Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Algoritmos y Programación

ANÁLISIS DEL PROYECTO N° 2

César Carios
Jhonatan Homsany
Sección C3

Para explicar el desarrollo de este proyecto es necesario dividir el mismo en partes y describir cómo se escribió el código de cada función por separado. Abordaremos cada parte del código en este orden:

1. Funciones y Acciones Globales
2. Clase Personaje
3. Personajes (Objetos)
4. Función Main

Funciones y Acciones Globales:

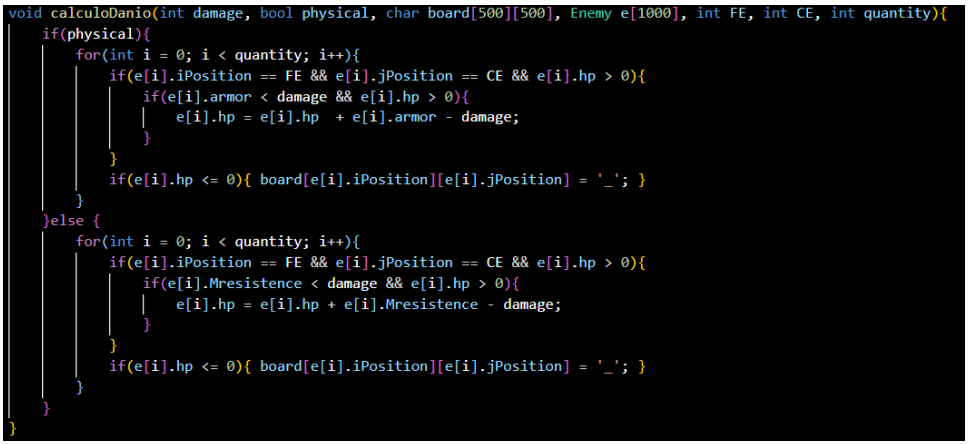
Función CharacterQuantity: Al leer el código fuente, lo primero que se observa después de incluir las librerías es una función llamada CharacterQuantity. Esta función tiene un propósito simple: recorrer el tablero de juego (matriz) y al momento de encontrar un carácter de algún personaje –héroe o enemigo– se le suma una unidad a una variable de tipo entero llamada quantity que funciona como contador. A su vez, la variable quantity nos sirve para tener la cantidad de personajes que habrá en los arreglos de cada personaje, sea héroe o enemigo.

```
int CharacterQuantity(int row, int column, char board[500][500], char inputValue){
    int quantity = 0;
    for(int i = 0; i < row; i++){
        for(int j = 0; j < column; j++){
            if(board[i][j] == inputValue){
                quantity++;
            }
        }
    }
    return quantity;
}
```

Los parámetros que recibe esta función son: la cantidad de filas y columnas de la matriz (int row, int column), la matriz (board) y los caracteres de los personajes de la simulación (inputValue).

Acción calculoDanio: Como su nombre lo indica, el objetivo de esta acción es calcular el daño que se le hace a los enemigos y a los héroes. Cada personaje de la simulación tiene una acción para calcular el daño por separado de los mismos. La primera acción para calcular el daño que se ve en la función es para

calcular el daño hecho por los héroes hacia los enemigos. Es una acción sencilla que basa su funcionamiento en un condicional: si el ataque recibido es de categoría física, hace daño teniendo en cuenta la defensa del enemigo. Pero si el ataque recibido es mágico, hace el daño teniendo en cuenta la resistencia mágica del enemigo.



```
void calculoDanio(int damage, bool physical, char board[500][500], Enemy e[1000], int FE, int CE, int quantity){
    if(physical){
        for(int i = 0; i < quantity; i++){
            if(e[i].iPosition == FE && e[i].jPosition == CE && e[i].hp > 0){
                if(e[i].armor < damage && e[i].hp > 0){
                    e[i].hp = e[i].hp  + e[i].armor - damage;
                }
            }
            if(e[i].hp <= 0){ board[e[i].iPosition][e[i].jPosition] = '_'; }
        }
    }else {
        for(int i = 0; i < quantity; i++){
            if(e[i].iPosition == FE && e[i].jPosition == CE && e[i].hp > 0){
                if(e[i].Mresistence < damage && e[i].hp > 0){
                    e[i].hp = e[i].hp + e[i].Mresistence - damage;
                }
            }
            if(e[i].hp <= 0){ board[e[i].iPosition][e[i].jPosition] = '_'; }
        }
    }
}
```

Los parámetros que recibe esta acción son los siguientes: la cantidad de daño que le hará el héroe al enemigo (int damage), una variable booleana que si es verdadera hace ataque físico y si es falsa hace ataque mágico (bool physical), el tablero de la simulación (char board), un arreglo llamado Enemy que contiene a los enemigos generados en la simulación (Enemy e [1000], la coordenada de algún enemigo que se encontró en el tablero para ejecutar en él el cálculo de daño (int FE e int CE) y la cantidad de enemigos que hay en la simulación (int quantity).

Dentro de esta función, se encuentra for que va desde i = 0 hasta la cantidad de enemigos que hay en la simulación, esto para incluir dentro un condicional que valida si el enemigo que se encuentre en la posición “i” del arreglo de enemigos contiene las mismas coordenadas que las que fueron pasadas como FE y CE, de ser así, se le realiza el respectivo cálculo de daño a ese enemigo y si ese enemigo no posee las mismas coordenadas que las suministradas, procede a aumentarse el valor de “i” tras terminar esa iteración del ciclo para así validar lo que sucede con el siguiente enemigo del arreglo.

Un detalle importante de esta función es que tiene un condicional que valida cuántos puntos de vida tiene el enemigo luego de recibir el ataque, si los puntos de vida son menores o iguales 0, el carácter E del enemigo se reemplaza por un espacio vacío en la posición donde estaba el enemigo, indicando de esta forma que el enemigo ha sido eliminado. Como se explicó en el primer párrafo de esta acción, cada personaje de la simulación tiene su propia acción para calcular el daño que reciben con la diferencia de que estos no reciben una variable de tipo

booleana, dado que el daño que realizarán los enemigos siempre será físico, por lo mismo, estas funciones no poseen un cálculo de daño para ataques mágicos. Funcionan de la misma manera que la acción calculoDanio con la única diferencia que cada acción recibe su respectivo arreglo de personajes y el nombre de esas acciones es calculoDanio seguido de la inicial del héroe que recibirá daño:

Cálculo de daño para el Guerrero:

```
void calculoDanioG(int damage, char board[500][500], Warrior g[100], int FE, int CE, int quantity){
    for(int i = 0; i < quantity; i++){
        if(g[i].iPosition == FE && g[i].jPosition == CE && g[i].hp > 0){
            if(g[i].armor < damage && g[i].hp > 0){
                g[i].hp = g[i].hp  + g[i].armor - damage;
            }
        }
        if(g[i].hp <= 0){
            board[g[i].iPosition][g[i].jPosition] = '_'; }
    }
}
```

Cálculo de daño para el Clérigo:

```
void calculoDanioK(int damage, char board[500][500], Healer k[100], int FE, int CE, int quantity){
    for(int i = 0; i < quantity; i++){
        if(k[i].iPosition == FE && k[i].jPosition == CE && k[i].hp > 0){
            if(k[i].armor < damage && k[i].hp > 0){
                k[i].hp = k[i].hp  + k[i].armor - damage;
            }
        }
        if(k[i].hp <= 0){ board[k[i].iPosition][k[i].jPosition] = '_'; }
    }
}
```

Cálculo de daño para el Mago:

```
void calculoDanioM(int damage, char board[500][500], Wizard m[100], int FE, int CE, int quantity){
    for(int i = 0; i < quantity; i++){
        if(m[i].iPosition == FE && m[i].jPosition == CE && m[i].hp > 0){
            if(m[i].armor < damage && m[i].hp > 0){
                m[i].hp = m[i].hp  + m[i].armor - damage;
            }
        }
        if(m[i].hp <= 0){ board[m[i].iPosition][m[i].jPosition] = '_'; }
    }
}
```

Cálculo de daño para el Arquero:

```
void calculoDanioA(int damage, char board[500][500], Archer a[100], int FE, int CE, int quantity){
    for(int i = 0; i < quantity; i++){
        if(a[i].iPosition == FE && a[i].jPosition == CE && a[i].hp > 0){
            if(a[i].armor < damage && a[i].hp > 0){
                a[i].hp = a[i].hp  + a[i].armor - damage;
            }
        }
        if(a[i].hp <= 0){ board[a[i].iPosition][a[i].jPosition] = '_'; }
    }
}
```

Acción Curación: Este conjunto de acciones son la solución al problema de cómo podíamos curar a los personajes con el clérigo. Al igual que la acción para

calcular el daño, hicimos una acción de curación exclusiva para cada héroe. Estas acciones funcionan basándose en un condicional: primero verificar que la vida del héroe que vaya a curar sea mayor que cero y que al curarlo, sus puntos de vida no sean mayores a los puntos de vida base que tenía. Si esto es así, se usa una variable de tipo entero llamada healing que contiene la cantidad de puntos de vida que se le sumarán a la vida del héroe. Si no cumple la primera condición, entonces la vida del héroe será igual a la vida base que tenía al iniciar la simulación, es decir, se curará por completo.

