



TÓPICOS DE PROGRAMACIÓN



TIAGO PUJIA
UNLAM – INGENIERÍA INFORMÁTICA
2024

Introducción

Este libro fue escrito durante el segundo cuatrimestre del año 2024, en el marco de la cursada del profesor *Pablo Soligo* en la comisión 2600 [martes (2), turno tarde (6)].

Para cualquier información faltante, errores o correcciones, se agradece contactar a través del usuario "Tiago_nahuel_".

- **Clases Grabadas:**

https://youtube.com/playlist?list=PLENvh_JZMnA6n-wdPgmxpLHa4OBW7Qxxo&si=TGvaM4ajN0egZ17Q

https://youtu.be/8ZEgML4sg-4?si=nn6xRYXVh_5KDo9Q

- **Guía de Ejercicios Resuelta:**

https://drive.google.com/drive/folders/1VaKiO-yaJnpOC805JUw7vsVxD8PSyKh8?usp=drive_link

Índice

Unidad 0: Buenas Prácticas.....	5
CodeBlocks.....	5
Crear Librerías.....	7
Tipos de Datos.....	8
Operador Ternario.....	10
Macros.....	11
Vectores	14
Matrices.....	15
Proceso Interno.....	15
Recorridos Óptimos	16
Tamaño Máximo.....	20
Unidad 1: Punteros	21
Definición.....	21
Aplicación a Vectores y Matrices	22
Aplicación a Cadenas	24
Declaraciones de Cadena.....	24
Recorrido.....	25
Funciones de "string.h"	26
Aplicación a Estructuras	31
Unidad 2: Memoria Dinámica.....	32
Explicación Memoria de la CPU.....	32
Uso Aplicado de Memoria Dinámica.....	33
Re Asignar Memoria.....	34
Aplicación a Matrices	35

Unidad 3: Funciones	37
Argumentos al Main.....	37
Tipo de Dato Universal	38
Funciones 'mem'	40
Funciones 'Map', 'Filter' y 'Reduce'	44
Algoritmo de Ordenamiento por Selección.....	47
Algoritmo de Búsqueda Binaria	49
Unidad 4: TDA.....	51
Definición.....	51
Implementación Genérica.....	52
Unidad 5: Archivos	56
Tipos de Archivos para C	56
Manipulación General de Archivos (Repaso).....	58
Manipulación Archivos de Texto.....	60
Texto No Estructurado	60
Texto Estructurado Variable	62
Texto Estructurado Fijo	69
Funciones de Tipo de dato Universal	76
Manipulación Archivos Binarios	77
Algoritmo Fusión Merge.....	79
Unidad 6: Recursividad	84
Extra: Uso de Librería SDL	85

Unidad 0: Buenas Prácticas

CodeBlocks

- **Definición**

CodeBlocks es un IDE (Entorno de Desarrollo Integrado) con servicios especializados para C, su **utilización es obligatorio** debido a que los parciales se hacen por medio de este. Por medio de este podemos: editar código, depurar, compilar y ejecutar.

- **Instalación**

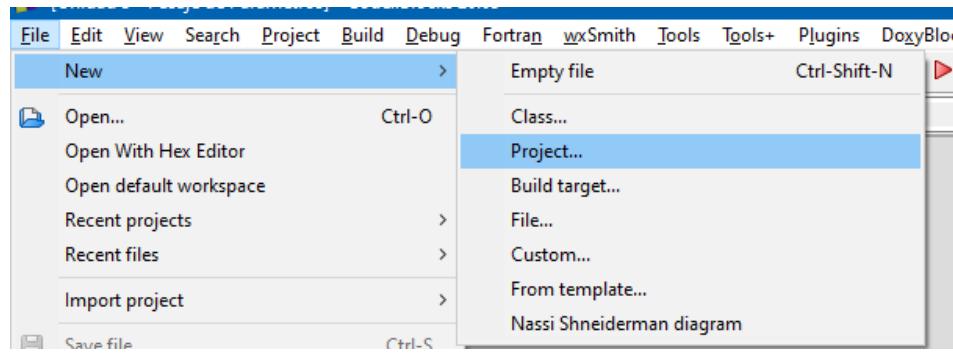
<https://www.codeblocks.org/downloads/binaries/>

Instalamos la versión que incluye el compilador mingw-32 bits (codeblocks-xx.xxmingw-32bit-setup.exe).

• Configuración

Este IDE se maneja mediante **proyectos**, esta es una estructura que organiza y gestiona los archivos, configuraciones y recursos. Para crear uno vamos a:

Barra de Herramientas > File > New > Project



Seleccionamos la categoría de **aplicación por consola**. Al momento de seleccionar la ruta hay que tener cuidado ya que debemos evitar:

- Caracteres raros (utilizar debajo de ASCII 128)
- Saltos
- Direcciones muy largas

En caso contrario no nos dará opciones como debugear. Tambien, debemos aceptar las opciones que se nos presenta (debug y release).

• Utilización

- Para compilar, ejecutar o ambos hacemos uso de:
- Para debugear:



• Crear Archivos

Para crear un archivo de cabecera o funciones vamos a:

Barra de Herramientas > File > New > File

Tendremos la opción de header y source (se utilizarán ambas). Continuamos en la opción y antes de crear aceptamos la opción de debug y release.

Crear Librerías

Es buena práctica hacer nuestras funciones lo más flexibles con tal de que sean re utilizadas y separarlos en distintos archivos (separados del archivo main).

Tambien es buena práctica crear nuestras propias librerías de C (como string.h) debido a que no todos los ordenadores soportan las librerías de C, exceptuando aquellas que van más allá de C puro como "scanf".

Para incluir un archivo C en el proyecto vamos a hacer uso de 2 tipos de archivos:

- **Funciones - nombre.c**

Incluimos las funciones que vamos a agregar al proyecto

- **Cabecera - nombre.h**

Incluimos los prototipos y más tarde el contenido del .c

Para incluir una librería de C hacemos uso de <> indicando la ruta del compilador. Para indicar la inclusión de un archivo en la ruta actual hacemos uso de las comillas dobles:

- **#include "ruta/nombre.h"**

- **#include <libreria.c>**

Como estamos haciendo uso de codeblocks, todos los archivos se encuentran en la misma carpeta, por lo tanto, tan solo debemos incluir el nombre.

Ejemplo:

Archivos de la Carpeta:

- main.c
- vectores.c
- vectores.h

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "vectores.h"
5 // Aca se imprime vectores.h
6
7 int main()
8 {
9     int vec[3] = {5,3,7};
10
11     mostrarVec(vec,3);
12 }
13
14 // Aca se imprime vectores.c
```

Tipos de Datos

El tipo de dato numérico (int) se le puede hacer ciertas modificaciones para modificar la longitud de bits o si esta con signo o no en su 1º bit. Las cláusulas se colocan detrás del int.

La longitud se puede definir con: **char, short int, int, long long int**. Y el signo se puede definir con **signed** y **unsigned**. Todas las combinaciones posibles:

- **Números Signados**

El primer bit representa el signo (positivo o negativo):

Tipo de Dato	Formato	Tamaño(bits)	Rango
signed char	%hhd	8	$[-2^7 ; 2^7 - 1]$
short int	%hd	16	$[-2^{15} ; 2^{15} - 1]$
int	%d	32	$[-2^{31} ; 2^{31} - 1]$
long long int	%lld	64	$[-2^{63} ; 2^{63} - 1]$

- **Números Sin Signar**

Toda la línea de bits representa el número:

Tipo de Dato	Formato	Tamaño(bits)	Rango
unsigned char	%hu	8	$[0 ; 2^8 - 1]$
unsigned short int	%hu	16	$[0 ; 2^{16} - 1]$
unsigned int	%u	32	$[0 ; 2^{32} - 1]$
size_t	%u	32	$[0 ; 2^{32} - 1]$
unsigned long long int	%llu	64	$[0 ; 2^{64} - 1]$

El tipo **size_t** proviene de la librería `<stdlib.h>` y es igual a **unsigned int** pero más abreviado, se utiliza para contar cantidad o longitudes. Aunque en vez de llamarlo por librería puede ser llamado por macros (tema próximo).

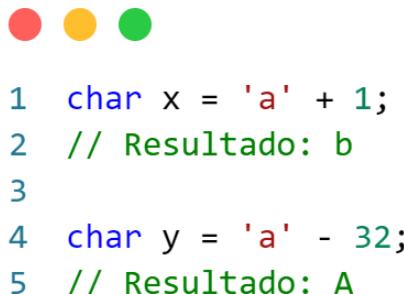
- **Números Flotantes**

Tipo de Dato	Formato	Tamaño(bits)	Rango
float	%f o %e o %g	32	[$\pm 3.4 \times 10^{38}$; $\pm 1.2 \times 10^{-38}$]
double	%f o %e o %g	64	[$\pm 1.7 \times 10^{308}$; $\pm 2.2 \times 10^{-308}$]
long double	%Lf o %Le o %Lg	96 - 152	

Mientras mejor categoría, más precisión de valor flotante. Long Double depende del S.O.

- **Uso del tipo Char**

Tipo de dato que ocupa 8 bits donde podemos representar valores de la tabla ASCII haciendo uso de las comillas simples. Como dentro contiene valores numéricos, podemos hacer operaciones aritméticas. Ejemplo:



```

1  char x = 'a' + 1;
2  // Resultado: b
3
4  char y = 'a' - 32;
5  // Resultado: A

```

- **Variable Constante**

Podemos hacer que una variable sea constante con el fin que no pueda ser modificada. Es recomendado para los parámetros de una función. Para esto tan solo agregamos la cláusula “**const**” al inicio del tipo de dato.

A pesar de tener tantas combinaciones de tipos de datos, la mejor opción es utilizar siempre el tipo int, size_t, char y float. Esto debido a que se encuentran más estandarizados y adaptados a funciones de librerías, evitando problemas futuros.

Operador Ternario

El operador ternario es un operador condicional que permite realizar una evaluación condicional de manera compacta, simplifica la expresión if-else en una sola línea (aunque se permiten los saltos).

