

Universidad de Costa Rica

Programación bajo plataformas abiertas

Grupo: Tres en línea

Desarrollo del juego: Gato

Integrantes:

**Alison Rivera Cisneros
C06510**

**María Paulette Pérez Monge
B95916**

2021

Índice:

Introducción	2
Diseño general	3
Principales retos	7
Conclusiones	13
Bibliografía	14

Introducción

Desde la infancia, la mayoría se ha familiarizado con el juego de Gato o tres en raya, por lo que pareció que no estaría demás una versión virtual del mismo, junto a que tiene reglas bastante simples.

En este documento se plantea exponer a detalle el camino que se siguió a lo largo de las clases de laboratorio y reuniones, para desarrollar el juego Gato en lenguaje de programación c. Para la elaboración de este proyecto se emplearon una serie de herramientas virtuales y tecnológicas para lograr el objetivo, entre las que se encuentran Replit, Github, zoom, entre otras.

El escrito se dividirá en tres distintas partes donde se plantean detalles alusivos al diseño general, que incluirá aspectos referentes al desarrollo del juego como la programación del menú principal, el almacenamiento de los datos de cada jugador como lo son el nombre y su puntuación en cada partida. En esta sección del documento también se encontrará información sobre el proceso para crear un repositorio en Github, darle acceso a externos y definir el archivo README.

Entre las otras secciones en las que se dividirá el documento están los principales retos, donde se explicará ampliamente las dificultades que se enfrentaron al escribir el código, incluyendo los detalles de cómo se planteó el desafío, si se logró solucionar y de ser así los pasos que se siguieron para lograrlo. Otra de las separaciones de este escrito serán las conclusiones donde se plantea hacer una síntesis de la experiencia identificando los conocimientos adquiridos y las incógnitas generadas.

Diseño general

En este apartado se describirán a detalle cómo se decidieron programar cada sección del juego desarrollando, especificando códigos, procesos y líneas de pensamiento tomados para lograr el objetivo. A continuación se subdividen cada una de las secciones generadas en el trabajo con su respectivas explicaciones.

Desarrollo del menú principal

Esta parte del juego fue una de las últimas en programarse, ya que a la hora de iniciar con el desarrollo, los esfuerzos se enfocaron principalmente en la programación de las partes más complejas. Una vez concluido con los apartados más desafiantes se procedió a establecer *switch* con dos casos concretos. El uno hace referencia a que el jugador digite el valor 1 para comenzar el juego, el segundo se basa en presentar las mejores puntuaciones del historial de juegos cuando el jugador selecciona el número 2, y para finalizar el *switch* se definió la opción *default* para cuando se decidiera cerrar el juego desde el inicio, conformando de esta manera el menú principal.

Solicitar y almacenar el nombre de los jugadores

Para este apartado se plantea en un inicio la opción de solicitar los nombres de los jugadores y desarrollar una forma en la que se elija aleatoriamente cuál jugador tiene el primer turno, o sea, cuál de los dos nombres de usuario es el jugador 1. Sin embargo esta posibilidad se descarta ya que conlleva el desarrollo de una función extra que complica el código y fácilmente la acción puede ser realizada manualmente por los jugadores al cambiar el orden en el que introducen su nombre de usuario por turno. Posterior a esta propuesta se inició el código para solicitar el nombre de cada jugador, el cual consta de 6 líneas en las cuales se definió un arreglo tipo *char*, que almacena los nombres como se observa en la figura 1.

```
char jug1[15];
char jug2[15];
printf("Ingrese nombre jugador 1\n");
scanf("%s",jug1);
printf("Ingrese nombre jugador 2\n");
scanf("%s",jug2);
```

Figura 1. Solicitud de nombres de jugadores.