Curación para el Guerrero:

```
void CuracionG(int healing, char board[500][500], Warrior g[100], int FE, int CE, int quantity){
    for(int i = 0; i < quantity; i++){
        if(g[i].iPosition == FE && g[i].jPosition == CE && g[i].hp > 0){
            if(g[i].hp + healing < g[i].hpMax && g[i].hp > 0){
                g[i].hp += healing;
            }
            else if(g[i].hp > 0) {g[i].hp = g[i].hpMax;;}
        }
    }
}
```

Curación para el Clérigo:

```
void CuracionK(int healing, char board[500][500], Healer k[100], int FE, int CE, int quantity){
    for(int i = 0; i < quantity; i++){
        if(k[i].iPosition == FE && k[i].jPosition == CE && k[i].hp > 0){
            if(k[i].hp + healing < k[i].hpMax && k[i].hp > 0){
                k[i].hp += healing;
            }
            else if(k[i].hp > 0){k[i].hp = k[i].hpMax;}
        }
    }
}
```

Curación para el Mago:

```
void CuracionM(int healing, char board[500][500], Wizard m[100], int FE, int CE, int quantity){
    for(int i = 0; i < quantity; i++){
        if(m[i].iPosition == FE && m[i].jPosition == CE && m[i].hp > 0){
            if(m[i].hp + healing < m[i].hpMax && m[i].hp > 0){
                m[i].hp += healing;
            }
            else if(m[i].hp > 0) {m[i].hp = m[i].hpMax;}
        }
    }
}
```

Curación para el Arquero:

```
void CuracionA(int healing, char board[500][500], Archer a[100], int FE, int CE, int quantity){
    for(int i = 0; i < quantity; i++){
        if(a[i].iPosition == FE && a[i].jPosition == CE && a[i].hp > 0){
            if(a[i].hp + healing < a[i].hpMax && a[i].hp > 0){
                a[i].hp += healing;
            }
            else if(a[i].hp > 0) {a[i].hp = a[i].hpMax;}
        }
    }
}
```

Los parámetros que reciben estas acciones son exactamente los mismos parámetros que reciben las acciones para calcular daño con la pequeña e importante diferencia que en vez de recibir la variable damage, recibe la variable healing.

Acción FillBoard: Esta acción se encarga de recibir como parámetros la cantidad de filas y columnas del tablero, el tablero y el archivo de entrada donde el usuario introdujo previamente el tamaño del tablero. Su funcionamiento es sencillo, un doble for que se usará para "dibujar" el tablero de juego de la simulación con el carácter "_", además del tamaño del mismo. Todo esto se hace al principio del programa donde se lee un archivo de entrada con las características antes descritas.

```
void FillBoard(int row, int column, char board[500][500], ifstream &entrada){
    for(int i = 0; i < row; i++){
        for(int j = 0; j < column; j++){
            entrada >> board[i][j];
        }
    }
}
```

Acción PrintBoard: Esta acción se encarga de imprimir en un archivo de salida cómo quedó el tablero de juego al final del programa. Por eso recibe los mismos parámetros que la acción FillBoard, pero en vez de recibir un archivo de entrada, recibe un archivo de salida.

```
void PrintBoard(int row, int column, char board[500][500], ofstream &salida){
    for(int i = 0; i < row; i++){
        for(int j = 0; j < column; j++){
            salida << board[i][j];
        }
        salida << endl;
    }

}
```

Clase Personaje:

En esta sección se tratará todo lo relacionado con la clase personaje que creamos en nuestro programa para que sirviera como plantilla de héroes y enemigos. La llamamos clase Character. Los atributos de esta clase estaban claros desde un principio: todas variables de tipo entero que almacenarían vida, vida máxima, ataque, armadura y resistencia mágica de los héroes y enemigos. Por supuesto, esta clase también tendría como atributos dos variables que almacenarían la posición de fila y columna de la matriz, la dirección en la que se moverían los personajes, cuantas casillas se desplazarían y la habilidad que

usarían. A continuación, se adjunta una imagen donde se aprecia la declaración de estas variables que fungen como atributos de la clase personaje:

```
class Character{
    public:
    int hp, hpMax, attack, armor, Mresistence;
    int iPosition;
    int jPosition;
    int direction;
    int movement;
    int ability;
```

Inmediatamente después de declarar esas variables como atributos de la clase, se procedió a escribir el constructor de la misma, que lo dejamos vacío porque no era nuestra intención inicializar los atributos que se declararon previamente. Ahora bien, después del constructor vacío vienen una serie de cuatro acciones que nos ayudaron a tener más claro y definido lo que harían los personajes de la simulación:

Acción AskAttributes: Esta acción recibe un solo parámetro, un documento de entrada de donde se leerán las propiedades de los personajes, que son vida, ataque, armadura y resistencia mágica.

```
void AskAttributes(ifstream &entrada){
    entrada >> this->hp;
    this->hpMax = this->hp;
    entrada >> this->attack;
    entrada >> this->armor;
    entrada >> this->Mresistence;
}
```

Acción changeCoordinates: Los parámetros que recibe esta acción son todas variables de tipo entero: los dos primeros parámetros se reciben por referencia porque son valores que modificaremos (iPosition que es la fila donde se encuentra algún personaje y jPosition que es la columna donde está el mismo personaje) y los dos últimos parámetros se reciben por valor. El funcionamiento de esta función es muy simple, le asigna el valor de la variable iPosition a la variable X y el valor de jPosition a Y. Sin embargo, la funcionalidad de esta acción no se queda allí, pues es una herramienta que nos ayuda a asignarles las

coordenadas a cada uno de los personajes. Esto se profundizará en la sección sobre la función main de este análisis.

```
void changeCoordinates(int &iPosition, int &jPosition, int x, int y){
    iPosition = x;
    jPosition = y;
}
```

Acción AskDirectionAndMovement: Esta acción sirve para recibir el valor entero de la dirección, movimiento y habilidad de los personajes. Como estamos recibiendo un número entero, tuvimos que aplicar las operaciones correspondientes con los operadores div y mod respectivamente para separar cada digito que se recibe, cabe destacar que estas operaciones las hicimos con ayuda de una variable de tipo entero auxiliar que inicializamos en cero. Como esta acción solo funciona para leer datos, recibe un solo parámetro que es el archivo de entrada del que se leerá el número entero cuyos dígitos contienen la dirección, movimiento y habilidad de los personajes.

```
void AskDirectionAndMovement(ifstream &entrada){
    int aux = 0;
    entrada >> aux;
    this->direction = aux / 100;
    this->movement = (aux / 10) % 10;
    this->ability = aux % 10;
}
```

Acción AskMovement: En esta acción está definido cómo se moverán todos los personajes que haya en la simulación, sea para arriba, derecha, abajo o izquierda. Primero recibe como parámetros la cantidad de filas y columnas de la matriz tablero, luego recibe el tablero y por último el carácter del personaje que vaya a hacer uso de esta acción. Luego declaramos una variable de tipo booleana y la inicializamos en falso. Si esta variable llegase a ser verdadera en algún momento, significa que el personaje que se está moviendo no puede seguir avanzando porque se encontró con otro personaje. A continuación, viene un switch cuyos casos son directamente la dirección a la que se vaya a mover el personaje (1, 2, 3 o 4). Por ejemplo, en el caso en el que el personaje se vaya a desplazar hacia arriba, el switch entra en el caso 1. Cada uno de los casos del 1 al 4 tienen la misma estructura: un for que dentro tiene condicionales. El primer condicional valida que el personaje al hacer su movimiento no se salga del tablero. Si esto es cierto, lo siguiente que se va a revisar es si el siguiente espacio está vacío y si esto se cumple, el personaje avanza. Pero si resulta que

la casilla delante del personaje no está vacía, el programa entra en el else y hace que el personaje avance hasta una casilla antes de toparse con otro personaje.

Posteriormente, el booleano CharacterFound pasa a ser verdadero y el programa sale de ese for mediante un break. Ahora bien, si se da el caso en el que no se cumple la primera condición en la que se verifica si el personaje al hacer su movimiento no se sale del tablero, entonces el programa entra el else de más abajo. Este else hace que el personaje se mueva hasta una casilla antes de salirse del tablero. Por último, hay una validación que luego de realizar los procesos anteriores verifica que si el personaje que se está moviendo no se encontró con nada y la última casilla donde termina su movimiento está vacía, entonces hace que el personaje se mueva hasta allá.

Este switch, como se mencionó antes, tiene cuatro casos. Cada uno de los casos posee la misma estructura, pero con una lógica distinta dependiendo de la dirección del personaje. Se adjunta una imagen del caso 1, donde el personaje se mueve hacia arriba:

```
bool CharacterFound = false;

switch(direction){

    case 1:
        for(int i = 1; i <= movement;){
            if(iPosition - i >= 0){
                if(board[iPosition - i][jPosition] == '_'){
                    i++;
                }
                else {
                    board[iPosition][jPosition] = '_';
                    board[iPosition - i + 1][jPosition] = inputValue;
                    iPosition = iPosition - i + 1;
                    CharacterFound = true;
                    break;
                }
            }
            else {
                board[iPosition][jPosition] = '_';
                board[iPosition - i + 1][jPosition] = inputValue;
                iPosition = iPosition - i + 1;
                CharacterFound = true;
                break;
            }
        }
        if (!CharacterFound && board[iPosition - movement][jPosition] == '_'){
            board[iPosition][jPosition] = '_';
            board[iPosition - movement][jPosition] = inputValue;
            iPosition = iPosition - movement;
            break;
        }
```

Objetos Instanciados de la Clase Character (Personajes):

En esta sección se especificará cómo se hicieron cada una de las habilidades de los héroes.