```
condicion ? expresión_verdadera : expresion_falsa;
```

Para crear este operador, primero especificamos una **condicional**, seguido agregamos los signos "?" y ":"; el contenido que este adelante del signo '?' es lo que devolverá en caso que sea verdadera la condicional, y lo que este adelante del ':' es lo que devolverá en caso que sea falso. Finaliza con punto y coma.

Ejemplo:



```
1 int a = 4, b = 3;
2
3 int mayor = a > b ? a : b;
4
5 printf("Valor Mayor: %d",mayor);
6 // Resultado: Valor Mayor: 4
```

Macros

El compilador al ver un código escrito con hash # define/include, lo que hace es copiar y pegar el contenido/archivo, a nuestro archivo donde sé que lo está llamando. Las macros pueden ser de 2 tipos:

- **Macros Simples (constantes)**

Las macros simples se utilizan para definir valores constantes que pueden emplearse en todo el código. Estas constantes son reemplazadas por su valor literal en el código fuente durante la etapa de preprocesamiento, antes de que el compilador procese el programa.

```
#define NOMBRE_VALOR valor
```

Ejemplo:



```
1 #define size_t unsigned int
```

Es recomendable que las macros constantes, como mensajes de texto, longitudes o valores repetidos, se utilicen de manera consistente en todo el proyecto. Siendo estos macros guardados en un archivo como "**globales.h**" con el fin de centralizar todas las definiciones. Facilita mucho el mantenimiento y errores. Ejemplo:

```
10 // Longitudes
11 #define TAM_TEXTO 100
12 #define LONGITUD_VEC 5
13
14 // Mensajes
15 #define INGRESAR_NUMERO "Ingrese un Numero: "
16 #define INGRESAR_TEXTO "Ingrese Texto: "
17 #define INGRESAR_LETRA "Ingresar Letra: "
18 #define INGRESAR_BUSQUEDA "Ingresar Busqueda: "
19 #define INGRESAR_LONGITUD "Ingresar Longitud: "
20 #define INGRESAR_FLOAT "Ingresar Float: "
21 #define SEPARADOR_1 "-----\n"
```

- **Macros con Parámetros (similar a función)**

Estas macros pueden aceptar parámetros y se comportan como funciones en tiempo de preprocesamiento. El preprocesador simplemente reemplaza el uso de la macro por el código correspondiente, con los parámetros sustituidos por los valores dados.

```
#define NOMBRE_MACRO(parametro1, parametro2) (parametro1 + parametro2)
```

Hay que tener cuidado con el uso de parámetros, siempre debemos encapsular con paréntesis cada contenido que se le manda como parámetro. Ejemplo:

```
1 # define MAX(X,Y) (X) > (Y) ? (X) : (Y)
2
3 printf("El maximo es: %d",MAX(50,10));
4 // Resultado = El maximo es 50
```

- **Operador Hash (#)**

- Mensajes Dinámicos

Otro uso de las macros es crear mensajes dinámicos donde tan solo recibe valores y este modifica una cadena, sería una combinación de macros con parámetros y simple.

Para esto debemos escribir hacer uso del hash (#) en el parámetro en la cadena, pero debe estar rodeado de comillas dobles (estando fuera de la cadena). Ejemplo:

```
1 #define INGRESAR_NUMERO_EXCEPCION(N) "Valor(" #N " fin): "
2 // Ejemplo: INGRESAR_NUMERO_EXCEPCION(-1)
3 // Resultado = "Valor(-1 fin): "
```

- ReDefinir a Cadena

El operador hash(#) se puede utilizar tambien para convertir un argumento cualquier (real) a una cadena. Ejemplo: `#define STRINGIFY(x) #x`

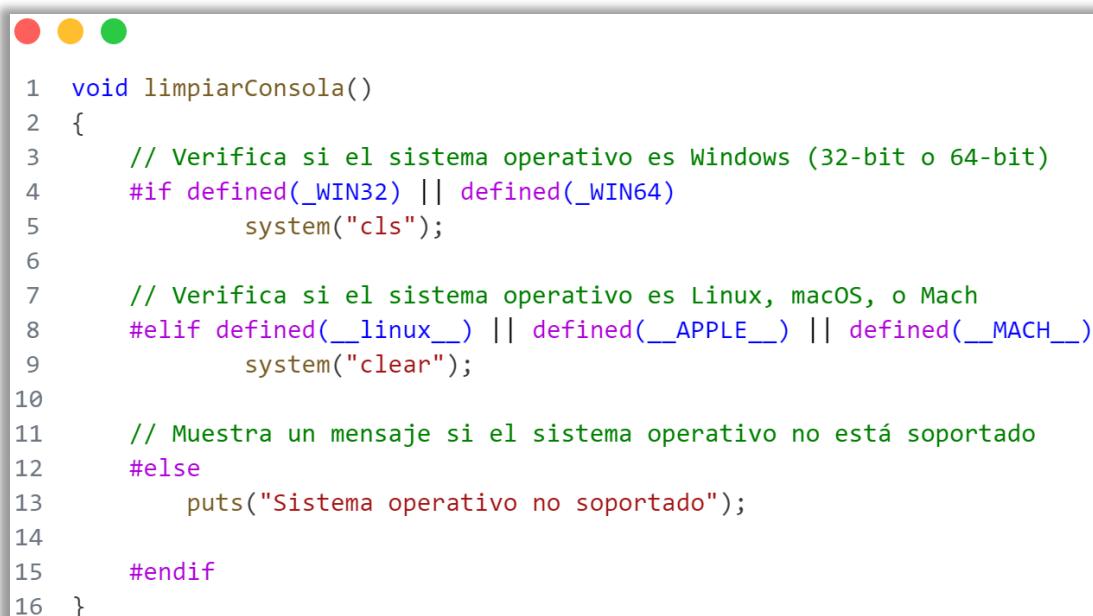
• Control Condicional

Se puede controlar la inclusión o exclusión de un macro con diferentes propósitos, como adaptarse a diferentes sistemas operativos, en depuración, entre otras... Hacemos uso de las cláusulas:

- **#ifdef**
Comprueba si una macro está definida (if defined).
- **#ifndef**
Comprueba si una macro no está definida (if not defined).
- **#if**
Evalúa una expresión constante.
- **#elif**
Se utiliza luego de ifdef, ifndef o if en caso que la primera fue falsa para probar con una nueva condicional.
- **#else**
Alternativa en caso que las anteriores condicionales fallaron.
- **#endif**
Se utiliza siempre para cerrar una estructura condicional.

Algunas utilidades:

Ejecutar comando en consola (en este caso limpiar consola) según el S.O:



```
1 void limpiarConsola()
2 {
3     // Verifica si el sistema operativo es Windows (32-bit o 64-bit)
4     #if defined(_WIN32) || defined(_WIN64)
5         system("cls");
6
7     // Verifica si el sistema operativo es Linux, macOS, o Mach
8     #elif defined(__linux__) || defined(__APPLE__) || defined(__MACH__)
9         system("clear");
10
11    // Muestra un mensaje si el sistema operativo no está soportado
12    #else
13        puts("Sistema operativo no soportado");
14
15    #endif
16 }
```

Vectores

• Administración del Espacio

Al momento de crear un array de N elementos, se reservan en memoria N cantidad de espacios multiplicado por el tamaño del tipo de dato:

$N * \text{sizeof}(\text{tipo dato})$

Ejemplo:



El tipo int ocupa 4 bytes. Por lo tanto, reservamos $5 * 4(\text{bytes}) = 20$ bytes.

```
1 int vector[5];
```

Es un tema meramente de comprensión que servirá para otros temas.

• Definir Tamaño del Vector

Es buena práctica definir el tamaño del array mediante constantes define, definiendo el tamaño máximo.

Tambien, mediante otra variable definir la longitud de espacios que se están utilizando en el array. Este va a ir cambiando en el programa según se elimina o inserta. Se Esta técnica se utiliza siempre para definir un vector.



```
1 #define TAM 50
2
3 int vector[TAM] = {4,3,6};
4 size_t cantEl = 3;
5
6 mostrarVec(vec,cantEl);
```

• Retornar Tamaño

Podemos exactamente la longitud de índices que ocupa un array con la siguiente formula:



```
1 sizeof(vector)/sizeof(int);
```

sizeof retorna tamaño, en este caso el tamaño del vector ($5 * 4 = 20$) y del tipo int(4 bytes). Por lo que ya no es necesario pasar a una función el tamaño del array.

Matrices

Proceso Interno

Para una mejor comprensión de cómo funcionan las matrices NxN, es importante entender que cada matriz se almacena en memoria de manera continua. Los elementos de cada fila se colocan uno tras otro, formando una secuencia lineal.

00	01	02	Array 0
10	11	12	Array 1
20	21	22	Array 2



Esta comprensión ayudara más adelante para el tema de punteros y memoria dinamica.

Recorridos Óptimos

La optimización de la manera de recorrer las matrices es muy importante.

- **Recorrido Triangular**

00	01	02
10	11	12
20	21	22

Recorremos los elementos que están por debajo de la diagonal principal.



```
1 size_t i, j;
2
3 // Recorrido de Matriz Triangular
4 for(i = 1 ; i < f ; i++)
5     for(j = 0 ; j < i ; j++)
6         printf("%d ",matriz[i][j]);
7
8 // Recorrido de Matriz Triangular + Diagonal Principal
9 for(i = 0 ; i < tamano ; i++)
10    for(j = 0 ; j <= i ; j++)
11        printf("%d ",matriz[i][j]);
```

- Recorrido Triangular Espejado

00	01	02
10	11	12
20	21	22

Podemos utilizar una técnica llamada *Matriz Espejada*, en la que podemos acceder a la casilla espejo de la posición en la que nos encontramos para diferentes usos. Solo necesitamos intercambiar i (F) y j (C) al recorrer la matriz de manera triangular.



```
1 for(i = 1 ; i < f ; i++)
2     for(j = 0 ; j < i ; j++){
3         printf("Valor: %d" , matriz[i][j] );
4         printf("Espejado: %d" , matriz[j][i] );
5     }
```

Valor: 10	Espejado: 01
Valor: 20	Espejado: 02
Valor: 21	Espejado: 12

- **Saber si es Matriz Identidad**

1	0	0
0	1	0
0	0	1

La matriz identidad es aquella donde solo tiene 0 en todas las casillas y 1 en la diagonal principal.



