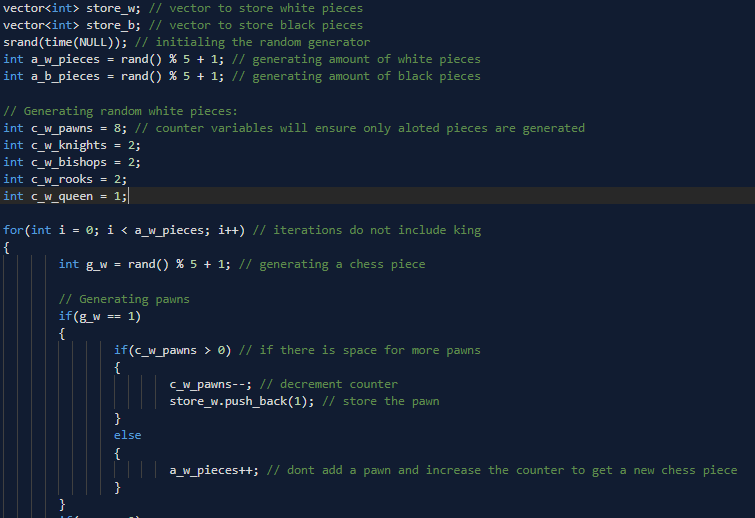
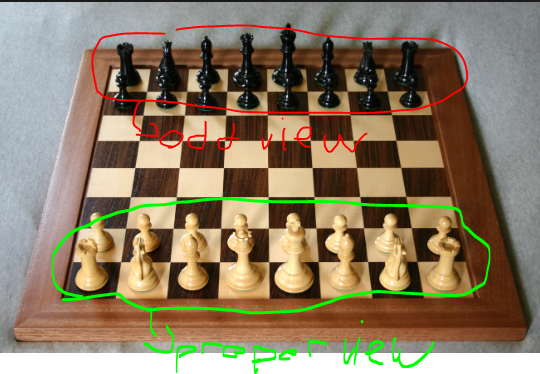
**Introduction:**

For my ICS4UI summative I created a two player chess game. Like most chess games, mine is played between two “live” players with all the functions of a regular chess game, including dynamic player controls, live movement display and the incorporation of chess rules (piece captures, game endings). What makes my chess game different is that it uses a TCP (Transmission Control Protocol) socket to connect both players. The game can be played from the same computer or different computers, as long as the Visual Studio IDE is installed, and that there is an internet and TCP connection setup (Details covered in *How Code Works*).

**Project Journey:**

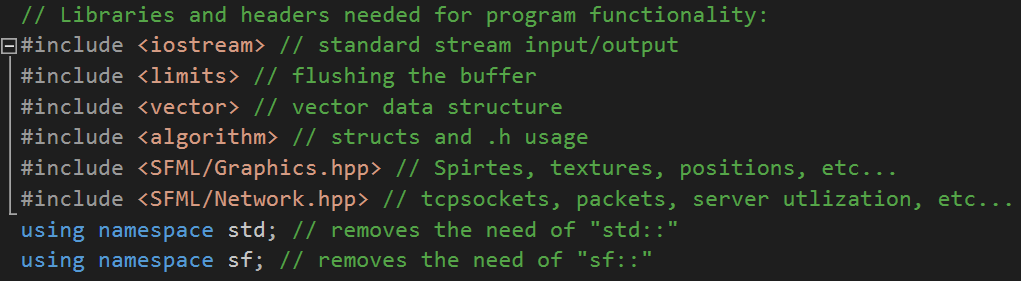
From the beginning to the end of this project, there have been many changes in code revisions that changed the structure of the entire program. Originally this project was developed as an Ai program aimed to solve a game of chess during its end game scenario. This was to be achieved by using a Min/Max algorithm for Ai decisions whether it be for the Ai to aim to capture a player’s chess piece or for the ai to defend one of it’s own chess pieces. I would then implement a A\* search algorithm so that the Ai can actually find the best way to go about its decision (attack or defend), such as bringing up one piece or using two chess pieces simultaneously. I began the project development using SDL. Soon however, I made the switch to SFML due to the 

many issues that arises with my use of SDL. While developing the project as a game solver, I soon decided that instead of focusing on a player vs Ai game, I wanted to work on a two player chess game instead. There were some changes I made to my project because of this. I abandoned some research and code from concepts such as the A\* search algorithm, the min/max algorithm, as well as the end game generator, that I needed to create the end game

