time each player has remaining. The main file then utilizes board.cpp to create an instance of the board class. It then initializes the board by calling the method Board::Initialize\_Board() which utilizes Piece.cpp and creates pointers to piece objects. During the game after each move, the main file will show how much time is remaining in each timer object and displays the board object.

**Relationships Between Classes & Data Structures**:

The main data structure of the program is a set of arrays White\_Pieces and Black\_Pieces which store pointers to Piece objects. Inside the piece object are data members including the color of the piece, the type of piece, whether or not it has moved, and its location. Board::Initialize\_Board() creates pointers to the 32 pieces of a chess game and stores them in the arrays. Afterwards, methods such as Board::GetPieceAt(int x, int y) and Board::King\_Exists() iterate through the array to check on the data of the pieces to do different things. GetPieceAt loops through the White\_Pieces and Black\_Pieces arrays and returns a pointer to the piece at that location while King\_Exists() loops through the arrays to see if the kings are alive.

**Interesting Core Algorithms (I don’t know if this is applicable to ours so we can delete it if we need to)**:

**Functions of Interest**:

timer1.start() - starts timer1

timer1.stop() - stops timer1.

timer1.reset() - resets timer1.

timer1.getTime() - gets the amount of time elapsed on timer1; to figure out how much time is remaining, we subtract the value returned from the number of minutes on the timer at the beginning of the game.

void Board::InitializeBoard() - Creates pointers to all the pieces at the starting position of a chess game and stores them in arrays White\_Pieces and Black\_Pieces.

Piece\* Board::GetPieceAt(int x, int y) - Loops through arrays White\_Pieces and Black\_Pieces of the board object and checks their location. If they match the x and y coordinates in the argument, it returns a pointer to that piece.

void Board::DisplayBoard() - Displays an 8x8 representation of the board where the pieces are represented by their piece\_symbol. It fills in empty squares with \* and displays the column letters and row numbers along the bottom and left edges to make it easier to visualize the board.

bool Board::MovePiece(int x\_current,int y\_current,x\_dest,y\_dest) - First calls Board::CheckMoveLegal(Piece\* piece, int x\_dest, y\_dest) to check if the move is legal, by comparing the type of piece and the direction it is moving in. It also calls Board::FriendlyPieceAt(Piece\* piece, int x\_dest, int y\_dest) which checks if the piece at the inputted x and y is the same color as the inputted piece. If the move passes both functions, it then makes the move by updating the position of the moved piece. If it captures a piece, it changes the location to (999,999) as well as all the pieces within 1 square of the destination excluding pawns which is the “atomic” part of the atomic chess game.

Void Board::PromotePawns() - Loops through White\_Pieces and Black\_Pieces arrays and checks if there are any white pieces on the eighth row or any black pieces on the first row. If there are any, it asks the user to enter a char Q,R,N,or B which then switches the piece\_type member of that pawn to the corresponding piece by calling Queen\_Pawns(), Rook\_Pawns, etc.

int Board::King Exists() - Loops through White\_Pieces and Black\_Pieces arrays and checks what the locations are of the kings of both colors. A location of (999,999) means they have been captured. If only the black king is captured, the method returns 1, if only the white king is captured, the method returns 2, if both kings are captured, the method returns 2, and if neither king is captured, the method returns 0. This method is used to determine if the game should end.

**Challenges**:

Our timer appears to be specific to Linux, as it only works on Macbooks in our demonstration of it; and it does not work on Windows in our attempts to make it run. This was an oversight on our part.

**Justification**:

Our project took a common, well-documented example of how classes in C++ can be utilized and added to it by implementing another class into the program. In order to make this program work, we had to both incorporate a new class into an already-working program, as well as into existing classes, without compromising their functionality or efficiency. Moreover, making our timers functional required the ability to make our timers a) duplicable, and b) separate from one another. In other words, we needed to not only create a timer, but also figure out how to stop and start multiple timers without unpausing the opposite timer (since in a true competitive chess match, the clock is wired such that one clock stopping automatically starts the other one).