Software Engineering Assignment 2-**19476496**

For our second assignment we were given the task of creating the board game Focus/Domination in c with the use of linked lists.

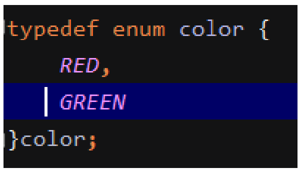
To do this the idea of divide and conquer is needed once again, as in order for us to take a manageable approach at the assignment, we must split it into smaller workable ideas which we could execute.
These were:

1. Initializing and creating key components like board layouts ,structs, enums and player information.
2. Implementing the printing of the board, the movement of the players and the placing of stacks on top of one another.
3. To make sure that we limit stack sizes to 5 and to create a function to remove excess pieces
4. Finally to set up winning game conditions.

Each part was then split again into smaller more handable problems.

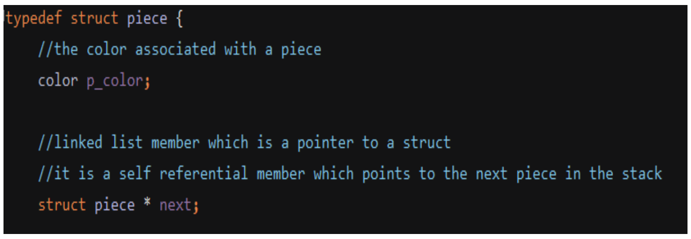
Part 1:(a) Defining enums and structs

Part 1 can be divided into 2 main parts, board initialization and player initialization.
Before both we had to define 2 structs and 2 enums which hold information about the board.

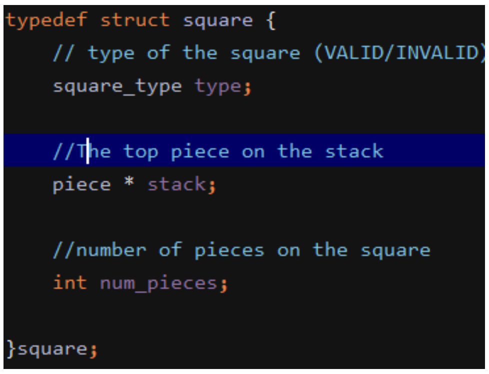


```
typedef enum color {
    RED,
    GREEN
}color;
```

Enums are used to represent a string with an associated integer value which can be used. In Enum color we define red and green, as they are the colours we will be using in a 2 player game. We also have the square_type enum which was used to keep track of locations on the board not traversable, the enum had 2 members INVALID and VALID.



```c
typedef struct piece {
    //the color associated with a piece
    color p_color;

    //linked list member which is a pointer to a struct
    //it is a self referential member which points to the next piece in the stack
    struct piece * next;
```



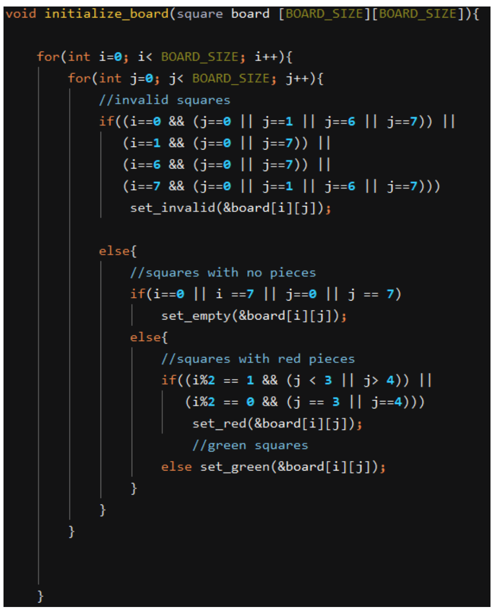
```c
typedef struct square {
    // type of the square (VALID/INVALID)
    square_type type;

    //The top piece on the stack
    piece * stack;

    //number of pieces on the square
    int num_pieces;

}square;
```

Structs were also an integral part of this assignment due to their use in linked lists and storing a range of data in one defined type

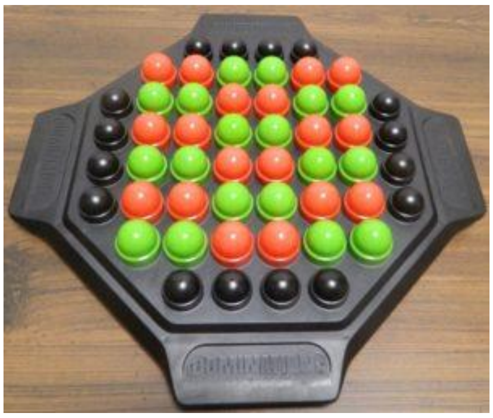
Struct piece contains 2 members, the colour of the piece and the self referential member next. That self referential member will point to another piece that it has been placed upon.