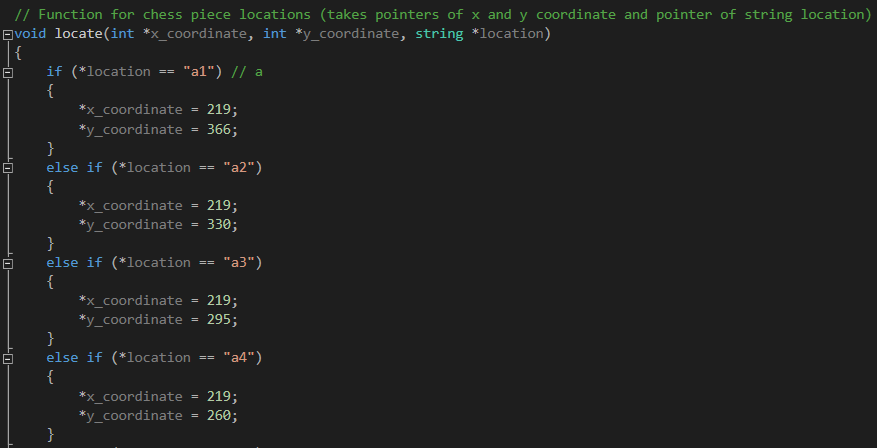
scenario of a chess game. I continued however, to use the chess rules and location generation functions, that I had developed as well as the mouse input functions (Details covered in *How Code Works*). Because the game would be played between two players, I wanted both players to be able to play on their own board with their own perspective instead of playing on the same board with one player ending up with an odd game perspective (eg. opposing player facing downwards). Due to this I began looking into internet connections in two player games. I researched TCP and UDP connections and experimented with both protocols. Eventually I chose to implement TCP sockets (due to its reliability), into my game to create a server-client connection between the two chess players. 

**How the Code Works:**

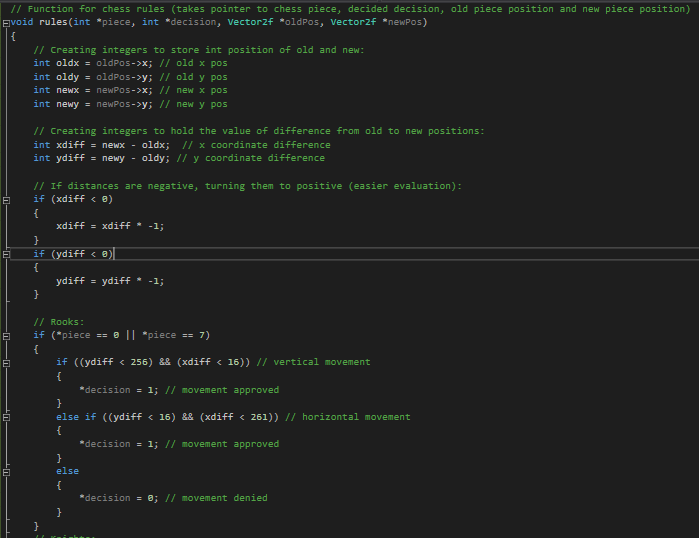
I will explain the code of my project code in sub categories.



