Rybnik, 03.02.2019

Semester: 3
Group: 3
Section: Wednesday 10:15

Computer Programming Laboratory

Yutnori - Board game

Author: Sebastian Chłodek
E-mail: sebachl594@student.polsl.pl
Tutor: Anna Gorawska

1. Task topic

The task was to create a console program imitating a japanese board game called “Yutnori”. In the creation of the program, there were supposed to be implemented at least 4 topics carried out during the laboratories and at least 5 classes. The topics I used are: Inheritance, Polymorphism, RTTI and Exceptions.

2. Project analysis

In my program I used classes, specifically class ‘Move’ and 5 derived classes ‘Do’, ‘Gae’, ‘Geol’, ‘Yut’ and ‘Mo’, class ‘Pawn’ and class ‘Baton’. Class ‘Baton’ imitates special batons, of which there are 4 and they are used instead of our dice. Each baton has 2 sides and depending on how many of them land with the plain side facing up, indicates by how many tiles the player can move one of his pawns. The 5 classes derived from class ‘Move’ are named after each possible combination of outcome of our throw of 4 batons. Finally from class ‘Pawn’ we get 4 unique pawns, pawn 1 and 2 for Player 1 and pawn 3 and 4 for Player

To create board I used 3 arrays. The main array ‘Board’ which has 20 tiles and represents the outer edges of our board, and the two arrays ‘Diagonal1’ and ‘Diagonal2’, each being 7 tiles long, which represent the diagonals of our board.

Other than that, there are 2 pointers, one to class ‘Move’ and one to class ‘Pawn’, indicating current move and which pawn is being moved, 3 1-tiled arrays used in the function, that moves pawns and finally some integers, ‘first_number’, ‘second_number’, ‘which_pawn’, ‘who_won’, ‘which_player’, ‘var_a’ and ‘var_b’.

3. External specification

The program is designed for 2 players. At the start of the program each player throws their batons and the one that achieved a lesser score will go first during the proper gameplay.
Rules are simple:

- Each player has 2 pawns (Player 1 - pawn 1 and 2, Player 2 - pawn 3 and 4)
- The goal is to come back to the starting point with both of your pawns
- If your pawn lands on the tile, which is occupied by a different pawn (no matter if it's yours or your opponents), the currently occupying one has to start from the beginning
- There are shortcuts on the board, on the corners and in the middle. There are 4 combinations of paths, that your pawn can take:

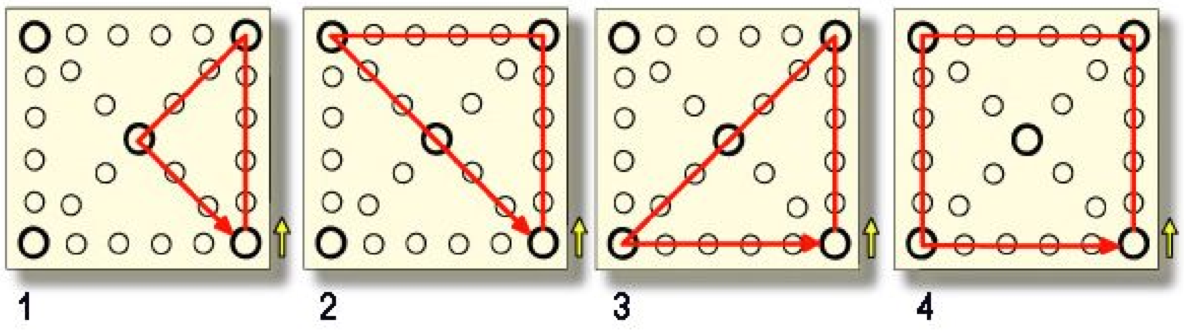


Image taken from https://pl.wikipedia.org/wiki/Yut

4. Internal specification

The program consists of 8 .cpp files: *main, classMove, classBaton, classPawn, functionDecidingOrder, functionDraw, functionGeneratingThrow*, *functionPrintingResultsOfThrow* and corresponding headers.

In *classMove* there is our main class '*Move*', which has pure virtual void function '*move_*', and pure virtual int function '*strength_of_move*'.

