I – INTRODUCTION

As part of the beginning of the second semester, we were asked to do a third project. This one is about making a game called "Kapture". The aim of this game is to catch the flag of the opposite player and to bring it back next to your flag. There are of course several rules we had to respect which will be describe all along our report.

We manage to do our best to respect and apply each of the rules of the game. In fact, only one is missing, this is the saving rule, we didn't find how to save the game even if the program was off. Except this, all is working fine.

One must know this project was time-consuming. Indeed, we spent more than 20 hours on it. However, as we observed for the previous projects, we learnt so many new things about C language and it was a real pleasure to code a game this time.

II-ARCHITECTURE OF THE KAPTURE

Obviously, we had to do several functions to code our game the best way we could. We will describe the most important to explain their purpose.

Before describing our functions, you should know that we have created a 2D dynamic array which has 10 lines and 16 columns, we called it "field".

Function fill:

This function receives our field and fills it with all the elements our field must have. For the obstacles, we assigned a random number between 0 and 100 to a variable x and we made two loops, one for our lines (i) and another for our columns (j), to fill the field of the obstacles. For instance, if x was between 0 and 60 Field [i][j] was grass, if it was between 60 and 85 it was forest and 85 and between 85 and 100 it was river. With this technique, we managed to regulate the chance of each obstacle to be created in the field. Moreover, in this function we "create" our pawns. All our game is reposing on the fact that our pawns are numbers for us and we know their initial position by placing them on the field in this function. We used this technique because it was easier for us to assimilate their moving points to the value of the pawn. We created 10 pawns, 5 for each team. 2 scouts and one Infantry and Trooper for each player.

For the player 1:

The flag is 0

The Infantry pawn is 30

The infantry pawn which is flag bearer is 53

The Scout 1 is 50

The Scout 2 is 51

The Trooper pawn is 20

The trooper pawn which is flag bearer is 52

For the player 2, we just add 100 for each pawn and its flag is -1.

Function Flag:

This function allows the user to decide if he wants to place his flag or put it randomly on the field. We also manage to make him re enter the coordinated of his flag if he enters coordinates that doesn't correspond to his camp or if the coordinates are outside the field.

Function pos:

Like many times in our game, there are again two functions of pos. 1 for each player. Before describing the architecture of this function, we must explain why we have to do this. In fact, we created 2 arrays, respectively called "Position" and "Position2" in which we put the position and the value where our pawn is. We also put the position of the corresponding flag of their team in both arrays. Let's take an example to explain. For player1, the Trooper pawn is the first pawn he has to play so imagine it is at line 5, columns 7 and on the case where a forest is, then at the first line of position it will be : Position[0][0]=5 Position[0][1]=7 and Position[0][2]=3.

The purpose of the function pos is to fill the arrays Position and pos2 for position2 which were already allocated. In this function, we enter the coordinates of the initial position of the pawns. Then, another function will change the 2 arrays in function of the displacement of each pawn on the field.

1	0	1
4	0	1
5	0	1
8	0	1
?	?	1

Position1 when the game start

1	15	1
4	15	1
5	15	1
8	15	1
?	?	1

Position2 when the game start

Function player:

This function is kind of complicated. It allows the player to move. There are again two functions player. First, we regulate the order of the pawn by initialize a cpt to 0. Then we put a while loop and we say that k=position[cpt][0] and L=position[cpt][1]. Thanks to this mechanism, the pawn which has to play is field[k][L]. Then, we check if field[k][L] is greater than 51 (or 151 for player 2). We do this check to verify if it's a flag bearer or not. If it is, we do field[k][L] modulo 50. If it's not, we divide field[k][L] by 10 and assimilate this value to our variable PM (moving point). For player 2 it is the same just we must subtract 100 before each calculus.

At this step of the function, we know which pawn has to move and how many PM he has. After this, we create a Do While loop inside the while loop and we say to the player that he can move his pawn, or he may enter h for help. If he enters "h", all the rules and the controls of the game will be printed on the screen.

One must know that we made a switch to allow the user to move, the player must move by entering a letter. Each letter represents a direction, “z” to move forward, “a” to move left forward, “q” to move left, etc… Let describe one case to see how it really works.