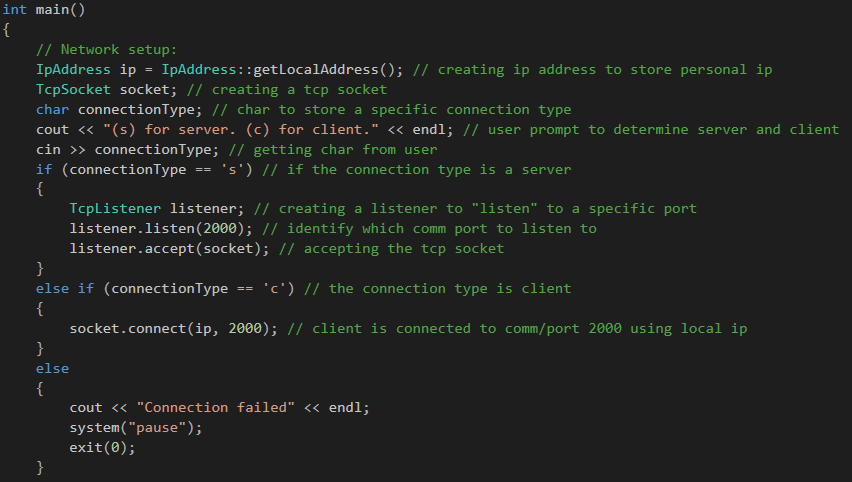
The first section of the code are the program headers. This included some of the basic libraries such as the iostream and vector. The **SFML/Graphics** header is specific to the sfml library. This included some of the crucial parts of my program such as loading images, files, creating sprites, as well as mouse inputs and general events. The **SFML/Network** header was used for setting up the TCP socket and connection. This included classes such as IpAddress, sockets, listeners, packets and receivers. I also added the sf namespace to avoid declaring **“sf::”** every time I use a class from the sf library.



The locate function was used to determine where the chess pieces are to be originally placed at the start of every chess game. The void function takes a pointer to variables already created and defined in the main function. This includes an empty integer for a x coordinate, a y coordinate as well as a filled string that holds a coordinate on the chess board (eg. f4, d2, a2). The function takes the location string and determines its associated x and y coordinate. Because integers are referenced and dereferenced with pointers, it removes the need of returning any values to the main function, and the integers can be manipulated, accessed and evaluated at any time, and almost everyplace in the program. Since I originally planned to use this code to output chess pieces from a randomly generated set of chess pieces, it significantly automated that process. However after, starting the chess game with all pieces in their original order, I found that the locate function to not be as useful as it was before. The function could have been replaced with lines in the main that simply state where to place the chess pieces, but since I had already made this function, I decided not to replace it.