There are also 5 classes that inherit from main class, called '*D0*', '*Gae*', '*Geol*', '*Yut*' and '*Mo*'. Each of them has unique integer '*move_by*' which will indicate by how many tiles can the player move one of his pawns. Inherited functions of these classes '*move_*' and '*strength_of_move*' respectively print the value of '*move_by*' and return the value '*move_by*'. There is also function '*uni*' which sets pointer '*how_much*' to one of the classes and carries out its function '*move_*'.

In *classBaton* there is a private integer '*binary_throw*', which indicates which side of the baton is facing up after a throw. There is also a constructor, which gives our '*binary_throw*' certain value and function '*draw_baton*', which prints the result of throw (either baton with plain side facing up or the engraved side facing up).

In *classPawn* there are 4 private integers: '*symbol*', '*move_by_how_much*', '*starting_point*' and '*finished*'. '*symbol*' is a unique number of particular pawn (1, 2, 3 or 4), '*move_by_how_much*' tells us by how many tiles the pawn can yet move, '*starting_point*' tells us if the pawn is on or off the board (1-off, 0-on), '*finished*' tells us if the particular pawn has already finished its path, came back to the start (1-yes, 0-no). There is also a constructor setting the value of '*symbol*' and 9 functions: *int get_number* - getting 'symbol', *void set_strength* - setting the value of '*move_by_how_much*', *int get_strength* - getting the value of '*move_by_how_much*', *void decrease_strength* - decreasing the value of '*move_by_how_much*' by 1, *void in_game* - setting the value of '*starting_point*' to 0, *void finished_the_game* - setting the value of '*finished*' to 1, *int get_starting_point* - getting the value of '*starting_point*', *int get_finished* - getting the value of '*finished*' and *void beaten* - setting the value of '*starting_point*' to 1.

functionDraw prints our board, using our 3 arrays '*Board*', '*Diagonal1*' and '*Diagonal2*'.

In *functionGeneratingThrow* there are 2 functions: '*generate_binary_number*' and '*generate_random_numbers*'. The first one simply generates pseudo-random number, which can be either 0 or 1 and assigns it to variable ‘*first_number*’. The second one uses the generated pseudo-random numbers and assigns them to ‘*binary_throw*’ of our batons, to finally print them in our console window. It also operates on variable ‘*second_number*’ which indicates the sum of our 4 batons (can be from 1 to 5).

In *functionPrintingResultsOfThrow* there is function ‘*results_of_throw’* which takes the variable ‘*second_number*’ and depending on its value dynamically casts pointer ‘*myMove*’ to a particular class ‘*Move*’. It then uses function ‘*typeid*’ to print the name of this class and finally sets pointer ‘*pointer*’ to this class to call function ‘*uni’.*

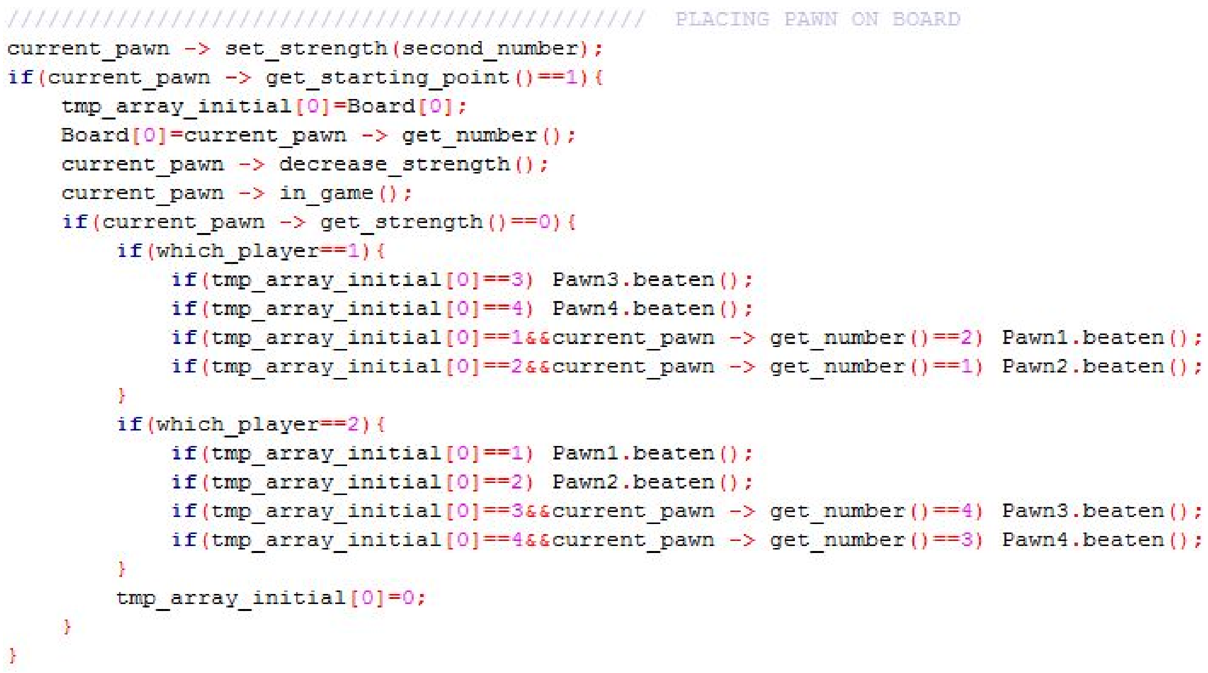
functionDecidingOrder uses functions ‘*generate_random_numbers*’ and ‘*results_of_throw*’ to decide which player will have the first move, by setting either value 1 or 2 to variable ‘*which_player*’.