```
1 size_t i, j;
2
3 // Chequear Diagonal Principal
4 for(i = 0 ; i < f ; i++)
5     if(1 != matriz[i][i])
6         return 0;
7
8 // Chequear Triangulos con Espejado
9 for(i = 1 ; i < f ; i++)
10    for(j = 0 ; j < i ; j++)
11        if(0 != matriz[i][j] || 0 != matriz[j][i])
12            return 0;
13
14 return 1;
```

- **Transponer Matriz**

Transponer una matriz es el proceso de intercambiar filas por columnas. Esto significa que el elemento que está en la posición (i, j) pasa a la posición (j, i) .

Ejemplo:

A	B	C		A	D	G
D	E	F		B	E	H
G	H	I		C	F	I

La optimización para transponer una matriz cuadrada consiste en aprovechar el recorrido triangular, es decir, solo intercambiar los elementos que están por fuera de la diagonal principal (ya que la diagonal principal permanece intacta). Hacemos uso del espejado de matriz junto una variable auxiliar al momento de realizar el intercambio.

```
1 void transponerMatriz(char matriz[][COL_MAX], int tamanio) {  
2     size_t i, j;  
3     char aux;  
4  
5     // Recorre la matriz de manera triangular  
6     for(i = 1 ; i < tamanio ; i++)  
7         for(j = 0 ; j < i ; j++)  
8         {  
9             // Intercambia los elementos de forma  
10            // espejada con una variable auxiliar  
11            aux = matriz[i][j];  
12            matriz[i][j] = matriz[j][i];  
13            matriz[j][j] = aux;  
14        }  
15 }
```

Tamaño MÁximo

Podemos eliminar la predefinición de tamaños de matrices con constantes para hacerlo de uso global en bibliotecas. Ejemplo de error:

```
● ● ●  
1 #define TAM_COL 4  
2 #define TAM_FIL 4  
3  
4 void mostrarMatriz(int matriz[][][TAM_COL],int f, c);
```

Una opción posible es poner una constante de tamaño máximo con un número muy grande. Teniendo la desventaja que reservamos memoria que no se va a utilizar, pero es lo que se utiliza (por ahora) hasta la unidad de memoria dinámica. Ejemplo:

```
● ● ●  
1 #define MAX_COL 50  
2 #define MAX_FIL 50  
3  
4 // Una matriz identidad es siempre cuadrada, tan solo le pasamos un parametro  
5 void es_identidad(int matriz[][][MAX_COL],int tamanio);  
6  
7 int main()  
8 {  
9     int resultado;  
10    int matriz[MAX_FIL][MAX_COL] = {  
11        {1,0,0},  
12        {0,1,0},  
13        {0,0,1}  
14    }  
15  
16    resultado = es_identidad(matriz,3);  
17    printf("Identidad: %d",resultado);  
18 }
```

Unidad 1: Punteros

Definición

Un puntero es una variable cuyo valor es una **dirección de memoria** en lugar de un valor de dato, apunta a la ubicación en la memoria donde se almacena un valor. Los punteros pueden apuntar a variables de diferentes tipos: int, char, float, double, estructuras o incluso otros punteros. Los vectores ya de por sí cuando son llamados, entregan siempre el inicio de la dirección de memoria.

Para definir un puntero, se antepone el operador "*" al tipo de dato que se espera que apunte. Para obtener la dirección de memoria de una variable, se utiliza el operador "&". Para acceder al valor almacenado en la dirección apuntada por el puntero se utiliza el operador "*".

Ejemplo 1:

```
● ● ●
1 char nombre = 'T';
2 int *direccionNombre = &nombre;
3
4 printf("Direccion de Memoria: %u\n",direccionNombre);
5 printf("Valor de la dirección: %c\n", *direccionNombre)
```

Direccion de Memoria: 12345678
Valor de la dirección: T

Ejemplo 2:

```
● ● ●
1 void miFuncion(int *valor)
2 {
3     *valor = 'U'; // Modificamos el valor original
4 }
5
6 int main()
7 {
8     char valor = 'R';
9
10    miFuncion(&valor); // Pasamos la dirección como argumento
11    printf("Valor: %c\n",valor);
12 }
```

Valor: U

Aplicación a Vectores y Matrices

• Vectores

Debemos hacer el uso aritmético de punteros en arrays, y evitar utilizar subíndices, esto debido a que es más eficiente y obligatorio.

Para esto tenemos varias maneras, entre ellas: indicamos con el puntero la posición final, y el vector original lo vamos incrementando en el bucle con 'vec++' hasta la posición final. Ejemplo:

```
1 // En este caso todos los incrementos son x4 (tamaño bytes int)
2 void mostrarVecInt(int *vec, size_t longitud)
3 {
4     int *fin = vec + longitud; // Definimos fin del vec
5
6     while(vec < fin) // Comparamos las direcciones de memoria
7     {
8         printf("%d\n", *vec); // Imprimimos valor de la posicion
9         vec++; // Incrementamos puntero a la siguiente posicion
10    }
11 }
```

El compilador comprende que debe incrementar sumar 4 bytes al puntero debido a que es tipo int, aplicaría lo mismo para el resto de tipos de datos. Por lo que no es necesario hacer uso de una línea "vec += 4", pero serían equivalencias. Al igual que el compilador también comprende que para definir la variable "fin" debe multiplicar por 4 bytes la variable longitud.

Otra manera de recorrer es mediante el uso del ciclo for. Ejemplo:

```
1 void mostrarVecInt(int *vec, size_t longitud)
2 {
3     // Declaramos el final en memoria
4     int *fin = vec + longitud;
5
6     for(vec = vec ; vec < fin ; vec++)
7     {
8         printf("%d ", *vec);
9     }
10 }
```

Ejemplo 2: Ejercicio 1.1

```
1 // Función para insertar un elemento en un vector int en una posición específica
2 void insVec(int *vec, size_t longitud, size_t *cantEl, int ingreso, size_t posicion)
3 {
4     // Verifica si la posición de inserción es válida
5     if (posicion > *cantEl)
6         // Si la posición es mayor que la cantidad de elementos, no se hace nada
7         return;
8
9     // Puntero al final del vector actual
10    int *finVec = vec + *cantEl;
11    // Puntero a la posición donde se insertará el nuevo elemento
12    int *posVec = vec + posicion;
13
14    // Desplaza los elementos a la derecha para hacer espacio para el nuevo elemento
15    while (finVec > posVec) // Mientras el final del vector esté después de la posición
16    {
17        *finVec = *(finVec - 1); // Copia el elemento anterior a la posición actual
18        finVec--; // Desplaza el puntero hacia la izquierda
19    }
20
21    // Inserta el nuevo elemento en la posición deseada
22    *posVec = ingreso;
23
24    // Incrementa la cantidad de elementos si hay espacio en el vector
25    if (*cantEl < longitud)
26        (*cantEl)++;
27 }
```

• Matrices

Con la aplicación de matrices para acceder a un elemento de un array de debe utilizar un doble puntero al estilo ***(*(matriz+F)+C)** donde con F indicamos la fila y C la columna (aunque no es obligatorio para matrices). Ejemplo:

```
1 int matriz[3][3] = {
2     {00,01,02},
3     {10,11,12},
4     {20,21,22}
5 };
6
7 printf("%d", *(*(matriz+1)+2));
8 // Resultado: 12
```

Aplicación a Cadenas

Declaraciones de Cadena

Tipo de dato compuesto por un array del tipo char, donde al final termina con el carácter nulo '\0', este indica cuando termina la cadena.

- **Declaración Burocrática**

Creamos un array formal e indicamos posición por posición el valor que tendrá la cadena en esa posición indicando el carácter nulo:



```
1 char cad[5] = {'H', 'o', 'l', 'a', '\0',}
```

- **Declaración Clásica**

Indicamos el texto mediante comillas dobles con el tamaño de la cadena y el carácter nulo se asignan solo:



```
1 char cad[] = "Hola";
```

- **Declaración con Puntero**

Se puede hacer el uso de punteros, pero no es recomendado. Porque crea la cadena de manera constante (no puede ser modificada en ningún sentido), a no ser que comencemos a operar como un puntero. Ejemplo:



```
1 char *cad = "Hola";
```

Recorrido

El recorrido de una cadena se debe utilizar la lógica de punteros como un array (sigue siendo igual de eficiente), solo que en una cadena es más sencillo debido a que ya sabemos que finaliza con '\0'.

Para recorrer una cadena de principio a fin ('\0') tan solo hacemos uso de un while donde verificamos si es correcto el valor donde se encuentra posicionado el array, la única manera que no lo sea es si el carácter nulo que en ASCII es 0. Luego, dentro del bucle vamos aumentando la posición del puntero.

Ejemplo:

Retorna la longitud de la cadena excluyendo el carácter nulo

```
1 size_t strlen(char* str)
2 {
3     // Guarda la dirección inicial del string en el puntero 'ini'.
4     const char* ini = str;
5
6     // Recorre el string hasta llegar al carácter nulo '\0'.
7     while(*str)
8         str++;
9
10    // Calcula la longitud restando la dirección inicial de la final.
11    return str-ini;
12 }
```

{'H', 'o', 'l', 'a', '\0'}
strlen: 4

Funciones de "string.h"

Las funciones de las librerías tendrán que ser creadas a mano:

- **strcpy**

Realiza una copia del contenido de un string a otro string:

```
● ● ●
1 char strcpy(char *destino, char *origen)
2 {
3     char *ini = destino; // Debemos retornar la dirección inicial de destino
4
5     // Copia cada carácter de 'origen' a 'destino' mientras no sea el carácter nulo
6     while(*origen)
7     {
8         *destino = *origen; // Asigna el carácter actual de 'origen' a 'destino'
9         destino++;          // Avanza el puntero 'destino'
10        origen++;          // Avanza el puntero 'origen'
11    }
12    *destino = '\0'; // Añade el carácter nulo al final de la cadena copiada
13
14    return ini;
15 }
```

- **strncpy**

Se utiliza para copiar hasta "n" caracteres de una cadena a otra.