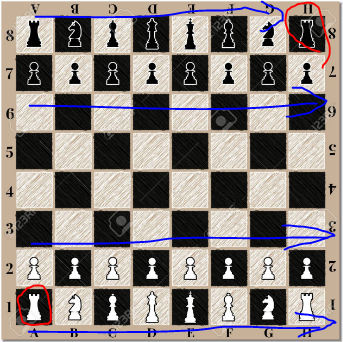
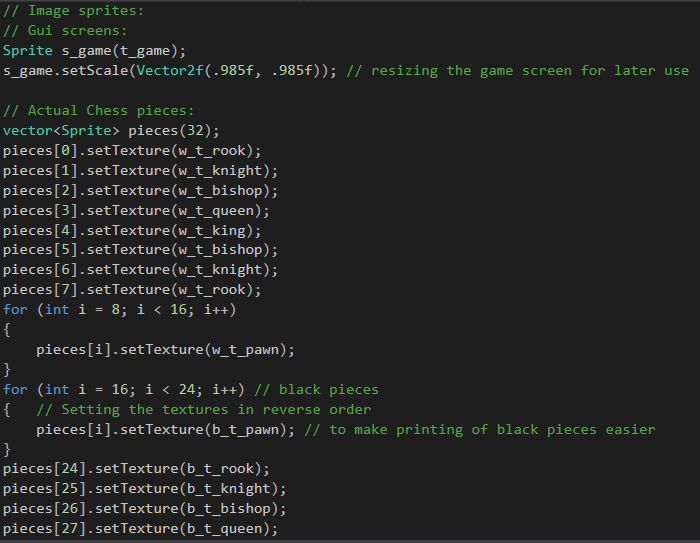
The rules function is used to determine if a certain chess move is legal or illegal based on the defined rules of chess. The void function takes a pointer to variables already created and defined in the main function. This includes a pointer to specific chess piece and a null integer that will hold the decision of whether a chess move was legal or illegal. The function also takes pointers to two variables belonging to a class called **Vector2f**. This class belongs to the sfml library and is simply a utility template class for manipulating 2-dimensional vectors, with two coordinates (x and y). It can be used to represent anything that has two dimensions: a size, a point or a velocity. In this case the variables are used to represent the **past location** of a chess piece and the **new location** to where it has been moved to. After receiving these variables, the function then proceeds to create integer copies of the old and new chess piece positions (for x and y). This makes it easier to access and evaluate the actual movement of the chess pieces. The function then creates an integer that holds the difference from the old position to the new position (for both x and y). If any of the distances are negative the function will change it to positive to make assessing and comparing the differences much easier. The function then determines what chess piece is being evaluated by determining the # of the **piece** variable. The function can now move on with the decision process. The concept behind deciding whether a move is legal or not is that if the difference between the old x/y and x/y position is a certain amount the function will determine whether the move was legal or illegal. This is because chess pieces can only move a certain distance in a chess board in a certain direction or method. The difference between the old position and the new position illustrate this virtual movement in a game of chess. If a move is determined legal, a 1 is said to be equal to the value of the decision. If the move is illegal, the value of the decision is made to be 0. When using these values in the main function, if it determines that the move is legal the program will allow the chess piece to move and will determine the opposing chess piece to be “captured”. If the move is determined illegal, then the moved piece is barred from moving to that position. This process in the function is done for all types of chess pieces.



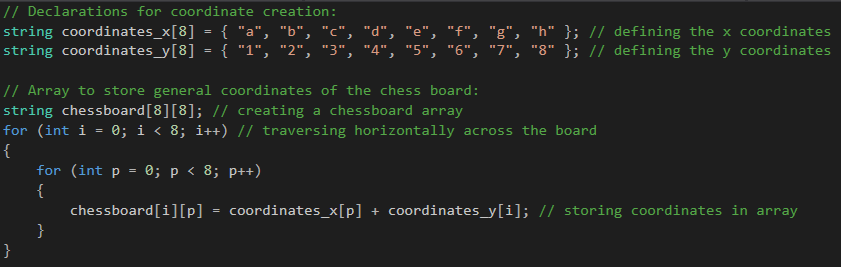
This fourth section of the code is the network setup. This step occurs at the beginning of the program and is necessary for establishing a server-client game connection. Briefly tcp or transmission control protocol is a network communication protocol designed to send data packets over the Internet. TCP is a transport layer protocol in the OSI layer and is used to create a connection between remote computers by transporting and ensuring the delivery of messages over supporting networks and the Internet. It begins by determining the local ip address of the computer the program is running on. This is done with **IpAddress::getLocalAddress();**. Ipaddress is a class in the sfml network library and is used to store the Ip address of the server. **TcpSocket** socket, is a socket port used for data communication between the two players (this will be a crucial aspect of the program). The program then proceeds to ask the user(s) if they wish to activate the gui as a server or a client using a char as a connection type. If the connection type is a server, the program activates a socket listener that reads the data coming from the port (in this case listening to port 2000) and accepts the socket as the main communication between the server and client. If the connection type is a client, the socket connects itself to the server using the Ip as a destination and the port as the pathway or means of communication. If the user does not specifically whether the connection type is a client or server, the program declares this as an error and ends the program, to avoid crashes generated from this error.