Guerrero: Para crear al héroe guerrero lo primero que se hizo fue declarar una clase llamada Warrior que hereda públicamente los atributos y métodos de la clase Character. Luego se crearon los tres métodos que necesita tener esta clase: Quebrajar, Estocada y Torbellino. A continuación, se darán detalles de cómo funcionan cada uno de estos métodos.

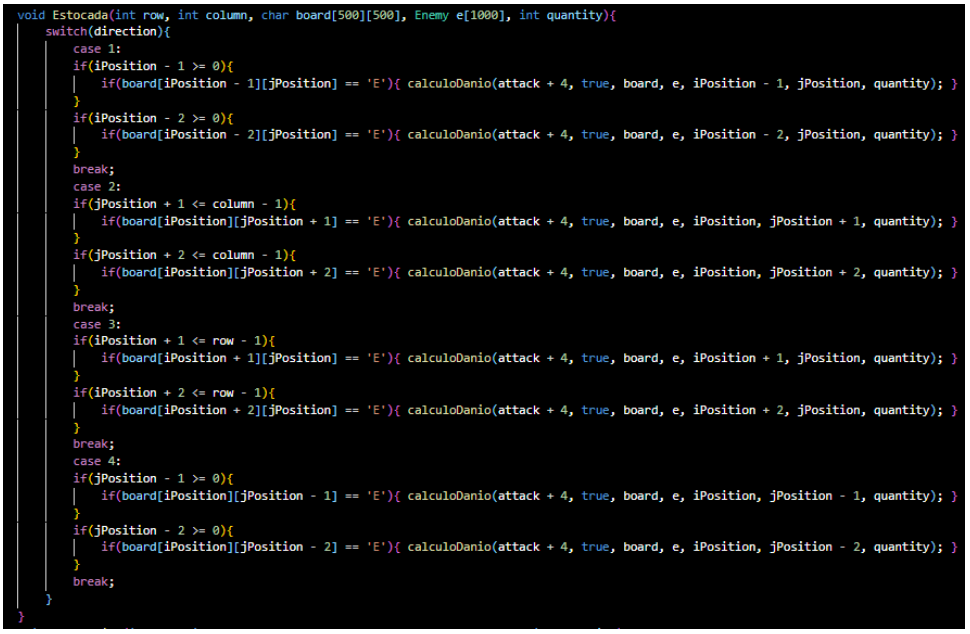
Quebrajar: Lo primero que hay dentro de esta acción son dos variables enteras de nombre X y Y, sirven para almacenar la posición en la fila y la posición en la columna respectivamente del guerrero que vaya a realizar Quebrajar. Ahora viene la parte más importante: un switch cuyos casos funcionan con base en la dirección a la que el guerrero vaya a usar la habilidad. Por ejemplo, en el caso 1 el guerrero va a usar la habilidad Quebrajar hacia arriba. Después de entrar en el caso 1, el programa pasa por 3 condicionales. Cada uno de estos tres condicionales se encarga de validar que la zona donde el guerrero va a pegar no se salga del tablero. Si esto es verdadero, entonces el programa entra en un segundo condicional que se encarga de llamar a la acción para calcular el daño si es que alguna casilla donde va a pegar la habilidad Quebrajar coincide con algún carácter "E" de enemigo. Cada uno de los siguientes casos (del 1 al 4) funciona exactamente igual que el caso descrito en este párrafo, con la pequeña diferencia que cada condicional para verificar que la habilidad no se salga del tablero varía dependiendo de la dirección a la que se vaya a aplicar la misma.

```
void Quebrajar(int row, int column, char board[500][500], Enemy e[1000], int quantity){
    int x = iPosition, y = jPosition;
    switch(direction){
        case 1:
        if(x - 1 >= 0 && y - 1 >= 0){if(board[x - 1][y - 1] == 'E'){calculoDanio(attack + 2, true, board, e, x - 1,
        if(x - 1 >= 0){if(board[x - 1][y] == 'E'){calculoDanio(attack + 2, true, board, e, x - 1, y, quantity);}}
        if(x - 1 >= 0 && y + 1 < column){if(board[x - 1][y + 1] == 'E'){calculoDanio(attack + 2, true, board, e, x
        break;
        case 2:
            if(x - 1 >= 0 && y + 1 < column){if(board[x - 1][y + 1] == 'E'){calculoDanio(attack + 2, true, board, e
            if(y + 1 < column){if(board[x][y + 1] == 'E'){calculoDanio(attack + 2, true, board, e, x, y + 1, quanti
            if(x + 1 < row && y + 1 < column){if(board[x + 1][y + 1] == 'E'){calculoDanio(attack + 2, true, board,
        break;
        case 3:
            if(x + 1 < row && y - 1 >= 0){if(board[x + 1][y - 1] == 'E'){calculoDanio(attack + 2, true, board, e, x
            if(x + 1 < row){if(board[x + 1][y] == 'E'){calculoDanio(attack + 2, true, board, e, x - 1, y, quantity)
            if(x + 1 < row && y + 1 < column){if(board[x + 1][y + 1] == 'E'){calculoDanio(attack + 2, true, board,
        break;
        case 4:
            if(x - 1 >= 0 && y - 1 >= 0){if(board[x - 1][y - 1] == 'E'){calculoDanio(attack + 2, true, board, e, x
            if(y - 1 >= 0){if(board[x][y - 1] == 'E'){calculoDanio(attack + 2, true, board, e, x, y - 1, quantity);
            if(x + 1 < row && y - 1 >= 0){if(board[x + 1][y - 1] == 'E'){calculoDanio(attack + 2, true, board, e, x

        break;
    }
}
```

Estocada: Similar a las demás habilidades, esta acción comienza con un switch que tiene 4 casos, estos casos indican la dirección en la que el Guerrero utilizará

estocada. En esta habilidad tuvimos que tener en cuenta que pega dos casillas hacia la dirección a la que se aplique la misma. Por esto mismo cada uno de los

casos del switch están siguen la misma de estructura de: condicional que valida que en la primera casilla donde golpeará estocada no es fuera de los límites del tablero, si esto se cumple entonces pasa a otro condicional donde llama a cálculo de daño dependiendo de si coincide la casilla en donde va a pegar con un enemigo. Luego pasa a un segundo condicional que valida si existe un enemigo en esa coordenada donde va a realizar el ataque, al igual que en el primer caso, si esto se cumple entonces entra a un segundo condicional donde se llama a la acción cálculo de daño si es que la casilla donde pegará coincide con un enemigo.



```
void Estocada(int row, int column, char board[500][500], Enemy e[1000], int quantity){
    switch(direction){
        case 1:
        if(iPosition - 1 >= 0){
            if(board[iPosition - 1][jPosition] == 'E'){ calculoDanio(attack + 4, true, board, e, iPosition - 1, jPosition, quantity); }
        }
        if(iPosition - 2 >= 0){
            if(board[iPosition - 2][jPosition] == 'E'){ calculoDanio(attack + 4, true, board, e, iPosition - 2, jPosition, quantity); }
        }
        break;
        case 2:
        if(jPosition + 1 <= column - 1){
            if(board[iPosition][jPosition + 1] == 'E'){ calculoDanio(attack + 4, true, board, e, iPosition, jPosition + 1, quantity); }
        }
        if(jPosition + 2 <= column - 1){
            if(board[iPosition][jPosition + 2] == 'E'){ calculoDanio(attack + 4, true, board, e, iPosition, jPosition + 2, quantity); }
        }
        break;
        case 3:
        if(iPosition + 1 <= row - 1){
            if(board[iPosition + 1][jPosition] == 'E'){ calculoDanio(attack + 4, true, board, e, iPosition + 1, jPosition, quantity); }
        }
        if(iPosition + 2 <= row - 1){
            if(board[iPosition + 2][jPosition] == 'E'){ calculoDanio(attack + 4, true, board, e, iPosition + 2, jPosition, quantity); }
        }
        break;
        case 4:
        if(jPosition - 1 >= 0){
            if(board[iPosition][jPosition - 1] == 'E'){ calculoDanio(attack + 4, true, board, e, iPosition, jPosition - 1, quantity); }
        }
        if(jPosition - 2 >= 0){
            if(board[iPosition][jPosition - 2] == 'E'){ calculoDanio(attack + 4, true, board, e, iPosition, jPosition - 2, quantity); }
        }
        break;
    }
}
```

Torbellino: A diferencia de las dos habilidades anteriores, esta habilidad no funciona con base a un switch y la variable dirección. Su funcionamiento es más simple. Como torbellino hace cierto daño dependiendo de la multiplicación del ataque del Guerrero más la unidad por la cantidad de enemigos a la que les hace daño, lo primero que hay dentro del método torbellino es una variable de tipo entero llamada contEnemigos, que como su nombre lo indica sirve para llevar registro de a cuantos enemigos les va a afectar la habilidad torbellino. Y seguido del contador de enemigos, hay declaradas dos variables que almacenan la posición en la fila y la posición en la columna del guerrero que use torbellino. Estas variables llevan por nombre X y Y. Posteriormente hay cuatro condicionales que validan que las casillas que están alrededor del Guerrero (sin incluir las diagonales) no sean casillas que no existan en el tablero. Si esto es cierto, el programa entra en un segundo condicional que verifica si hay un

enemigo en dicha casilla, y si lo hay, suma una unidad al contador de enemigos. Después de esto, hay cuatro condicionales más. Estos últimos condicionales validan que el torbellino no se vaya a salir del tablero, y si no se sale, entonces verifica si en las casillas que va a pegar hay un enemigo. Si esto es así, entonces se llama a la acción para calcular el daño.