Struct square however has 3 members: the number of pieces on a board position, its type and the pointer to the top of the stack in that location. Piece* stack and piece* next are used in conjunction to join stacks and split stacks later in the program.

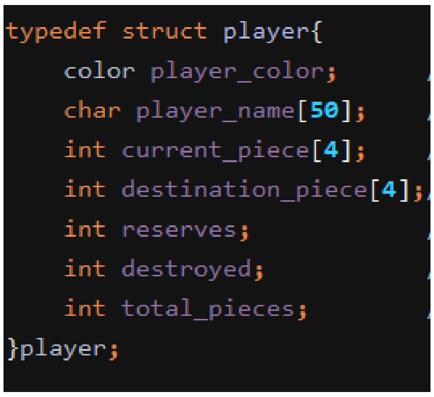
Part 1(b) Initializing the board
To Initialize the board, we must use a series of functions which set pieces to green, red, valid and invalid. The basic layout sets a pointer to a stack, defines its type, color and next pointer to null. We then use a series of loops and ifs to initialize the board (set invalid, green etc) These functions lays the game out like this



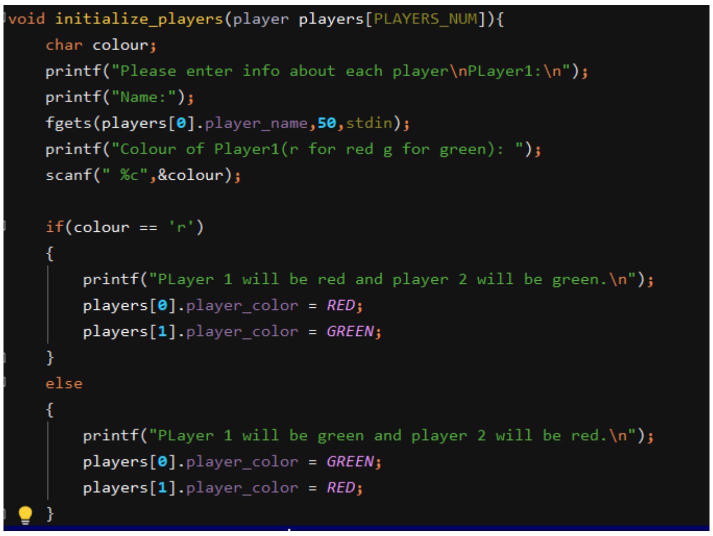
```c
void initialize_board(square board [BOARD_SIZE][BOARD_SIZE]){

    for(int i=0; i< BOARD_SIZE; i++){
        for(int j=0; j< BOARD_SIZE; j++){
            //invalid squares
            if((i==0 && (j==0 || j==1 || j==6 || j==7)) ||
               (i==1 && (j==0 || j==7)) ||
               (i==6 && (j==0 || j==7)) ||
               (i==7 && (j==0 || j==1 || j==6 || j==7)))
                set_invalid(&board[i][j]);

            else{
                //squares with no pieces
                if(i==0 || i ==7 || j==0 || j == 7)
                    set_empty(&board[i][j]);
                else{
                    //squares with red pieces
                    if((i%2 == 1 && (j < 3 || j> 4)) ||
                       (i%2 == 0 && (j == 3 || j==4)))
                        set_red(&board[i][j]);
                        //green squares
                    else set_green(&board[i][j]);
                }
            }
        }


    }
```



Part1(c) Initializing the player:
To initialize the player, we need to find important information pieces about the player. These were name, colour, reserves, total pieces, destroyed and 2 locational pieces dest and cur. Below we see that total_pieces is set to 18. The total holds the current amount of visible pieces i.e pieces on the top of a stack of your colour, this information is useful as when this is 0 we have lost. We also have a destination_piece that will be used in part 2 for holding the location of pieces in which we want to move.