Before entering the switch, we initialize the variable “pos” to field[k][L]. So pos is equal to the value of our pawn. Then imagine the player enters “z”, which means he wants to move forward. Therefore, we enter in the case “z”, we initialize the variable save to the value which is in field[k-1][L] (indeed, move forward means the line where we are minus one.). Then we say that, at the case where the pawn will be, its value is equal to the value of the pawn by entering field[k-1][L]=pos. Moreover, we want to give the value which was in field[k-1][L] to field[k][L] so we enter : field[k][L]=position[cpt][2]. Finally, we must give to the array “position” the value where our pawn is (river, forest or grass) so we enter: position[cpt][2]=save.

Because the pawn has moved forward, we need to decrement k (the variable which represent lines.)

At the end of the movement, we say that k is equal to the line where our pawn is, and j is equal to the column where our pawn is.

After this, we do two loops, those loops will serve to check what is around our pawn without checking its own position obviously. If it’s finds a number greater than 53 and minus 200 or a number equal to -1 (this is the number, we attributed to the flag of player 2) it starts our function called “battle” which will be described next.

Then, we close our Do while loop if the PM-the case where is our pawn is less or equal to 0.

When this loop is closed, we still are in the while loop, so we increment cpt and we pass to the other pawn.

Finally, at each action or each changing, we print the field with our function print which will be described as well.

Function battle:

In fact, there are two functions for the battles, one for the player 1 and one for the player 2 but of course I will describe only one because they are quite identic. So, this function is a huge switch. As a reminder, when this function occurs, this is because there is either a pawn or enemy flag around our pawn. However, for each case (it can be an infantry or a trooper or a scout, and my pawn can be also either an infantry, a trooper or a scout). In fact, there are different ends with different beginning, so we had to do each case one by one for the player 1 and for the player 2. Obviously, in Battle 1, there are only the cases for player 1. In this function, field[p1i][p2j] is equal to the coordinates of our pawn and field[p2i][p2j] is equal to the coordinates of the enemy pawn. Moreover, the save variable is equal to our position.

I will describe two cases (One when there is battle between two pawns so that I hope it will be clear.)

For instance, when our pawn is an infantry and it is against a scout1, we will enter in the case 150 when save is equal to 30. in this case, we will return the scout manually in our code at his initial position and reinitialize our arrays “position”.

When it’s a flag, if only it’s not a scout and for example it’s an infantry pawn, then we change it’s value from 30 to 53 by entering in the case -1 when save is equal to 30 so that the function player can see it’s a flag bearer and conserve the same PM than before even if it’s considered as a scout in battle.

Function fog:

This function fills the array visited with 1 when a pawn moves. In fact, when a pawn will move, it will fill the array with 1 all around him in the area you asked to unveil. Thanks to this change in 1, the position filled by 1 will always be understood as visible for the function print regarded as which player is moving his pawn and which visited array it is. We fill one array for the player 1(visited1) and another array for player 2 (visited2) and which array will save the part of the field which has been unveiled so that each player will see its own visited area or unveiled because of the combination of the function print and fog.

Function print:

This function allows to print the field (with colours). It receives the field, the array visited and the coordinates of the pawn that you are moving. Visited is an array of the size of the field which has been filled with zeros. We change this array with function fog by transforming the 0 into 1 in the area he deblocks when a pawn moves. Then we print the field only where there are 1 or in the area around the pawn. The area is decided thanks to the coordinates of the pawn we are moving which are given by the calling of the function. Finally, this function prints in grey the area where there are 0 and around the pawn but not in the area we have to unveil.

Function flagdrop:

This function makes the flag drop on a square adjacent to a pawn which was bearing it. We use it when the bearing pawn loses a battle. We managed to make drop the flag on a free square.

Function game over:

At the end of a turn of a player, we search his flag on the field and when we find it we check if it is at its starting position. If it's the case, then we check around the flag and if we find a number which signifies that this a flagbearer, then we return 1, which will stop the player function and enters in the variable over. As soon as the variable is equal to 1, we stop the while loop on the main and we don't launch the player2 function. Finally, we print a message of victory for the corresponding player.

Function movement:

This function receives the case where a player wants to move his pawn and it checks if it's in the field and if the case is not occupied by another pawn or by his flag. If it is, we return 1 which will make the function player print that he can't not move here, and he will be able to move again.

