**UNIVERSIDAD CENTROAMERICANA**

**JOSÉ SIMEÓN CAÑAS**

**DEPARTAMENTO DE INGENIERÍA E INFORMÁTICA**

**Fundamentos de Programación**

**Proyecto de C++**

Integrantes:

Engelhard Robert, Diego, 00356119

Martínez Ventura, Enrique Gabriel, 00389819

Catedrática: Ingeniera Elisa Aldana

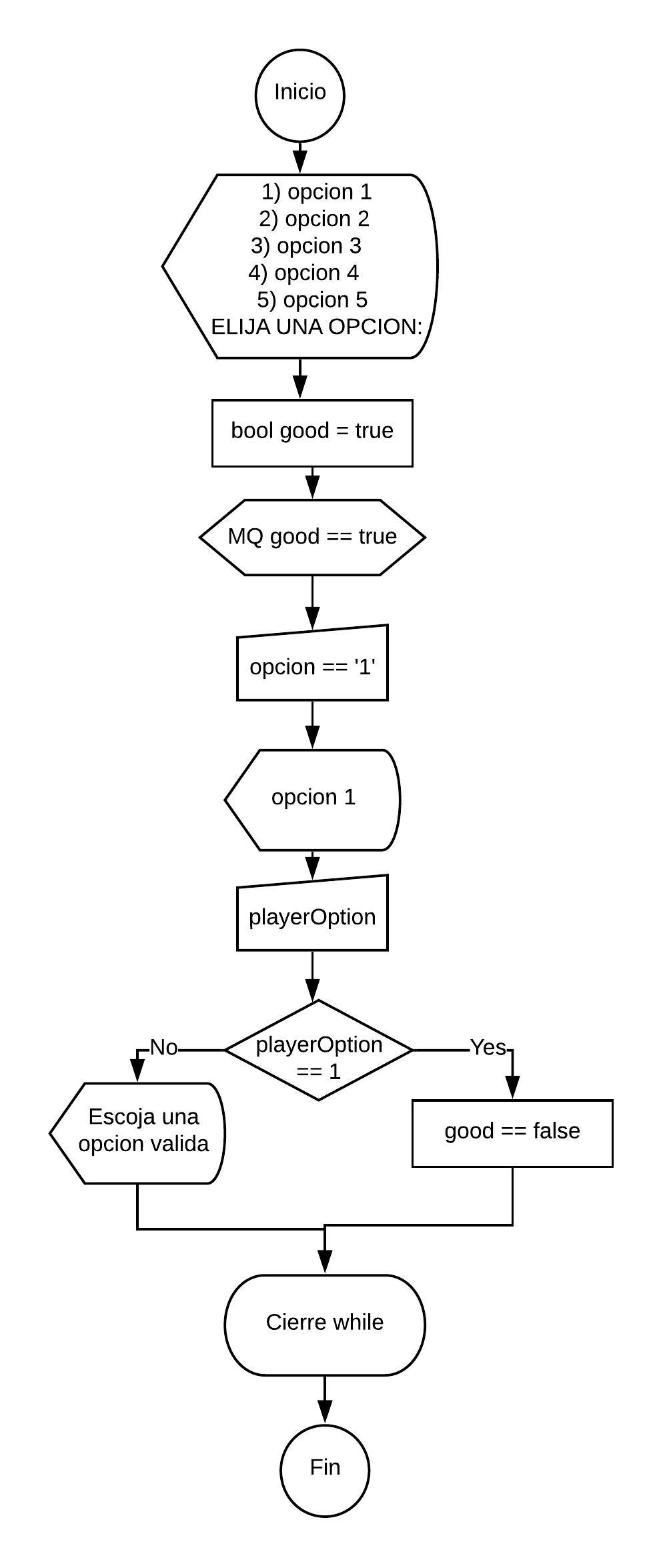
Fecha de entrega: miércoles 20 de noviembre de 2019

**DESCRIPCIÓN DEL PROYECTO**

Debido a los objetivos que nos hemos puesto para el proyecto de fundamentos, al igual que los limites a los que nos debemos apegar por el ámbito de la materia, se nos hace indispensable el crear una explicación sobre el modelo sobre el cual nos hemos guiado:

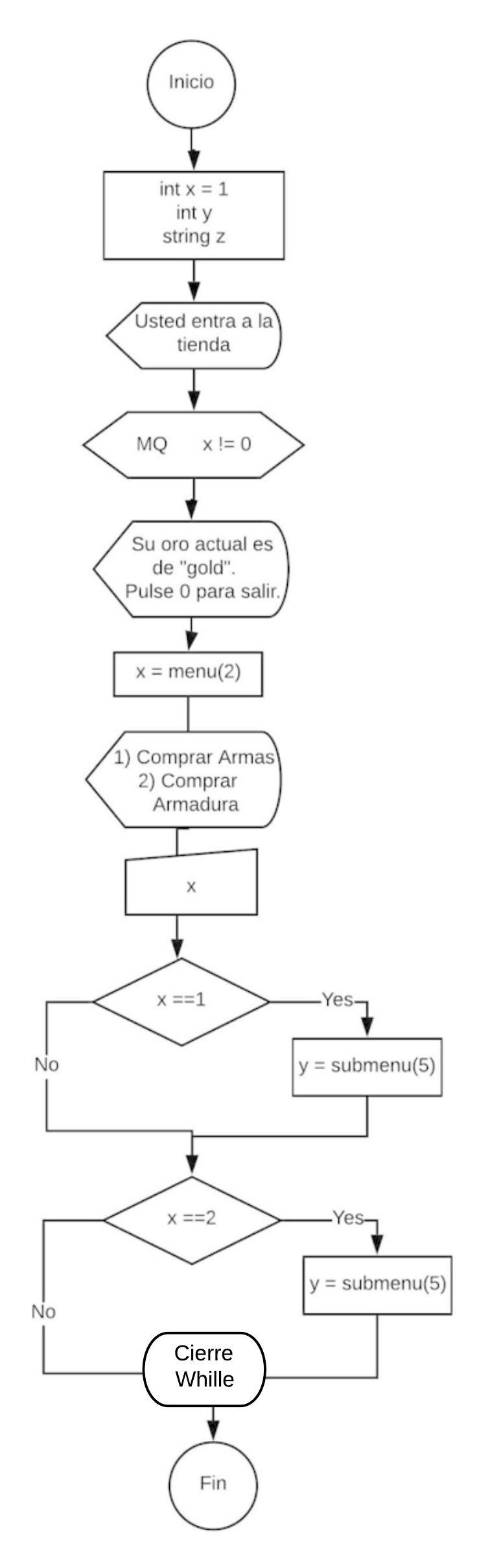
1. El juego está basado en texto, escrito directamente a la consola.
2. Los sistemas del juego serán basados en un estilo simplificado de un juego RPG. Estos sistemas son:
3. Un sistema de movimiento, que proporciona descripciones cuando el jugador entra a una nueva área. Por igual, este sistema crea un numero aleatorio con la librería ‘math.h’ que se utiliza para medir la probabilidad de que el jugador sea atacado por un enemigo. El sistema funcionara por medio de una función simple que tomen un parámetro de tipo entero ‘x’, y un switch que compare a la variable ‘x’ con los casos posibles. De ser un caso aceptado, se ejecutará un cambio a una variable global y se reiniciará la función mediante un bucle.
4. Un sistema de combate, en el cual el jugador peleara con enemigos hasta que los puntos de vida de alguno de los contrincantes lleguen a ser 0. Esto será hecho por medio de un ciclo principal ‘while’ con dos parámetros, la vida del jugador y la vida del enemigo. Se inicia con el turno del jugador seguido por el del enemigo. Los enemigos son creados con la función del movimiento, siendo asignados directamente a cada “área”. Esta función es inicializada con una función simple, la cual consiste en una variable local ‘x’ que utilizamos como un identificador y la cual es un parámetro de la función. Esta variable pasa a un ‘switch’ que inicializa el combate usando parámetros predeterminados para cada valor de ‘x’. Los parámetros se refieren a la vida del enemigo, el valor mínimo de daño que puede hacer, un rango de valores arriba de ese daño mínimo que pueda causar, y un rango de valores que se refieran a las posibles recompensas en caso que el jugador gane la pelea. Las acciones del jugador se rigen por un menú, y sus atributos son cambiados dependiendo de las variables globales pertinentes, siendo reseteadas antes de que reinicie el bucle.
5. Un sistema de tienda, el cual permite al jugador obtener ciertos ‘objetos’, aunque no son inicializados como tales, que les permita subir sus propios ‘atributos’ en forma de ciertas variables globales. Aquí también se utilizará un sistema parecido al del sistema de movimiento, utilizando un arreglo para crear funciones que asignen un valor a las variables locales de costo y de beneficios. Al ser comprado, un ‘objeto’ cambia la variable necesaria instantáneamente, y resta el oro de su costo de los valores del jugador.
6. Un sistema de subida de nivel, el cual permite que los ‘atributos’ del jugador suban de manera natural cuando la variable de experiencia llegue a valores predeterminados. Esto sera hecho mediante una función simple que se repite al final de cada instancia de combate.
7. El objetivo del juego será simplemente llegar al nivel mas alto posible y triunfar contra un enemigo especial en el ‘área’ mas alejada del inicio del juego, en cuyo punto termina la partida.