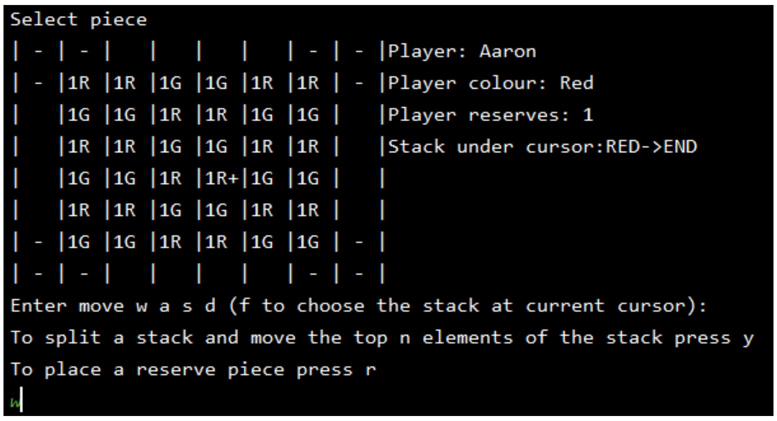
```c
typedef struct player{
    color player_color;
    char player_name[50];
    int current_piece[4];
    int destination_piece[4];
    int reserves;
    int destroyed;
    int total_pieces;
}player;
```



```c
void initialize_players(player players[PLAYERS_NUM]){
    char colour;
    printf("Please enter info about each player\nPLayer1:\n");
    printf("Name:");
    fgets(players[0].player_name,50,stdin);
    printf("Colour of Player1(r for red g for green): ");
    scanf(" %c",&colour);

    if(colour == 'r')
    {
        printf("PLayer 1 will be red and player 2 will be green.\n");
        players[0].player_color = RED;
        players[1].player_color = GREEN;
    }
    else
    {
        printf("PLayer 1 will be green and player 2 will be red.\n");
        players[0].player_color = GREEN;
        players[1].player_color = RED;
    }
```

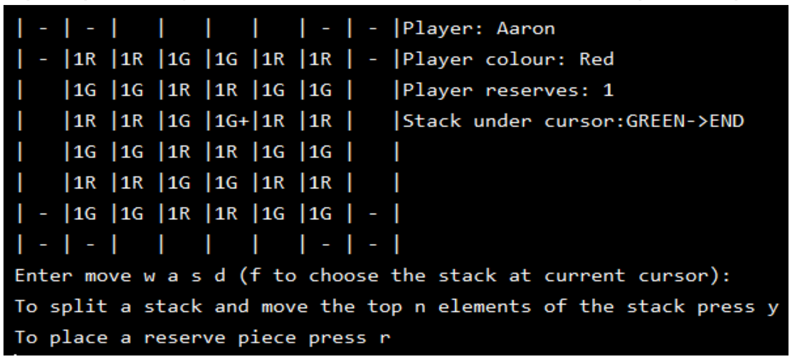
Part 2:

Part 2(a) Reading in information about user input and the location of the piece in which they want to move to. This can be done by inputting coordinates of the piece you wish to choose; however I felt that it was oftentimes hard to find the exact location of that piece fast. To counteract this I programmed it to print the board each time the user inputs a directional key and update a cursor under that location. This cursor is notified by a + sign. With this I can hit w and enter and then a and enter to move the cursor to the piece I want to choose up and to the left.



```
Select piece
| - | - |   |   |   |   | - | - |Player: Aaron
| - |1R |1R |1G |1G |1R |1R | - |Player colour: Red
|   |1G |1G |1R |1R |1G |1G |   |Player reserves: 1
|   |1R |1R |1G |1G |1R |1R |   |Stack under cursor:RED->END
|   |1G |1G |1R |1R+|1G |1G |   |
|   |1R |1R |1G |1G |1R |1R |   |
| - |1G |1G |1R |1R |1G |1G | - |
| - | - |   |   |   |   | - | - |
Enter move w a s d (f to choose the stack at current cursor):
To split a stack and move the top n elements of the stack press y
To place a reserve piece press r
w
```

