Universidad Central de Venezuela

Facultad de Ciencias

Escuela de Computación

Licenciatura en Computación

Algoritmo y Programación

Noviembre de 2022

Proyecto #2. Dungeons & Dragons.

Sofía Marcano C.I:29.765.263 Sección: C3 & Nicole Llerena C.I: 30.444.890 Sección: C2

Con el siguiente informe, damos por describir las instrucciones que se realizaron para ejecutar el algoritmo indicado.

Para ‘int main’ pasamos por parámetros una variable de tipo int ‘argc’ y un arreglo de strings ‘argv’ para obtener una cantidad de argumentos y tenerlos en un arreglo. Allí tendremos los atributos para el archivo de entrada, el tablero y el archivo de salida. Definimos tres objetos para abrir el archivo, obtener los datos del tablero y las salidas. Para luego invocar a dos métodos que leerán entradas y así imprimir las salidas en el tablero.

```
int main (int argc, char* argv[]) {
    Input input;
    Board board;
    Output output;

    input.readInput(argv, board);

    output.printOutput(argv, board);

    return 0;
}
```

Estos métodos estarán dentro de la clase ‘Input’. En esta clase, tendremos por atributos un fichero ‘fstream’ para input, un ‘string’ como una variable auxiliar llamada 'aux', dos enteros que definirán la cantidad de turnos y las acciones tomadas por cada personaje en dichos turnos, y un objeto para el ataque.

Empezamos explicando el método que leerá los atributos de los personajes existentes en la simulación, que por parámetros se pasa un arreglo que guarda los personajes y el número como entero de la cantidad del tipo de personaje que se está evaluando existentes en la simulación, y con un ciclo se irá leyendo sus puntos de vida,

ataque, armadura, resistencia mágica y se define que el personaje está vivo.

Luego se mueve el cursor a la siguiente línea con la función ‘getline´.

```
class Input {
    fstream input;
    string aux;
    int action;
    int turn;
    Attack fire;

    void readAttributes (Character array[], int num) {
        for (int i = 0; i < num; i++) {
            input >> array[i].healthP;
            array[i].maxHealth = array[i].healthP;
            input >> array[i].attackP;
            input >> array[i].armorP;
            input >> array[i].magicRP;
            array[i].alive = true;

            getline(input, aux);
        }
    }
```

Pasamos a la siguiente acción para leer los turnos de los personajes, por parámetros pasamos el arreglo de personajes, el número de personajes existentes, el tablero por referencia. También declaramos la dirección, distancia y ataque con enteros.

```
void readTurns(Character array[], int num, Board &board) {
    int direction, distance, attack;

    for (int i = 0; i < num; i++) {
        input >> action;

        direction = action / 100;
        distance = action / 10 % 10;
        attack = action % 10;

        if (array[i].alive) {

            board.moveCharacter (direction, distance, array[i]);


            if ((array[i].simbol == 'K' || array[i].simbol == 'M') && (attack == 2 || attack == 3)) {
                input >> array[i].attackCoor[0].X;
                input >> array[i].attackCoor[0].Y;
            }

            fire.attack(attack, direction, array[i], board);
        }

        getline (input, aux);
    }
}
```

Allí mismo, junto a un ciclo se leerán las acciones de los personajes, luego se separa ese entero en tres números de una cifra para definir la dirección de movimiento, la distancia que el personaje se moverá y el ataque que debe realizar.

Con un ‘if’ se verifica que el personaje cuyo turno se está leyendo siga con vida mediante su atributo 'alive', en caso de que sí, movemos al personaje con el método ‘moveCharacter’ y le pasamos la dirección, la distancia y el personaje a moverse.

Con el otro ‘if’ verificamos si el personaje es un mago o clérigo y si el número del ataque a realizar corresponde a una habilidad apuntada, en caso de que se cumpla la condición se hacen dos lecturas más para obtener las coordenadas del ataque y se guardan en la primera casilla de un arreglo de coordenadas.

Luego se llama el método 'attack' que se encarga de definir qué ataque realizar y llevarlo a cabo.

Finalmente se mueve el cursor a la siguiente línea del archivo de lectura.