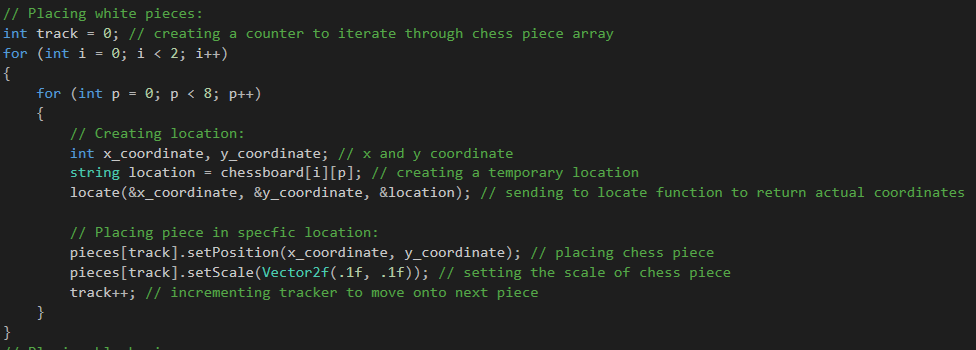
The fifth section of the code is the Texture definitions and image imports. The program defines a texture for every image that will be used in the program. It is important to note that the texture itself is not going to be displayed in the game gui, but simply attached to a variable (explained in the next section) that will actually be drawn and displayed. For each texture the program loads an image that imports itself onto the texture. This includes images for chess pieces and the game background.



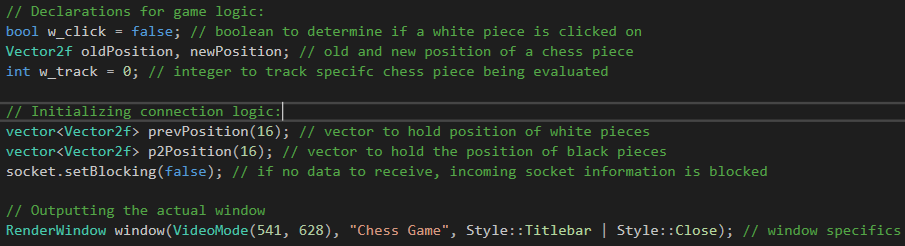
The sixth section is where images are actually created and outputted to the gui screen. The **sprite** class allows, images to be changed in color, scale, cropped, rotated, as well as position. When changing any of a sprites characteristics, it is done so by declaring a Vector2f number. This is because vector coordinates control many of a sprite’s attributes. So when changing the scale of the **s\_game** screen as shown above, the **Vector2f**, numbers are used to change those values. In this program, the sprites for each chess piece are declared within a vector structure. This makes accessing each chess piece easy for all purposes. Although all the chess pieces are declared with the same sprite, they are individually connected to their own texture using the **.setTexture()** function. I declared each sprite in an order so that when they are outputted to the board, it is done in the proper format, where the pieces start with a white rook and end with a black rook.



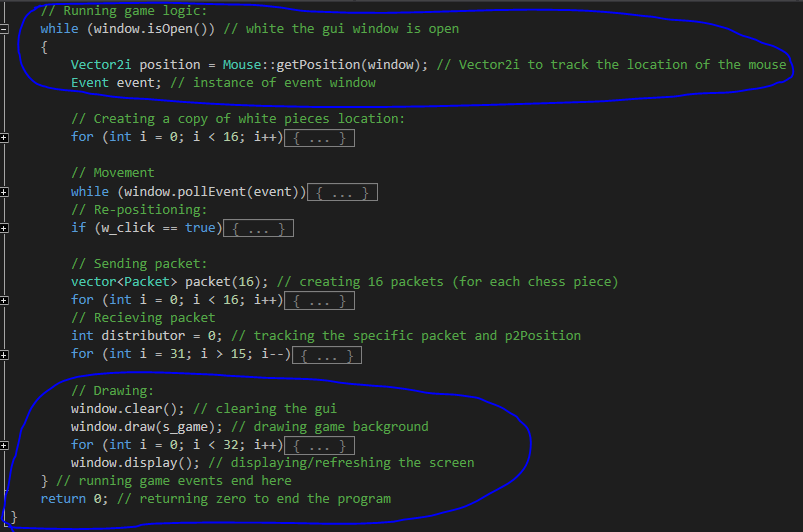
In this section of the program, I have created two string arrays that hold chess board coordinates. Then I create a general chess board array that will hold all 64 coordinates of the chess board. I use a while loop to iterate through both string arrays and create a combination of 64 coordinates. All of which are sent to the general chessboard array. Since I originally planned to use this code to help the chess AI move its pieces around the board, it significantly automated that process. However after changing the program to be controlled by two players who physically choose where the pieces are placed (within the chess game rules), I have found that this piece of code is not as useful as it used to be. Its general purpose as of now is to initially help place the white and black chess pieces on the board at the start of every game, with no use later on.