```
char c1[TAM] = "May S", c2[TAM];
strncpy(c2,c1,3);
printf("%s",c2);
// Resultado = May
```

```
● ● ●
1 char strncpy(char *destino, char *origen, size_t bytes)
2 {
3     char *ini = destino; // Debemos retornar la dirección inicial de destino
4
5     // Copia cada carácter de 'origen' a 'destino' mientras no sea el carácter nulo
6     while(*origen && bytes)
7     {
8         *destino = *origen; // Asigna el carácter actual de 'origen' a 'destino'
9         destino++;          // Avanza el puntero 'destino'
10        origen++;          // Avanza el puntero 'origen'
11        bytes--;           // Decrementa el contador de bytes a copiar
12    }
13    *destino = '\0'; // Añade el carácter nulo al final de la cadena copiada
14
15    return ini;
16 }
```

• strcmp

Compara 2 cadenas y devuelve el resultado, estos pueden ser:

- str1 = str2 => 0 (cero)
- str1 > str2 => Numero Positivo
- str1 < str2 => Numero Negativo



```
1 int strcmp(const char *str1, const char *str2)
2 {
3     // Compara ambas cadenas mientras sean iguales y no lleguen al final.
4     while(*str1 == *str2 && *str1 && *str2)
5     {
6         str1++; // Avanza en la cadena 1.
7         str2++; // Avanza en la cadena 2.
8     }
9
10    // Retorna la diferencia entre los primeros caracteres diferentes.
11    return *str1 - *str2;
12 }
```

```
char c1 = "Alo A",
      c2 = "Alo B";
strcmp(c1,c2);
// Resultado = A(65) - B(66) = -1
```

• strncmp

Compara los primeros N bytes de 2 cadenas y devuelve el mismo resultado



```
1 int strncmp(const char* str1, const char* str2, size_t bytes)
2 {
3     // Mientras ambas cadenas no lleguen al final y queden bytes por comparar
4     while(*str1 && *str2 && bytes)
5     {
6         // Si los caracteres actuales no coinciden, retorna la diferencia
7         if(*str1 != *str2)
8             return *str1 - *str2;
9
10        str1++; // Avanza al siguiente carácter
11        str2++; // Avanza al siguiente carácter
12        bytes--; // Decrementa el número de bytes a comparar
13    }
14
15    // Si una cadena termina antes o no se agotaron los bytes, retorna la diferencia
16    return *str1 - *str2;
17 }
```

```
char c1 = "qwerty",
      c2 = "qwerth";
printf("%d",strncmp(c2,c2,4));
// Resultado = 0
// q w e r t
// 0 1 2 3 4
```

• **strchr**

Se utiliza para buscar la primera aparición de un carácter:

```
● ● ●  
1 int strchr(char *str, const char buscar)  
2 {  
3     const char *ini = str; // Guarda el puntero inicial de la cadena.  
4     while (*str) { // Itera mientras no se alcance el carácter nulo.  
5         if (*str == buscar) // Si el carácter actual es igual a 'buscar'.  
6             return str - ini; // Retorna la posición relativa en la cadena.  
7         str++; // Avanza el puntero a la siguiente posición.  
8     }  
9     return -1; // Si no se encuentra el carácter, retorna -1.  
10 }
```

```
char c[] = "Tiago";  
strchr(c, 'a');  
// Resultado = 2
```

Se le puede hacer una modificación y es que te devuelve la dirección de memoria dentro de la cadena si lo encuentra o el carácter nulo en caso contrario, en vez del índice de la posición.

• **strrchr**

Se utiliza para buscar la última aparición de un carácter:

```
● ● ●  
1 int strrchr(char *str, const char buscar)  
2 {  
3     int posicion = -1; // Inicializa la posición como -1  
4     const char *ini = str; // Guarda la dirección inicial de la cadena  
5  
6     // Recorre la cadena hasta que se llega al final ('\0')  
7     while (*str) {  
8         // Si el carácter actual es el buscado, actualiza la posición  
9         if (*str == buscar)  
10            posicion = str - ini; // Calcula la posición actual del carácter buscado  
11  
12            str++; // Avanza al siguiente carácter de la cadena  
13    }  
14  
15    // Retorna la última posición encontrada, o -1 si no se encontró  
16    return posicion;  
17 }
```

- **strcat**

Concatena/Añade una cadena detrás de otra, uniéndolas:



```
1 char* strcat(char *destino, char *origen)
2 {
3     // Guarda la dirección inicial del destino para retornarla al final.
4     char *ini = destino;
5
6     // Avanza el puntero 'destino' hasta el final de la cadena actual.
7     while(*destino)
8         destino++;
9
10    // Copia los caracteres de 'origen' a 'destino' hasta encontrar el carácter nulo.
11    while(*origen)
12    {
13        *destino = *origen; // Asigna el carácter actual de 'origen' a 'destino'.
14        destino++;        // Avanza el puntero 'destino'.
15        origen++;        // Avanza el puntero 'origen'.
16    }
17
18    // Añade el carácter nulo al final de la nueva cadena concatenada.
19    *destino = '\0';
20
21    // Retorna el puntero inicial de 'destino'.
22    return ini;
23 }
```

```
char c1 = "Hola",
      c2 = "Mundo",
      c3 = strcat(c1,c2);
// Resultado: Hola Mundo
```

- **strstr**

Busca la primera aparición de una subcadena en una cadena.

```
● ● ●

1 char* strstr(const char* cadena, const char* subcadena)
2 {
3     // Si la subcadena es vacía, se devuelve la cadena original
4     if (!*subcadena)
5         return cadena;
6
7     const char *c, *s;
8
9     // Recorre la cadena principal hasta el final
10    while (*cadena)
11    {
12        c = cadena;      // Puntero auxiliar para recorrer la cadena principal
13        s = subcadena; // Puntero auxiliar para recorrer la subcadena
14
15        // Mientras los caracteres de ambas cadenas coincidan y no se llegue al final
16        while (*c && *s && *c == *s)
17        {
18            c++;
19            s++;
20        }
21
22        // Si se recorrió toda la subcadena, significa que se encontró la subcadena
23        if (!*s)
24            // Retorna la posición de la primera coincidencia en la cadena principal
25            return cadena;
26
27        cadena++; // Avanza al siguiente carácter en la cadena principal
28    }
29
30    // Si no se encontró la subcadena, retorna NULL
31    return NULL;
32 }
```

Aplicación a Estructuras

- **Tamaño de la Estructura**

Las estructuras se siguen declarando de la misma manera, recordando que se debe tener cuidado al momento de contar a mano la cantidad de bytes que ocupa, debido que al final de cada estructura el S.O agrega un dato. Ejemplo:

```
● ● ●

1 #define TAM 50
2
3 typedef
4 {
5     int legajo;
6     char nombre[TAM];
7     float sueldo;
8 } EMPLEADO;
9
10 // sizeof(EMPLEADO) != 4(int) + 50(char[50]) + 4(float) = 58 bytes
```

- **Aplicación de Punteros**

Se le puede pasar como argumento a una función la dirección de memoria de la estructura, con el fin de no duplicar el dato que se utilizara y ahorrar recursos.

La manipulación del mismo con punteros es el mismo ya conocido, la única diferencia es que para acceder a una propiedad mediante la dirección de memoria se utiliza la sintaxis de flecha `->` y no la del punto. Ejemplo:

```
● ● ●

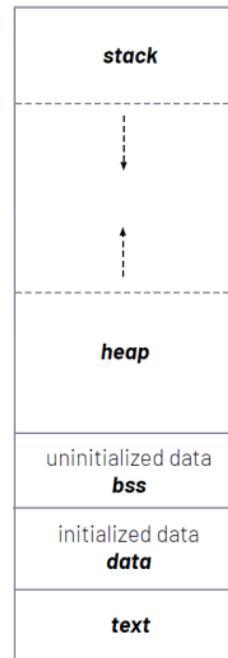
1 int aumentarSueldo(EMPLEADO *emp)
2 {
3     // Acceso a un dato con flecha
4     float incremento = (emp->sueldo) * 0.20;
5
6     // Modificar el valor de dirección
7     emp->sueldo += incremento;
8 }
```

Unidad 2: Memoria Dinámica

Explicación Memoria de la CPU

Dentro de la memoria de la CPU, se encuentra dividida por varias secciones:

1. **text** - Almacena las instrucciones en lenguaje ensamblador (Assembly).
2. **data** - Almacena datos inicializados; variables globales de Assembly (aca están las macros)
3. **bss** - Almacena datos no inicializados; variables globales al inicio del programa
4. **heap** (memoria dinámica) - Asignación de memoria en tiempos de ejecución
5. **stack** (pila) - Almacena variables locales y en el contexto de las funciones.



- **Gestión de Memoria**

En C, las variables se almacenan en la pila (stack), mientras que el contenido que necesitan manejar se almacena en la memoria dinámica (heap). La pila gestiona variables automáticamente, mientras que la memoria dinámica debe ser gestionada por el programador mediante funciones como malloc, realloc, y free.

- **Ventajas**

La memoria dinámica ofrece flexibilidad al permitir la asignación y liberación de memoria durante la ejecución. Esto evita las limitaciones de tamaño fijo en estructuras como matrices, permitiendo definirlo en tiempos de ejecución. Optimiza el uso de recursos del sistema, ya que la memoria utilizada por variables o funciones es liberada cuando ya no es necesaria y porque el acceso a datos es mucho más rápido en memoria dinámica.

Resumen, es recomendado para el uso de arrays y para variables de uso constante.

Uso Aplicado de Memoria Dinámica