En esa misma clase pasamos a la acción ‘readInput’ donde tenemos un arreglo con los argumentos pasados por consola y el objeto del tablero, allí abrimos el archivo de entrada, y con un ‘if’ verificamos si el archivo se abrió correctamente.

```
public:
void readInput (char* argv[], Board &board) {

    input.open(argv[1], ios::in);


    if (input.is_open()) {

        input >> board.row;
        input >> board.col;

        getline(input, aux);

        board.fillBoard(input);

        readAttributes(G, board.numWarriors);
        readAttributes(K, board.numClerics);
        readAttributes(M, board.numMages);
        readAttributes(A, board.numArchers);
        readAttributes(E, board.numEnemies);

        input >> turn;

        getline(input, aux);
```

Dentro de ese ‘if’ leemos el tamaño del tablero, movemos el cursor a la línea de abajo, y se llena el tablero. Se lee en orden los atributos de los personajes mediante el método previamente descrito, pasando en orden los arreglos de tipo de personaje y el número de ese tipo de personaje existente. Se lee la cantidad de turnos y volvemos a mover el cursor a la línea de abajo.

Con el ‘for’ se leen los turnos y acciones de los personajes con el método ‘readTurns’ y con esto se cierra el archivo.

```
            for (int i = 0; i < turn; i++) {


                readTurns(G, board.numWarriors, board);
                readTurns(K, board.numClerics, board);
                readTurns(M, board.numMages, board);
                readTurns(A, board.numArchers, board);
                readTurns(E, board.numEnemies, board);
            }
        }

        input.close();
    }
};
```

Para la clase output, declaramos una variable de archivo para salida. Tenemos un método donde pasamos el arreglo de los personajes y la cantidad, con un ‘for’ imprimimos en orden los atributos de un tipo de personaje a la vez, luego pasamos a la acción con la cual abrimos el archivo de salida y con un ‘for’ se imprime el estado final del tablero.

```
class Output {

    fstream output;


    void printCharacter (Character array[], int num) {

        for (int i = 0; i < num; i++) {
            output << array[i].healthP << " ";
            output << array[i].attackP << " ";
            output << array[i].armorP << " ";
            output << array[i].magicRP << endl;
        }

    }
```

Y así terminamos con las impresiones de los atributos finales de los personajes, y se arma el archivo de salida.

```
    public:
    void printOutput (char* argv[], Board board) {

        output.open(argv[2], ios::out);

        for (int i = 0; i < board.row; i++) {
            for (int j = 0; j < board.col; j++) {
                output << board.board[i][j];
            }
            output << endl;
        }

        printCharacter(G, board.numWarriors);
        printCharacter(K, board.numClerics);
        printCharacter(M, board.numMages);
        printCharacter(A, board.numArchers);
        printCharacter(E, board.numEnemies);

        output.close();
    }
};
```

Ahora bien, al inicio del código empezamos con dos constantes que definen la cantidad máxima de cada tipo de héroe y de enemigos en 100 y 1000.

Tenemos la clase de coordenadas, con sus atributos públicos, donde tendremos a (x) y (y) para las coordenadas, y un constructor donde inicializamos a (x) y (y) en -1, y una acción para eliminar esas coordenadas luego.

```
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

const int maxHeroes = 100;
const int maxEnemies = 1000;

class Coordinate {
    public:
    int X;
    int Y;

    Coordinate () {
        X = -1;
        Y = -1;
    }

    void deleteCoordinate () {
        X = -1;
        Y = -1;
    }
};
```

En la clase de personaje, tenemos por atributos públicos el máximo de vida, la cantidad de vida, cantidad de ataque, armadura y resistencia mágica como enteros, con ‘boolianos’ tenemos a héroe inicializado en ‘True’ caso contrario sería un enemigo, y ‘magicAttack’ en falso para verificar si los personajes tienen un ataque mágico, y ‘alive’ para ver si sigue con vida o no, también tenemos las coordenadas de movimiento del personaje y un arreglo con las coordenadas de su ataque, y un ‘char’ para identificar a los personajes.

Junto a un constructor inicializamos la cantidad de vida, ataque, armadura y resistencia mágica en cero, y el método para eliminar coordenadas. En el siguiente pasamos los mismos atributos más el ‘booliano’ de la vida en falso.

Con el método ‘cleanAttackcoor’ limpiamos el tablero con un ‘for’ eliminando las coordenadas del ataque una vez que se termina de ejecutar un ataque.

```
void deleteCharacter () {
    alive = false;
    coor.deleteCoordinate();
}

void cleanAttackCoor () {
    for (int i = 0; i < 16; i++) {
        attackCoor[i].deleteCoordinate();
    }
}
```

Tenemos la acción ‘lowerHealthP’ donde pasamos el daño y el bool de magia, y junto al ‘if’ se verifica el tipo de daño recibido y si el personaje debe recibir daño o no.