In our *main* I included the whole algorithm of 1) choosing which pawn current player wants to move and 2) moving said pawn along the board.

In the first part the program takes a value of variable ‘*which_pawn*’ from the player, to assign pointer ‘*current_pawn*’ to one of the objects of class ‘*Pawn*’. If the players input is different from values of his pawns, the program throws exception, informing which pawns he can move. If player inputs the value of his pawn, that has already completed its path (its ‘*finished*’ value is 1) the program informs player of that and automatically assigns the pointer to the players other pawn. Also if one of the players pawns has already finished its path, then the program will print that information on the screen on which the player tells which pawn he wants to move.

The second part of the program is divided into 8 sections via comments included in the code.

The first one ‘*PLACING PAWN ON BOARD’* simply checks if the pawns variable ‘*starting_point’* is equal to 1, and if it is, then it places that pawn on the first tile of array ‘*Board*’ and sets the value of ‘*starting_point’* to 0.



```
///////////////////////////////////////////////  PLACING PAWN ON BOARD
current_pawn -> set_strength(second_number);
if(current_pawn -> get_starting_point()==1){
    tmp_array_initial[0]=Board[0];
    Board[0]=current_pawn -> get_number();
    current_pawn -> decrease_strength();
    current_pawn -> in_game();
    if(current_pawn -> get_strength()==0){
        if(which_player==1){
            if(tmp_array_initial[0]==3) Pawn3.beaten();
            if(tmp_array_initial[0]==4) Pawn4.beaten();
            if(tmp_array_initial[0]==1&&current_pawn -> get_number()==2) Pawn1.beaten();
            if(tmp_array_initial[0]==2&&current_pawn -> get_number()==1) Pawn2.beaten();
        }
        if(which_player==2){
            if(tmp_array_initial[0]==1) Pawn1.beaten();
            if(tmp_array_initial[0]==2) Pawn2.beaten();
            if(tmp_array_initial[0]==3&&current_pawn -> get_number()==4) Pawn3.beaten();
            if(tmp_array_initial[0]==4&&current_pawn -> get_number()==3) Pawn4.beaten();
        }
        tmp_array_initial[0]=0;
    }
}
```