```
void* malloc(size_t tamaño_en_bytes);
```

Pasos para hacer uso de Memoria Dinámica:

1. Creamos un puntero que almacenará la dirección de la memoria reservada dinámicamente.
2. Utilizamos la función **malloc(tamaño_bytes)** para asignar memoria en la sección de memoria dinámica (heap). Esta función devuelve la dirección de la memoria asignada. El tamaño a reservar se especifica en bytes, usando sizeof(tipo). Es fundamental verificar que malloc devuelva una dirección válida (es decir, que no sea NULL), lo que indicaría que la reserva fue exitosa.
3. Con la dirección ya asignada, podemos almacenar datos en la memoria dinámica utilizando la sintaxis del puntero ***puntero = valor**; Esto coloca los valores en la memoria reservada.
4. Cuando ya no necesitamos la memoria reservada, liberamos su espacio usando **free(puntero)**. Este paso es crucial para evitar fugas de memoria, ya que el sistema no libera automáticamente la memoria asignada dinámicamente.

Ejemplo:

```
1 int *pi; // Creamos nuestra variable tipo puntero
2 pi = malloc(sizeof(int)); // Apuntamos hacia una dirección heap
3
4 // Chequear que retorno una dirección de memoria
5 if(pi != NULL)
6 {
7     *pi = 7;
8     // ... Hacemos lo que queramos con ella
9     printf("Valor asignado %d en la dirección %u", *pi, pi);
10 }
11
12 free(pi); // Liberamos la memoria
```

Si queremos almacenar un vector por ejemplo tan solo debemos multiplicar el valor de sizeof por la cantidad de espacios que queremos. Luego, la manipulación (lectura y escritura) es la misma que la lógica de punteros.

Re Asignar Memoria

```
void* realloc(void* dir, size_t nuevo_tamaño_en_bytes);
```

Se utiliza para redimensionar el tamaño de un bloque de memoria previamente asignado por malloc sin perder memoria. Recibe como primer argumento el puntero de memoria ya asignado y segundo el espacio nuevo que puede ser mayor o menor que la original:



```
1 // Reserva inicial de memoria
2 int *arr = (int*) malloc(5 * sizeof(int));
3
4 // Asignar valores
5 for (int i = 0; i < 5; i++)
6     arr[i] = i + 1; // 1, 2, 3, 4, 5
7
8 // Aumentar el tamaño a 10
9 arr = (int*) realloc(arr, 10 * sizeof(int));
10
11 // Inicializar nuevos valores
12 for (int i = 5; i < 10; i++)
13     arr[i] = i + 1; // 6, 7, 8, 9, 10
14
15 // Imprimir valores
16 for (int i = 0; i < 10; i++)
17     printf("%d ", arr[i]);
18
19 // Liberar memoria
20 free(arr);
```

Hay que tener cuidado al momento de utilizar esta función porque su costo de recursos de computadora puede ser muy elevado.

Aplicación a Matrices

Haciendo uso de este concepto, podemos crear matrices de tamaño dinámico, permitiendo funciones más flexibles sin depender de técnicas que limitan el tamaño máximo de columnas y filas.

A cada función que manipula la matriz debemos pasarle un puntero a un puntero, por ejemplo: **int **matriz**, indicando que se trata de una matriz.

- **Liberar Memoria**

Para liberar la memoria, recorremos cada fila y la liberamos una por una. Luego de liberar cada fila, liberamos el arreglo de punteros que apuntan a las filas.



```
1 void liberarMatriz(int** matriz, const int filas) {  
2     for (int i = 0; i < filas; i++) // Liberar cada fila  
3         free(*(matriz+i));  
4     free(matriz); // Liberar el arreglo de punteros  
5 }
```

• Crear Matriz

Usamos malloc inicialmente para crear la matriz, reservando espacio para el arreglo de punteros a filas (int*). Luego, recorremos cada fila y reservamos espacio en memoria según la cantidad de columnas. Es importante siempre verificar si hay errores durante la asignación de memoria.



```
1 int **crearMatrizInt(size_t filas, size_t columnas)
2 {
3     size_t i, j;
4     int **matriz = (int**) malloc(filas * sizeof(int*));
5
6     // Verificar si se retorno la dirección
7     if(matriz == NULL)
8         return NULL;
9
10    // asignamos memoria para cada fila (columnas)
11    for(i = 0; i < filas ; i++)
12    {
13        *(matriz+i) = (int*) malloc(columnas * sizeof(int));
14
15        // Si ocurrio un error, liberamos toda la memoria
16        if(*(matriz+i) == NULL)
17        {
18            liberarMatriz(matriz,i);
19            return NULL;
20        }
21    }
22
23    return matriz;
24 }
```

La ventaja de usar este método es que podemos definir un tamaño diferente de columnas para cada fila, creando matrices irregulares si es necesario.

Unidad 3: Funciones

Argumentos al Main

Los argumentos del main en C permiten que un programa reciba entradas desde la línea de comandos cuando se ejecuta. Estos argumentos se pasan a la función main a través de dos parámetros:

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

- **int argc** – “Argument Count”

Entero que indica la cantidad de argumentos que se pasaron a la función main. Incluye el nombre del programa, por lo que siempre es al menos 1

- **char* argv[]** – “Argument Vector”

Array puntero de cadena. Cada elemento de este array es uno de los argumentos pasados a la línea de comandos. El primer elemento (argv[0]) es siempre la ruta del programa.

Es un uso que se le da únicamente a los programadores debido que tiene la característica que se le puede ingresar datos como un scanf sin necesidad de llamarlo, ahorrando varias líneas de código y por lo tanto tiempo.

Para hacer uso de esta característica abrimos el CMD, copiamos la ruta del archivo y ejecutamos: “*ruta archivo*” *Argumentos*

Ejemplo:

Ejecutamos: “*ruta archivo*” Hola Mundo

```
1 int main(int argc, char* argv[])
2 {
3     size_t i;
4
5     printf("Número de argumentos: %d\n", argc);
6     for(i = 0; i < argc; i++)
7         printf("Argumento [%d]: %s\n", i, argv[i]);
8
9     return 0;
10 }
```

```
Resultado:
Argumento [0]: prueba.c
Argumento [1]: Hola
Argumento [2]: Mundo
```

Tipo de Dato Universal

Para esto hacemos uso del tipo de dato puntero genérico **void*** para los parámetros y retorno, permitiendo pasar diversos datos sin preocuparse por el tipo. Debemos tener en cuenta los siguientes detalles para todas las funciones:

- El incremento del puntero debemos hacerlo a mano (el compilador no va a entenderlo). Ósea: `vec += tamano`
- Debemos especificar la longitud y tamaño del tipo de dato que utiliza
- La dirección final de vector la calculamos a mano como tambien debe ser tipo `void*`

Ejemplo:

```
1 void mostrarVector(void *vec, size_t longitud, size_t tamano)
2 {
3     void *fin = vec + longitud * tamano;
4
5     while(vec < fin)
6     {
7         printf("%d "); // ERROR
8         vec += tamano;
9     }
10}
11
12 int main()
13 {
14     int vec[] = {0,1,2,3};
15     size_t tamano = sizeof(*vec), longitud = sizeof(vec) / tamano;
16
17     mostrarVector(vec,longitud,tamano);
18
19     return 0;
20 }
```

El problema que tiene esta función es ¿Cómo especificamos el formato para mostrar? Para solución esto se le dará a la función otra función como argumento que contendrá el `printf` con su formato. En otras funciones van a requerir otra acción.

Para pasar como argumento debe cumplir los mismos valores de parámetros y retorno que pide la función global. Y esta tendrá como proceso recibir un valor global void*, en cual será transformada al tipo de dato que se utiliza y hará su acción.

Ejemplo:

```
1 void mostrarEntero(void *elemento)
2 {
3     int *elementoTransformado = (int*)(elemento);
4     printf("%d",*elementoTransformado);
5 }
6
7 void* mostrarVector(void *vec, size_t longitud, size_t tamanio, void mostrar(void*))
8 {
9     void *fin = vec + longitud * tamanio;
10
11    while(vec < fin)
12    {
13        mostrar(vec);
14        printf(" ");
15        vec += tamanio;
16    }
17    printf("\n");
18 }
19
20 int main()
21 {
22     int vec[] = {0,1,2,3};
23     size_t tamanio = sizeof(*vec),
24           longitud = sizeof(vec) / tamanio;
25
26     mostrarVector(vec,longitud,tamanio,mostrarEntero);
27
28     return 0;
29 }
```

Si la función debe recibir por argumento 3 funciones o más, con especialidad para su tipo de dato, no es recomendado transformar la función en uso de tipo de dato global.

Funciones 'mem'

Las funciones mem (memcpy, memmove, memset, memcmp, etc...) forman parte de la biblioteca estándar "string.h", y están diseñadas para trabajar con bloques de memoria en lugar de cadenas de caracteres. Estas funciones tratan la memoria como una secuencia de bytes y no dependen de terminadores como el carácter nulo '\0' o de tipos de datos.

- **memcpy**

Copia un bloque de memoria de una ubicación a otra. Copia **n bytes** desde la dirección apuntada por **origen** a la dirección apuntada por **destino**.