Con el método ‘heal’ cumplimos cuando a un personaje se le debe subir los puntos de vida o no en caso de que haya sido atacado, sino se deja con su vida máxima.

```
    void lowerHealth (int damage, bool magic) {

        if (magic) {
            damage -= magicRP;
        } else {
            damage -= armorP;
        }

        if (damage > 0) {
            healthP -= damage;
        }

    }

    void heal (int heal) {
        if (healthP + heal <= maxHealth) {
            healthP += heal;
        } else {
            healthP = maxHealth;
        }
    }

};
```

Aquí pasamos a describir las clases de los personajes:

En la clase ‘Warrior’ tenemos al constructor el cual inicializa a ‘symbol’ en ‘G’ para identificar al guerrero el cual hereda de la clase ‘Character’ públicamente, los atributos, y así sucesivamente con los otros personajes con la única diferencia que en la clase ‘Cleric’ y ‘Mage’ tiene el ‘bool’ de ‘magicAttack’ en (True), y en el enemigo ‘bool’ de héroe en (False). Luego se declaran globalmente 5 arreglos de cada clase del tamaño del máximo de héroes o enemigos que pueden existir en la simulación.

```
class Warrior: public Character {
    public:
    Warrior () {
        simbol = 'G';
    }

};

Warrior G[maxHeroes];
Cleric K[maxHeroes];
Mage M[maxHeroes];
Archer A[maxHeroes];
Enemy E[maxEnemies];
```

En la clase ‘board’ nos fijaremos en el tablero, tendremos por atributos públicos las filas y columnas por entero, al tablero como una matriz estática de 500x500 de tipo 'char', y el número de personajes en enteros inicializados en cero.

```
class Board {
    public:
    int row, col;
    char board[500][500];
    int numWarriors = 0;
    int numClerics = 0;
    int numMages = 0;
    int numArchers = 0;
    int numEnemies = 0;


    Board () {

        for (int i = 0; i < 500; i++) {
            for (int j = 0; j < 500; j++) {
                board[i][j] = ' ';
            }
        }
    }
```

Un constructor donde inicializamos el tablero vacío con un ‘for’, y pasamos a los métodos:

Primero tenemos al método ‘readBoard’ donde tenemos un auxiliar y las coordenadas de (x) y (y), aquí se cuenta la cantidad de

personajes y se les asigna sus coordenadas iniciales mediante un ‘switch’.

En la acción ‘fillBoard’ tenemos a ‘fstream’ y el ‘input’ por referencia, un auxiliar por ‘string’, para rellenar el tablero con un ‘for’ y así leerlo.

```
void readBoard (char aux, int coorY, int coorX) {

    switch (aux) {
        case 'G':
            G[numWarriors].coor.Y = coorY;
            G[numWarriors].coor.X = coorX;
            numWarriors++;
        break;

        default:
        break;
    }
}

void fillBoard (fstream &input) {
    string aux;

    for (int i = 0; i < row; i++) {

        getline(input, aux);
        for (int j = 0; j < col; j++) {
            board[i][j] = aux[j];

            readBoard(aux[j], i, j);
        }
    }
}
```

En el método ‘moveCharacter’, tenemos la dirección, distancia, el personaje y sus símbolos, con esto por medio de un ‘switch’ se modificarán las coordenadas del personaje, para que vaya de arriba abajo y de derecha a izquierda. Y con el ‘if’ evitamos que el personaje salga del tablero si sus coordenadas son mayores en filas o columnas, o que salte a otro personaje en caso de encontrarse a alguien en su camino.

```
void moveCharacter (int direction, int distance, Character &chara) {

    board[chara.coor.Y][chara.coor.X] = '_';

    switch (direction) {
        case 1:

            for (int i = 0; i < distance; i++) {
                if (board[chara.coor.Y - 1][chara.coor.X] == '_') {
                    chara.coor.Y--;
                } else {

                    break;
                }
            }

        break;
    }

    board[chara.coor.Y][chara.coor.X] = chara.simbol;
}
```

Tenemos la función ‘findCharacter’ donde está el arreglo de personajes, los enteros (x) y (y), y un número. Y con un ‘for’ e ‘if’ se ubica al personaje específico dentro de su arreglo de tipo de personaje y retornamos su índice al terminar.