The second one ‘*MOVING PAWN ON BOARD’* moves the pawn along the array ‘*Board*’.

```
////////////////////////////////////////////////  MOVING PAWN ON BOARD
for(int i=0;i<20;i++){
    if(Board[i]==current_pawn -> get_number()){
    m=i;
    break;
    }
}
if(m!=-1){
while(current_pawn -> get_strength()>0){
    if(m==19){
        current_pawn -> finished_the_game();
        Board[19]=0;
        break;
    }
    tmp_array2[0]=Board[m+1];
    Board[m+1]=current_pawn -> get_number();
    Board[m]=tmp_array1[0];
    tmp_array1[0]=tmp_array2[0];
    tmp_array2[0]=0;
    current_pawn -> decrease_strength();
    if(m==0) Board[0]=tmp_array_initial[0];
    tmp_array_initial[0]=0;
    if(current_pawn -> get_strength()==0){
        if(which_player==1){
            if(tmp_array1[0]==3) Pawn3.beaten();
            if(tmp_array1[0]==4) Pawn4.beaten();
            if(tmp_array1[0]==1&&current_pawn -> get_number()==2) Pawn1.beaten();
            if(tmp_array1[0]==2&&current_pawn -> get_number()==1) Pawn2.beaten();
        }
        if(which_player==2){
            if(tmp_array1[0]==1) Pawn1.beaten();
            if(tmp_array1[0]==2) Pawn2.beaten();
            if(tmp_array1[0]==3&&current_pawn -> get_number()==4) Pawn3.beaten();
            if(tmp_array1[0]==4&&current_pawn -> get_number()==3) Pawn4.beaten();
        }
        tmp_array1[0]=0;
    }
    m++;
}
```

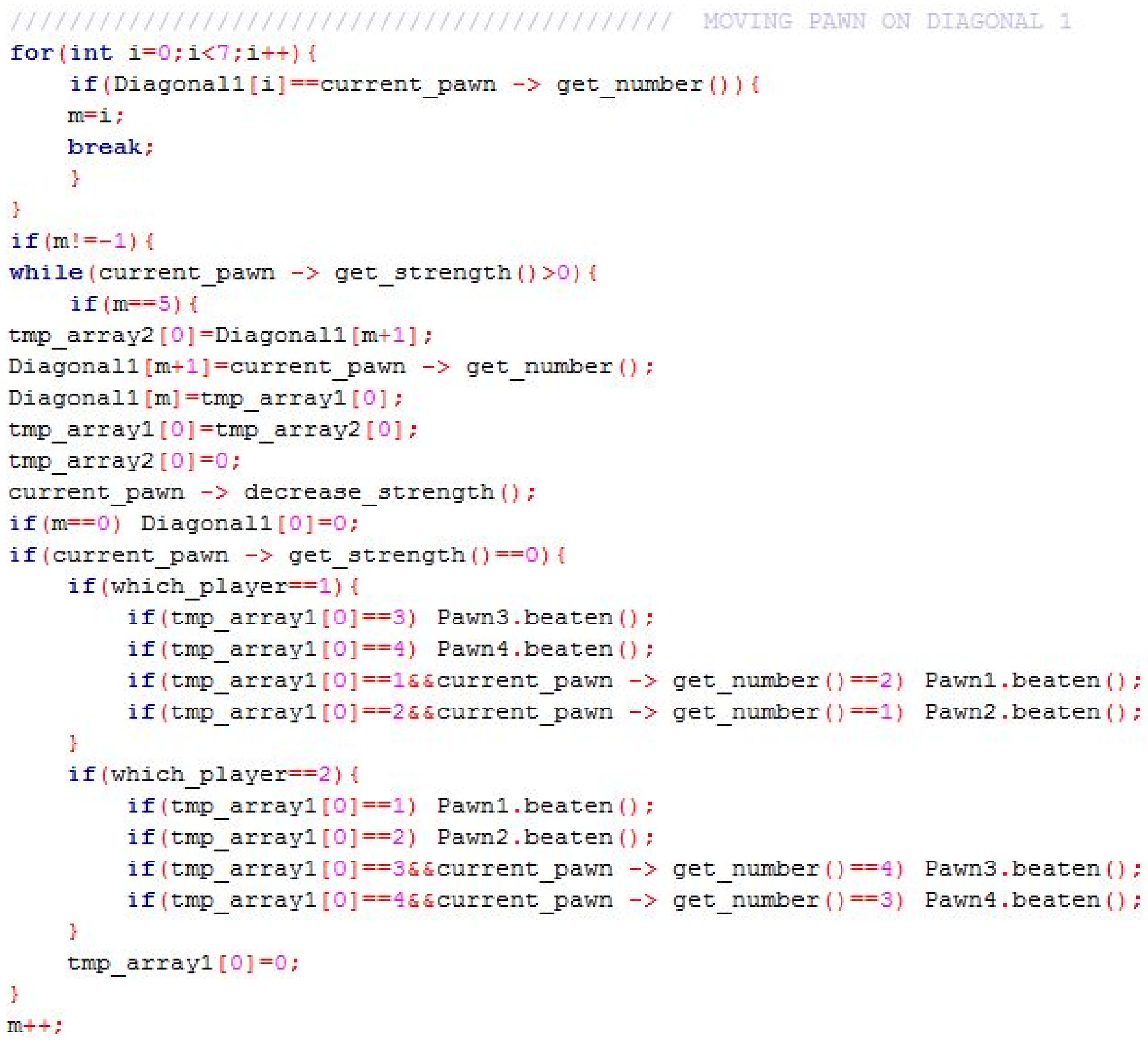
The third one '*PLACING PAWN ON DIAGONAL 1*' checks if the pawn is on upper-right corner of the board and value of its variable '*move_by_how_much*' is equal to 0. If both of these conditions are true, then the program places the pawn on the first tile of array '*Diagonal1*'.

```
///////////////////////////////////////////////  PLACING PAWN ON DIAGONAL 1
if(current_pawn -> get_strength()==0&&m==5){
    if(which_player==1){
        if(Diagonal1[0]==3) Pawn3.beaten();
        if(Diagonal1[0]==4) Pawn4.beaten();
        if(Diagonal1[0]==1&&current_pawn -> get_number()==2) Pawn1.beaten();
        if(Diagonal1[0]==2&&current_pawn -> get_number()==1) Pawn2.beaten();
    }
    if(which_player==2){
        if(Diagonal1[0]==1) Pawn1.beaten();
        if(Diagonal1[0]==2) Pawn2.beaten();
        if(Diagonal1[0]==3&&current_pawn -> get_number()==4) Pawn3.beaten();
        if(Diagonal1[0]==4&&current_pawn -> get_number()==3) Pawn4.beaten();
    }
    Diagonal1[0]=Board[5];
    Board[5]=0;
}
```