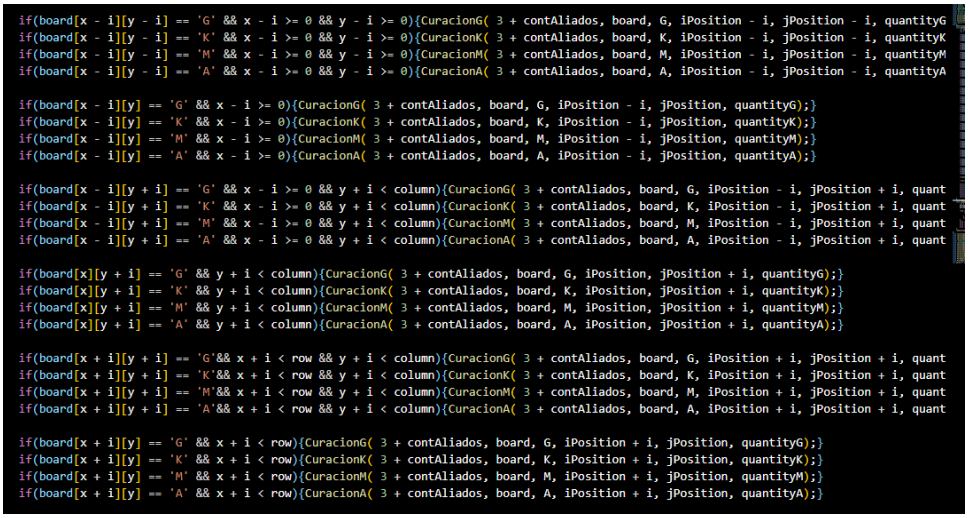
```
void Torbellino(int row, int column, char board[500][500], Enemy e[1000], int quantity){
    int contEnemigos = 0;
    int x = iPosition, y = jPosition;

    if(x - 1 >= 0){ if(board[x - 1][y] == 'E'){contEnemigos++;}}
    if(y + 1 < column){ if(board[x][y + 1] == 'E'){contEnemigos++;}}
    if(x + 1 < row){if(board[x + 1][y] == 'E'){contEnemigos++;}}
    if(y - 1 >= 0){ if(board[x][y - 1] == 'E'){contEnemigos++;}}

    if(x - 1 >= 0){if(board[x - 1][y] == 'E'){calculoDanio((attack + 1)* contEnemigos, true, board, e, x - 1,
    if(y + 1 < column){if(board[x][y + 1] == 'E'){calculoDanio((attack + 1)* contEnemigos, true, board, e, x,
    if(x + 1 < row){if(board[x + 1][y] == 'E'){calculoDanio((attack + 1)* contEnemigos, true, board, e, x + 1
    if(y - 1 >= 0){if(board[x][y - 1] == 'E'){calculoDanio((attack + 1)* contEnemigos, true, board, e, x, y -
}
```

Clérigo: En el momento en el que íbamos a hacer al personaje Clérigo junto con sus métodos nos percatamos de que, para que sus habilidades funcionaran bien, tenía que estar declarado después de los demás personajes y para poder recibir el daño debía estar declarado antes que la clase enemigo, sin embargo, esto nos arrojaría un error ya que, al declararlo de esa manera, el clérigo no podría realizarle ningún tipo de daño al enemigo ya que el enemigo aún no estaría declarado en el algoritmo. Para resolver esto, llegamos a la conclusión de que una buena manera de manejar la situación que se nos presentó sería declarar dos clérigos. El primero sería el Clérigo original, sin ningún método, solo con sus respectivos constructores y destructores vacíos, el cual será el encargado de almacenar la vida y los atributos del clerigo en la simulación, además de recibir los daños proporcionados por los enemigos y las curaciones que se pueden realizar los clérigos entre sí. El segundo Clérigo al cual denominamos HealerCopy estaría declarado después de todos los demás personajes de la simulación y sería el encargado de tener las habilidades del Clérigo. Entonces la función del Clérigo original es recibir daño y pasarle a su copia (HealerCopy) todos los atributos que tiene, incluida la habilidad que va a utilizar y el movimiento que hará. Mientras que la copia del Clérigo se encarga de ejecutar las habilidades que recibe del Clérigo original y de moverse por el tablero según sea el caso. En los siguientes párrafos se darán detalles de cómo funcionan las habilidades del Clérigo. Por último, es importante añadir que el Clérigo hace un daño mágico, así que las habilidades que tenga que dañen a los enemigos reciben el booleano de ataque físico en falso.

Nova de Luz: Debido a que esta habilidad cura aliados y daña a enemigos, el código de este método está divido en dos partes: cómo y cuándo llevar a cabo las curaciones y luego cómo y cuándo dañar a los enemigos. Empecemos por la

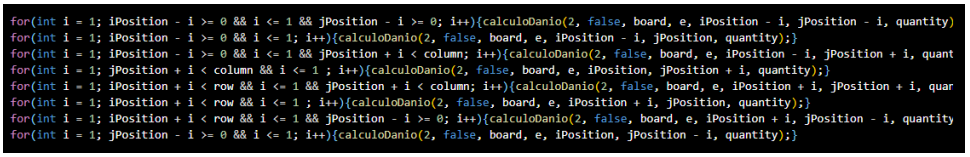
estructura que tiene el código para curar a los aliados. Lo primero que se observa al ver el método Nova de Luz, es que hay una variable de tipo entero inicializada en 1 llamada contAliados. La función de esta variable es llevar la cuenta de cuantos aliados hay alrededor del clérigo, pues esta habilidad cura una cantidad de puntos de vida específica en función de a cuántos aliados esté curando. La variable para contar aliados está inicializada en 1 porque el Clérigo se cura a sí mismo, entonces siempre se debe a empezar a contar a partir de 1. Después de la variable contAliados, se declararon dos variables de nombre X y Y que almacenan la posición del Clérigo en la fila y columna donde esté respectivamente. Seguidamente hay una serie de condicionales que verifican si alrededor del Clérigo (incluidas las diagonales) existen aliados y que alrededor de él no haya casillas que no existan en el tablero (este código se repite para cada uno de los héroes y para cada una de las direcciones en las que la habilidad tenga alcance). Si estas dos cosas se cumplen, entonces se le suma la unidad a la variable para contar aliados. Después de que se terminan los condicionales para verificar lo que se describió anteriormente, vienen otras líneas de código más con condicionales que verifican lo mismo que los condicionales anteriores, con la diferencia de que si encuentra a un aliado y la habilidad no se sale del tablero, entonces llama a la acción Curación para el personaje que haya encontrado (es el mismo código anterior solo que llama a la acción curación y también está hecho para cada posible casilla que la habilidad pueda afectar y para cada héroe que pueda haber en la simulación). Por último, hay una línea de código donde la acción Curación se aplica en el Clérigo que esté usando la habilidad.