Desarrollo del juego

En primer lugar se determinaron los requerimientos del juego por desarrollar; al ser gato, el elemento que se necesitaba era una cuadrícula que funcionara como el campo de juego y dos especies de fichas para cada jugador. Se eligió desplegar el tablero a partir de un arreglo de arreglos vacíos de modo ilustrativo, para lo cual se crea una función que imprimiera matrices, de modo que se pudiera seguir empleando a lo largo del juego cada que se realice una movida. Durante el proceso surgió la duda de cómo los jugadores podrían indicar la ubicación que le querían dar a su ficha en el juego, y se concluyó que la mejor opción sería establecer coordenadas en la matriz, por lo que se modifica la función de imprimir como se observa en la figura 2 para que junto con el tablero se imprimieran los indicadores de coordenadas, cuyo resultado se puede apreciar en la figura 3 donde a través de un ciclo *for* se procedió a llenar la matriz con caracteres vacíos obteniendo de este modo el resultado esperado.

```
void imprimir_matriz(char b[3][3])
{
    printf(" a b c\n");
    for(int i=0;i<3;i++)
    {printf("%d", i);
    for(int j=0;j<3;j++)
    {printf("[%c]", b[i][j]);}
    printf("\n");
    }
}
```

Figura 2. Función para imprimir matriz

```
  a b c
0[ ][ ][ ]
1[ ][ ][ ]
2[ ][ ][ ]
```

Figura 3. Tablero de juego

El siguiente paso en el desarrollo del juego era el lograr recibir coordenadas que elija cada jugador, para esto se empleó el comando *scanf* de modo que recibiera un entero para seleccionar fila y un carácter que indicará la columna. Sin embargo, surgió el problema de que las casillas seleccionadas se podían sobrescribir, lo que impedía llevar a cabo el juego. Esto se solucionó al aplicar condicionales al código que solicitaba las coordenadas al jugador, y se determinó establecer esta solicitud como una función para emplear recursividad en la misma. Luego de programar el manejo de posibles errores a la hora de digitar las casillas la función resultó como la que se aprecia en la figura 4.

```

void casilla()
{ printf("Ingrese casilla %s\n", jugador);
scanf("%s", str1);
char str2[]={str1[0]};
char str3[]={str1[1]};
sscanf(str2, " %d", &fila);
sscanf(str3, " %c", &columna);
if((fila==0||fila==1||fila==2)&&(columna=='a'||columna=='b'||columna=='c'))
{if (mat[fila][letra(columna)] == ' ')
{mat[fila][letra(columna)] = movida;
system("cls");
imprimir_matriz(mat);}
else
{printf("\nCasilla ocupada, elija otra\n");
casilla();}
}
else
{printf("\nCaracter invalido\n");
casilla();}
}

```

Figura 4. Función que recibe las coordenadas del juego

El segundo elemento primordial del juego eran las fichas, y se establecieron como una variable de nombre “*movida*” asignada a cada jugador. Para el uno sería el carácter “x” y para el dos el “o”, como se observa en la figura 5 donde se aprecia la distribución antes mencionada dentro de un ciclo *while*, el cual se colocó con el objetivo de que el juego corriera de modo continuo hasta que uno de los jugadores ganara o se acabaran las casillas disponibles. Para esta tarea se decidió establecer los 16 casos posibles por los que se pueda ganar, como se observa en la figura 6, para que si alguno se cumpliese la variable *hay_ganador* se vuelva igual a 1 y el juego terminase por medio del comando *break*.

```

while(contadorG<9){
    if (contadorG % 2 == 0)
    {
        jugador = jug1;
        movida = 'x';
    }
    else
    {
        jugador = jug2;
        movida = 'o';
    }
    casilla();
}

```

Figura 5. Asignación de variable “movida”

```

    if (
//filas x
    (mat[0][0]=='x' && mat[0][1]=='x' && mat[0][2]=='x') || (mat[1][0]=='x' && mat[1][1]=='x' && mat[1][2]
    == 'x') || (mat[2][0]=='x' && mat[2][1]=='x' && mat[2][2]=='x')
//filas o
    || (mat[0][0]=='o' && mat[0][1]=='o' && mat[0][2]=='o') || (mat[1][0]=='o' && mat[1][1]=='o' && mat[1][2]
    == 'o') || (mat[2][0]=='o' && mat[2][1]=='o' && mat[2][2]=='o') ||
//columnas x
    (mat[0][0]=='x' && mat[1][0]=='x' && mat[2][0]=='x') || (mat[0][1]=='x' && mat[1][1]=='x' && mat[2][1]
    == 'x') || (mat[0][2]=='x' && mat[1][2]=='x' && mat[2][2]=='x') ||
//columnas o
    (mat[0][0]=='o' && mat[1][0]=='o' && mat[2][0]=='o') || (mat[0][1]=='o' && mat[1][1]=='o' && mat[2][1]
    == 'o') || (mat[0][2]=='o' && mat[1][2]=='o' && mat[2][2]=='o')
//diagonal °125 x
    || (mat[0][0]=='x' && mat[1][1]=='x' && mat[2][2]=='x')
// diagonal °45 x
    || (mat[0][2]=='x' && mat[1][1]=='x' && mat[2][0]=='x')
//diagonal °125 o
    || (mat[0][0]=='o' && mat[1][1]=='o' && mat[2][2]=='o')
// diagonal °45 o
    || (mat[0][2]=='o' && mat[1][1]=='o' && mat[2][0]=='o')
    )
    {
        printf("\nFin del juego, gana %s", jugador);
        hay_ganador=1; //alguien ganó
        break;
    }
    contadorG++;
}