```
● ● ●

1 void* memcpy(void *destino, void *origen, size_t bytes)
2 {
3     // Convertimos los punteros genéricos a tipo char*
4     // para poder copiar byte a byte.
5     char *dest = (char*)destino,
6         *orig = (char*)origen;
7
8     // Mientras queden bytes por copiar
9     while(bytes--)
10    {
11        // Copiamos el valor del byte de origen a destino
12        *dest = *orig;
13
14        // Avanzamos los punteros para copiar el siguiente byte
15        dest++;
16        orig++;
17    }
18
19 /*
20 Tambien se puede escribir el bucle como:
21 while(bytes--)
22     *dest++ = *orig++;
23 */
24
25 // Devolvemos el puntero destino
26 return destino;
27 }
```

- **memmove**

Similar a memcpy, pero maneja correctamente el solapamiento de regiones de memoria. Es más seguro que memcpy pero menos eficiente.

Copia n bytes desde origen a destino, pero permite que las regiones de memoria se superpongan sin problemas. Si hay solapamiento, memmove asegura que la copia se realice correctamente. La única diferencia con memcpy es que almacena en un bloque auxiliar el bloque que se quiere copiar. Ejemplo:

```
1 void *memmove(void *destino, void* origen, size_t bytes)
2 {
3     // Si destino y origen son iguales, no hay nada que hacer
4     if( destino == origen )
5         return destino;
6
7     // Reserva un bloque de memoria auxiliar del tamaño de bytes
8     void *aux = malloc(bytes);
9
10    // Verifica si la reserva de memoria fue exitosa
11    if(!aux)
12        return NULL; // Error, finaliza el programa
13
14    // Copia los datos del origen a la memoria auxiliar
15    memcpy(aux, origen, bytes);
16
17    // Copia los datos de la memoria auxiliar a destino
18    memcpy(destino, aux, bytes);
19
20    // Libera la memoria auxiliar
21    free(aux);
22
23    // Retorna el puntero al bloque de memoria destino
24    return destino;
25 }
```

- **memcmp**

Compara dos bloques de memoria byte por byte. Compara los primeros n bytes de las ubicaciones de memoria apuntadas por dir1 y dir2. Los resultados pueden ser:

- str1 = str2 => 0 (cero)
- str1 > str2 => Numero Positivo
- str1 < str2 => Numero Negativo



```
1 int memcmp(void *dir1, void *dir2, size_t bytes)
2 {
3     // Convertimos los punteros a char para realizar comparacion
4     char *dirc1 = (char*)dir1,
5         *dirc2 = (char*)dir2;
6
7     // Recorremos byte a byte hasta que no halla mas
8     while(bytes--)
9     {
10         if(*dirc1 != *dirc2)
11             return *dirc1 - *dirc2;
12
13         // Avanzamos al siguiente byte
14         dirc1++;
15         dirc2++;
16     }
17
18     // Si todas las comparaciones fueron iguales, retornamos 0.
19     return 0;
20 }
```

- **memset**

Establece a un bloque de memoria un valor específico. Llena los primeros n bytes de la memoria apuntada por dir con el valor especificado (interpretado como un unsigned char).

```
char nombre[] = "pablo";
memset(nombre, '&', 5);
// Resultado = "|||||"
```

```
1 void* memset(void *dir, int valor, size_t bytes)
2 {
3     // Convertimos el puntero a unsigned char* para operar byte por byte
4     unsigned char *dirChar = dir;
5     // Convertimos el valor a unsigned char, ya que solo necesitamos un byte
6     unsigned char byte_valor = valor;
7
8     // Llenamos la región de memoria especificada con el valor proporcionado
9     while (bytes--)
10    {
11        *dirChar = byte_valor; // Asignamos el valor al byte actual
12        dirChar++; // Avanzamos al siguiente byte
13    }
14
15    return dir; // Retornamos el puntero al bloque de memoria
16 }
```

Funciones 'Map', 'Filter' y 'Reduce'

Son funciones comunes en todo lenguaje de programación, pero no estándar, donde permite realizar diversas operaciones dentro del recorrido de un vector.

- **Map**

Aplica una función a cada elemento de un vector.

```
● ● ●  
1 void map(void *vec, size_t longitud, size_t tamanio, void accion(void*))  
2 {  
3     void* fin = vec + longitud * tamanio;  
4     while(vec < fin)  
5     {  
6         accion(vec);  
7         vec += tamanio;  
8     }  
9 }
```

Ejemplo:

```
● ● ●  
1 void restarInt(void* el)  
2 { *(int*)el -= 1; }  
3  
4 int main()  
5 {  
6     int vec[] = {2,3,4,5,6,7};  
7     map(vec,6,4,restarInt);  
8     // Resultado: {1,2,3,4,5,6}  
9 }
```

- **Filter**

Filtrá los elementos de un vector basándose en una función condicional.

Retorna solo los elementos que cumplen con la condición.

```
1 void filter(void *vec, size_t *longitud, size_t tamano, int condicion(void*)) {
2     void* fin = vec + *longitud * tamano; // Calcular el final del array
3     void* pos;
4
5     // Recorrer el array mientras no se haya llegado al final
6     while (vec < fin) {
7         // Aplicar la condición a cada elemento
8         if (condicion(vec)) { // Si se cumple la condición, eliminar el elemento
9             pos = vec + tamano;           // Posición del siguiente elemento
10            memmove(vec, pos, fin - pos); // Mover los elementos restantes hacia atrás
11
12            fin -= tamano; // Actualizar el puntero de fin
13            (*longitud)--; // Reducir el tamaño del array
14        } else {
15            // Avanzar solo si no se elimina el elemento
16            vec += tamano;
17        }
18    }
19 }
```

Ejemplo:

```
1 int filtrarInt(void *el)
2 { return *(int*)el <= 5 ? 1 : 0; }
3
4 int main()
5 {
6     int vec[] = {1,2,3,4,5,6,7};
7     size_t tamano = sizeof(*vec),
8            longitud = sizeof(vec)/tamano;
9
10    filter(vec,&longitud,tamano,filtrarInt);
11    // Resultado: {6,7}
12 }
```

- **Reduce**

Aplica una función acumulativa o de reducción a los elementos de vector de manera que se reduce a un solo valor.

```
1 void* reduce(void *vec, size_t lon, size_t tam, void reductor(void*, void*), void* inicial) {  
2     void *acumulador = inicial,  
3         *fin = vec + lon * tam;  
4  
5     while(vec < fin)  
6     {  
7         // Toma el acumulador actual y el elemento actual del array para combinarlos.  
8         reductor(acumulador, vec);  
9         vec += tam;  
10    }  
11  
12    return acumulador;  
13 }
```

Ejemplo:

```
1 // Función reductora: suma de enteros  
2 void sumaInt(void* acumulador, void* actual) {  
3     *(int*)acumulador += *(int*)actual;  
4 }  
5  
6 int main()  
7 {  
8     int vec[] = {1,2,3,4},  
9         inicial = 5,  
10        *resultado;  
11     size_t tamanio = sizeof(*vec), longitud = sizeof(vec)/tamanio;  
12  
13     resultado = reduce(vec, longitud, tamanio, sumaInt, &inicial);  
14     // Resultado = 15 => 5 + 1 + 2 +3 + 4  
15  
16     return 0;  
17 }
```

Algoritmo de Ordenamiento por Selección

Algoritmo de ordenamiento recomendado para el uso de vectores de gran magnitud o para un vector en general. Es parte del estándar y se llama **qsort**.

El ordenamiento por selección divide la lista en dos partes: la parte ordenada y la parte desordenada. En cada iteración, se selecciona el elemento más pequeño de la parte desordenada y se intercambia con el primer elemento de esa parte. Este proceso se repite hasta que toda la lista está ordenada:

50	26	7	9	15	27	Array Original
----	----	---	---	----	----	----------------

7	26	50	9	15	27	Se coloca el 7 en 1° posición. Intercambia: 7 y 50
7	9	50	26	15	27	Se coloca el 9 en 2° posición. Intercambia: 9 y 26
7	9	15	26	50	27	Se coloca el 15 en 3° posición. Intercambia: 15 y 50
7	9	15	26	50	27	26 no cambia de posición
7	9	15	26	27	50	Se coloca el 27 en 5° posición. Intercambia: 27 y 50
7	9	15	26	27	50	Resultado Final

Parte Ordenada

Para crear este algoritmo debemos:

1. Iniciar:

Comienza en la función 'ordenarSelección', donde se pasan el vector vec, la longitud del vector, el tamaño de cada elemento, y una función de comparación cmp.

2. Iteración Externa:

Se establece un bucle que itera sobre cada elemento del vector hasta longitud - 1. Esta iteración busca ubicar cada elemento en su posición correcta en el vector ordenado.

3. Búsqueda Elemento Menor

En cada iteración del bucle, se llama a *buscarElementoMenor*, pasando el vector y el número restante de elementos desordenados.

4. Intercambiar Elementos

Se compara su dirección con la dirección del elemento en la posición actual. Si el menor elemento no está en la posición actual, se llama a *intercambiarElementos*.

5. Avance al Siguiente Elemento

Después de intercambiar se avanza el puntero vec al siguiente elemento.

6. Repetición:

El proceso se repite hasta que todos los elementos han sido procesados.

```

1 // Función que ordena un vector usando el método de selección con lógica genérica
2 void ordenarSeleccion(void *vec, size_t longitud, size_t tamanio, int cmp(void*, void*))
3 {
4     void *vecFin = vec + tamanio * (longitud - 1), // Puntero para controlar el final
5         *posMenor; // Puntero que almacena la dirección del menor elemento encontrado
6
7     // Itera en todos los items menos en el ultimo que no es necesario
8     while (vec < vecFin)
9     {
10        // Busca el menor en el rango actual (desde 'vec' hasta el final)
11        posMenor = buscarElementoMenor(vec, longitud, tamanio, cmp);
12
13        // Si el menor no es el elemento actual, se intercambian
14        if (posMenor != vec)
15            intercambiarElementos(vec, posMenor, tamanio);
16
17        vec += tamanio; // Avanza al siguiente elemento en el array
18        longitud--; // Como se avanzado en el vec, reducimos la longitud de búsqued
19    }
20}

```

```

1 void* buscarMenor(void *vec, size_t longitud, size_t tamanio, int cmp(void*, void*))
2 {
3     void *fin = vec + (longitud - 1) * tamanio, // Calcular la dirección final del vec
4         *pos = vec; // Apunta al elemento actual o primero, donde se guarda el menor
5
6     while(vec < fin) // Itera cada elemento del vector comparando cuál es menor
7     {
8         vec += tamanio; // Avanza al siguiente elemento del vector (razón por la resta 1)
9         if(cmp(vec, pos) < 0) // Si el elemento actual es menor, actualiza pos
10            pos = vec;
11    }
12
13    return pos; // Retorna la dirección del menor elemento
14}

```