Ahora pasemos a la parte del código de Nova de Luz donde se hacen los daños a los enemigos. Esta sección es más corta que la de curación y más sencilla. Está formado por una serie de ocho for´s (un for para cada dirección alrededor del Clérigo) que hacen daño a los enemigos que estén en las casillas donde esta habilidad tenga alcance.



```
if(board[x - i][y - i] == 'G' && x - i >= 0 && y - i >= 0){CuracionG( 3 + contAliados, board, G, iPosition - i, jPosition - i, quantityG
if(board[x - i][y - i] == 'K' && x - i >= 0 && y - i >= 0){CuracionK( 3 + contAliados, board, K, iPosition - i, jPosition - i, quantityK
if(board[x - i][y - i] == 'M' && x - i >= 0 && y - i >= 0){CuracionM( 3 + contAliados, board, M, iPosition - i, jPosition - i, quantityM
if(board[x - i][y - i] == 'A' && x - i >= 0 && y - i >= 0){CuracionA( 3 + contAliados, board, A, iPosition - i, jPosition - i, quantityA

if(board[x - i][y] == 'G' && x - i >= 0){CuracionG( 3 + contAliados, board, G, iPosition - i, jPosition, quantityG);}
if(board[x - i][y] == 'K' && x - i >= 0){CuracionK( 3 + contAliados, board, K, iPosition - i, jPosition, quantityK);}
if(board[x - i][y] == 'M' && x - i >= 0){CuracionM( 3 + contAliados, board, M, iPosition - i, jPosition, quantityM);}
if(board[x - i][y] == 'A' && x - i >= 0){CuracionA( 3 + contAliados, board, A, iPosition - i, jPosition, quantityA);}

if(board[x - i][y + i] == 'G' && x - i >= 0 && y + i < column){CuracionG( 3 + contAliados, board, G, iPosition - i, jPosition + i, quant
if(board[x - i][y + i] == 'K' && x - i >= 0 && y + i < column){CuracionK( 3 + contAliados, board, K, iPosition - i, jPosition + i, quant
if(board[x - i][y + i] == 'M' && x - i >= 0 && y + i < column){CuracionM( 3 + contAliados, board, M, iPosition - i, jPosition + i, quant
if(board[x - i][y + i] == 'A' && x - i >= 0 && y + i < column){CuracionA( 3 + contAliados, board, A, iPosition - i, jPosition + i, quant

if(board[x][y + i] == 'G' && y + i < column){CuracionG( 3 + contAliados, board, G, iPosition, jPosition + i, quantityG);}
if(board[x][y + i] == 'K' && y + i < column){CuracionK( 3 + contAliados, board, K, iPosition, jPosition + i, quantityK);}
if(board[x][y + i] == 'M' && y + i < column){CuracionM( 3 + contAliados, board, M, iPosition, jPosition + i, quantityM);}
if(board[x][y + i] == 'A' && y + i < column){CuracionA( 3 + contAliados, board, A, iPosition, jPosition + i, quantityA);}

if(board[x + i][y + i] == 'G'&& x + i < row && y + i < column){CuracionG( 3 + contAliados, board, G, iPosition + i, jPosition + i, quant
if(board[x + i][y + i] == 'K'&& x + i < row && y + i < column){CuracionK( 3 + contAliados, board, K, iPosition + i, jPosition + i, quant
if(board[x + i][y + i] == 'M'&& x + i < row && y + i < column){CuracionM( 3 + contAliados, board, M, iPosition + i, jPosition + i, quant
if(board[x + i][y + i] == 'A'&& x + i < row && y + i < column){CuracionA( 3 + contAliados, board, A, iPosition + i, jPosition + i, quant

if(board[x + i][y] == 'G' && x + i < row){CuracionG( 3 + contAliados, board, G, iPosition + i, jPosition, quantityG);}
if(board[x + i][y] == 'K' && x + i < row){CuracionK( 3 + contAliados, board, K, iPosition + i, jPosition, quantityK);}
if(board[x + i][y] == 'M' && x + i < row){CuracionM( 3 + contAliados, board, M, iPosition + i, jPosition, quantityM);}
if(board[x + i][y] == 'A' && x + i < row){CuracionA( 3 + contAliados, board, A, iPosition + i, jPosition, quantityA);}
```

Fragmento de código donde se aprecia una parte de las validaciones para sanar a los aliados



```
for(int i = 1; iPosition - i >= 0 && i <= 1 && jPosition - i >= 0; i++){calculoDanio(2, false, board, e, iPosition - i, jPosition - i, quantity)
for(int i = 1; iPosition - i >= 0 && i <= 1; i++){calculoDanio(2, false, board, e, iPosition - i, jPosition, quantity);}
for(int i = 1; iPosition - i >= 0 && i <= 1 && jPosition + i < column; i++){calculoDanio(2, false, board, e, iPosition - i, jPosition + i, quant
for(int i = 1; jPosition + i < column && i <= 1 ; i++){calculoDanio(2, false, board, e, iPosition, jPosition + i, quantity);}
for(int i = 1; iPosition + i < row && i <= 1 && jPosition + i < column; i++){calculoDanio(2, false, board, e, iPosition + i, jPosition + i, quar
for(int i = 1; iPosition + i < row && i <= 1 ; i++){calculoDanio(2, false, board, e, iPosition + i, jPosition, quantity);}
for(int i = 1; iPosition + i < row && i <= 1 && jPosition - i >= 0; i++){calculoDanio(2, false, board, e, iPosition + i, jPosition - i, quantity
for(int i = 1; jPosition - i >= 0 && i <= 1; i++){calculoDanio(2, false, board, e, iPosition, jPosition - i, quantity);}
```

Código donde se hacen las validaciones y se llama a la acción para calcular el daño a los enemigos

Impacto Sagrado: Al ser una habilidad apuntada, lo primero que hay dentro de este método son dos variables de tipo entero inicializadas en cero. El propósito de estas variables es almacenar las coordenadas que se lean del documento de entrada para ejecutar el daño o la curación sobre la casilla que se aplique la habilidad. Luego, hay un condicional que verifica si las coordenadas que se recibieron en la entrada coinciden con las coordenadas del Clérigo que usó la habilidad, y si esto es así se valida si la vida que se va a curar al Clérigo es menor o mayor que la vida base del mismo. Si es mayor que la vida base, el Clérigo recupera todos sus puntos de vida, si no, entonces se le suman los puntos de vida correspondientes a la vida del Clérigo. Después vienen una serie de 5 condicionales que verifican si las coordenadas introducidas coinciden con las de algún personaje de la simulación. Si las coordenadas son las mismas que las de algún héroe, llama a su acción curación. En cambio, si las coordenadas introducidas coinciden con la de un enemigo, le hace el daño correspondiente.

```
void ImpactoSagrado(char board[500][500], Warrior g[100], Healer k[100], Wizard m[100], Archer a[100], Enemy e[100], ifstream &entrada, int
    int x = 0, y = 0;
    entrada >> y;
    entrada >> x;

    if(x == iPosition && y == jPosition){
        if(hp + 4 + attack > hpMax){
            hp = hpMax;
        }
        else {hp += 4 + attack;}
    }
    else if(board[x][y] == 'G'){ CuracionG( 4 + attack, board, g, x, y, quantityG);}
    else if(board[x][y] == 'K'){ CuracionK( 4 + attack, board, k, x, y, quantityK);}
    else if(board[x][y] == 'M'){ CuracionM( 4 + attack, board, m, x, y, quantityM);}
    else if (board[x][y] == 'A'){ CuracionA(4 + attack, board, a, x, y, quantityA);}
    else if(board[x][y] == 'E'){ calculoDanio( 2 + attack, false, board, e, x, y, quantityE);}

}
```

Luz Sagrada: Esta habilidad funciona exactamente igual que Impacto Sagrado, pero con la diferencia de que no le hace daño a ningún enemigo y que cura más puntos de salud a sus aliados y a él mismo. Tiene la misma estructura: primero verifica si las coordenadas introducidas coinciden con las del Clérigo que usa la habilidad y lo cura. Si no, verifica que las coordenadas coincidan con algún aliado y lo cura.

```
void LuzSagrada(char board[500][500], Warrior g[100], Healer k[100], Wizard m[100], Archer a[100], ifstream &entr
    int x = 0, y = 0;
    entrada >> x;
    entrada >> y;

    if(x == iPosition && y == jPosition){
        if(hp + 8 + attack > hpMax){
            hp = hpMax;
        }
        else {hp += 8 + attack;}
    }
```

Mago: Las habilidades del mago fueron sencillas de plantar y luego hacer en el código. Además, el mago hace ataque mágico, por lo que sus ataques reciben al booleano de ataque físico en falso. Los métodos son los siguientes:

Teletransportación*:* Hacer esta habilidad en código no fue muy complicado. Reutilizamos el método de la clase Personaje que tiene todas las instrucciones para realizar los movimientos, pero le añadimos algunos cambios. En primer lugar, dependiendo de a donde se vaya a teletransportar el mago, el programa verifica que hay en la segunda casilla a la que se va a mover. Si esta casilla está vacía, entonces el mago se teletransporta a ella. En cambio, si en la segunda casilla hay un personaje o es el límite del mapa, el programa verifica la primera casilla delante del mago y valida si se puede mover a ella o no. Si se puede, se teletransporta a ella. Es importante recalcar que todo el proceso anterior se hace luego de validar que el mago no se vaya a salir del tablero usando su teletransportación. Este planteamiento de verificar casillas y luego moverse se hizo para cada dirección a la que el mago pueda moverse.

```
switch(direction){

    case 1:

    if(iPosition - 2 >= 0){

        if(board[iPosition - 2][jPosition] == '_'){

            board[iPosition][jPosition] = '_';
            board[iPosition - 2][jPosition] = 'M';
            iPosition -= 2;
        }
        else if(board[iPosition - 1][jPosition] == '_'){

            board[iPosition][jPosition] = '_';
            board[iPosition - 1][jPosition] = 'M';
            iPosition -= 1;
        }
    }
    else if(iPosition - 1 >= 0 && board[iPosition - 1][jPosition] == '_'){

        board[iPosition][jPosition] = '_';
        board[iPosition - 1][jPosition] = 'M';
        iPosition -= 1;
    }
    break;
```

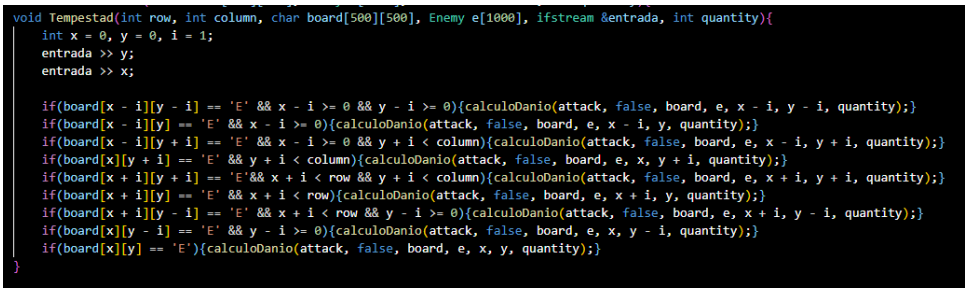
Caso del switch en el que el Mago se teletransporta hacia arriba