The fourth one ‘*PLACING PAWN ON DIAGONAL 2*’ checks if the pawn is on upper-left corner of the board and value of its variable ‘*move_by_how_much*’ is equal to 0. If both of these conditions are true, then the program places the pawn on the first tile of array ‘*Diagonal1*’.

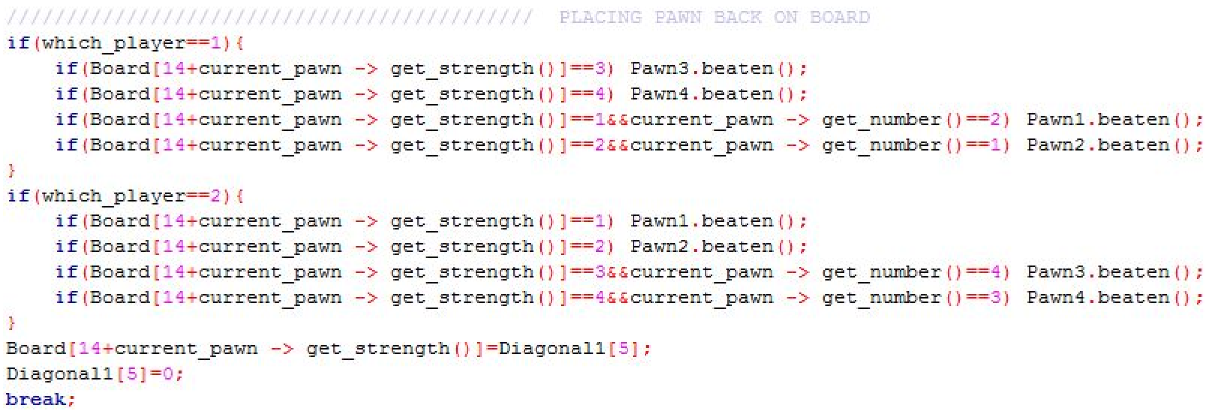
```
///////////////////////////////////////////////  PLACING PAWN ON DIAGONAL 2
if(current_pawn -> get_strength()==0&&m==10){
    if(which_player==1){
        if(Diagonal2[0]==3) Pawn3.beaten();
        if(Diagonal2[0]==4) Pawn4.beaten();
        if(Diagonal2[0]==1&&current_pawn -> get_number()==2) Pawn1.beaten();
        if(Diagonal2[0]==2&&current_pawn -> get_number()==1) Pawn2.beaten();
    }
    if(which_player==2){
        if(Diagonal2[0]==1) Pawn1.beaten();
        if(Diagonal2[0]==2) Pawn2.beaten();
        if(Diagonal2[0]==3&&current_pawn -> get_number()==4) Pawn3.beaten();
        if(Diagonal2[0]==4&&current_pawn -> get_number()==3) Pawn4.beaten();
    }
    Diagonal2[0]=Board[10];
    Board[10]=0;
}
}
m=-1;
```