```

Figura 6. Casos de gane posibles

Despliegue de las 10 mejores puntuaciones

Luego de finalizar el juego o al pedirlo en el menú principal, se imprime la lista de las 10 mejores puntuaciones, o sea, las menores cantidades de movimientos hechas para ganar, en orden de introducción. Si la puntuación puede reemplazar a una mayor (a la mayor de todas), entra a la lista, sino, no entra. En el caso de que todas las puntuaciones sean iguales, se reemplazará la puntuación más antigua. Estas se almacenan en tres archivos externos distintos: *puntuaciones.txt* para las puntuaciones, *archivo externo.txt* para los nombres de usuario y *extra.txt* para datos adicionales; las puntuaciones y nombres se almacenan en arrays que se analizan, modifican y posteriormente se imprimen en los archivos, actualizándolos.

Crear el repositorio en Github y definir el archivo README

Para esta sección, primeramente se procedió a crear cuentas respectivas en la plataforma. Posteriormente se crea un repositorio en Github de nombre “gato”, de carácter público y se determina para el mismo un archivo README en el cual se detalla el nombre de los integrantes del grupo, el del grupo, además del juego desarrollado e instrucciones que ayudan a cualquier persona a compilarlo.

Principales retos

1. Definir el “tablero” del juego

Este fue uno de los primeros retos enfrentados a la hora de desarrollar el juego, esto debido a que, en un inicio, se tenía la idea de emplear comandos o funciones que permitieran introducir valores en cualquier parte de la pantalla, de modo que no se tenía delimitada el área de juego. Luego de realizar varias búsquedas y conversar con el profesor, se encontró que esto era posible pero requiere un nivel más avanzado, por lo que se concluyó que lo mejor sería emplear una matriz vacía como tablero y que esta se fuese modificando durante el juego, sin embargo, al poner esta idea en práctica ocurre que los caracteres vacíos no eran reconocidos, como se observa en la figura 7. Luego de repasar el código gran cantidad de veces se le consultó al profesor por el fallo y resultó que era un error en la tipografía de las comillas usadas para expresar el carácter vacío, pues se usaron comillas dobles y debían ser simples.

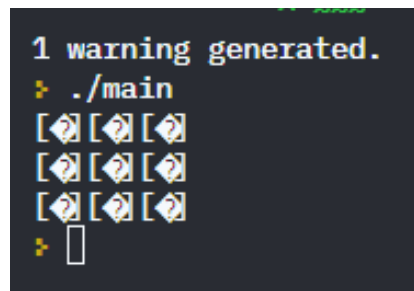


Figura 7. Caracteres no reconocidos

2. Incluir funciones en Replit

Para desarrollar el código en conjunto se empleó la plataforma Replit que permite a varios programadores trabajar con el mismo código a la vez, sin embargo, a la hora de querer integrar funciones desde un archivo de funciones auxiliares, Replit marcaba un error y no permitía correr el código. Este problema se generó específicamente cuando se intentó incluir una función que transformaba coordenadas de letras a números pero generaba el error “file not found with angled include use quotes instead”. Investigando durante las sesiones de laboratorio se determinó que en el Replit el llamado al archivo con las funciones auxiliares no se debía escribir con signos de mayor que, o menor que, (<>), si no con comillas (“ ”)

3. Casos de gane en tres en raya

Este fue uno de los principales retos enfrentados ya que, para hacer el código más concreto, se intentó realizar una forma automatizada que evaluara las casillas de la matriz de juego y de este modo identificar cuando un jugador lograba conseguir que tres de sus fichas se encontraran alineadas. Para llevar esto a cabo se tuvo la idea de comparar de tres en tres las casillas de la matriz por medio de un ciclo *while* e ir cambiando la posición a través de un iterador por medio de un ciclo *for*, sin embargo no funcionó. La idea se continuó trabajando pero resultaba muy complicada y no hacía el objetivo concreto, por lo que al final se decidió hacer la comparación de casillas, con sus coordenadas introducidas manualmente, de modo que se considerasen los 16 casos posibles en los que se podía ganar, como se observa en la figura 6.