MAIN:

We allocate all the arrays we need thanks to our function allocate. Then we fill them thanks to the functions we have described before. After this, we ask the user if he wants to choose the coordinate of his flag or not with the function flag. Moreover, we start a while loop which will call function player 1 and player 2, the while loop will continue until one of those two functions returns 1 in over 1 or over 2 thanks to the function game over. Finally, when one these two variables become 1, we stop the while loop and we print the victory message for the corresponding winner and we free all our arrays.

III- DIFFICULTIES

We had not a lot of difficulties to find our ideas to solve the different problems of this game. But we have encountered difficulties in making our ideas working with C language. At first, we had a problem with functions because we thought that we had to use call by reference to change what was inside the field. Therefore, we lost much time trying to find why our program didn't work.

After this we lost again more time trying to make battle work. In fact, our algorithm was good but maybe the idea of a function battle for each player was not the best because it makes a huge function and its hard to find the errors in those functions. For example, it was a mess to check all the different cases for each player for battles.

Loosing so much time has been sometimes discouraging to finish the game, but we persevered.

IV SOME ALGORITHMS

```
player1(field: 2D array of integer, position: 2D array of integer, position2: 2D array of integer, visited1: 2D array of integer): integer

Changed parameters: field, position, position2, visited1

Local variable: i, j, k, l, PM, pos, save, cpt, test, cpt1, cpt2, end: integer

                deplacement: character

BEGIN

cpt <- 0

while (cpt<4) do

        k <- position[cpt][0]

        l <- position[cpt][1]

        if (field[k][l]>51) then

                PM <- (field[k][l] % 50)

        else

                PM <- field[k][l]/10

        End if

        Do

                Write ("PLAYER 1")

                Print (field, visited1, k, l)

                Write ("Move your pawn or enter h for help")

                Write ("You have:”, PM, "PM”)

                Read (deplacement)

                 PM <- PM-position[cpt][2]

                pos <- field[k][l];

                switch(deplacement)

                case 'z':

                test <- movement1(field[k-1][l], k-1, l, field[k][l])
```

```
if(test=2)
        save <- field[k-1][l]
    field[k-1][l] <- pos
    field[k][l] <- position[cpt][2]
        position[cpt][2] <- save
        k <- k-1
else
        Write ("You can't move here, try again.")
        PM <- PM+position[cpt][2]
    End if
case 'a':
test <- movement1(field[k-1][l-1], k-1, l-1, field[k][l]);
if(test=2)
    save <- field[k-1][l-1]
        field[k-1][l-1] <- pos
        field[k][l] <- position[cpt][2]
        position[cpt][2] <- save
        k <- k-1
        l <- l-1
else
        Write ("You can't move here, try again.")
        PM <- PM+position[cpt][2]
End if
case 'q':
test=movement1(field[k][l-1], k, l-1, field[k][l]);
if(test=2)
        save <- field[k][l-1]
        field[k][l-1] <- pos
    field[k][l] <- position[cpt][2]
        position[cpt][2] <- save
     l <- l-1
    else
        Write ("You can't move here, try again.");
        PM <- PM+position[cpt][2]
End if
case 'w':
test <-movement1(field[k+1][l-1], k+1, l-1, field[k][l])
 if(test=2)
```

```
                save <- field[k+1][l-1];
                field[k+1][l-1] <- pos;
                field[k][l] <- position[cpt][2];
                position[cpt][2] <- save;
        k <- k+1
                l <- l-1
else
                Write ("You can't move here, try again.");
                PM <- PM+position[cpt][2]
End if
case 'x':
test <- movement1(field[k+1][l], k+1, l, field[k][l]);
if(test=2)
                save <- field[k+1][l]
                field[k+1][l] <- pos
                field[k][l] <- position[cpt][2]
        position[cpt][2] <-save
        k <- k+1
 else
                Write ("You can't move here, try again. ")
                PM <- PM+position[cpt][2]
case 'c':
test <- movement1(field[k+1][l+1], k+1, l+1, field[k][l]);
if(test=2)
                save <- field[k+1][l+1]
        field[k+1][l+1] <- pos
        field[k][l] <- position[cpt][2]
                position[cpt][2] <- save
                k <- k+1
                l <- l+1
else
                Write ("You can't move here, try again.")
                PM <- PM+position[cpt][2]
case 'd':
test <- movement1(field[k][l+1], k, l+1, field[k][l])
if(test=2)
                save <- field[k][l+1]
                field[k][l+1] <- pos
```

```
                    field[k][l] <- position[cpt][2]
                    position[cpt][2] <- save
                    l <- l+1
else
                    Write ("You can't move here, try again.")
                    PM <- PM+position[cpt][2]
case 'e':
test <- movement1(field[k-1][l+1], k-1, l+1, field[k][l])
if(test=2)
                    save <- field[k-1][l+1]
                    field[k-1][l+1] <- pos
                    field[k][l] <- position[cpt][2]
                    position[cpt][2] <- save
                    l <- l+1
          k = k-1
else
                    Write ("You can't move here, try again.")
                    PM <- PM+position[cpt][2]
case 'h':
rules()
default:
PM=PM-100
End switch
position[cpt][0] <- k
position[cpt][1] <- l
fog(visited1, k, l)
cpt1 <- k
cpt2 <- l

for i <- -1 to 2 do
          for j<- -1 to 2
          if(cpt1+i>=0 and cpt2+j>=0 and cpt1+i<10 and cpt2+j<=15)
                              if((field[cpt1+i][cpt2+j]>53 and field[cpt1+i][cpt2+j]<200) or field[cpt1+i][cpt2+j]=-1)
                                        Write ("PLAYER 1")
                                        Write ("/!/ BATTLE /!/ ")
                                        print(field, visited1, k, l)
                                        Write new line
                                        battlep1(field, position2, position, field[k][l], k+i, l+j, &k, &l)
                    End if
                End if
            End for
        End for
        End <- gameover (field, position, 0, 52, 53)
        if(end=1)
        print (field, visited1, k, l)
        return 1
        End if
    while(PM-position[cpt][2]>=0)
    cpt <- cpt+1
End while
Write ("PLAYER 1 ")
print (field, visited1, k, l);
return 0;
END


Battlep1 (Field, position2, position, save, p2i, p2j, p1i, P1j)
Changed variables (Field, position2, position, p1i, p1j)
Copied variables (Save, p2i, p2j)
Local variables: x, i, j
Begin
Switch(field[p2i][p2j]
    case -1:
        if(save=20)
            field[p1i][p1j] <- 52
            field[p2i][p2j] <- position2[4][2]
        End if
        if(save=30)
            field[p1i][p1j] <- 53
            field[p2i][p2j] <- position2[4][2]
        End if
        if(save=51 or save=50)
            Write  ("You can't pick a flag with a scoot")
        End if
    case 150:
        if(save=30 or  save=20)
            field[5][15] <- 150
```

```
                    field[p2i][p2j] <- position2[2][2]
                    position2[2][2] <- 1
                    position2[2][0] <- 5
                    position2[2][1] <- 15
            End if
    End switch
END
```