Esfera de Hielo: Al ser una habilidad apuntada, primero se declararon dos variables para almacenar las coordenadas que se leen desde el archivo de salida que es donde caerá el ataque. Después hay un condicional que valida si las coordenadas que se introdujeron coinciden las coordenadas de algún enemigo en el tablero. Si esto es cierto, entonces se procede a llamar a la acción para calcular el daño que se le tiene que hacer al enemigo.

```
void EsferaDeHielo(char board[500][500], Enemy e[1000], ifstream &archivo, int quantity){
    int x = 0, y = 0;
    archivo >> y;
    archivo >> x;

    if(board[x][y] == 'E'){
        calculoDanio(attack * 3, false, board, e, x, y, quantity);
    }
}
```

Tempestad: Esta habilidad también es apuntada, pero no hace daño en una sola casilla, sino que hace daño en la casilla que corresponda a la coordenada que se leyó del archivo de entrada y también hace daño alrededor de esa casilla, incluyendo las diagonales. Planteamos esta habilidad similar a como se planteó Nova de Luz, solo que es apuntada y hace daño en el centro de las coordenadas

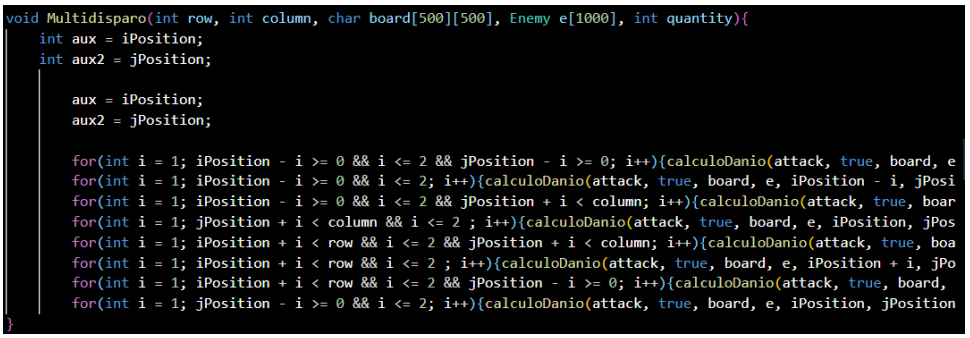
suministradas. Entonces, lo primero que hay dentro de este método son dos variables de tipo entero que almacenan las coordenadas que se leen del archivo de entrada. Luego vienen nueve condicionales que validan si hay algún enemigo en las casillas donde hace daño esta habilidad y si la habilidad no se sale del tablero. Si eso es cierto, se llama a la acción para calcular daño.



```
void Tempestad(int row, int column, char board[500][500], Enemy e[1000], ifstream &entrada, int quantity){
    int x = 0, y = 0, i = 1;
    entrada >> y;
    entrada >> x;

    if(board[x - i][y - i] == 'E' && x - i >= 0 && y - i >= 0){calculoDanio(attack, false, board, e, x - i, y - i, quantity);}
    if(board[x - i][y] == 'E' && x - i >= 0){calculoDanio(attack, false, board, e, x - i, y, quantity);}
    if(board[x - i][y + i] == 'E' && x - i >= 0 && y + i < column){calculoDanio(attack, false, board, e, x - i, y + i, quantity);}
    if(board[x][y + i] == 'E' && y + i < column){calculoDanio(attack, false, board, e, x, y + i, quantity);}
    if(board[x + i][y + i] == 'E'&& x + i < row && y + i < column){calculoDanio(attack, false, board, e, x + i, y + i, quantity);}
    if(board[x + i][y] == 'E' && x + i < row){calculoDanio(attack, false, board, e, x + i, y, quantity);}
    if(board[x + i][y - i] == 'E' && x + i < row && y - i >= 0){calculoDanio(attack, false, board, e, x + i, y - i, quantity);}
    if(board[x][y - i] == 'E' && y - i >= 0){calculoDanio(attack, false, board, e, x, y - i, quantity);}
    if(board[x][y] == 'E'){calculoDanio(attack, false, board, e, x, y, quantity);}
}
```

Arquero: Para hacer las habilidades del Arquero fue de mucha utilidad fijarse en las habilidades del Guerrero. A continuación, sus habilidades:

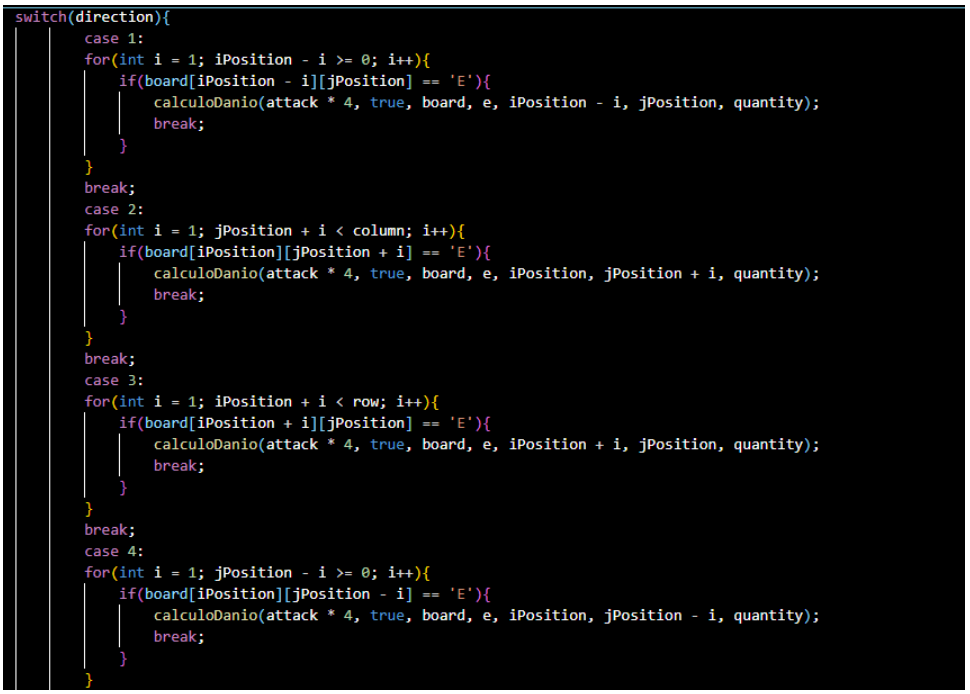
Multidisparo: Multidisparo puede pensarse como una Estocada del Guerrero pero que afecta a todas las direcciones, incluyendo las diagonales. Para empezar, lo primero que hay en el método Multidisparo son dos variables de tipo entero que sirven para almacenar las coordenadas del Arquero que utilice esta habilidad (fila y columna). Luego, hay nueve for´s. Cada uno de ellos es una dirección a la que el Multidisparo puede llegar. Por supuesto, se valida que la habilidad no se salga del tablero y llama a la acción calcular daño para los enemigos a los que se encuentre.



```
void Multidisparo(int row, int column, char board[500][500], Enemy e[1000], int quantity){
    int aux = iPosition;
    int aux2 = jPosition;

        aux = iPosition;
        aux2 = jPosition;

        for(int i = 1; iPosition - i >= 0 && i <= 2 && jPosition - i >= 0; i++){calculoDanio(attack, true, board, e
        for(int i = 1; iPosition - i >= 0 && i <= 2; i++){calculoDanio(attack, true, board, e, iPosition - i, jPosi
        for(int i = 1; iPosition - i >= 0 && i <= 2 && jPosition + i < column; i++){calculoDanio(attack, true, boar
        for(int i = 1; jPosition + i < column && i <= 2 ; i++){calculoDanio(attack, true, board, e, iPosition, jPos
        for(int i = 1; iPosition + i < row && i <= 2 && jPosition + i < column; i++){calculoDanio(attack, true, boa
        for(int i = 1; iPosition + i < row && i <= 2 ; i++){calculoDanio(attack, true, board, e, iPosition + i, jPo
        for(int i = 1; iPosition + i < row && i <= 2 && jPosition - i >= 0; i++){calculoDanio(attack, true, board,
        for(int i = 1; jPosition - i >= 0 && i <= 2; i++){calculoDanio(attack, true, board, e, iPosition, jPosition
}
```

Disparo Longevo: Esta habilidad se hizo con base en un switch para poder cubrir las cuatro direcciones a las que el arquero podría disparar. La lógica fue la siguiente: en cada caso hacer un for que iterara casilla por casilla hasta encontrar un enemigo o hasta una casilla antes de salirse del tablero. Si en su trayectoria se encuentra un enemigo, entonces se llama a la acción para calcular el daño correspondiente a esta habilidad para el enemigo y posteriormente realiza un break para detener el movimiento ya que el mismo no traspasa al enemigo que se encuentre. Se repitió este proceso para las cuatro direcciones,

teniendo en cuenta los ligeros cambios correspondientes al validar que no se saliera la habilidad del mapa dependiendo de la dirección en la que el Arquero use esta habilidad.



```
switch(direction){
        case 1:
        for(int i = 1; iPosition - i >= 0; i++){
            if(board[iPosition - i][jPosition] == 'E'){
                calculoDanio(attack * 4, true, board, e, iPosition - i, jPosition, quantity);
                break;
            }
        }
        break;
        case 2:
        for(int i = 1; jPosition + i < column; i++){
            if(board[iPosition][jPosition + i] == 'E'){
                calculoDanio(attack * 4, true, board, e, iPosition, jPosition + i, quantity);
                break;
            }
        }
        break;
        case 3:
        for(int i = 1; iPosition + i < row; i++){
            if(board[iPosition + i][jPosition] == 'E'){
                calculoDanio(attack * 4, true, board, e, iPosition + i, jPosition, quantity);
                break;
            }
        }
        break;
        case 4:
        for(int i = 1; jPosition - i >= 0; i++){
            if(board[iPosition][jPosition - i] == 'E'){
                calculoDanio(attack * 4, true, board, e, iPosition, jPosition - i, quantity);
                break;
            }
        }
```