**DIAGRAMAS DE FLUJO Y ALGORITMO DE CADA PROCESO**



**Función menú**: La función menú, esta caracterizada por ser una función de carácter reutilizable a lo largo de todo el programa. Debido a que nuestro programa es un juego basado, en su mayor parte, por menús, resulta conveniente tener una función dedicada a la elección de opciones. Ya que cuando sea que se requiera la elección de mas de una opción en particular, para la realización de una acción indicada por el usuario, resulta conveniente mandar a llamar a la función ‘menú’ e indicarle la cantidad de opciones con las que se trabajará, dependiendo del caso. En caso de que el jugador no elija una opción válida, se utiliza un bucle que no terminara mientras una variable booleana sea verdadera. Solo se le cambia el valor cuando el jugador escoja una opción válida.

*Comentario:* Aplica la misma lógica desde la opción ‘1’ (la cual es mostrada en el diagrama de flujo), hasta la opción ‘5’.



Int x = 1, y

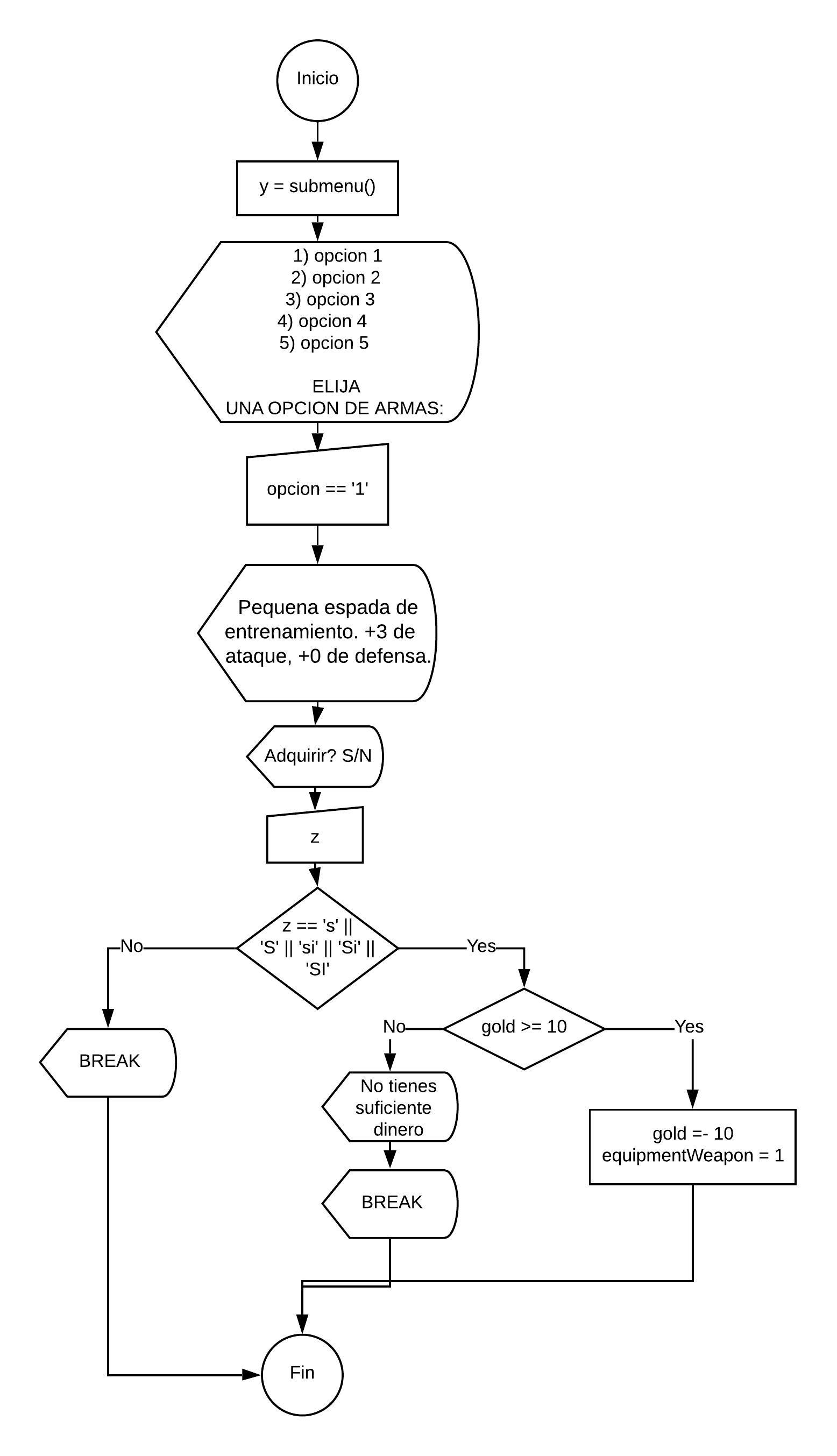
String z

**Función Tienda**: la función tienda, es la primera función del programa. Al iniciar el juego, se le presenta la opción al usuario de ‘entrar’ a la tienda. En la cual, se pueden comparar objetos, tales como armas o armadura. Para poder adquirir dichos objetos, el usuario necesitará disponer de una cantidad suficiente de oro para poder comprarlos. Para cada menu se llama a la funcion menu.

En la función *‘tienda’*, primero se declaran las variables ‘x’ y ‘y’ de tipo integer, y la variable ‘z’ de tipo string.

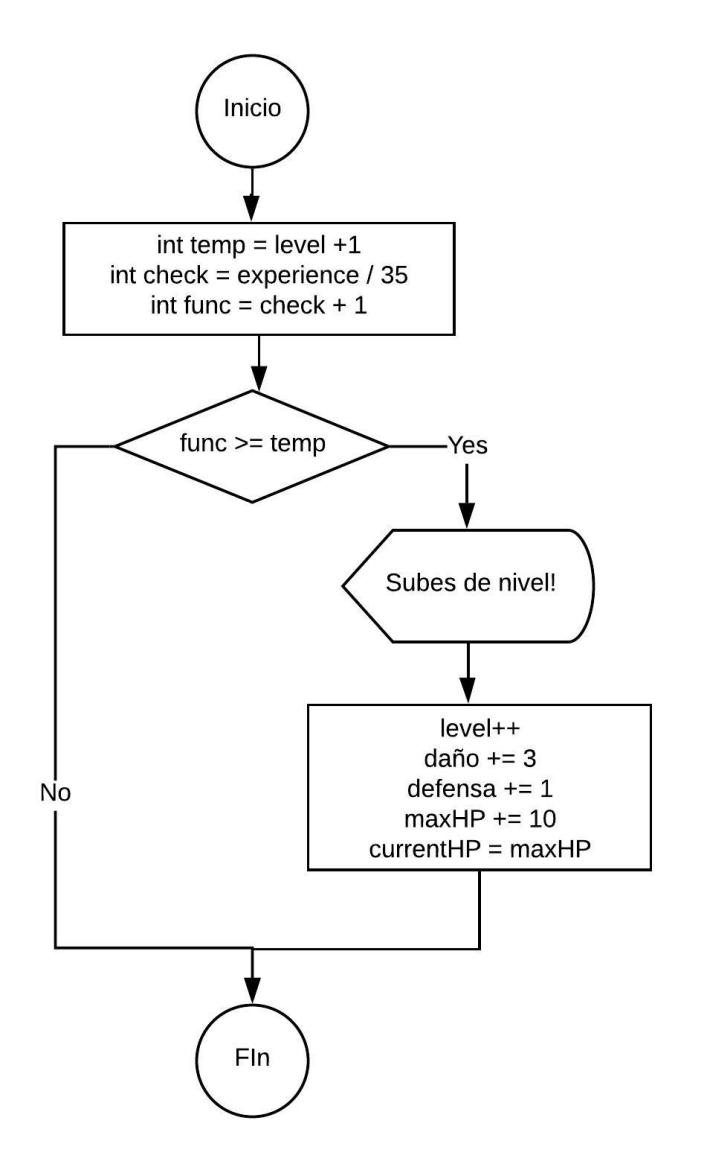
Luego, se presentan dos opciones en estilo sub-menú: la primera, de comprar armas y la segunda, de comprar armadura *(ambas mostradas en el diagrama de flujo como ‘y = submenu()’)*. El usuario podrá ingresar la opción deseada, es decir, 1 o 2. Para este caso, se utilizó un bucle while, ya que cada vez que el usuario ingrese una de esas dos opciones, es posible que deseen comprar otro objeto.