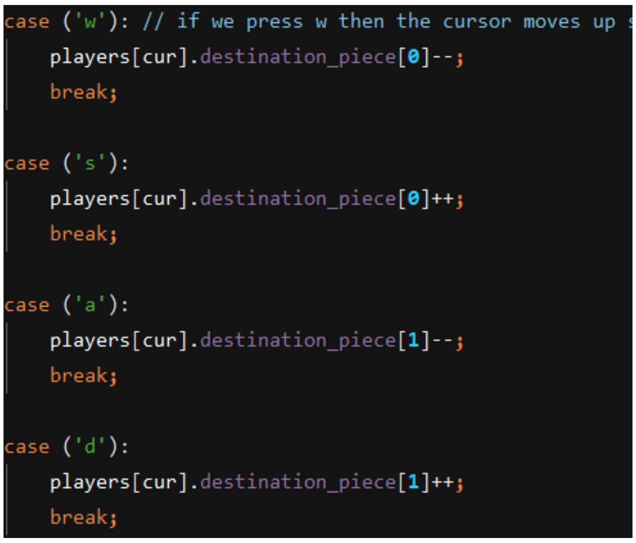
Upon pressing w and enter the + cursor moves up one space.



```
| - | - |   |   |   |   | - | - |Player: Aaron
| - |1R |1R |1G |1G |1R |1R | - |Player colour: Red
|   |1G |1G |1R |1R |1G |1G |   |Player reserves: 1
|   |1R |1R |1G |1G+|1R |1R |   |Stack under cursor:GREEN->END
|   |1G |1G |1R |1R |1G |1G |   |
|   |1R |1R |1G |1G |1R |1R |   |
| - |1G |1G |1R |1R |1G |1G | - |
| - | - |   |   |   |   | - | - |
Enter move w a s d (f to choose the stack at current cursor):
To split a stack and move the top n elements of the stack press y
To place a reserve piece press r
```

This can be repeated until the user decides they want a certain element in the stack.

This is done using destination piece coordinates and a switch which reads what character the user entered.



```
case ('w'): // if we press w then the cursor moves up
    players[cur].destination_piece[0]--;
    break;

case ('s'):
    players[cur].destination_piece[0]++;
    break;

case ('a'):
    players[cur].destination_piece[1]--;
    break;

case ('d'):
    players[cur].destination_piece[1]++;
    break;
```

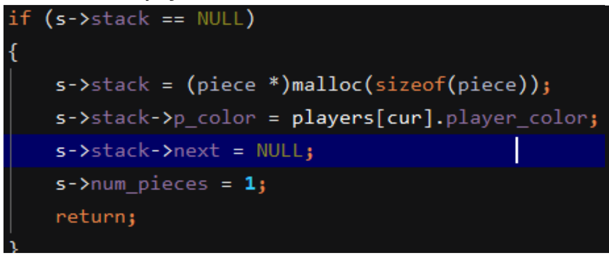
Part 2(b) From this point if the user decides they want to choose this piece they can do 1 of 3 things.
1.place a reserve which will attack a piece from the players reserve bank to the top of the stack.

2.Move the stack which will join that stack with another stack that the user indicates.

3.Split the stack,the user can split a stack that is 2 or more high and move the top n elements.

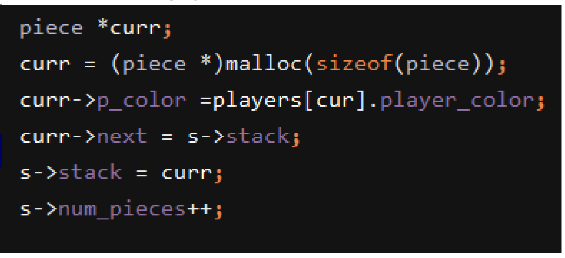
1. If the player decides to use a reserve piece, a function called place_reserve is called which takes in the current user's location and places a reserve at that point. The function has 2 cases: if the location is empty or not

Empty



```
if (s->stack == NULL)
{
    s->stack = (piece *)malloc(sizeof(piece));
    s->stack->p_color = players[cur].player_color;
    s->stack->next = NULL;
    s->num_pieces = 1;
    return;
}
```

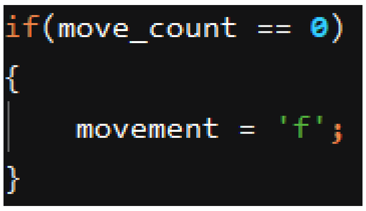
Not empty



```
piece *curr;
curr = (piece *)malloc(sizeof(piece));
curr->p_color =players[cur].player_color;
curr->next = s->stack;
s->stack = curr;
s->num_pieces++;
```

The difference between the 2 is that when we are deciding where the new piece should point to, we define a piece pointer called curr which will point the new piece to the top of the stack it was placed on should it not be empty.