The fifth one ‘*MOVING PAWN ON DIAGONAL 1*’ moves the pawn along the array ‘*Diagonal1*’.



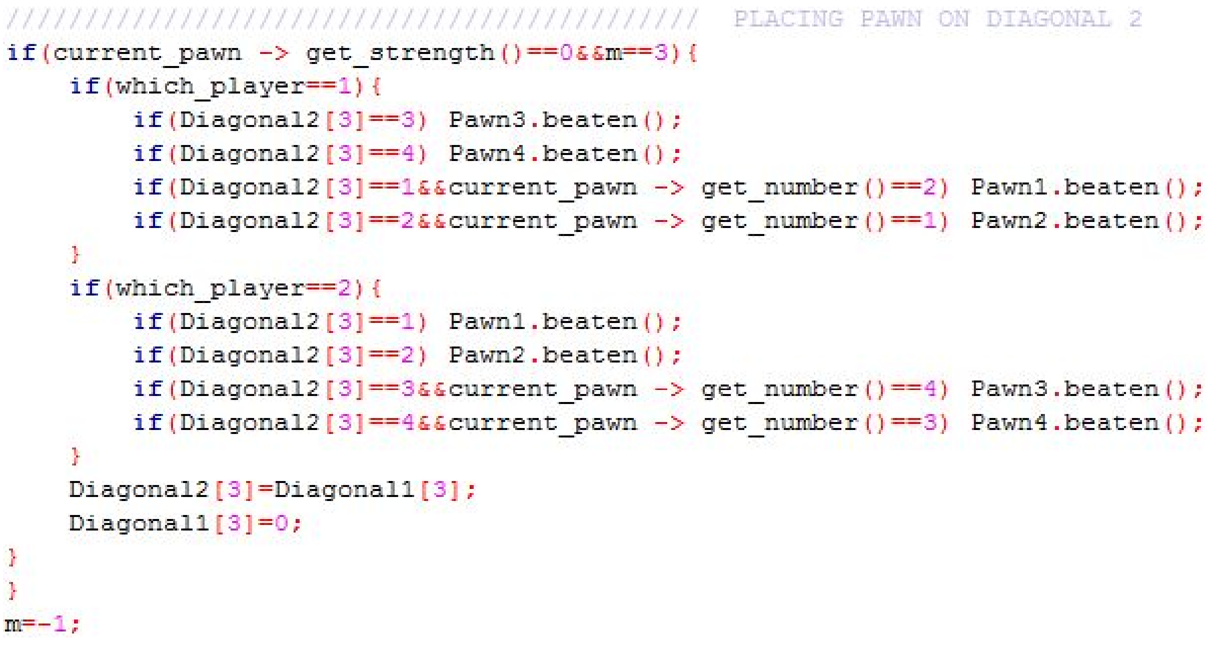
```
/////////////////////////////////////////////  MOVING PAWN ON DIAGONAL 1
for(int i=0;i<7;i++){
    if(Diagonal1[i]==current_pawn -> get_number()){
    m=i;
    break;
    }
}
if(m!=-1){
while(current_pawn -> get_strength()>0){
    if(m==5){
tmp_array2[0]=Diagonal1[m+1];
Diagonal1[m+1]=current_pawn -> get_number();
Diagonal1[m]=tmp_array1[0];
tmp_array1[0]=tmp_array2[0];
tmp_array2[0]=0;
current_pawn -> decrease_strength();
if(m==0) Diagonal1[0]=0;
if(current_pawn -> get_strength()==0){
    if(which_player==1){
        if(tmp_array1[0]==3) Pawn3.beaten();
        if(tmp_array1[0]==4) Pawn4.beaten();
        if(tmp_array1[0]==1&&current_pawn -> get_number()==2) Pawn1.beaten();
        if(tmp_array1[0]==2&&current_pawn -> get_number()==1) Pawn2.beaten();
    }
    if(which_player==2){
        if(tmp_array1[0]==1) Pawn1.beaten();
        if(tmp_array1[0]==2) Pawn2.beaten();
        if(tmp_array1[0]==3&&current_pawn -> get_number()==4) Pawn3.beaten();
        if(tmp_array1[0]==4&&current_pawn -> get_number()==3) Pawn4.beaten();
    }
    tmp_array1[0]=0;
}
m++;
```