Posteriormente, cuando el usuario ingrese una de las dos opciones indicadas, se desplegará un ‘submenú’. Aquí es donde entra en juego la variable ‘y’, llamando a la función menú nuevamente. El valor de la variable y se usa en un switch para determinar el objeto que se desea comprar. Se le presenta al jugador una breve descripcion de los efectos del objeto, y seguidamente se le pregunta si desea comprarlo. En caso de que no tengan suficiente oro para realizar la transaccion, se le informa al jugador y se reinicia el bucle. Esto tambien ocurre si el jugador no ingresa un valor reconocido de respuesta, la cual puede ser ‘s’, ‘S’, ‘Si’, ‘si’, y ‘SI’. No se verifican otras respuestas ya que el resultado sera el mismo, el reinicio del bucle.



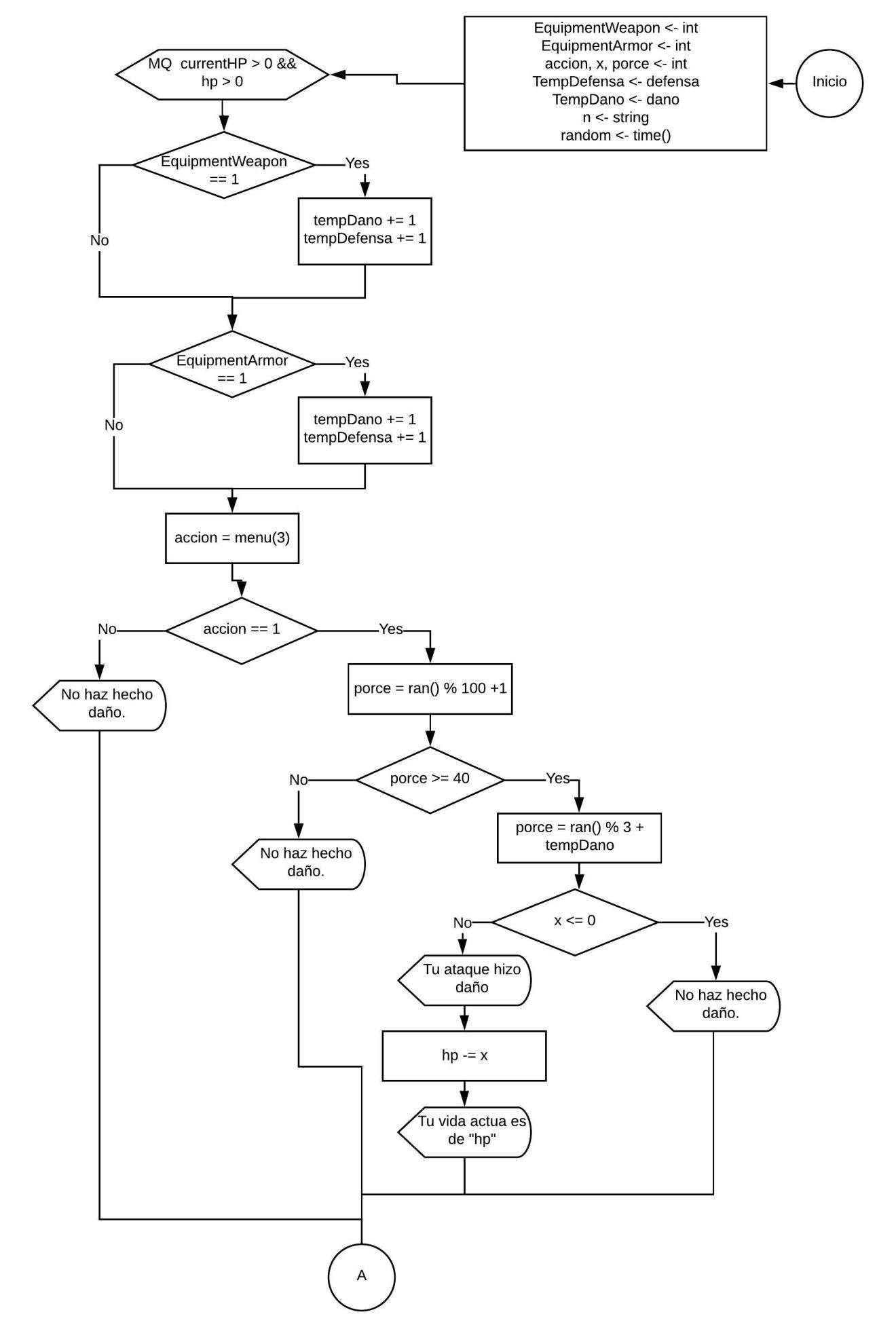
*Comentario*: Solo se muestra el flujograma del submenú de armas *(declarado como ‘ y = submenu()’ en el flujograma principal)* y su respectivo caso ‘1’, ya que la lógica siempre está sujeta a la misma, lo único a cambiar son los parametros dependiendo del caso y submenú a trabajar.

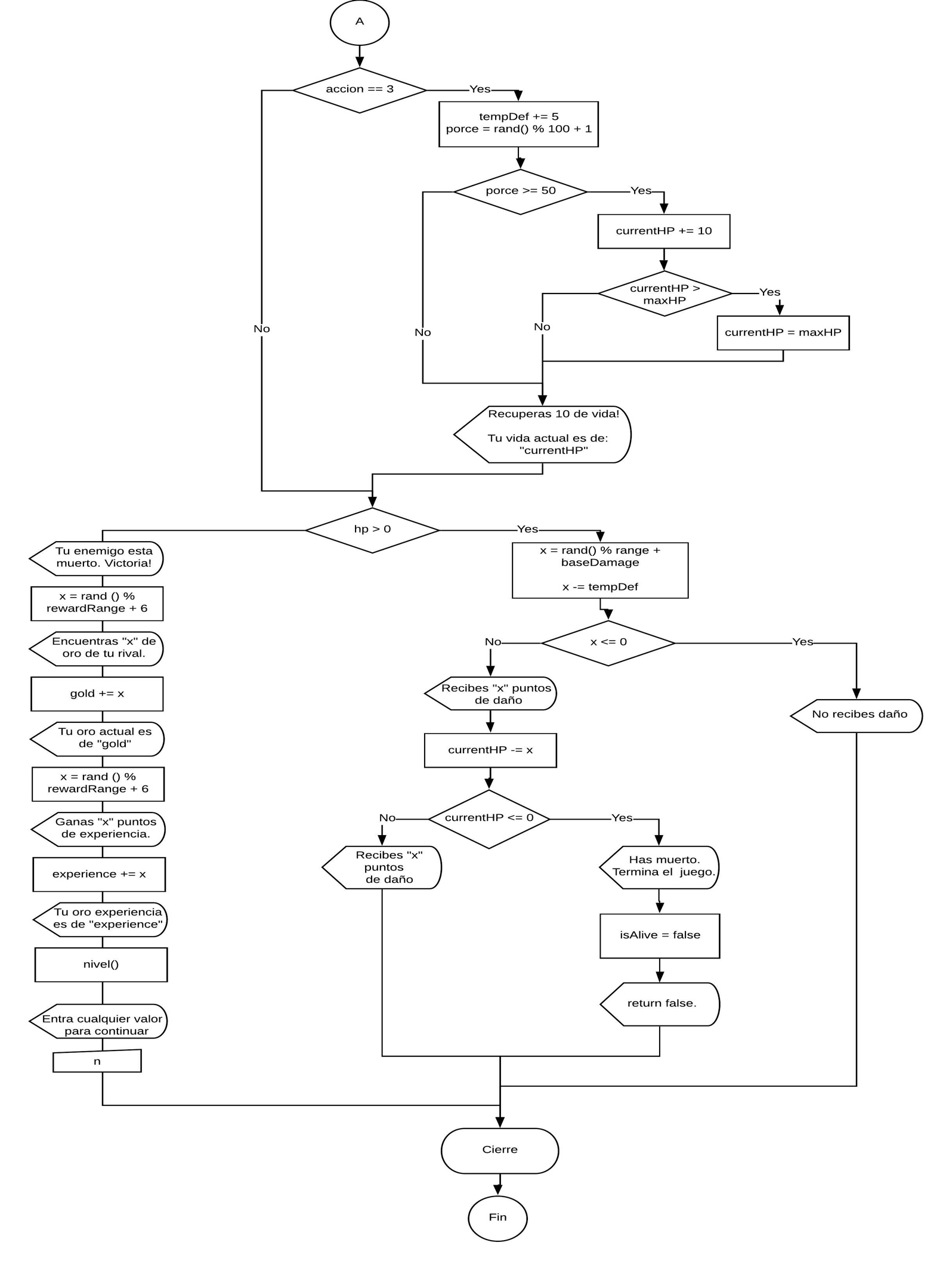
**Submenus de Armas y Armaduras**



**Función Nivel**: la función ‘nivel’, es la encargada en subir de nivel al usuario. Esta tomará efecto, cuando se cumpla que si el valor de la variable ‘func’ es mayor al de la variable ‘temp’. Si la condición se cumple, se mostrará un mensaje en pantalla, haciendo referencia a que el usuario, en efecto, ha subido de nivel. Posteriormente, se incrementarán el nivel daño, defensa, y vida. Todo esto, mediante un contador, tal y como se muestra en el diagrama de flujo. Es decir, que las variables irán aumentando, a medida que el usuario incremente de nivel; cabe recalcar, que la variable de vida actual del usuario (currentHP) será igualada a la cantidad máxima de vida (maxHP), es decir, el valor que irá aumentando a medida que el usuario suba de nivel.