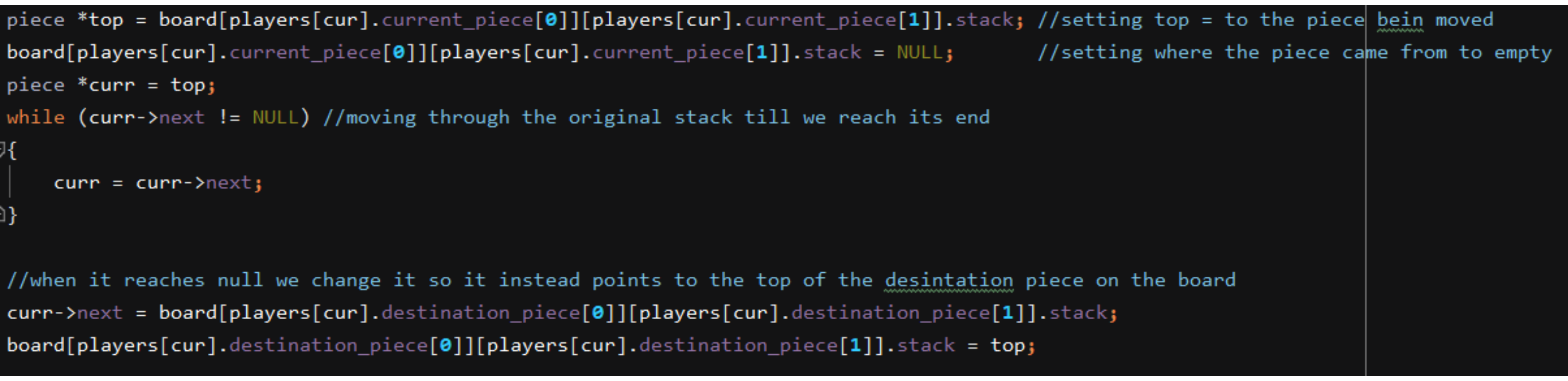
2.If the player decides to move the piece onto another stack 2 functions will be used: 1 to choose where the user wants to place the stack and another for actually placing the stack on the desired location. The first function move_piece is similar to selection piece in which the user can move using w a s d to pick their desired location however we create a movecounter which counts the amount of times the user has moved as the user can only move the same amount of times as the size of the stack.



```
if(move_count == 0)
{
    movement = 'f';
}
```

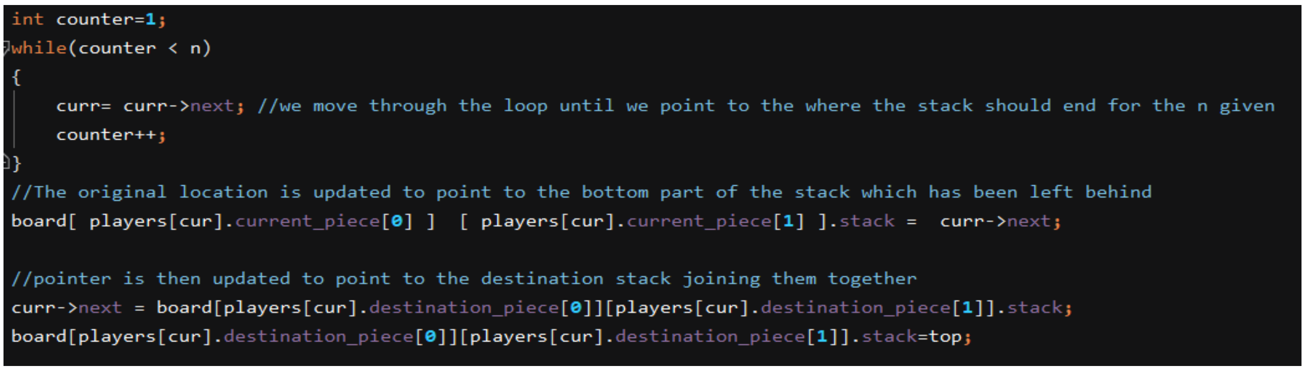
If the user moves 2 times on a stack of size 2 there move counter will decrease to 0 and if they try to move again they will automatically choose the piece they were on currently