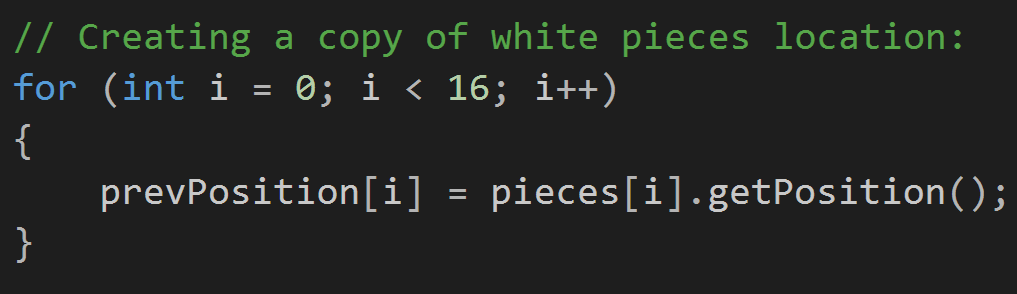
In this section, the chess pieces are actually positioned around the chessboard. This is done with help from the chessboard array from the section above. Here the placement of white chess pieces is detailed. . The process is also repeated for the black set of chess pieces. To begin a counter is created to help track what piece is being manipulated. A for loop is intilated to loop 32 times (through every white piece). While the for loop is iterating, a null x and y coordinate are declared along with a location string that holds one coordinate from the chessboard string array. These three variables are then sent to the **locate** function, and the x and y coordinates are given actual integer coordinate. Each chess piece is then placed onto the board using the **.setPosition();** function and the x and y coordinate as a location. The scale of each piece is also set to the proper amount. The tracking counter is then incremented to move onto manipulating the next chess piece. As this process continues, all the chess pieces are automatically positioned to their proper starting spot on the chessboard.



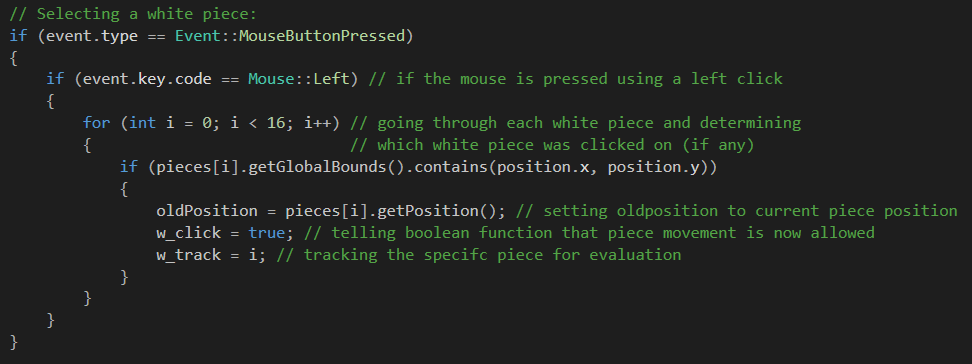
Before the game is actually started, I specify some declarations to be used throughout the running game logic. To begin, I declare a boolean called **w\_click**. This is so that a user can dynamically click onto their chess pieces and drop it, all while the movement is being evaluated in terms such as legality as well as piece captures. When the boolean is true, the movement is evaluated, and while the boolean is not true, there is no movement evaluation. Two vector2f variables are created, known as **oldPosition** and **newPosition**. These positions will be what determines if a chess move was legal or illegal and will be used in the further sections of the program. I have also created a variable known as **w\_track**, which will track the specific chess piece being evaluated. The next thing the program does is to initialize the connection logic. Here a vector of player one and player two positions are created (one for each chess piece). These positions will also track the positions of chess pieces before and after they are moved, and will be the data sent across the communication port, and will result in the live drawing of the opponent’s chess moves. Essentially one player’s chess moves will become the opposing player’s chess moves on the other user’s gui screen. At the end of this section, I output the actual window of the gui. I create an instance of the window, known as **window**, and give it characteristics such as dimensions, the title bar name. **Style::Close** teels the program to insure that the gui window cannot be resized (if the window were to be resized, it would break the foundation of my game).