La variable ‘func’ es igual a el valor de la experiencia dividido entre 35, redondeado hacia abajo, + 1. La variable ‘temp’ es igual al contador del nivel + 1. De este modo, la primera subida de nivel ocurre a los 35 puntos de experiencia, la segunda a los 70, la tercera a los 105, y así sucesivamente.



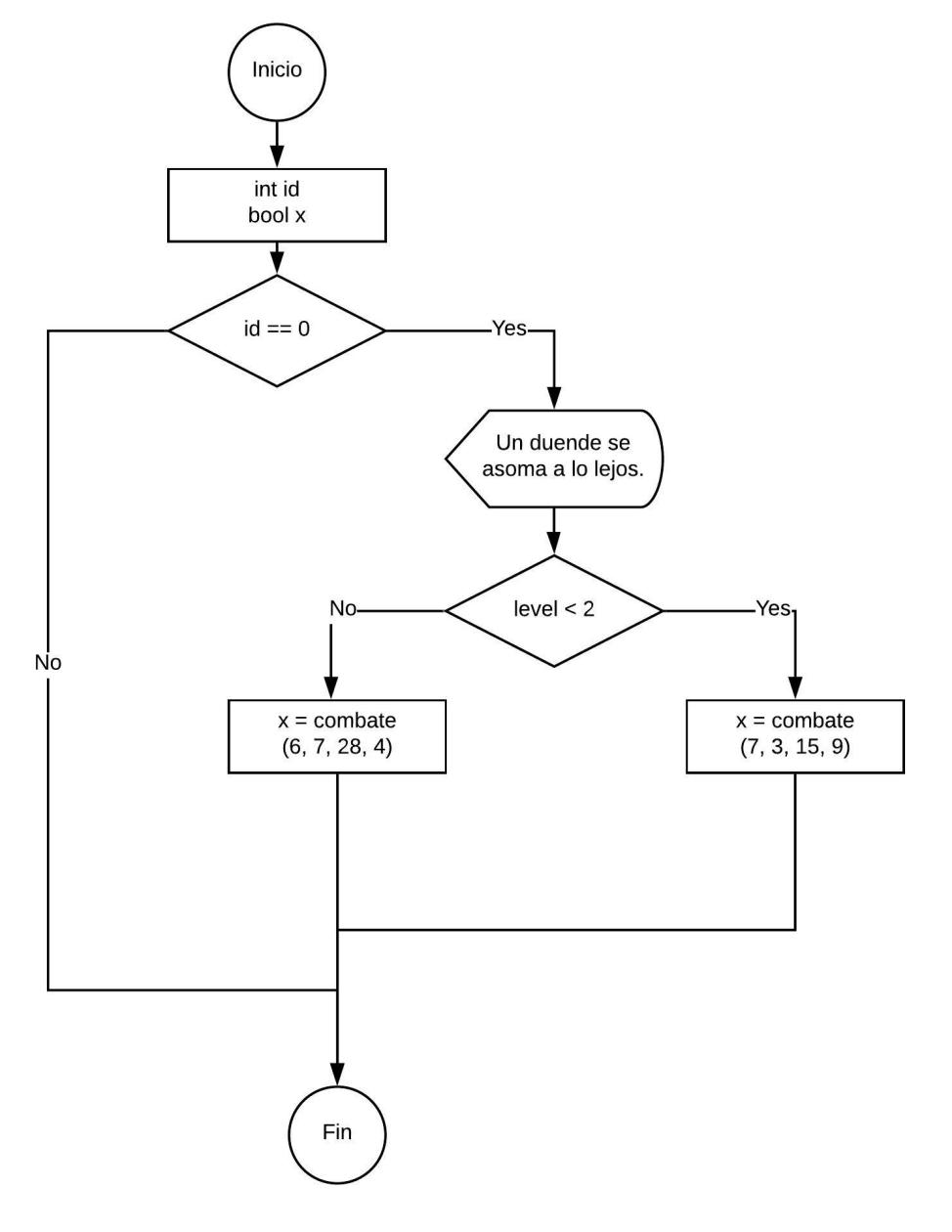


**Función Combate**: La función ‘combate’, es la encargada de todas las acciones que ocurrirán durante un enfrentamiento contra un enemigo a lo largo del juego. La función esta dividida en facetas, las cuales se encuentran dentro de un bucle cuyas condiciones son que la vida del jugador sea mayor a 0, y que la vida del enemigo igualmente sea mayor a 0:

1. *Switch EquipmentWeapon*: a lo largo de un combate, el daño y la defensa, podrá a estar afectado dependiendo el tipo de arma que se tenga. Por ejemplo, en este diagrama de flujo que se presenta, presenta una opción, en la cual, si el usuario tiene el arma ‘1’, afectará en la cantidad de daño que realizará y la cantidad de defensa que gana o pierde el usuario. (Se muestra solo un caso, ya que aplica la misma lógica para el resto de armas que se puedan llegar a elegir).
2. *Switch EquipmentArmor*: Aplica la misma lógica que el switch anterior, con la diferencia, que en este caso se trata de armaduras que el usuario pueda llegar a elegir.
3. *Menú Acción*: durante un combate, el usuario será preguntado que acción desea hacer, ya sea un ataque ‘fuerte’, ataque ‘suave’ o el de defenderse. Para el caso ‘1’ (ataque fuerte) se aplicará un porcentaje mayor o igual del cuarenta porciento, en el cual el ataque sucederá. Este valor es del veinte porciento para el caso ‘2’. De lo contrario, no surtirá efecto alguno el ataque hacia el enemigo. En el caso ‘1’, el daño que puede aplicar el jugador es mayor que en el caso ‘2’, pero tiene una menor probabilidad de surtir efecto. Luego para el caso ‘3’ (defender), se aplicará mayor o igual al cincuenta porciento, si esta condición se cumple, el usuario regenerará 10 puntos de vida, y ganara 5 puntos de defensa, de lo contrario, no tomará efecto la acción.

Comentario: en el diagrama de flujo, solo se muestra la acción ‘1’ y la acción ‘3’, esto es debido a que la acción ‘2’, esta bajo la misma lógica que la acción ‘1’, con la diferencia que la acción ‘2’ esta sujeta a distintos parámetros.

1. *Condición ‘HP > 0’*: Esta condición revisa si la vida del enemigo sigue siendo mayor a 0. Si lo es, procede a atacar al jugador usando los parámetros del enemigo individual. Si no lo es, le notifica al jugador que ha sido victorioso. También se le provee al jugador experiencia y oro en base a una función rand que utiliza un parámetro del enemigo. Aquí se llama la función “Nivel” para revisar si el nuevo valor de experiencia amerita subir de nivel.
2. *Si ‘HP > 0’ es verdadera:* En caso de que el enemigo no haya muerto, el jugador tomara daño. Aquí se realiza este calculo, y se revisara que la vida del jugador aun sea mayor a cero. En caso que no sea, se terminará su partida por medio de un valor booleano que crea un while en la función ‘main’. Si sigue mayor a 0, se repite el ciclo.