4. Finalizar el juego

En un inicio no se tenía idea de cómo se iba a continuar con esta sección del programa, sin embargo de experiencia en cursos anteriores de programación, se recordó que la mejor manera para llevar la continuidad del juego era a partir de un ciclo *while*, de modo que se podía finalizar el juego deteniendo el ciclo. Esta propuesta fue la que se decidió desarrollar, por lo que se analizó cuántos turnos eran posibles para que los jugadores llenaran la matriz y así terminarían el juego. Se concluyó que eran nueve turnos máximo por lo que se definió un ciclo *while* que corriera mientras el *contador* (que representa la cantidad de veces que jugaba cada jugador) fuese menor a nueve. También se incluyeron los casos de gane donde lleguen a lograr los tres en línea en menos de nueve movimientos, para que si esto ocurría se le asignaba automáticamente al contador un valor de 10 para que dejase de correr el ciclo. Sin embargo esto no funcionó debido a que el ciclo evalúa la condición, no en el punto donde cambiase, sino que hasta que corriera la totalidad de su contenido. Luego de una búsqueda exhaustiva para solucionar el problema se encontró el comando *break*, que permite cortar el ciclo en cualquiera de sus puntos, por tanto, se empleó para lograr finalizar el juego en el punto en el que algún jugador lograra tres en línea.

5. Manejar coordenadas inválidas

A la hora de llevar a cabo el juego es muy posible que el jugador cometa errores al digitar las coordenadas de donde desea colocar su ficha, por lo que se decidió emplear un manejo de errores. Esta tarea fue realmente compleja ya que eran probables una gran cantidad de tipos de errores. Primeramente se consideró lo que pase si el jugador introduce una gran cantidad de caracteres, para lo que se planteó el uso de arreglos para evaluar la cantidad de datos introducidos, de modo que los valores que el jugador eligiera se almacenaran en *string*. Posteriormente se intentó implementar el comando *strlen()* para determinar la longitud del arreglo y de este modo controlar la cantidad de caracteres que se ingresaban, pero a la hora de intentar compilar el código este presentaba fallos, ya que el comando retornó un valor cero a pesar de que, probándolo en un archivo aparte, sí funcionaba, por lo que se decidió emplear condicionales como se observa en la figura 8, manteniendo la idea de usar una *string* y verificando que su primer y segundo valor fueran válidos. También se solucionó el error de digitar una casilla que no exista o seleccionar una ocupada, para lo que se continuó con el uso de condicionales, que permiten evaluar los posibles errores para cada turno, de modo que cada vez que se incurriera en una de estas faltas solicitara valores nuevos para las coordenadas correspondientes usando recursividad.

```
void casilla()
{ printf("Ingrese casilla %s\n", jugador);
  scanf("%s", str1);
  char str2[]={str1[0]};
  char str3[]={str1[1]};
  sscanf(str2, " %d", &fila);
  sscanf(str3, " %c", &columna);
  if((fila==0||fila==1||fila==2)&&(columna=='a' ||
  columna=='b' || columna=='c'))
  {if (mat[fila][letra(columna)] == ' ')
    {mat[fila][letra(columna)] = movida;
    system("cls");
    imprimir_matriz(mat);}
  else
  {printf("\nCasilla ocupada, elija otra\n");
   casilla();}
  }
  else
  {printf("\nCaracter invalido\n");
   casilla();}
}
```

Figura 8. Manejo de errores en la función *casilla()*

6. Limpiar la pantalla

Para llevar a cabo un juego más ordenado se propuso la idea de limpiar la pantalla e imprimir la nueva matriz cada turno, y que de este modo no resultase confuso para los jugadores ver tantos tableros en la pantalla. Para esta tarea se encontró el comando `system("cls")`, que en un inicio no funcionaba por que hizo falta importar su librería `stdlib.h`. Al notar este detalle se agregó por medio del comando `#include <stdlib.h>`. Una vez funcionando el comando, el siguiente desafío fue ubicarlo, ya que debía borrar el contenido anterior de la pantalla para luego imprimir una nueva matriz y que el siguiente jugador elija una nueva casilla. Cuando fue ubicado el comando se intentó correr el código, sin embargo no funcionaba; esto se debía a que la plataforma que se estaba empleando para compilar no era del todo compatible con el lenguaje. El comando `system("cls")` le indica a la consola que el usuario desea limpiar la pantalla, por lo que al no estar corriendo el código como un archivo externo desde una terminal no era posible ejecutar el comando. Usando la IDE Codeblocks, el comando funcionó perfectamente.