Once f is pressed the function to join the 2 stacks is called



```
piece *top = board[players[cur].current_piece[0]][players[cur].current_piece[1]].stack; //setting top = to the piece bein moved
board[players[cur].current_piece[0]][players[cur].current_piece[1]].stack = NULL;       //setting where the piece came from to empty
piece *curr = top;
while (curr->next != NULL) //moving through the original stack till we reach its end
{
    curr = curr->next;
}

//when it reaches null we change it so it instead points to the top of the desintation piece on the board
curr->next = board[players[cur].destination_piece[0]][players[cur].destination_piece[1]].stack;
board[players[cur].destination_piece[0]][players[cur].destination_piece[1]].stack = top;
```

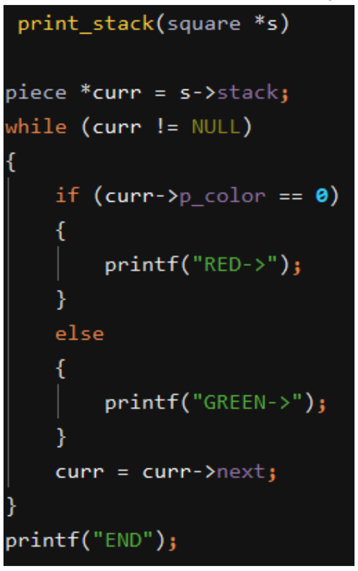
3.If the player decides they want to split the stack and move the top n pieces they will press y during selection and this will call a function called split_function which is similar to move piece however it has an extra argument holding the top n part of the stack they wish to move. Split_function then calls split stack which is similar again to the function to join 2 stacks



```
int counter=1;
while(counter < n)
{
    curr= curr->next; //we move through the loop until we point to the where the stack should end for the n given
    counter++;
}
//The original location is updated to point to the bottom part of the stack which has been left behind
board[ players[cur].current_piece[0] ]  [ players[cur].current_piece[1] ].stack =  curr->next;

//pointer is then updated to point to the destination stack joining them together
curr->next = board[players[cur].destination_piece[0]][players[cur].destination_piece[1]].stack;
board[players[cur].destination_piece[0]][players[cur].destination_piece[1]].stack=top;
```

Counter here moves through until the curr pointer is the nth value in the stack,
The original location is given the value found at curr next;
curr->next is then updated to point to the stack in which we want to place the split stack onto.

Part 2(C) This is printing the board and printing the current stack. Printing the current stack is done with the function print_stack

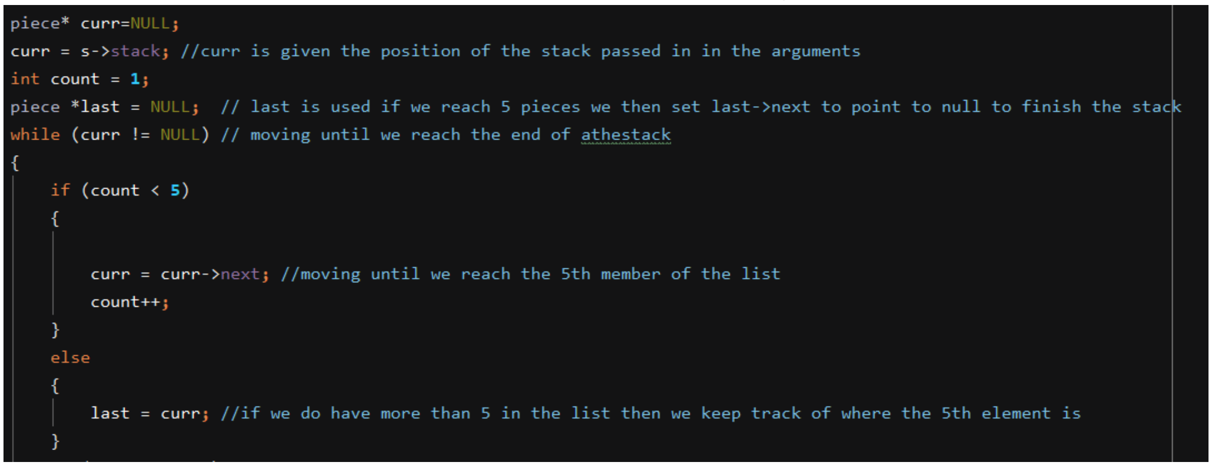


```
 print_stack(square *s)

piece *curr = s->stack;
while (curr != NULL)
{
    if (curr->p_color == 0)
    {
        printf("RED->");
    }
    else
    {
        printf("GREEN->");
    }
    curr = curr->next;
}
printf("END");
```