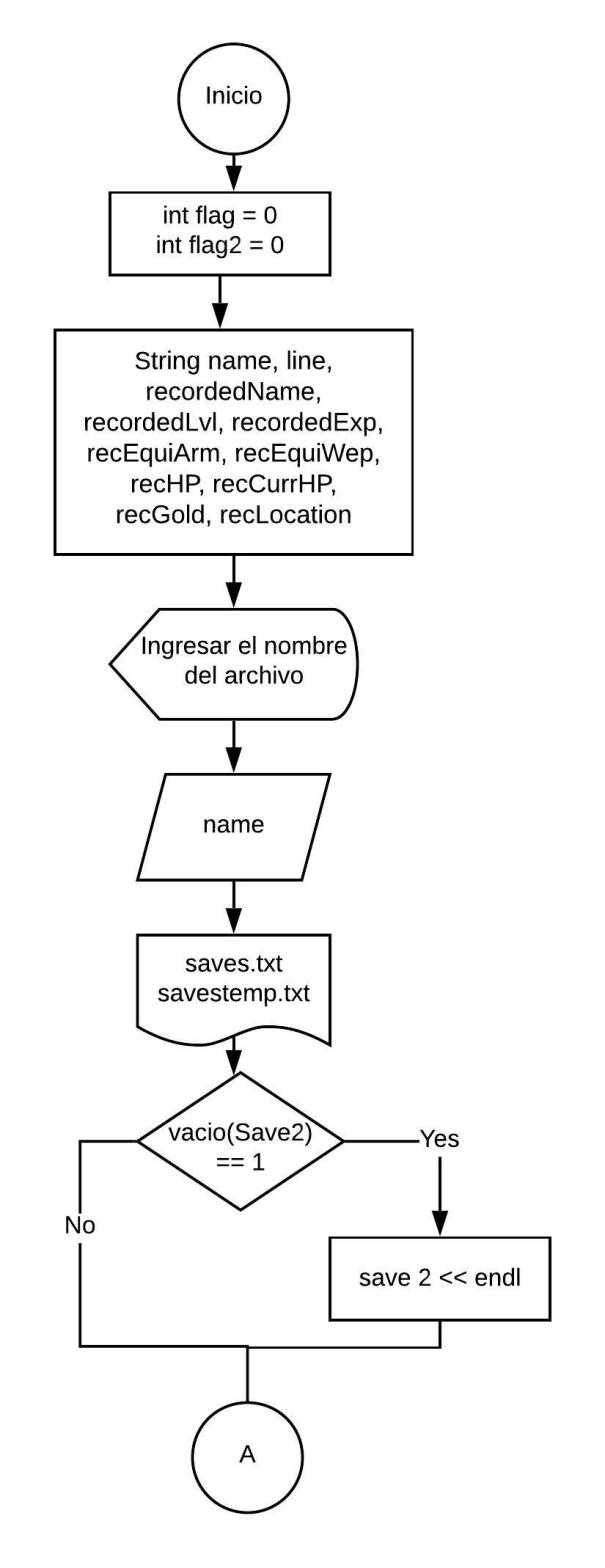
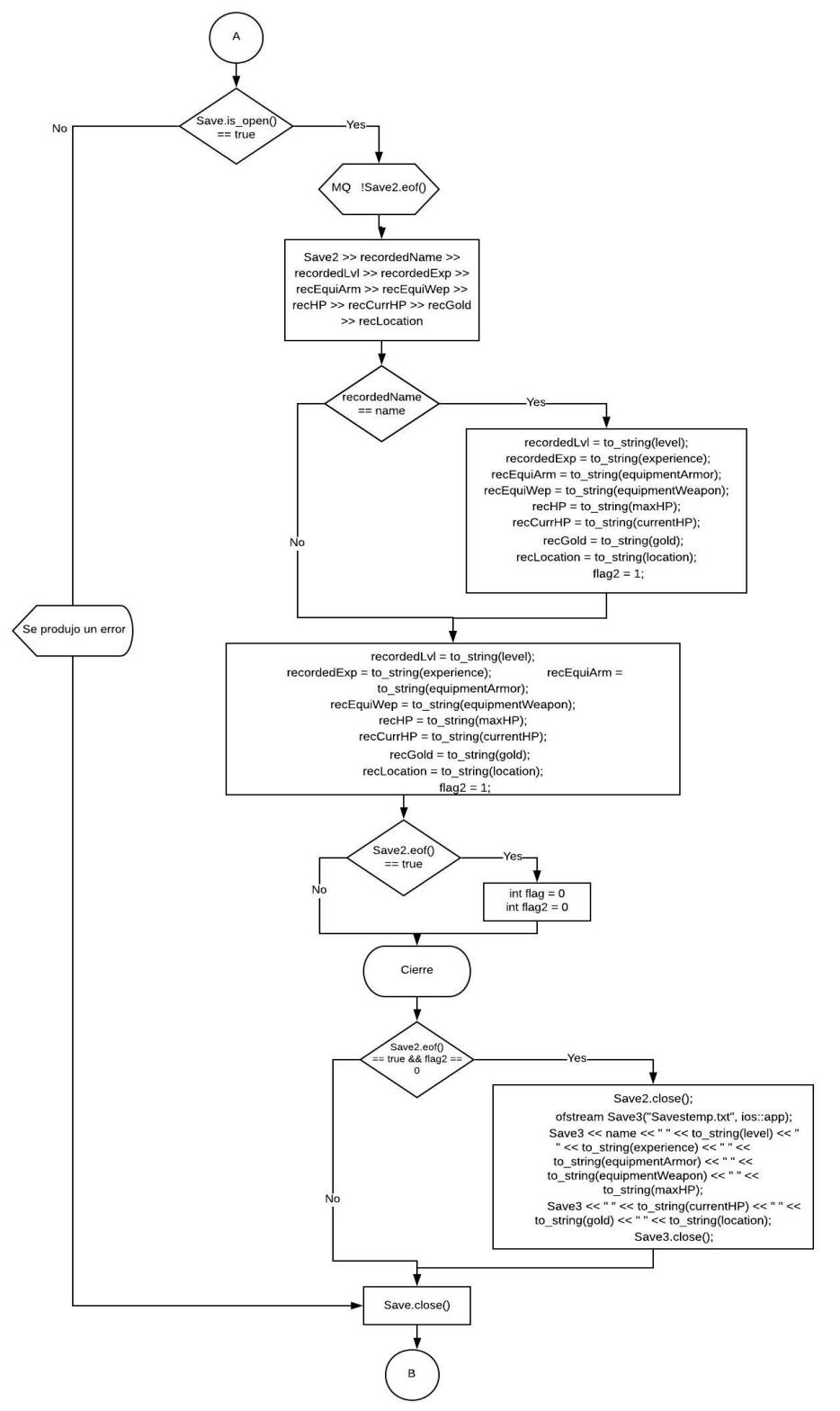


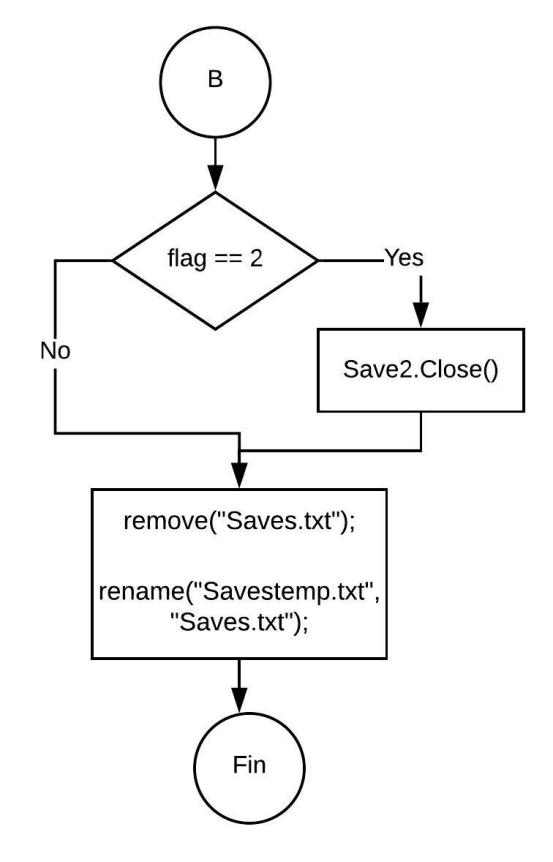
**Función Enemigo**: La función ‘enemigo’, es la encargada en darle los parámetros del enemigo a enfrentar. Con parámetros, se refiere a:

1. El rango de valores que el enemigo pueda causar en daño.
2. El daño mínimo que el enemigo pueda realizar.
3. El nivel de vida del enemigo.
4. La cantidad de recompensa, que pueda llegara a dar el enemigo cuando se le venza.

Estos parámetros estarán sujetos a cambios dependiendo de si el usuario es menor al nivel que se muestra en la condición del enemigo al que se enfrente. Este es el valor ‘2’ en el ‘if’ del diagrama de flujo y todos estos parámetros son los que inicializan a la función ‘combate’.