Con el método ‘checkhealthP’, tenemos un entero ‘i’ y un personaje. Donde verificamos si el personaje sigue con vida, de no ser así lo eliminamos del mapa.

```
int findCharacter (Character array[], int Y, int X, int num) {

    for (int i = 0; i < num; i++) {
        if (array[i].coor.Y == Y && array[i].coor.X == X) {
            return i;
        }
    }
    return -1;
}

void checkHealth (int i, Character &chara) {
    if (chara.healthP <= 0) {
        board[chara.coor.Y][chara.coor.X] = '_';
        chara.deleteCharacter();
    }
}
```

A continuación se describirán ciertos métodos especiales de algunos ataques de personajes:

Con la acción ‘checkDisparoLongevo’ pasamos un arreglo de personajes, el personaje que ataca, un entero para daño, a (x) y (y), y el número de personajes del tipo del arreglo pasado que existen. Definimos un auxiliar donde se guardará el índice retornado por el método de ‘findCharacter’, con esto verificamos que al momento de ejecutarse el ataque de disparo longevo haya cumplido con el daño de todo personaje que se haya encontrado.

```
void checkDisparoLongevo (Character array[], Character attacker, int damage, int y, int x, int num) {
    int aux;
    aux = findCharacter (array, y, x, num);

    array[aux].lowerHealth(damage, attacker.magicAttack);
    checkHealth (aux, array[aux]);
}
```

Con el método ‘checkAttack’ pasamos a los personajes, el ataque, el daño, un bool para ‘torbellino’ y un auxiliar. Con esto se calcula el total de daño que hace el personaje, junto a los ‘if donde vemos si el atacante es un héroe, entonces se cuentan a los enemigos a los que hace daño en caso contrario si el atacante es un enemigo se cuentan los héroes a los que hace daño.

Por medio de un ‘for’ y un ‘switch’, verificamos si el atacante es un héroe o un enemigo para evitar el fuego amigo y en otro caso verificamos si existe un héroe en el rango del ataque con cada personaje. Y al final limpiamos las coordenadas de los ataques.

```
void checkAttack (Character attacker, int damage, bool torbellino) {
    int aux;


    if (torbellino) {
        int numCharacter = 0;
        if (attacker.heroe) {

            for (int i = 0; i < 16; i++) {
                if (board[attacker.attackCoor[i].Y][attacker.attackCoor[i].X] == 'E') {
                    numCharacter++;
                }
            }
        } else {

            for (int i = 0; i < 16; i++) {
                switch (board[attacker.attackCoor[i].Y][attacker.attackCoor[i].X]) {
                    case 'G':
                        numCharacter++;
                    break;

        damage *= numCharacter;
    }
```

Con el método ‘checkhealP’ pasamos a ‘heal’, un ‘bool’ para ‘nova’, el símbolo del personaje más un auxiliar. Entonces si témenos al ataque ‘nova’, e inicializamos un número de personajes en cero, esto para luego saber cuantos personajes de cada tipo hay, y que junto al puntaje los cure o ataque.

```
void checkHeal (int heal, bool nova, Character chara) {
    int aux;
    if (nova) {
        int numCharacter = 0;
        for (int i = 0; i < 16; i++) {
            switch (board[chara.attackCoor[i].Y][chara.attackCoor[i].X]) {
                case 'G':
                    numCharacter++;
                break;
            }
        }
        heal += numCharacter;
    } else {
        heal += chara.attackP;
    }
```

Con esto terminado pasamos a la clase ‘Attack’ donde se describirán los métodos de habilidad y/o ataques de los personajes.Entonces por atributos tenemos el rango, el daño y la sanación.

Empezamos con el método ‘quebrajar’ donde pasamos la dirección, el símbolo de personaje y los atributos del tablero. Y donde definimos a rango en 3 y a daño en 2, junto con un ‘switch’ vemos la dirección en donde se ejecuta el ataque y se le aplica el daño al personaje que esté en dicha casilla.

```
class Attack {
    int range;
    int damage;
    int heal;

    void quebrajar(int direction, Character chara, Board &board) {
        range = 3;
        damage = 2;

        switch (direction) {
            case 1:

                for (int i = 0; i < range; i++) {
                    chara.attackCoor[i].Y = chara.coor.Y - 1;
                    chara.attackCoor[i].X = chara.coor.X + i - 1;
                }
            break;

            break;
        }

        damage += chara.attackP;

        board.checkAttack (chara, damage, false);
    }
```