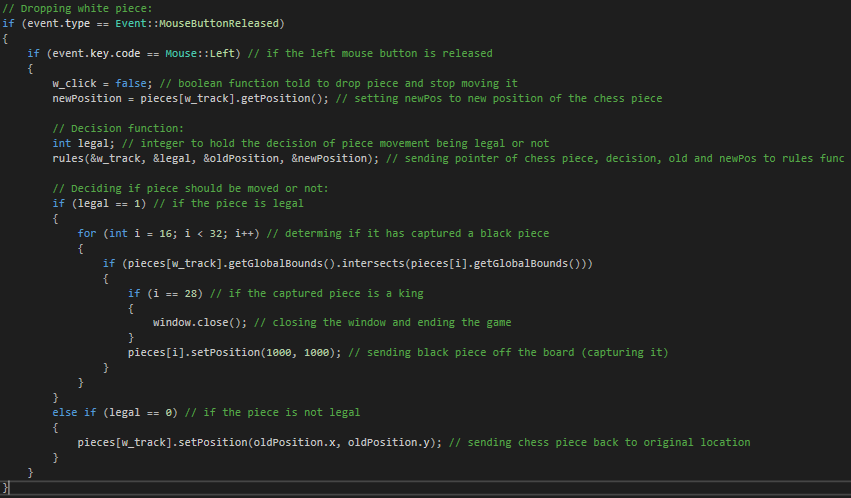
In this general section, **while(while.isOpen())** is where all of the running game events and most of the logic takes place. In the while loop the program starts by taking a live x and y position of the mouse in the window. This position is recorded as **Vector2i**. Vector2i stores coordinates as integers whereas Vector2f stores integers as floats. The program also creates an instance of the game events. This is used for mouse inputs such as clicks, traces, releases as well as screen events such as closing or dragging the screen. Near the end of the for loop is where all sprites are drawn to the screen consecutivity. **Window.clear();**, initially clears the screen of any leftover elements. **Window.draw();** then proceeds to draw all the sprites with the first sprites drawn at the bottom and the last sprites being drawn at the top of the gui. **Window.display();** then displays the sprites to the gui. To the user this total process would be seen as the screen simply updating its contents.



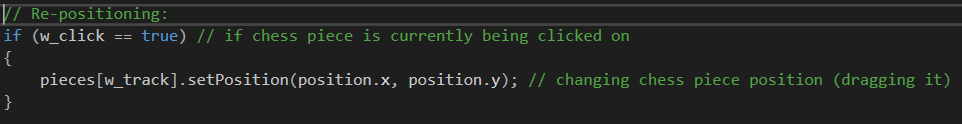
Before chess piece movement is made, a copy of each chess piece location is stored inside of the player1 vector known as **prevPosition**. This vector will be compared later against the updated chess piece positions to determine if any data needs to be communicated through the Tcp port.