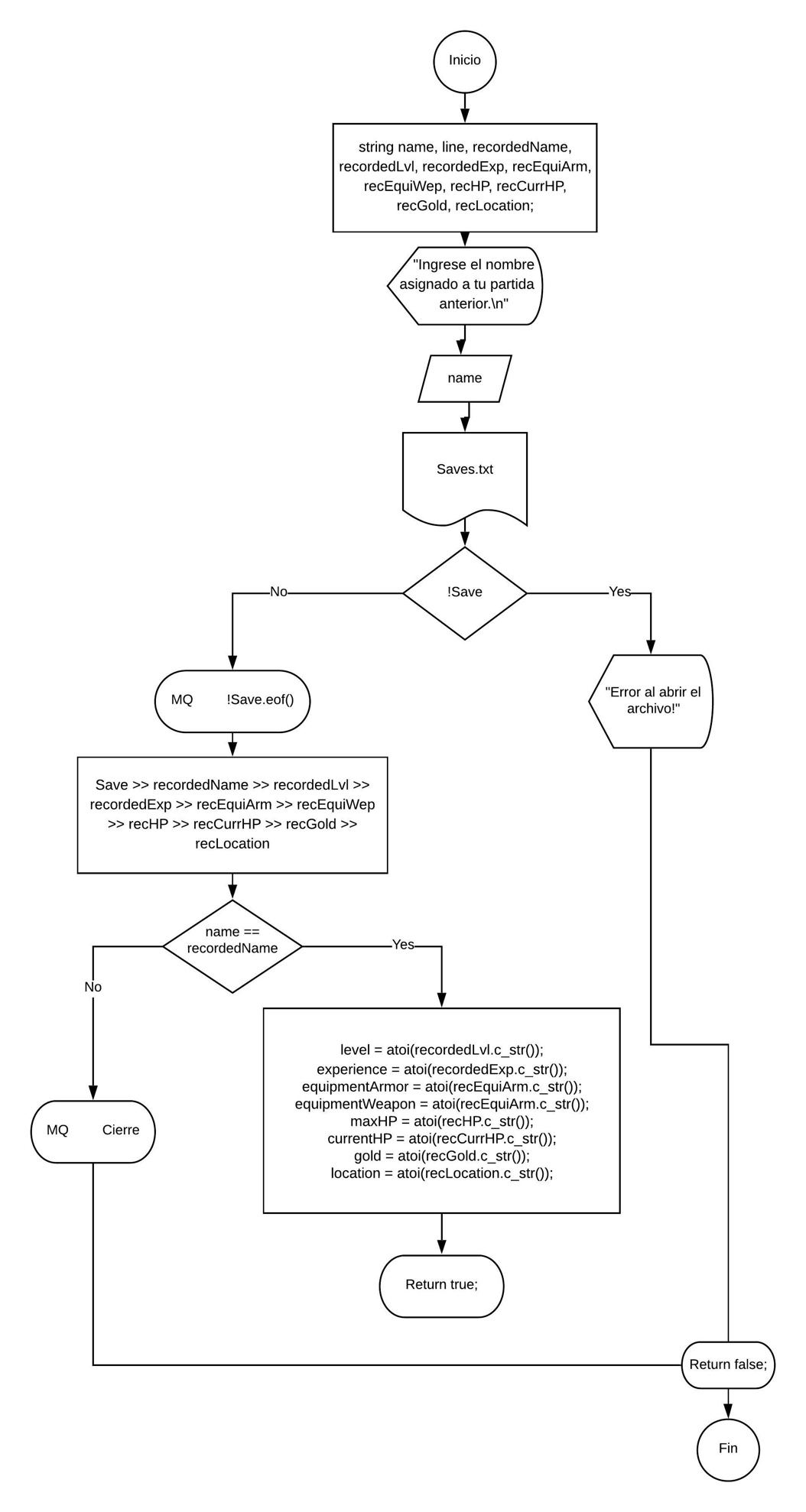
*Comentario*: el diagrama de flujo, solo muestra el caso para un enemigo, ya que, lo único que ira cambiando dependiendo del enemigo, serán los parámetros. La lógica será siempre la misma.





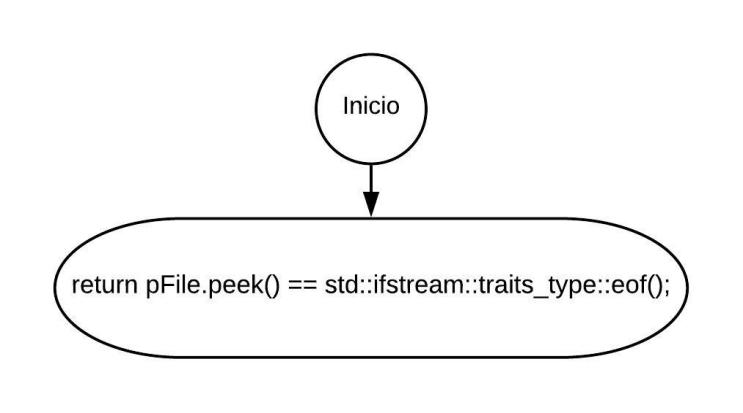
**Función Save**: En esta función se guarda la partida del jugador en varios pasos.

1. Se le pide al usuario un nombre para su partida
2. Para empezar, se abre el archivo “Saves.txt” en una variable ‘ifstream’ con nombre Save2, y se crea un archivo llamado “Savestemp.txt” en una variable ‘ofstream’ llamada ‘Save’. También se declaran las variables locales que se utilizaran por el programa, al igual que dos valores de tipo entero que utilizamos como ‘flags’
3. Se llama a la función vacío para revisar si la variable ‘Save2’ esta vacía.
4. Se revisa que la variable ‘Save’ sea abierta correctamente. Si no, se da un mensaje de error. Si lo es, se continua.
5. Se inicia un bucle que no termina hasta que se llegue al final de la variable Save2, sin que la variable ‘flag’ se haya utilizado.
6. Se leen a variables locales los valores pertinentes a la partida de la variable Save2.
7. Se revisa el nombre de la partida actual con los nombres dentro del archivo. Si algún nombre coincide, y aun no se ha utilizado la primer ‘flag’, se sobrescriben los valores anteriores con los valores actuales y se utiliza la variable ‘flag2’ para que no se repita este código.
8. Sin importar la ejecución del ‘if’, se guardan las variables locales a la variable ‘Save’.
9. Si se llega al final del documento, se cambia el valor de la variable ‘flag’ para que el bucle no se repita.
10. Fuera del bucle, si se encuentra al final del documento y la variable ‘flag2’ no fue utilizada, se le agrega al documento una nueva línea con las variables de la partida actual. Para esto, cerramos la variable ‘Save’, y abrimos el mismo documento con una variable ‘ofstream’ llamada Save3. Esta variable contiene el argumento ‘ios::app’, con el cual se asegura que solo se le agregue los valores al final del documento. Esta variable se cierra dentro del mismo ‘if’.
11. Se cierra la variable ‘Save’ y se revisa que la variable ‘flag2’ no se haya utilizado. Si se utilizo, entonces cerramos la variable Save2, la cual aun esta abierta si solo se reemplazaron valores.
12. Se renombra “Savestemp.txt” a “Saves.txt” después de eliminar a “Saves.txt”, con el fin de tener un solo documento ordenado para almacenar los archivos guardados.