Curr pointer runs through the pieces in the stack of the board location. If it finds the colour to be red:RED-> is printed.If its not red GREEN-> is printed the curr is updated to the next piece.

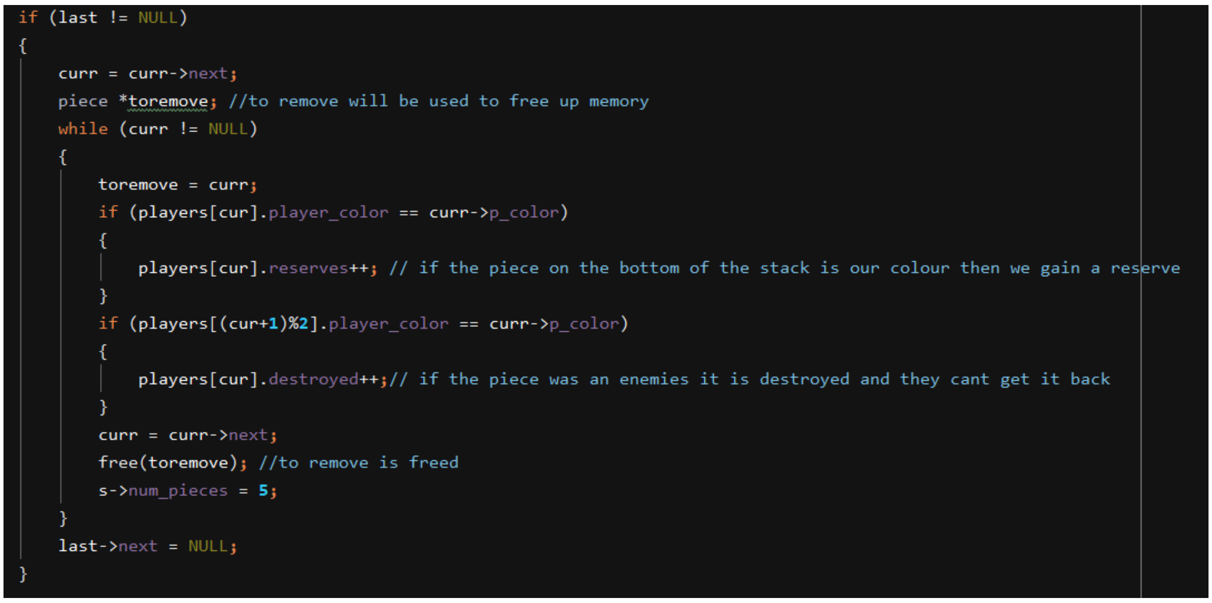
Part 3: Ensuring limit of 5 on each stack

For part 3 a function was created which takes in a stack and moves through it to see if it contains more than 5 pieces.
I created a function for this as I knew it would be called many times.



```
piece* curr=NULL;
curr = s->stack; //curr is given the position of the stack passed in in the arguments
int count = 1;
piece *last = NULL;  // last is used if we reach 5 pieces we then set last->next to point to null to finish the stack
while (curr != NULL) // moving until we reach the end of athestack
{
    if (count < 5)
    {

        curr = curr->next; //moving until we reach the 5th member of the list
        count++;
    }
    else
    {
        last = curr; //if we do have more than 5 in the list then we keep track of where the 5th element is
    }
```

In the above part if we find that last != NULL then we know that there are move than five pieces. We define a piece pointer called to remove which will point to the current piece over 5 in the stack and using the free() command will free up that memory.