Voltereta: De la misma forma en la que en la habilidad anterior se usó un switch para cada uno de los casos en el que el Arquero se puede mover, en esta habilidad se aplicó también un switch. Esta habilidad la pensamos como el Quebrajar del Guerrero con el extra de que el Arquero se mueva hacia atrás. Primero utilizamos dos variables de tipo entero para almacenar allí las coordenadas del Arquero que use esta habilidad. Después viene el switch del que se habló con los cuatro casos para cada una de las direcciones. Y dentro de cada caso hay primero tres condicionales que validan que la habilidad no se salga del tablero y que si se encuentra a un enemigo entonces llama a la acción para calcular el daño. Después de esos tres condicionales hay un último condicional con el único objetivo de validar si detrás del Arquero hay un espacio donde pueda moverse, y que, si lo hay que se mueva. Esto se repite para los cuatro casos con las adaptaciones correspondientes cada dirección.

```
switch(direction){
    case 1:
        if(x - 1 >= 0 && y - 1 >= 0){if(board[x - 1][y - 1] == 'E'){calculoDanio(attack, true, board, e, x - 1, y - 1, quantity);}}
        if(x - 1 >= 0){if(board[x - 1][y] == 'E'){calculoDanio(attack, true, board, e, x - 1, y, quantity);}}
        if(x - 1 >= 0 && y + 1 < column){if(board[x - 1][y + 1] == 'E'){calculoDanio(attack, true, board, e, x - 1, y + 1, quantity);}

        if(iPosition + 1 < row){
            if(board[iPosition + 1][jPosition] == '_'){
                board[iPosition][jPosition] = '_';
                iPosition += 1;
                board[iPosition][jPosition] = 'A';
            }
        }
    break;
    case 2:
        if(x - 1 >= 0 && y + 1 < column){if(board[x - 1][y + 1] == 'E'){calculoDanio(attack, true, board, e, x - 1, y + 1, quantity);}
        if(y + 1 < column){if(board[x][y + 1] == 'E'){calculoDanio(attack, true, board, e, x, y + 1, quantity);}}
        if(x + 1 < row && y + 1 < column){if(board[x + 1][y + 1] == 'E'){calculoDanio(attack, true, board, e, x + 1, y + 1, quantity);


        if(jPosition - 1 >= 0){
            if(board[iPosition][jPosition - 1] == '_'){
                board[iPosition][jPosition - 1] = 'A';
                board[iPosition][jPosition] = '_';
                jPosition -= 1;
            }
```

Primeros dos casos de la habilidad Voltereta

Enemigo: Para el personaje Enemigo, se hizo exactamente lo mismo que con el personaje Clérigo: además del Enemigo original, se hizo una copia. Ya que para el correcto funcionamiento del Enemigo es necesario que él que esté declarado después de todos los demás personajes de la simulación para que pueda realizarle el respectivo a daño a cada uno de los héroes de la simulación, y a su vez es necesario que esté declarado antes que todos los héroes para que estos puedan dañarlo. El Enemigo original y su copia funcionan de la misma forma que el Clérigo original y su copia, siendo el Enemigo original quien se encarga de recibir todos los daños y la copia el encargado de realizar todos los movimientos y ataques. Como el enemigo no tiene ninguna habilidad nueva que no se haya hecho antes para un héroe, se reutilizaron los códigos de Disparo Longevo del Arquero, Teletransportación del Mago y Torbellino del Guerrero. Por supuesto, se hicieron los ajustes necesarios para que no hubiese ningún error en la simulación. En Disparo Longevo en cada caso del switch se añadieron las líneas de código necesarias para cada calcular el daño en cada uno de los héroes. Lo mismo se hizo en Torbellino. En la Teletransportación del enemigo solo se cambió el carácter “M” por el carácter “E” en cada caso que fuera necesario, pues este código es, en esencia, idéntico a la teletransportación del Mago.

```
void DisparoLongevoE(int row, int column, char board[500][500], Warrior G[100], Healer K[100], Wizard M[100], Archer A[100], int
    switch(direction){
        case 1:
        for(int i = 0; iPosition - i >= 0; i++){
            if(board[iPosition - i][jPosition] == 'G'){
                calculoDanioG(attack * 4, board, G, iPosition - i, jPosition, quantityG);
                break;
            }
            if(board[iPosition - i][jPosition] == 'K'){
                calculoDanioK(attack * 4, board, K, iPosition - i, jPosition, quantityK);
                break;
            }
            if(board[iPosition - i][jPosition] == 'M'){
                calculoDanioM(attack * 4, board, M, iPosition - i, jPosition, quantityM);
                break;
            }
            if(board[iPosition - i][jPosition] == 'A'){
                calculoDanioA(attack * 4, board, A, iPosition - i, jPosition, quantityA);
                break;
            }
        }
        break;
```

Fragmento del Disparo Longevo donde se aprecia el caso 1 del switch

```
switch(direction){

    case 1:

    if(iPosition - 2 >= 0){

        if(board[iPosition - 2][jPosition] == '_'){

            board[iPosition][jPosition] = '_';
            board[iPosition - 2][jPosition] = 'E';
            iPosition -= 2;
        }
        else if(board[iPosition - 1][jPosition]){

            board[iPosition][jPosition] = '_';
            board[iPosition - 1][jPosition] = 'E';
            iPosition -= 1;
        }
    }
    else if(iPosition - 1 >= 0){

        board[iPosition][jPosition] = '_';
        board[iPosition - 1][jPosition] = 'E';
        iPosition -= 1;
    }
    break;
```

Fragmento de la Teletransportación del Enemigo donde se aprecia el caso 1 del switch

```
int x = iPosition, y = jPosition;

if(x - 1 >= 0){if(board[x - 1][y] == 'G'){contEnemigos++;}}
if(y + 1 < column){if(board[x][y + 1] == 'G'){contEnemigos++;}}
if(x + 1 < row){if(board[x + 1][y] == 'G'){contEnemigos++;}}
if(y - 1 >= 0){if(board[x][y - 1] == 'G'){contEnemigos++;}}

if(x - 1 >= 0){if(board[x - 1][y] == 'K'){contEnemigos++;}}
if(y + 1 < column){if(board[x][y + 1] == 'K'){contEnemigos++;}}
if(x + 1 < row){if(board[x + 1][y] == 'K'){contEnemigos++;}}
if(y - 1 >= 0){if(board[x][y - 1] == 'K'){contEnemigos++;}}

if(x - 1 >= 0){if(board[x - 1][y] == 'M'){contEnemigos++;}}
if(y + 1 < column){if(board[x][y + 1] == 'M'){contEnemigos++;}}
if(x + 1 < row){if(board[x + 1][y] == 'M'){contEnemigos++;}}
if(y - 1 >= 0){if(board[x][y - 1] == 'M'){contEnemigos++;}}

if(x - 1 >= 0){if(board[x - 1][y] == 'A'){contEnemigos++;}}
if(y + 1 < column){if(board[x][y + 1] == 'A'){contEnemigos++;}}
if(x + 1 < row){if(board[x + 1][y] == 'A'){contEnemigos++;}}
if(y - 1 >= 0){if(board[x][y - 1] == 'A'){contEnemigos++;}}

if(x - 1 >= 0){if(board[x - 1][y] == 'G'){calculoDanioG((attack + 1) * contEnemigos, board, G, iPosition - 1, jPosition, quantityG);
if(y + 1 < column){if(board[x][y + 1] == 'G'){calculoDanioG((attack + 1) * contEnemigos, board, G, iPosition, jPosition + 1, quantit
if(x + 1 < row){if(board[x + 1][y] == 'G'){calculoDanioG((attack + 1) * contEnemigos, board, G, iPosition + 1, jPosition, quantityG)
if(y - 1 >= 0){if(board[x][y - 1] == 'G'){calculoDanioG((attack + 1) * contEnemigos, board, G, iPosition, jPosition - 1, quantityG);

if(x - 1 >= 0){if(board[x - 1][y] == 'K'){calculoDanioK((attack + 1) * contEnemigos, board, K, iPosition - 1, jPosition, quantityK);
if(y + 1 < column){if(board[x][y + 1] == 'K'){calculoDanioK((attack + 1) * contEnemigos, board, K, iPosition, jPosition + 1, quantit
if(x + 1 < row){if(board[x + 1][y] == 'K'){calculoDanioK((attack + 1) * contEnemigos, board, K, iPosition + 1, jPosition, quantityK)
if(y - 1 >= 0){if(board[x][y - 1] == 'K'){calculoDanioK((attack + 1) * contEnemigos, board, K, iPosition, jPosition - 1, quantityK);
```