The sixth one '*PLACING PAWN BACK ON BOARD*' checks if the pawn is on lower-left corner of the board and value of its variable '*move_by_how_much*' is greater than 0. If both of these conditions are true, then the program places the pawn back on the 15th tile of array '*Board*' and moves it by number corresponding to value of '*move_by_how_much*'.



```
//////////////////////////////////////////  PLACING PAWN BACK ON BOARD
if(which_player==1){
    if(Board[14+current_pawn -> get_strength()]==3) Pawn3.beaten();
    if(Board[14+current_pawn -> get_strength()]==4) Pawn4.beaten();
    if(Board[14+current_pawn -> get_strength()]==1&&current_pawn -> get_number()==2) Pawn1.beaten();
    if(Board[14+current_pawn -> get_strength()]==2&&current_pawn -> get_number()==1) Pawn2.beaten();
}
if(which_player==2){
    if(Board[14+current_pawn -> get_strength()]==1) Pawn1.beaten();
    if(Board[14+current_pawn -> get_strength()]==2) Pawn2.beaten();
    if(Board[14+current_pawn -> get_strength()]==3&&current_pawn -> get_number()==4) Pawn3.beaten();
    if(Board[14+current_pawn -> get_strength()]==4&&current_pawn -> get_number()==3) Pawn4.beaten();
}
Board[14+current_pawn -> get_strength()]=Diagonal1[5];
Diagonal1[5]=0;
break;
```

The seventh one ‘*PLACING PAWN ON DIAGONAL 2*’ checks if the pawn is in the middle of the board and value of its variable ‘*move_by_how_much*’ is equal to 0. If both of these conditions are true, then the program places the pawn on the fourth tile of array ‘*Diagonal2*’.



```
////////////////////////////////////////////////  PLACING PAWN ON DIAGONAL 2
if(current_pawn -> get_strength()==0&&m==3){
    if(which_player==1){
        if(Diagonal2[3]==3) Pawn3.beaten();
        if(Diagonal2[3]==4) Pawn4.beaten();
        if(Diagonal2[3]==1&&current_pawn -> get_number()==2) Pawn1.beaten();
        if(Diagonal2[3]==2&&current_pawn -> get_number()==1) Pawn2.beaten();
    }
    if(which_player==2){
        if(Diagonal2[3]==1) Pawn1.beaten();
        if(Diagonal2[3]==2) Pawn2.beaten();
        if(Diagonal2[3]==3&&current_pawn -> get_number()==4) Pawn3.beaten();
        if(Diagonal2[3]==4&&current_pawn -> get_number()==3) Pawn4.beaten();
    }
    Diagonal2[3]=Diagonal1[3];
    Diagonal1[3]=0;
}
}
m=-1;
```