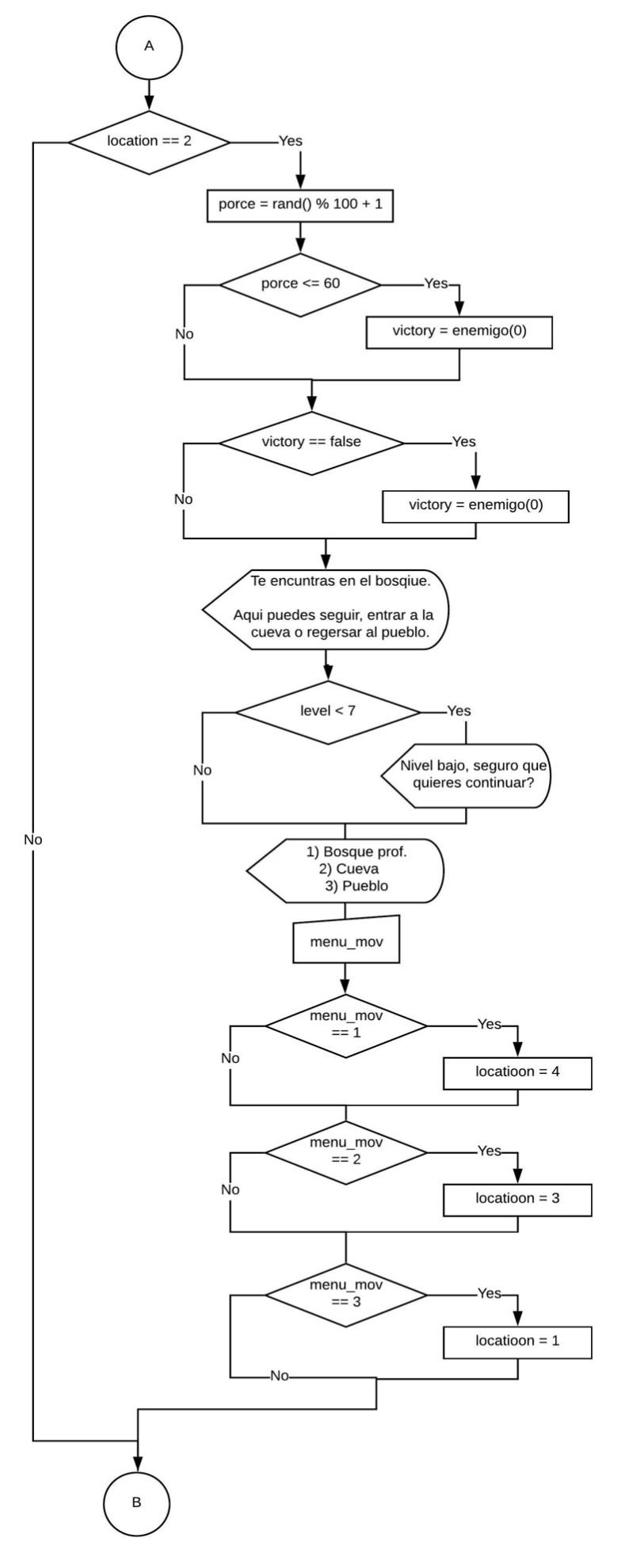
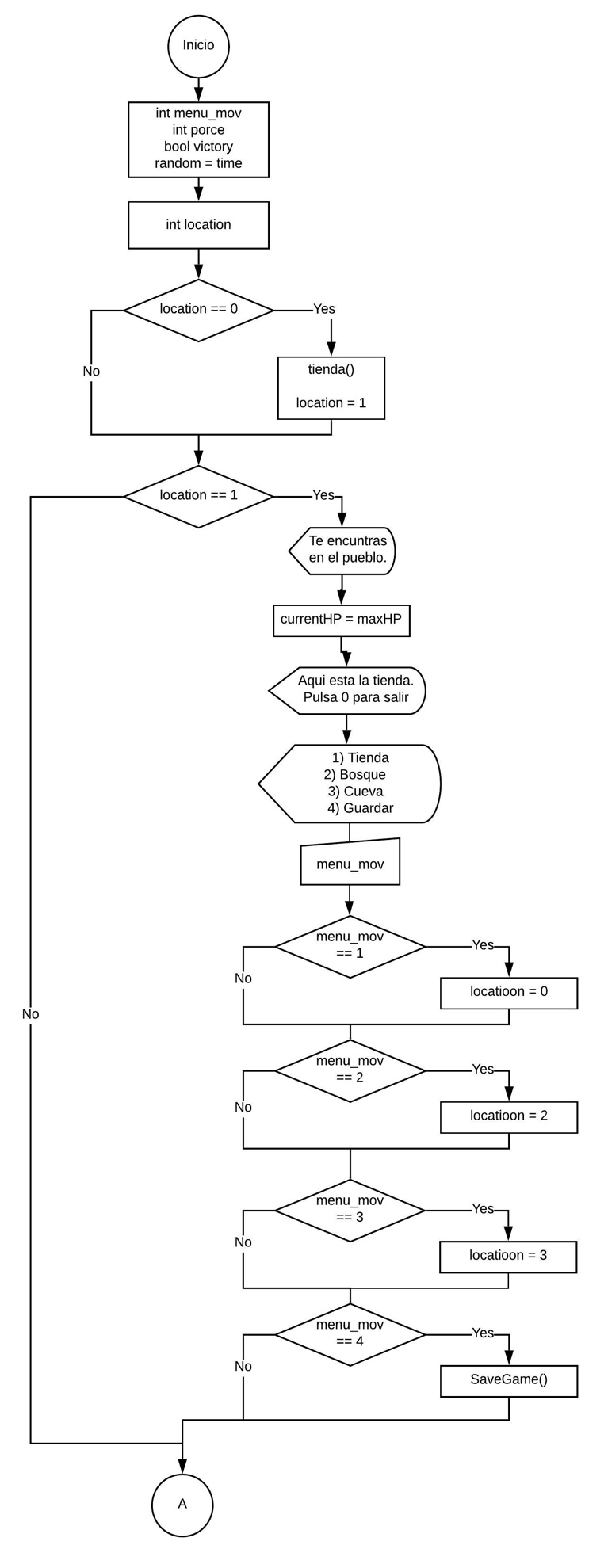


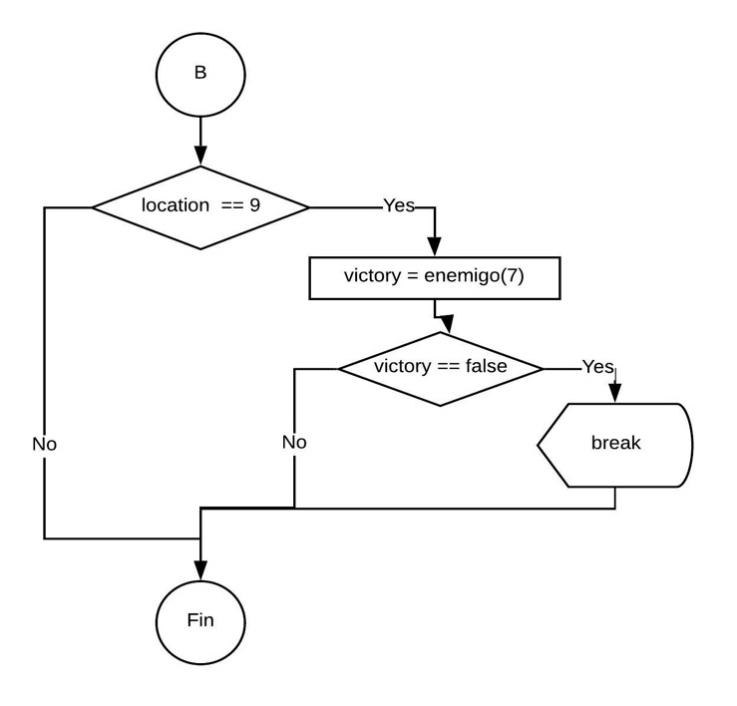
**Función Load**: En esta función se leen los valores del archivo “Saves.txt” para resumir una partida previa en ciertos pasos.

1. Se le pide al jugador el nombre de su partida anterior.
2. Se abre en una variable ‘ifstream’, llamada ‘Save’, el archivo “Saves.txt”.
3. Se revisa que el archivo se haya abierto de forma correcta. Si no, se le informa al jugador y la función regresa un valor falso.
4. Si se abre, entra a un ciclo while que no termina hasta llegar al final del archivo.
5. Se leen a variables locales los valores escritos en el archivo
6. Se revisa la variable del nombre que se le pidió al jugador, con los nombres ya escritos en el archivo. Si alguno es igual, se reemplazan las variables globales con los valores del archivo y la función regresa “true”.
7. Si ninguno es igual, se sale del while y regresa “false”



**Función ‘Vacío’**: Esta función se encarga de leer un archivo y determinar si esta vacío o no. Fue proporcionada por un instructor y se utiliza en la función ‘Save’.





**Función Movimiento:** La función ‘movimiento’, es la encargada en permitir al usuario moverse en los distintos lugares presentados a lo largo del juego.