7. Almacenar las 10 mejores puntuaciones y los nombres de los jugadores

Este fue uno de los apartados más complejos. Lo que se intentó primero fue usar un archivo externo para guardar las últimas 10 puntuaciones junto con los usuarios de los jugadores, para lo que se recurrió a juntar ambos en una string con la función `sprintf()` y adjuntarla al archivo mediante `fopen()` en modo *append*, que generaría un total de 11 usuarios con su debida puntuación, hacer un array bidimensional con sólo los 10 últimos nombres del archivo, dejando fuera el primero de la lista, para luego reemplazar con `fopen()` en modo *write* y los valores del nuevo array todo el contenido de dicho archivo. Esto no funcionó porque, al leer bien las instrucciones del proyecto, nos dimos cuenta de que se buscan las 10 mejores puntuaciones, no las últimas, pero sirvió de base para lo que haríamos después.

Intentando hacer la menor cantidad de cambios posible, tratamos de obtener únicamente la puntuación escrita en dicho archivo mediante la función `strtok()`, pero resultaba muy complicado, por lo que se prefirió tener un archivo para las puntuaciones llamado *puntuaciones.txt* y otro para los nombres de usuario llamado *archivo externo.txt*.

El archivo de usuarios debe tener 10 líneas llenas de cualquier cosa porque la función *fscanf()* que se usa para leerlo no toma en cuenta las líneas vacías, lo que modifica el orden en el que se imprimen junto con la respectiva puntuación. En cambio, esto no afecta al archivo de puntuaciones, pues el array se llena con 0s si no encuentra nada, para posteriormente imprimir esos 0s en el archivo cuando se actualice.

Mediante las funciones *maxval()* y *minval()* se lograron las siguientes implementaciones, en orden de prioridad:

1. Reemplazar las puntuaciones iguales a 0, que representan espacios vacíos en los array, dado que no se han hecho 10 partidas todavía.
2. Reemplazar el valor más alto si no hay 0s y se es menor a éste.
3. No entrar al array (osea, no registrar la puntuación) si no se completa alguna de las dos funciones anteriores.

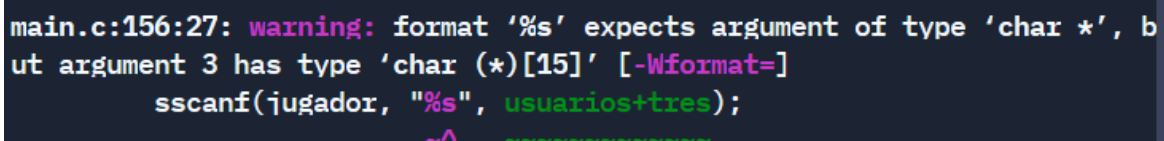
Se comprendió, posteriormente, que se necesitaba contemplar el caso en el que todas las puntuaciones fueran iguales, lo que puede pasar al sólo existir tres cantidades posibles de movimientos: 3, 4 y 5. Para esto, se decidió hacer otro archivo llamado *extra.txt* lleno de 9s, para que en el momento en el que el archivo de puntuaciones se llene sólo con determinado valor, empezar a reemplazar la puntuación más antigua introducida y se vaya actualizando el lugar donde ésta se encuentra conforme se introducen valores nuevos. Hay una variable designada a esto para cada una de las tres puntuaciones posibles (*tres*, *cuatro* y *cinco*), las cuales se actualizan leyendo *extra.txt* y luego lo modifican. *extra.txt* se llena con 9s porque, en teoría, si todo el arreglo está compuesto del mismo entero y estos se llenan de la posición 9 a la 0, los más antiguos por reemplazar serán los de la posición 9, por lo que se debe empezar por ahí. La cuarta cifra en este archivo es utilizada para llenar la variable *ultimo_valor*, que designa cuál fue la última casilla del array modificada y que fue incluida en la función *imprimir_todo()* para que esta muestre desde la puntuación más nueva a la más antigua, pues las introducidas primero están en las primeras casillas del arreglo, desde la 0 hasta la número:*ultimo_valor*, y las últimas desde la número: *ultimo_valor* hasta la casilla 9.