And same for all the case (130, 120, 152, 153, 151) with respect to all the possibilities of value that can take our pawn(30, 20, 52, 53, 51, 50)…

```
ALGORITHM: MAIN
VARIABLES: field, position, position2, visited1, visited2 : 2D arrays of integer
           over1, over2 : integer
BEGIN
alloc(&field, 10, 16)
fill(field)
alloc(&position, 5, 3)
alloc(&position2, 5, 3)
alloc(&visited1, 10, 16)
fill0(visited1)
alloc(&visited2, 10, 16)
fill0(visited2)
over1 <- 0
over2<- 0
flag1(field, position)
flag2(field, position2)
pos1(position)
pos2(position2)
while(over1=0 and over2=0)
    over1 <- player1(field, position, position2, visited2)
    if(over1=0)
        over2<-player2(field, position2, position, visited1)
    End if
End while
if(over1=1)
    Write ("WELL PLAYED PLAYER 1")
End if
if(over2=1)
    Write ("WELL PLAYER PLAYER 2")
```

```
End if
END
```

V- CONCLUSION:

To put in a nutshell, this project was very hard and long to achieve. However, it was instructive and entertaining to code a game. Indeed, playing to our game on our computer and let our relatives test it made us proud and more mature for the future. Nevertheless, we know that our code isn’t optimised as best as he could be, but we tried to do our best and it will be a great example for future projects. The project has tested our limits in terms of patience and perseverance and we have further increased them. Hope you will enjoy our game.