En el método ‘estocada’ donde pasamos la dirección, el personaje y los atributos del tablero, con el rango en 2 y el daño en 4, y un ‘switch’ para su respectivo movimiento y que al final y se le aplica el daño al personaje que este en la casilla.

```
    void estocada (int direction, Character chara, Board &board) {
        range = 2;
        damage = 4;
        switch (direction) {
            case 1:
                for (int i = 0; i < range; i++) {
                    chara.attackCoor[i].Y = chara.coor.Y - i - 1;
                    chara.attackCoor[i].X = chara.coor.X;
                }
            break;

        }

        damage += chara.attackP;

        board.checkAttack (chara, damage, false);
    }
```

En el método ‘torbellino’ pasamos la dirección, el personaje y los atributos del tablero con su rango en 4 y daño en 1, centramos al

personaje y desde allí se ejecutará la dirección del ataque por medio de las coordenadas (x) y (y), y así ejecuta el ataque y lo verificamos.

```
void torbellino (int direction, Character chara, Board &board) {
    range = 4;
    damage = 1;

    chara.attackCoor[0].Y = chara.coor.Y;
    chara.attackCoor[0].X = chara.coor.X + 1;

    chara.attackCoor[1].Y = chara.coor.Y;
    chara.attackCoor[1].X = chara.coor.X - 1;

    chara.attackCoor[2].Y = chara.coor.Y - 1;
    chara.attackCoor[2].X = chara.coor.X;

    chara.attackCoor[3].Y = chara.coor.Y + 1;
    chara.attackCoor[3].X = chara.coor.X;

    damage += chara.attackP;

    board.checkAttack (chara, damage, true);
}
```

Con el método ‘novaDeLuz’ pasamos la dirección, el personaje y los atributos del tablero con su rango en 9 y daño en 2, más la opción de curar con cantidad 3 puntos.

Junto a un ‘for’ se ejecuta la dirección del ataque en casillas y columnas, y luego verificamos si hay algún personaje para curar o atacar y se ejecutan ambos.

```
void novaDeLuz (int direction, Character chara, Board &board) {
    range = 9;
    damage = 2;
    heal = 3;

    for (int i = 0, g = 0; i < range / 3; i++) {
        for (int j = 0; j < range / 3; j++, g++) {
            chara.attackCoor[g].Y = chara.coor.Y + i - 1;
            chara.attackCoor[g].X = chara.coor.X + j - 1 ;
        }
    }

    board.checkHeal (heal, true, chara);

    damage;

    board.checkAttack (chara, damage, false);
}
```

En el método ‘impactoSagrado’ pasamos la dirección, el personaje y los atributos del tablero con su rango en 1 y daño en 2, más la opción de curar con cantidad de 4 puntos, como es una habilidad apuntada solo ataca en una sola casilla.

```
void impactoSagrado (int direction, Character chara, Board &board) {

    range = 1;
    damage = 2;
    heal = 4;

    board.checkHeal (heal, false, chara);

    damage += chara.attackP;

    board.checkAttack (chara, damage, false);
}
```

En el método ‘luzSagrada’ pasamos la dirección, el personaje y los atributos del tablero con su rango en 1 y daño en 0, más la opción de curar con cantidad de 8 puntos, como es una habilidad apuntada solo ataca en una sola casilla.

```
void luzSagrada (int direction, Character chara, Board &board) {

    range = 1;
    damage = 0;
    heal = 8;

    board.checkHeal (heal, false, chara);

    damage *= chara.attackP;

    board.checkAttack (chara, damage, false);
}
```

En el método ‘teletransportar’ pasamos la dirección, el personaje y las opciones del tablero y con el método para mover al personaje, lo movemos en la dirección indicada.

```
void teletransportacion (int direction, Character &chara, Board &board) {
    int distance = 2;
    board.board[chara.coor.Y][chara.coor.X] = '_';

    switch (direction) {
        case 1:

            chara.coor.Y -= distance;
        break;

    }

    if (chara.coor.Y >= board.row) {
        chara.coor.Y = board.row - 1;
    }
    if (chara.coor.Y < 0) {
        chara.coor.Y = 0;
    }
    if (chara.coor.X >= board.col) {
        chara.coor.X = board.col - 1;
    }
    if (chara.coor.X < 0) {
        chara.coor.X = 0;
    }
```

En método ‘esferaDeHielo’ pasamos la dirección, el símbolo de personaje y los atributos del tablero con su rango en 1 y daño en 3, y como es apuntada solo ataca en una sola casilla.

```
void esferaDeHielo (int direction, Character chara, Board &board) {

    range = 1;
    damage = 3;

    damage *= chara.attackP;

    board.checkAttack (chara, damage, false);
}
```