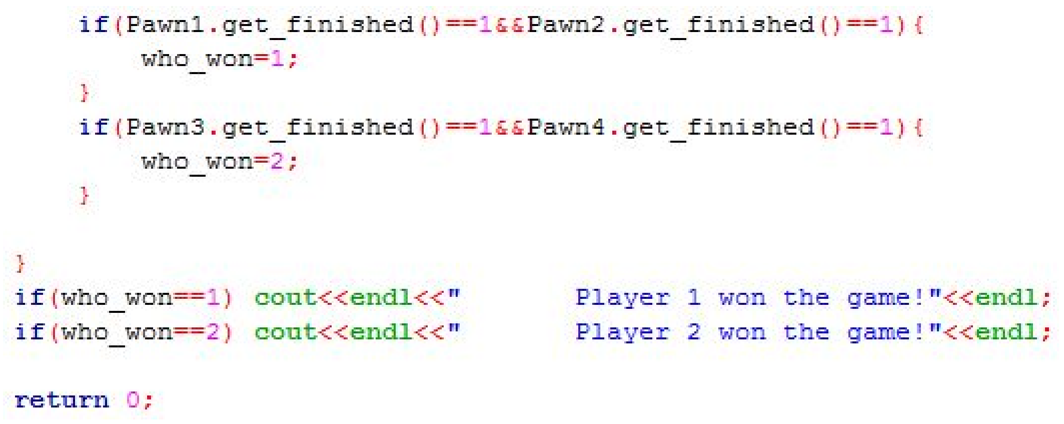
The eighth one ‘*MOVING PAWN ON DIAGONAL 2*’ moves the pawn along the array ‘*Diagonal2*’.

```
/////////////////////////////////////////////  MOVING PAWN ON DIAGONAL 2
for(int i=0;i<7;i++){
    if(Diagonal2[i]==current_pawn -> get_number()){
    m=i;
    break;
    }
}
if(m!=-1){
while(current_pawn -> get_strength()>0){
    if(m==5){
        current_pawn -> finished_the_game();
        Diagonal2[5]=0;
        break;
    }
    tmp_array2[0]=Diagonal2[m+1];
    Diagonal2[m+1]=current_pawn -> get_number();
    Diagonal2[m]=tmp_array1[0];
    tmp_array1[0]=tmp_array2[0];
    tmp_array2[0]=0;
    current_pawn -> decrease_strength();
    if(m==0) Diagonal2[0]=0;
    if(current_pawn -> get_strength()==0){
        if(which_player==1){
            if(tmp_array1[0]==3) Pawn3.beaten();
            if(tmp_array1[0]==4) Pawn4.beaten();
            if(tmp_array1[0]==1&&current_pawn -> get_number()==2) Pawn1.beaten();
            if(tmp_array1[0]==2&&current_pawn -> get_number()==1) Pawn2.beaten();
        }
        if(which_player==2){
            if(tmp_array1[0]==1) Pawn1.beaten();
            if(tmp_array1[0]==2) Pawn2.beaten();
            if(tmp_array1[0]==3&&current_pawn -> get_number()==4) Pawn3.beaten();
            if(tmp_array1[0]==4&&current_pawn -> get_number()==3) Pawn4.beaten();
        }
        tmp_array1[0]=0;
    }
    m++;
}
```

In all of these sections I have implemented a method of replacing the pawns currently occupying tile with the new ones. The old ones are deleted from the board by calling function ‘*beaten*’.

The algorithm of moving the pawn along one of the arrays looks like that: The program searches for our pawn on all the arrays and gets its location (sets it to variable ‘*m*’). Then it moves the pawn by 1 tile until its value of ‘*move_by_how_much*’ is equal to 0.

Finally there is one last method. After each iteration of choosing which pawn to move and moving said pawn the program checks if both of players pawns have completed their path. If so, then it assigns appropriate value to variable ‘*who_won*’, prints this information on screen and stops the program.



```
    if(Pawn1.get_finished()==1&&Pawn2.get_finished()==1){
        who_won=1;
    }
    if(Pawn3.get_finished()==1&&Pawn4.get_finished()==1){
        who_won=2;
    }

}
if(who_won==1) cout<<endl<<"       Player 1 won the game!"<<endl;
if(who_won==2) cout<<endl<<"       Player 2 won the game!"<<endl;

return 0;
```

5. Conclusions

Regarding the topics I used in my program:

- Inheritance, Polimorphism and RTTI are used on objects of class ‘*Move*’. All the classes ‘*Move*’ could have been rewritten into 1 class, as they don’t have much content, but I could not think of any other method of implementation of Inheritance and RTTI and any other classes I could create, so I left it as it is.
- Exceptions are used in part of *main* in which the player chooses the pawn he wants to move.

There is some problem with code::blocks where after calling function ‘*typeid’* it prints out the name of class, but also number of letters in that name (3Gae, 2D0, 4Geol). I don’t know why there’s such error.

In *main* I included the whole algorithm of choosing which pawn current player wants to move and moving said pawn along the board. I tried to move these functions into separate file, but the program stopped working as intended and as I could not find the reason for its behaviour I left it all in *main* making it roughly 400 lines long.