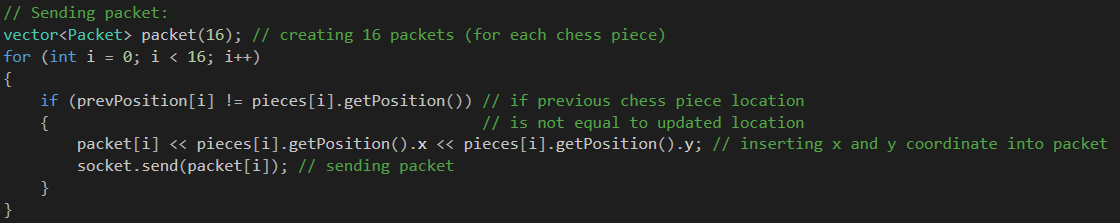
Since players on each screen will only be controlling the white chess pieces, they will only be able to move those pieces and any other movements will not be detected or registered. The program uses the event instance to declare a if statement that asks if the mouse has been clicked upon. A nested if statement below asks if the click was made by the left mouse button. A for loop then iterates through all thewhite chess pieces and determines if the mouse clicked upon any of them by using the live mouse position (**position.x** and **position.y**). If true, the **oldPosition** is set to the current position of the chess piece before it is moved. The chess piece tracker is set to the current iteration and boolean function **w\_click** is set to true to allow the actual movement of the chess piece.



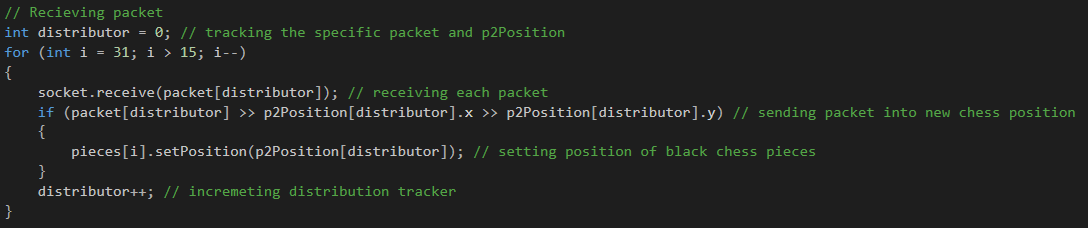
Here the program uses the event instance to determine if a mouse press has been released. This if statement will only apply after a mouse click has been made. A nested if statement then asks if the click release was made by the left mouse button. If true, the program then proceeds to set the boolean function **w\_click** to false to insure chess piece movement stops and sets the **newPosition** variable to the current position of the chess piece. A null variable is then created to hold the decision of the program that will determine whether the chess move is legal or not. It alongside the old and new chess positions are sent to the **rules** function. If the returned decision is l (legal), a for loop will iterate through black pieces and determine if a collision (capture) is made. This is done by asking if the hitbox around the chess piece (**.getGlobalBounds();**) intersects with the hitbox of the opposing chess piece. If a capture is made, that piece is sent off the board, and the opposing player will lose control of that piece. If that captured piece is a king, then the game closes to signify the end of the game. If the returned decision is 0 (illegal), the moved piece will be moved back to its original location.



Although I explained previously, this is a boolean function that dynamically controls the movement of a chess piece. While the function is deemed true, the repositioning of the chess piece is allowed and as soon as the boolean is deemed false, the positioning is discontinued, which effectively gives the effect of dragging and dropping the chess piece.



In this section, the program sends packets of data through the socket port. The program begins by creating a vector of packets (one for each of the chess pieces). Inside of the for loop, a if statement determines if the previous position of the white chess piece is the same as the current position of the piece after all the chess piece movements where made. If true the current position of the chess piece (x and y) are sent to the packet through the **<<** operator. The packet is then sent through the socket using the **.send();** function.



In this final section, the packet is received and data is updated in both gui screens. The program begins by creating a tracking variable that will track the specific packet and the packet distribution. A for loop iterates through the black pieces in reverse order. This is done to apply the packet distribution in reverse order, to create a mirror perspective for opposing players. During each iteration a packet is received through the socket port using the **.receive();** function. The packet is then sent into the player two x and y position and position of the chess piece is set to that of the communicated x and y position. At the end of the for loop the distributor is then incremented to move onto the next packet and playertwo position.