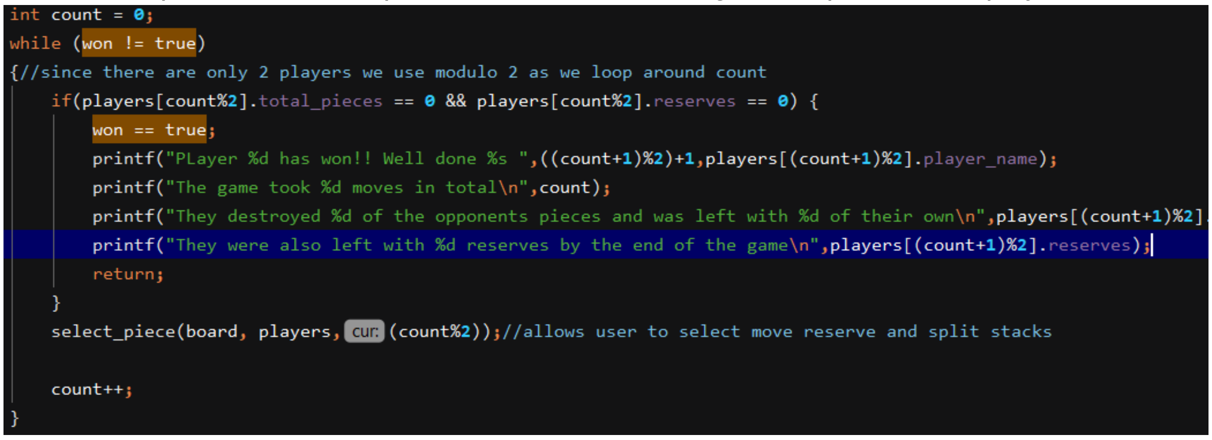


```c
if (last != NULL)
{
    curr = curr->next;
    piece *toremove; //to remove will be used to free up memory
    while (curr != NULL)
    {
        toremove = curr;
        if (players[cur].player_color == curr->p_color)
        {
            players[cur].reserves++; // if the piece on the bottom of the stack is our colour then we gain a reserve
        }
        if (players[(cur+1)%2].player_color == curr->p_color)
        {
            players[cur].destroyed++;// if the piece was an enemies it is destroyed and they cant get it back
        }
        curr = curr->next;
        free(toremove); //to remove is freed
        s->num_pieces = 5;
    }
    last->next = NULL;
}
```

If the bottom piece was yours you get an extra reserve if it was your opponents then they lose that piece completely.

Part 4: Implementing a winning condition.

How do we know when a player wins? We know this if the players opponent has no visible pieces left on the top of any stack i.e they may still have pieces but they are under the control of green. The opponent must also have no reserves left. There are 2 ways to test this we can scan the entire board each time a player goes to check or we can create a variable called total_pieces which keeps a track of the remaining visible pieces for a player



```c
int count = 0;
while (won != true)
{//since there are only 2 players we use modulo 2 as we loop around count
    if(players[count%2].total_pieces == 0 && players[count%2].reserves == 0) {
        won == true;
        printf("PLayer %d has won!! Well done %s ",((count+1)%2)+1,players[(count+1)%2].player_name);
        printf("The game took %d moves in total\n",count);
        printf("They destroyed %d of the opponents pieces and was left with %d of their own\n",players[(count+1)%2].
        printf("They were also left with %d reserves by the end of the game\n",players[(count+1)%2].reserves);
        return;
    }
    select_piece(board, players, cur: (count%2));//allows user to select move reserve and split stacks

    count++;
}
```

There are many cases in which a player either gains or loses a visible piece

If a player places a reserve on either an empty or an enemies space they gain one visible piece(if they move it onto an enemies space the enemy will lose one visible piece).

If a player moves a stack of their colour onto an enemies piece the enemies total pieces goes down.

If a player moves a stack of their colour onto their own piece then they lose one visible piece.

When a player splits a stack and moves it 2 pieces change:
The top piece that is left after moving the stack will increase that players total visible pieces

If the stack that the stack was moved onto had the players colour than their total pieces go down otherwise the enemies do

Finally if by the end of the game a player has no reserves and no visible pieces then the game ends and prints info about the user who won.

I used this as it is faster than doing a full scan of the board every single time a player takes a go