8. ¿Qué pasa si no gana nadie?

Al finalizar el código recordamos que hay casos en el juego del gato donde se llena el tablero sin completar tres en raya. Nos dimos a la tarea de lograr que, si esto sucede, se imprima que no ganó nadie, la puntuación del último jugador no se analice y los archivos externos no se modifiquen. En caso contrario de que si haya un ganador, la variable *hay_ganador* se hace 1 y se procede a analizar la nueva puntuación a ver si se incluirá dentro de las 10 mejores para imprimirlas, modificar los archivos externos y cerrar el programa. Hacer esto último fue simple, pues sólo se necesitó colocar la variable y el condicional en el lugar correcto; el reto fue darse cuenta de que la necesitamos poner, pues requirió de probar el juego varias veces con diferentes casos.

Conclusiones

- Para almacenar y registrar valores que se van a necesitar cada vez que se corra el programa, es bastante útil usar un archivo externo, pero, dependiendo del tipo de valor que se quiera escanear, las funciones para interactuar con estos servirán mejor o peor.
- Hay que tener conocimiento de si los argumentos de las funciones son arreglos o punteros, ya que a veces logran compilarse pero los resultados difieren de los esperados, especialmente si se trabaja con arrays bidimensionales donde es confuso usarlos y saber a qué refieren estos, lo que requiere mucho conocimiento previo o mucho ensayo y error.
- En varias de las funciones la consola nos genera la advertencia que se observa en la figura 9.



```
main.c:156:27: warning: format '%s' expects argument of type 'char *', but argument 3 has type 'char (*)[15]' [-Wformat=]
    sscanf(jugador, "%s", usuarios+tres);
                   ~^
```

Figura 9. Advertencia de la consola

Es un puntero al segundo nivel de un arreglo bidimensional, osea a cada conjunto, en este caso, strings completos. En teoría apunta a la dirección del primer valor de la string. No entendemos cuál es el problema ni cómo solucionarlo todavía, pero, a pesar de la advertencia, el programa corre perfectamente.

- Al usar *strlen()* en nuestro código siempre obtuvimos 0s aunque las *strings* fueran de sólo dos caracteres, lo que es extraño ya que, al probarla en otro archivo sí funcionaba y aparte podía retornar un *double*. No sabemos el por qué de este comportamiento.
- Es muy complicado organizar todos los criterios para decidir si una puntuación entrará en las 10 mejores. No siempre se imprimen de mayor a menor, pero siempre se imprimen las mejores, el cual era el requerimiento.

Bibliografía

NLed. (2017, 5 mayo). *Create a basic matrix in C (input by user !)*. stackoverflow.

Recuperado 10 de Noviembre de 2021, de
<https://stackoverflow.com/questions/2776397/create-a-basic-matrix-in-c-input-by-user>

C library function - sprintf(). (2021). tutorialspoint. Recuperado 10 de noviembre de 2021,

https://www.tutorialspoint.com/c_standard_library/c_function_sprintf.htm

C File Handling. (2020). programiz. Recuperado 17 de noviembre de 2021, de
<https://www.programiz.com/c-programming/c-file-input-output>

freeCodeCamp.org. (2021, 8 enero). *Pointers in C / C++ [Full Course]*.

[Video]. YouTube. <https://www.youtube.com/watch?v=zuegQmMdy8M>

Aviram Shiri. (2013, 30 noviembre). *The program doesn't stop on scanf("%c", &ch) line, why? [duplicate]*. stackoverflow. Recuperado 20 de noviembre de 2021, de

<https://stackoverflow.com/questions/20306659/the-program-doesnt-stop-on-scanf-ch-line-why>

scanf. (2021, 5 octubre). cppreference. Recuperado 10 de noviembre de 2021, de
<https://en.cppreference.com/w/c/io/fscanf>

meaning-matters (2013, 4 julio). *Use of external C++ headers in Objective-C*. stackoverflow. Recuperado 20 de noviembre de 2021, de

<https://stackoverflow.com/questions/17465902/use-of-external-c-headers-in-objective-c>