Fragmento de código del Torbellino del Enemigo donde se aprecia toda la sección donde se usan cada uno de los contadores para los héroes de la simulación y luego como se aplica el daño de la habilidad al Guerrero y al Clérigo después de validar que la habilidad no se saliera del tablero

Función Main:

Al declarar la función Main le introdujimos dos parámetros: uno de tipo entero y el otro un arreglo de caracteres. Esto se hizo con la intención de poder leer los archivos de entrada y escribir en un archivo de salida que son recibidos desde el main. De esta forma se pueden leer y escribir los archivos que queramos desde la consola al momento de ejecutar el programa. Lo primero que hay dentro de esta función son dos variables declaradas: una ifstream y otra ofstream. Sirven para poder manejar archivos de lectura (entradas) y escritura (salidas) respectivamente. En la línea siguiente a esas variables, hay una de tipo entero que tiene por nombre "iteraciones" y está inicializada en 0. Esta variable la usaremos luego en un for que dictará cuántas iteraciones tendrá el programa. Seguidamente escribimos las líneas de código que nos permitirán acceder al documento de entrada y al de salida desde la consola.

```
int main(int argc, char *argv[]){

    ifstream entrada;
    ofstream salida;
    int iteraciones = 0;

    entrada.open(argv[1]);
    salida.open(argv[2]);
```

En las siguientes líneas se declararon otras tres variables de tipo entero llamadas row, column y aux, las tres inicializadas en 0. A continuación leemos la cantidad de filas y columnas del documento de entrada, el funcionamiento de la variable aux será explicada posteriormente junto con el de la acción changeCoordinates:

```
int row = 0, column = 0, aux = 0;

entrada >> row;
entrada >> column;
```

Después, se creó la matriz que es el tablero de juego de la simulación y también se llamó a la acción FillBoard para llenar dicha matriz.

```
char board[500][500];
FillBoard(row, column, board, entrada);
```

En esta parte del código se declararon y se inicializan cinco variables (uno por cada personaje) que contienen la cantidad de caracteres de cada personaje en la simulación. Y después de eso, se declaró un arreglo para cada personaje que puede existir en la simulación, además de dos arreglos más que son para la copia del Clérigo y la copia del Enemigo. El tamaño de cada uno de estos arreglos es la cantidad de personajes que existen en la simulación.

```
int quantityG = CharacterQuantity(row, column, board, 'G');
int quantityK = CharacterQuantity(row, column, board, 'K');
int quantityM = CharacterQuantity(row, column, board, 'M');
int quantityA = CharacterQuantity(row, column, board, 'A');
int quantityE = CharacterQuantity(row, column, board, 'E');

Warrior G[100];
Healer K[100];
Wizard M[100];
Archer A[100];
Enemy E[1000];
HealerCopy H[quantityK];
EnemyCopy Enemigo[quantityE];
```

Las siguientes líneas del código en su mayoría son bucles for para pedir los atributos de los personajes que existirán en la simulación (un bucle para cada personajes). Posteriormente se realiza un bucle el cual se encarga de leer el tablero en busca de los respectivos personajes, de esta manera, la coordenada donde se halló el carácter del personaje a encontrar es pasada a la función changeCoordinates para proceder a asignarle esa coordenada al personaje ubicado en la posición “aux” del arreglo. Posteriormente se incrementa el valor de aux para así continuar leyendo la matriz desde el punto donde se encontró el carácter y repetir el proceso pero esta vez para un valor mayor de aux. Este proceso se repite hasta culminar de definir la ubicación de cada uno de los personajes incluidos en el arreglo.

```
for(int i = 0; i < quantityM; i++){ M[i].AskAttributes(entrada); }

for(int i = 0; i < row; i++){
    for(int j = 0; j < column; j++){
        if(board[i][j] == 'M'){
            if(aux - 1 < 0){
                M[aux].changeCoordinates(M[aux].iPosition, M[aux].jPosition, i, j);
                aux++;
            }
            else{
                M[aux].changeCoordinates(M[aux].iPosition, M[aux].jPosition, i, j);
                aux++;
            }
        }
    }
}

aux = 0;
```

Inmediatamente después de introducir los atributos que tendrá cada personaje de la simulación, se procede a leer del documento de entrada la cantidad de iteraciones que habrá en la misma.

```
entrada >> iteraciones;
```

Después de leer la cantidad de iteraciones, se procede a leer el movimiento, dirección y habilidad de cada uno de los personajes introducidos. Esto se hace con la ayuda de las acciones AskDirectionAndMovement y AskMovement. Esta última se realiza posteriormente a validar que el personaje aún esté vivo, para así en caso contrario, evitar que estas acciones se ejecuten y afecten la simulación. Sin embargo, la acción AskDirectionAndMovement se instancia previo a validar la vida del personaje ya que esa entrada debe ser leída sin importar la vida del personaje.

```
for(int i = 0; i < quantityA; i++){
    A[i].AskDirectionAndMovement(entrada);
    if(A[i].hp > 0){
        A[i].AskMovement(row,column, board, 'A');
        switch(A[i].ability){
            case 1:
                A[i].Multidisparo(row, column, board, E, quantityE);
            break;
            case 2:
                A[i].DisparoLongevo(row, column, board, E, quantityE);
            break;
            case 3:
                A[i].Voltereta(row, column, board, E, quantityE);
            break;
        }
    }
}
```

Bucle donde se pide la dirección, movimiento y habilidad de los Arqueros de la simulación

```
for(int i = 0; i < quantityK; i++){
    H[i].hp = K[i].hp;
    H[i].attack = K[i].attack;
    H[i].armor = K[i].armor;
    H[i].Mresistence = K[i].Mresistence;
    H[i].iPosition = K[i].iPosition;
    H[i].jPosition = K[i].jPosition;
    H[i].AskDirectionAndMovement(entrada);

    if(H[i].hp > 0){
        H[i].AskMovement(row,column, board, 'K');
        switch(H[i].ability){
            case 1:
                H[i].NovaDeLuz(row, column, board, G, K, M, A, E, quantityE, quantityG, quantityK, quantityM, quantityA);
            break;
            case 2:
                H[i].ImpactoSagrado(board, G, K, M, A, E, entrada, quantityG, quantityK, quantityM, quantityA, quantityE);
            break;
            case 3:
                H[i].LuzSagrada(board, G, K, M, A, entrada, quantityG, quantityK, quantityM, quantityA);
            break;
        }
    }
    K[i].iPosition = H[i].iPosition;
    K[i].jPosition = H[i].jPosition;
}
```

Este segmento del código representa el bucle for que se encarga de llamar a las acción AskDirectionAndMovement de cada personaje así como su acción AskMovement. Sin embargo, al ser este el perteneciente al Clérigo, dado que fue utilizada una copia, en la ejecución de este bucle For es donde se procede a asignarle a cada una de las copias los atributos del clérigo original en cada una de las iteraciones. El código realizado para la copia del enemigo tiene un funcionamiento idéntico.

```
for(int i = 0; i < quantityE; i++){
    Enemigo[i].hp = E[i].hp;
    Enemigo[i].attack = E[i].attack;
    Enemigo[i].armor = E[i].armor;
    Enemigo[i].Mresistence = E[i].Mresistence;
    Enemigo[i].iPosition = E[i].iPosition;
    Enemigo[i].jPosition = E[i].jPosition;
    Enemigo[i].AskDirectionAndMovement(entrada);
    if(Enemigo[i].hp > 0){
        Enemigo[i].AskMovement(row,column, board, 'E');
        switch(Enemigo[i].ability){
            case 1:
                Enemigo[i].DisparoLongevoE(row, column, board, G, K, M, A, quantityG, quantityK, quantityM, quantityA
            break;
            case 2:
                Enemigo[i].TpE(row, column, board);
                E[i].iPosition = Enemigo[i].iPosition;
                E[i].jPosition = Enemigo[i].jPosition;
            break;
            case 3:
                Enemigo[i].TorbellinoE(row, column, board, G, K, M, A, quantityG, quantityK, quantityM, quantityA);
            break;
        }
    }
    E[i].iPosition = Enemigo[i].iPosition;
    E[i].jPosition = Enemigo[i].jPosition;
}
```

Código del bucle for para la copia del Enemigo

Por último, se imprime cómo quedó el tablero al final de la simulación junto con los atributos de todos los personajes que estaban desde el principio de la misma, y luego se procede a cerrar el archivo de entrada y el archivo de salida.

```
PrintBoard(row, column, board, salida);
for(int i = 0; i < quantityG; i++){salida << G[i].hp << " " << G[i].attack << " " << G[i].armor << " " << G[i].Mresistence << endl; }
for(int i = 0; i < quantityK; i++){salida << K[i].hp << " " << K[i].attack << " " << K[i].armor << " " << K[i].Mresistence << endl; }
for(int i = 0; i < quantityM; i++){salida << M[i].hp << " " << M[i].attack << " " << M[i].armor << " " << M[i].Mresistence << endl; }
for(int i = 0; i < quantityA; i++){salida << A[i].hp << " " << A[i].attack << " " << A[i].armor << " " << A[i].Mresistence << endl; }
for(int i = 0; i < quantityE; i++){salida << E[i].hp << " " << E[i].attack << " " << E[i].armor << " " << E[i].Mresistence << endl; }

entrada.close();
salida.close();

return 0;
```