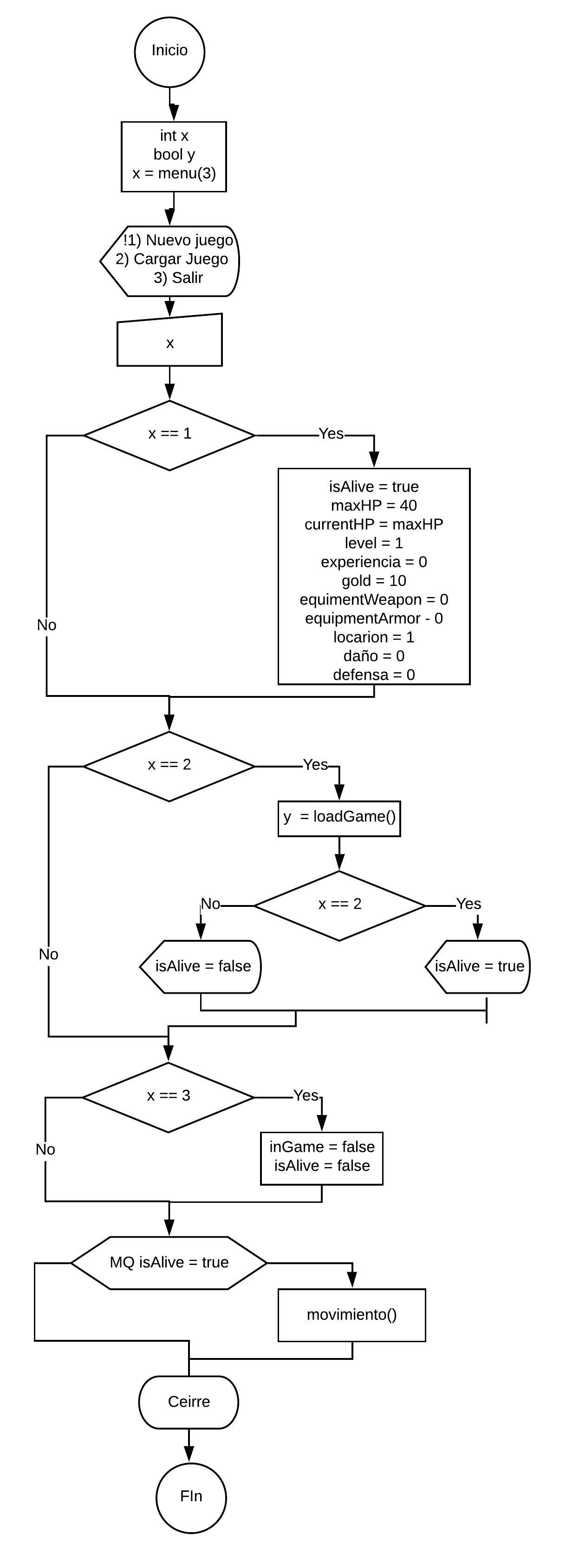
Si se elige el caso ‘0’, el usuario será presentado con la tienda del juego, llamando a la función. Seguidamente, serán regresados a la local ‘1’, la cual es un pueblo.

Luego, si se elije el caso ‘1’, al usuario se le presentaran 5 opciones: del 1 al 3, de moverse al lugar de elección, la opción 4 si se desea guardar la partida, llamando a la función *‘*Save’, y la opción 5 le permitirá salir del juego completamente.

Posteriormente, si la elección es el caso ‘2’, se presentará un porcentaje de probabilidad del sesenta porciento, en la que el usuario enfrentará o no, a un enemigo (Utilizando a la función ‘*enemigo’*). Si no se llega a cumplir la probabilidad, se continua en el juego.

De cumplirse dicha probabilidad y si el usuario resulta vencedor contra el enemigo enfrentado, proseguirá en el juego; por consiguiente, será presentado con opciones de moverse a otras locales del juego, y continuar en el mismo. De lo contrario, el juego terminará, y el usuario tendrá que volver a empezar desde el último punto guardado, o bien desde el inicio. *(La misma lógica aplica desde el caso ‘2’ hasta caso ‘8’, solo irán cambiando los parámetros de la probabilidad del enemigo, y de las locales a las cuales se puede mover.)*

Finalmente, el caso ‘9’ será la ‘batalla final’. Si el usuario resulta vencedor contra el enemigo enfrentado, resultará victorioso y el juego terminará, dejando al jugador saber mediante un mensaje. De lo contrario, el juego terminará, y el usuario tendrá que volver a empezar desde el último punto guardado.



**Función Menu\_Principal**: la función ‘menu\_principal’ es la que posteriormente, se ejecutara en el main.

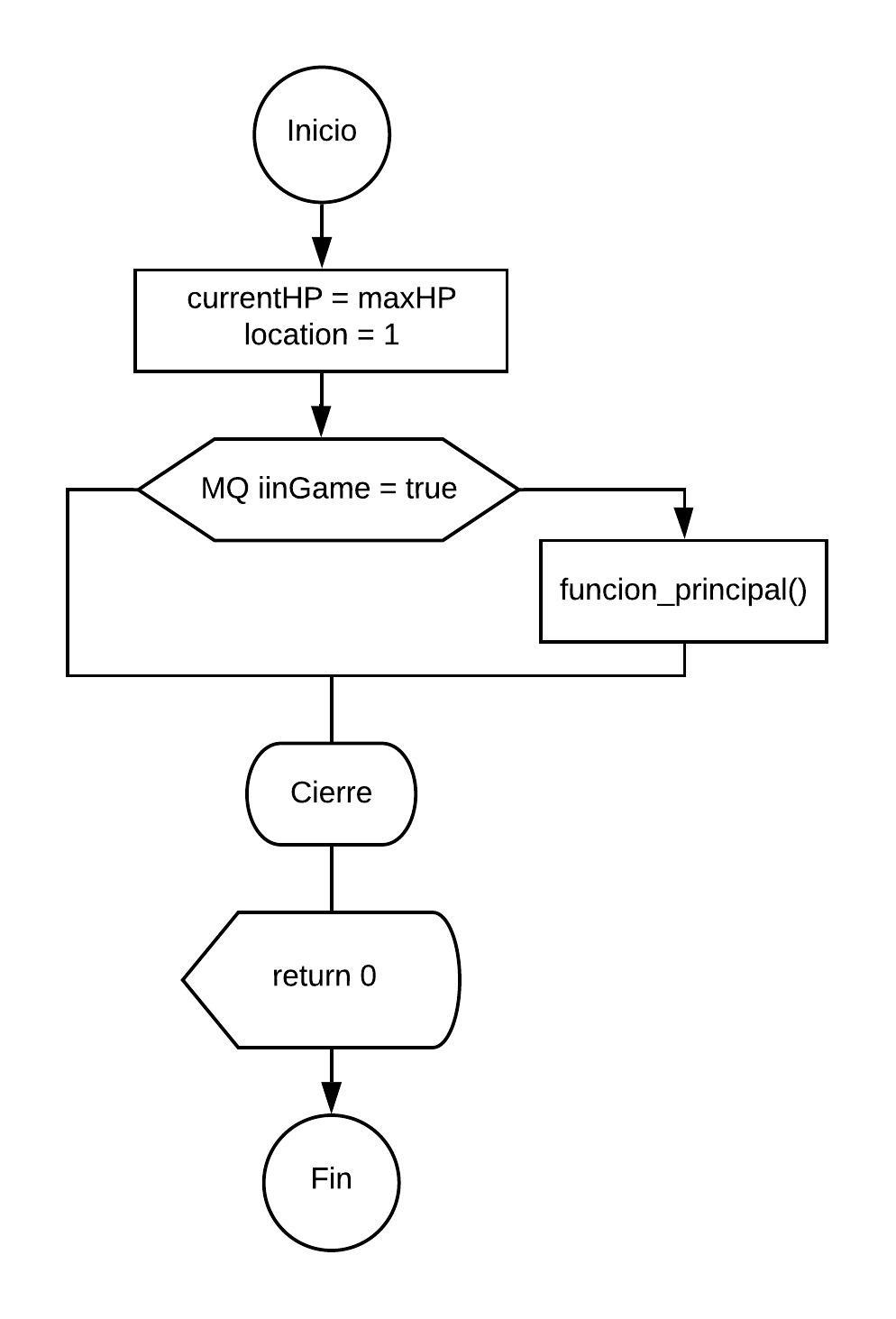
Dicha función, toma forma de un menú, en la cual el usuario tendrá tres opciones a elegir. La primera, si quiere iniciar un nuevo juego. La segunda, si se desea cargar una partida previamente guardada. Y la tercera, si se quiere salir.

En la misma, si se desea elegir la opción uno, se le brindará al usuario a la local ‘1’ del juego, con los valores predeterminados que hemos asignado para el inicio de una partida. Tambien se inicia el valor isAlive con true para poder entrar al bucle posterior.

Posteriormente, si se elije la opción dos, el programa cargará cualquier partida y sus datos correspondientes previamente guardados por el usuario. Si no puede cargar la partida especificada (La funcion regresa ‘false’) entonces no se inicia el valor isAlive, con el fin de que el programa no se atore en un bucle infinito y vuelva a mostrar el menu.

Luego, la opción tres, tal y como está indicada en el menú mostrado, terminará la ejecución del programa al cambiar el valor de las booleanas “inGame” y “isAlive” para evitar la ejecucion del programa.

Por último, se utilizará un bucle while, en el cual su condición será, si el usuario está vivo, se mandará a llamar a la función movimiento, que es donde el juego ‘empieza’. Este bucle es el que permite que la funcion movimiento se repita, lo cual permite que el jugador vaya de lugar en lugar y luche contra enemigos de manera consistente y rapida.



**Función Main**: la función ‘main’, es donde todo se ensambla. Se utilizará un bucle while, en el cual su condición será, si el usuario desea estar en el juego (variable “inGame”), se mandará a llamar a la función *menu\_principal*, aquí mismo, es donde el juego podra correr completamente. Ya que existe un bucle dentro de la funcion llamada, el menu no se repetira a menos que el jugador haya muerto en su partida.