En el método ‘tempestad’ pasamos la dirección, el personaje y los atributos del tablero con su rango en 9 y daño en ‘chara.attackP’, con un ‘for’ se indicará la dirección en la que el ataque se va ejecutar y verificamos si hubo un personaje allí y se le aplica la cantidad de daño.

```
void tempestad (int direction, Character chara, Board &board) {

    range = 9;
    damage = chara.attackP;

    for (int i = 0, g = 1; i < range / 3; i++) {
        for (int j = 0; j < range / 3; j++, g++) {
            chara.attackCoor[g].Y = chara.attackCoor[0].Y + i - 1;
            chara.attackCoor[g].X = chara.attackCoor[0].X + j - 1 ;
        }
    }

    board.checkAttack (chara, damage, false);
}
```

En el método ‘multiDisparo’ pasamos la dirección, el personaje y los atributos del tablero con su rango en 5 y daño en ‘chara.attackP’, con un ‘for’ definimos el rango en la dirección que será ejecutada y declaramos un ‘if’ para las casillas en donde no se atacara.

```
void multiDisparo (int direction, Character chara, Board &board) {
    range = 5;
    damage = chara.attackP;

    for (int i = 0, g = 0; i < range; i++) {
        for (int j = 0; j < range; j++) {
            if ((i == 0 || i == 4) && (j == 1 || j == 3)) {
                continue;
            } else if ((i == 1 || i == 3) && (j == 0 || j == 4)) {
                continue;
            } else if (i == 2 && j == 2) {
                continue;
            }

            chara.attackCoor[g].Y = chara.coor.Y + i - 2;
            chara.attackCoor[g].X = chara.coor.X + j - 2;
            g++;
        }
    }

    board.checkAttack (chara, damage, false);
}
```

Con el método de ‘disparoLongevo’ pasamos la dirección, el personaje y los atributos del tablero con daño en 4 y un auxiliar. Con un ‘switch’ definiremos por casos la dirección del ataque que ira hasta lo máximo del tablero pero antes verificando si hay algún personaje al cual atacar.

```
void disparoLongevo (int direction, Character chara, Board &board) {
    int aux;
    damage = 4;

    damage *= chara.attackP;


    switch (direction) {
        case 1:
            for (int i = chara.coor.Y - 1; i >= 0; i--) {
            if (chara.heroe) {
                if (board.board[i][chara.coor.X] == 'E') {
                        board.checkDisparoLongevo (E, chara, damage, i, chara.coor.X, board.numEnemies);

                        break;
                    }
            } else {
                if (board.board[i][chara.coor.X] == 'G') {
                    board.checkDisparoLongevo (G, chara, damage, i, chara.coor.X, board.numWarriors);

                    break;
                }
```

Con el método de ‘voltereta’ pasamos la dirección, el personaje y los atributos del tablero con su rango en 3 y daño en ‘chara.attackP’, y con un ‘switch’ evaluamos los casos en el que el personaje dará una voltereta hacia atrás según su movimiento, moviéndolo en dirección contraria.

```
void voltereta (int direction, Character &chara, Board &board) {
    range = 3;
    damage = chara.attackP;

    switch (direction) {
        case 1:

            for (int i = 0; i < range; i++) {
                chara.attackCoor[i].Y = chara.coor.Y - 1;
                chara.attackCoor[i].X = chara.coor.X + i - 1;
            }

            board.moveCharacter (3, 1, chara);
        break;

    }

    board.checkAttack (chara, damage, false);
}
```

Con esto terminamos los métodos de ataque y/o habilidades de los personajes, y tenemos una acción de ataque donde pasaremos el número del ataque, la dirección, el personaje y los atributos de tablero.

Entonces con un ‘switch’ evaluamos los distintos casos, empezando por el símbolo que representa al personaje y allí se ejecutara el ataque y/o habilidad del personaje según el numero escogido.

```
    public:

    void attack (int numAttack, int direction, Character &chara, Board &board) {
        switch (chara.simbol) {
            case 'G':
                switch (numAttack) {
                case 1:
                    quebrajar (direction, chara, board);
                break;
                case 2:
                    estocada (direction, chara, board);
                break;
                case 3:
                    torbellino (direction, chara, board);
                break;
                }
            break;
        }
    }
};
```

Y así concluimos con las indicaciones para realizar el algoritmo, por el cual al terminar esta parte pasamos a imprimir las salidas en un archivo.