```

1 void intercambioElementos(void *el1, void *el2, size_t bytes)
2 {
3     // Definimos los punteros char para recorrer byte por byte los elementos
4     char *elChar1 = (char*)el1, *elChar2 = (char*)el2;
5     char aux; // Variable auxiliar para almacenar temporalmente el valor de un byte
6
7     // Mientras queden bytes por intercambiar
8     while (bytes)
9     {
10
11        aux = *elChar1; // Guardamos el valor del byte actual de elChar1 en aux
12        *elChar1 = *elChar2; // Copiamos el valor de elChar2 a elChar1
13        *elChar2 = aux; // Colocamos el valor guardado en aux en elChar2
14
15        elChar1++; // Avanzamos ambos punteros al siguiente byte
16        elChar2++;
17        bytes--; // Reducimos la cantidad de bytes restantes
18    }
19}

```

Algoritmo de Búsqueda Binaria

Método de búsqueda de un dato extremadamente eficiente, necesita que este si o si ordenado. Funciona al dividir repetidamente el rango de búsqueda a la mitad hasta que encuentra el elemento o determina que no está presente en el conjunto:

Buscar : 2

2	4	6	8	10	12	14
0	1	2	3	4	5	6

2 < 8

2	4	6	8	10	12	14
0	1	2	3	4	5	6

2 < 4

2	4	6	8	10	12	14
0	1	2	3	4	5	6

2 = 2

Los pasos a seguir:

1. Divide al Conjunto a la Mitad:

Se comienza buscando en la mitad de la lista. Comparamos el valor de la mitad con el valor que estamos buscando.

2. Comparación:

Pueden ocurrir 3 casos:

- El valor de la mitad es igual al que buscamos (finaliza)
- Si es mayor, significa que el valor buscado está en la mitad inferior del array. Por lo tanto, descartamos la otra mitad
- Si es menor, significa que el valor buscado está en la mitad superior del array. Por lo tanto, descartamos la otra mitad.

3. Repetir:

Cada vez que descartamos una mitad, repetimos el proceso con el nuevo rango de búsqueda, tomando nuevamente la mitad y comparando el valor hasta que encontremos el elemento o el rango de búsqueda se reduzca a cero.

4. Finalizar:

Si llegamos a un punto en el que ya no queda ninguna parte del conjunto por buscar y no hemos encontrado el elemento, podemos concluir que el valor no está presente.

Ejemplo:

```
● ● ●

1 void* busquedaBinaria(void* buscar, void* vec, size_t longitud, size_t tamano, int cmp(void*, void*)) {
2     // Inicializa los punteros de inicio y fin que delimitan el rango de búsqueda
3     void *mitad;                                // Puntero a la posición central actual
4     void *inicio = vec;                          // Puntero al primer elemento del array
5     void *fin = vec + (longitud - 1) * tamano; // Puntero al último elemento del array
6     int resCmp;                                // Variable para almacenar el resultado de la comparación
7
8     // El bucle continúa mientras el puntero de inicio no supere al puntero de fin
9     while (inicio <= fin) {
10         // Calcula la dirección de la mitad del rango actual utilizando aritmética de punteros
11         mitad = inicio + ((fin - inicio) / (2 * tamano)) * tamano;
12
13         // Compara el valor buscado con el valor en la posición de mitad
14         resCmp = cmp(buscar, mitad);
15
16         // Si la comparación da cero, significa que hemos encontrado el valor
17         if (resCmp == 0)
18             // Devuelve el puntero al valor encontrado
19             return mitad;
20
21         // Si el valor buscado es menor que el valor en la posición de mitad
22         else if (resCmp < 0)
23             // Ajusta el puntero de fin para reducir la búsqueda a la mitad inferior
24             fin = mitad - tamano;
25
26         // Si el valor buscado es mayor que el valor en la posición de mitad
27         else
28             // Ajusta el puntero de inicio para reducir la búsqueda a la mitad superior
29             inicio = mitad + tamano;
30     }
31
32     // Si el bucle termina sin encontrar el valor, se retorna NULL
33     return NULL;
34 }
```

Unidad 4: TDA

Definición

El Tipo de Dato Abstracto (TDA) es una representación organizada de información, comúnmente una estructura de datos, que encapsula solo los aspectos más relevantes de un conjunto de datos, como un vector. En este contexto, el programador no necesita conocer los detalles internos del tipo de dato o cómo se almacena la información. Los datos en un TDA son manipulados únicamente mediante funciones específicas que operan sobre ellos, tales como:

- Constructor (Inicialización)
- Destructor (eliminar)
- Inserción
- Inserción ordenada
- Búsqueda
- Entre muchas otras

El uso de un TDA permite simplificar la programación, ya que las funciones recurrentes, como aquellas que operan sobre vectores, siempre requieren los mismos datos: dirección, longitud, cant elementos, tamaño. Con TDA solo necesitan recibir la referencia, sin que el programador deba preocuparse por los detalles internos.

```
1  typedef struct
2  {
3      void *vec;
4      size_t longitud;
5      size_t cantEl;
6      size_t tamanio;
7  } TDAVec;
```

Los TDA tiene ciertas características (similar a POO):

- Abstracción
Se enfoca en el "qué" hace y no en el "cómo".
- Encapsulamiento
Ocultar detalles internos de implementación de una estructura de datos, exponiendo solo lo necesario a través de una interfaz pública (funciones).
- Independencia de Implementación
Permite cambiar la implementación sin afectar el programa o interfaz.
- Reutilización del Código

Implementación Genérica

No es ningún conjunto de prácticas, técnicas o temas nuevos de programación, trata sobre aplicar a mayor escala lo aprendido. Las funciones de vectores son las mismas aplicadas en la unidad 0 y 1. El contenido de TDA es recomendado almacenarlo en un archivo/librería aparte. Ejemplo:

- **Estructura**

Se utilizará la estructura anteriormente mostrada

```
1 typedef struct
2 {
3     void *vec;      // Puntero genérico a la memoria del vector
4     size_t longitud; // Cantidad máxima de elementos que puede contener
5     size_t cantEl; // Cantidad actual de elementos insertados
6     size_t tamano; // Tamaño en bytes de cada elemento del vector
7 } TDAVec;
```

El vector esta con puntero porque va a ser dirigido con memoria dinámica.

- **Main**

Se creará este main y se irán desarrollando las funciones poco a poco.

```
1 int main()
2 {
3     // Inicializar variable
4     TDAVec vec;
5
6     // Inicializar TDA de tipo int de lon 15
7     crearVec(&vec, 15, sizeof(int));
8
9     // Insertar Valores
10    for(size_t i = 10 ; i <= 15 ; i++)
11        insertarVec(&vec, &i);
12
13    // Recorrer y realizar una acción
14    recorrerVec(&vec, mostrarInt);
15    // Resultado: 10-11-12-13-14-15
16
17    // Destruir TDA
18    destruirVec(&vec);
19
20    return 1;
21 }
```

- **Crear Vector (constructor)**

Se ejecuta 1 vez al iniciar el programa

```
● ● ●

1 // Inicializa la estructura TDAVec, asignando memoria dinámica para el vector
2 int crearVec(TDAVec *TDA, size_t longitud, size_t tamanio)
3 {
4     // Asigna memoria para el vector en base a la longitud y tamaño
5     TDA->vec = malloc(longitud * tamanio);
6
7     // Verifica si se pudo asignar memoria
8     if(!TDA->vec)
9         return 0; // Fallido
10
11    // Inicializa los campos del TDA
12    TDA->cantEl = 0;           // Al principio, el vector está vacío
13    TDA->longitud = longitud; // Asigna la capacidad máxima
14    TDA->tamanio = tamanio;   // Tamaño en bytes de cada elemento
15
16    return 1; // Éxito
17 }
```

- **Destruir Vector (destructor)**

Se ejecuta una vez al finalizar el programa:

```
● ● ●

1 // Esta función elimina el vector para evitar fugas de memoria
2 void destruirVec(TDAVec *TDA)
3 {
4     free(TDA->vec); // Libera la memoria reservada para el vector
5     // Resetear valores
6     TDA->cantEl = 0;
7     TDA->longitud = 0;
8     TDA->tamanio = 0;
9 }
```

- **Redimensionar Longitud del Vector**

Se hace uso del realloc.

```
1 // Redimensiona el vector a una nueva longitud usando realloc
2 int extenderVec(TDAVec *TDA, size_t nuevaLongitud)
3 {
4     // Redimensiona la memoria asignada al vector
5     TDA->vec = realloc(TDA->vec, nuevaLongitud * TDA->tamanio);
6
7     // Verifica si la reallocación fue exitosa
8     if(!TDA->vec)
9         return 0; // Falla
10
11    // Actualiza la longitud del vector con la nueva capacidad
12    TDA->longitud = nuevaLongitud;
13    return 1; // Exito
14 }
```

- **Insertar Elementos**

Existen múltiples funciones de inserción por lo que se vio (insertarOrdenado, insertarPrincipio, InsertarMedil, etc...). En este caso, es la implementación más sencilla, insertar al final:

```
1 // Copia el contenido del nuevo elemento en la última posición libre del vector
2 int insertarVec(TDAVec *TDA, void* ingreso)
3 {
4     // Verifica si hay espacio en el vector para un nuevo elemento
5     if(TDA->cantEl >= TDA->longitud)
6         return 0; // Vector Lleno
7
8     // Calcula la posición de la próxima inserción
9     void *fin = TDA->vec + TDA->tamanio * TDA->cantEl;
10
11    // Copia el nuevo elemento en la posición calculada
12    memcpy(fin, ingreso, TDA->tamanio);
13
14    // Incrementa la cantidad de elementos en el vector
15    TDA->cantEl++;
16
17    return 1; // Exito
18 }
```

Conociendo ya esta implementación, se debe saber ya cómo hacer el resto de funciones de inserción y como crear las de eliminación.

- **Recorrido del Vector (igual que map)**

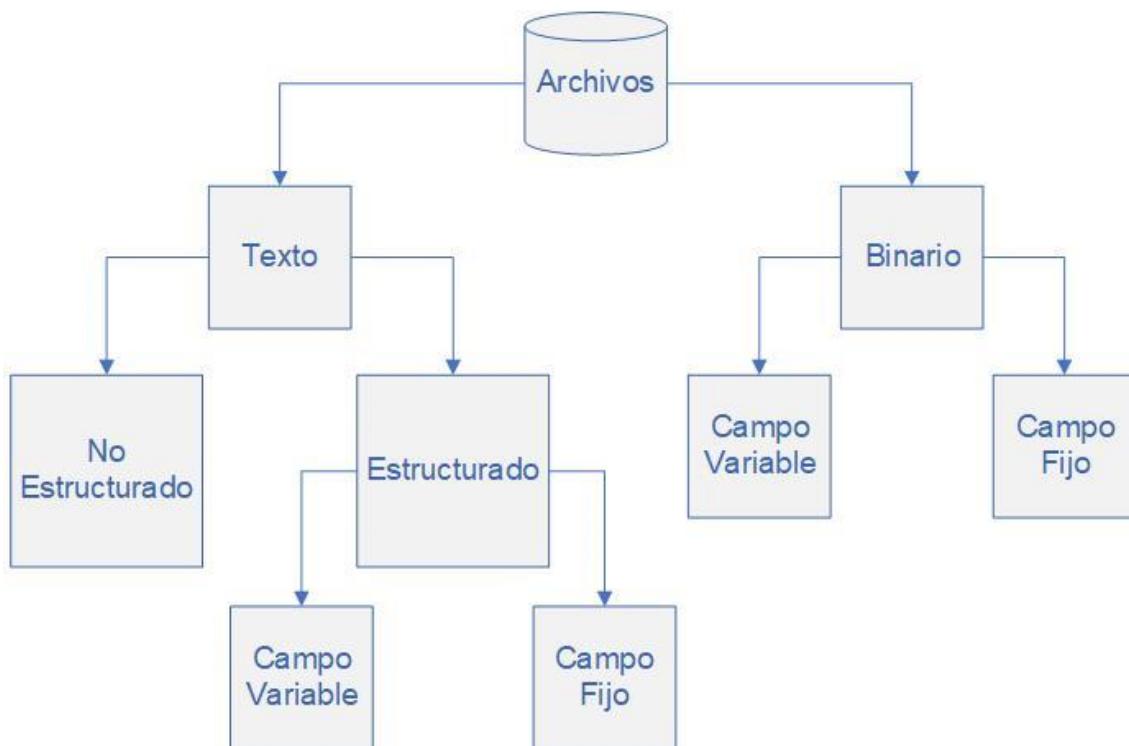
```
1 // Toma una función como parámetro que define qué hacer con cada elemento
2 void recorrerVec(TDAVec *TDA, void accion(void*))
3 {
4     void *ini = TDA->vec;
5     void *fin = TDA->vec + TDA->tamanio * TDA->cantEl;
6
7     // Bucle que recorre el vector mientras no se alcance el final
8     while(ini < fin)
9     {
10         accion(ini);          // Aplica la función 'accion' al elemento actual
11         ini += TDA->tamanio;// Avanza al siguiente elemento
12     }
13 }
```

Cuidado de no aumentar el puntero almacenado en '*TDA', por eso se utiliza una variable 'ini'.

Unidad 5: Archivos

Tipos de Archivos para C

Existen diferentes tipos de archivos para S.O. Pero para C, solo existen de 2 tipos con subcategorías:



- **Texto**

- No Estructurado

Contiene datos sin un formato fijo ni delimitadores. Un ejemplo es un archivo de texto libre extenso.

- Estructurado

Diseñado para almacenar datos organizados. Los datos se estructuran en registros y campos que siguen un formato determinado. En este formato entregan registros las bases de datos. Ejemplos: JSON o XML.

- Campo Variable

Los campos no tienen un tamaño fijo. Ejemplos: CSV o JSON. Ocupa menos tamaño, pero es menos eficiente.

- Campo Fijo

Cada campo en el archivo tiene un número predefinido de bytes. Ocupa más, pero es más eficiente.

- **Binario**

- Campo Variable

Tipo más eficiente para almacenar datos, el archivo contiene directamente las representaciones binarias de las estructuras de datos en memoria (y no con puro ASCII). Tipo utilizado por bases de datos.

- Campo Fijo (no lo utilizamos)

Los archivos binarios con campos de tamaño variable se usan en otros contextos, como en archivos multimedia (audios o videos).

Manipulación General de Archivos (Reaso)

Estos son funciones que se utilizan independientemente del tipo de archivo.
Reaso de la materia anterior:

```
FILE* fopen(const char* nombre, const char* modo);
```

Antes de utilizar un archivo debe ser abierto. Esta función retorna la dirección de memoria (puntero) donde se encuentra la estructura para su manejo, si hubo error retorna null. Los modos disponibles son:

Modo	Sub Modos	Utilidad
r	rt - rb	Lectura
w	wt - wb	Escritura (sobrescribir)
a	at - ab	Agregar contenido
r+b		Lectura y Escritura
w+b		Escritura y Lectura
a+b		Agregar y Lectura

La "t" corresponde a archivo de texto, la "b" a archivo binario, si no se especifica quiere decir que son archivo de texto (r = rt por ejemplo). Debemos especificar la ruta absoluta o relativa desde el archivo del ejecutable, y si escribimos la barra debe ser doble () -> (\).

```
int fclose(FILE *archivo);
```

Cuando un programa deja de necesitar un archivo debe cerrarse.

```
int remove(const char *nombre);
```

Elimina un archivo. Retorna un cero indicando que es correcto, y recibe como parámetro el string del nombre del archivo.

```
int rename (const char *nombreViejo, const char *nombreNuevo);
```

Cambia el nombre de un archivo. Retorna un cero indicando que es correcto, y recibe como parámetro 1º el nombre viejo (actual) y 2º el nombre nuevo.

```
int fseek(FILE puntero, long cantidadBytesDesplazar, int referencia);
```

Nos permite desplazarnos en el archivo para querer leer, modificar o escribir datos. Su desplazamiento es igual que en el array. Recibe 3 parámetros:

1. Archivo Puntero
2. Cantidad de Bytes a Desplazarse
3. Referencia

Desde que ubicación se va a realizar el desplazamiento, en las opciones podemos utilizar los números o constantes:

- 0 o SEEK_SET - Desde el comienzo del archivo
- 1 o SEEK_CUR - Desde la posición actual
- 2 o SEEK_END - Desde el final del archivo

Podemos realizar lecturas como escrituras sobre un archivo, pero antes de cambiar de modo es necesario reposicionarse en el archivo usando fseek o utilizando fflush. No podemos luego de fread/fwrite realizar una acción sin antes de hacer uso de las funciones dichas.

```
long ftell(FILE* archivo);
```

Se le da como parámetro una variable FILE y retorna un tipo de dato long int indicando la cantidad de bytes desde el origen.

```
void rewind(FILE* archivo);
```

Coloca el puntero de archivo al inicio.

Manipulación Archivos de Texto

Texto No Estructurado

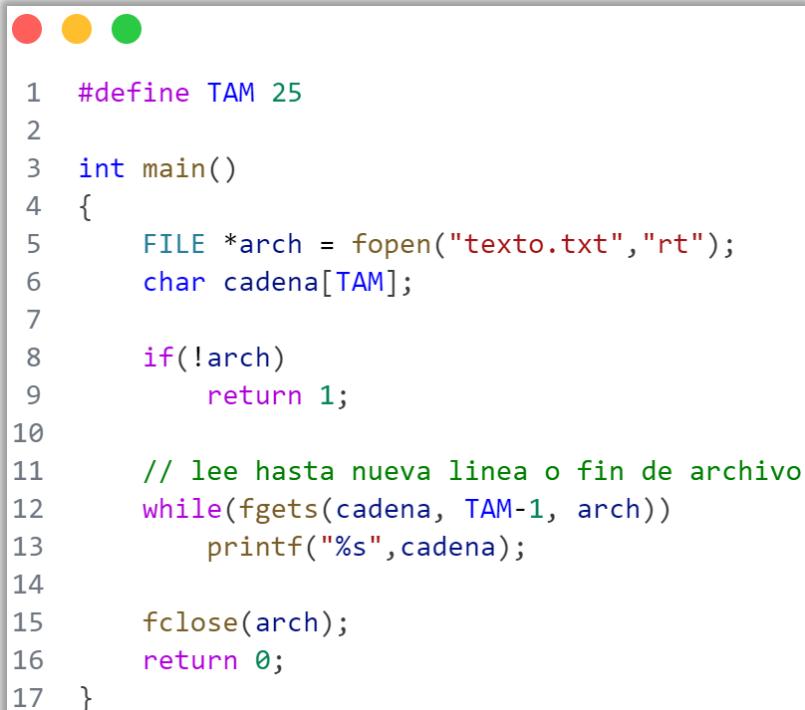
- **Lectura Línea por Línea**

```
char *fgets(char *str, int bytes, FILE *arch);
```

Lee una línea de texto desde un archivo y la almacena en un buffer (cadena de caracteres). Recibe 3 argumentos:

1. Str – Puntero a la cadena donde se almacenara
2. Bytes – Número maximo de bytes(caracteres) a leer
3. Arch – Puntero del archivo

Ejemplo:



The screenshot shows a terminal window with three colored window control buttons (red, yellow, green) at the top. The window contains the following C code:

```
1 #define TAM 25
2
3 int main()
4 {
5     FILE *arch = fopen("texto.txt","rt");
6     char cadena[TAM];
7
8     if(!arch)
9         return 1;
10
11    // lee hasta nueva linea o fin de archivo
12    while(fgets(cadena, TAM-1, arch))
13        printf("%s",cadena);
14
15    fclose(arch);
16    return 0;
17 }
```

- **Escribir Cadena**

```
int fputs(const char *str, FILE *arch);
```

Escribe una cadena de caracteres en un archivo. Retorna un valor positivo si la operación fue exitosa o EOF (-1) en caso contrario. Ejemplo:



```
1 #define TAM 25
2
3 int main()
4 {
5     size_t i;
6     FILE *arch = fopen("texto.txt", "wt");
7     char cadena[][][TAM] = {
8         {"Melisari"},
9         {"Vega"},
10        {"Aranibar"}
11    };
12
13    if(!arch)
14        return 1;
15
16    for(i = 0 ; i < sizeof(cadena)/TAM ; i++)
17    {
18        fputs(cadena[i], arch);
19        fputs("\n", arch);
20    }
21
22    fclose(arch);
23    return 0;
24 }
```

- **Escribir Cadenas con Formato**

```
int fprintf(FILE *arch, const char *formato, ...);
```

Esta función cumple con el mismo objetivo y formato que la función de printf. La única diferencia es que primero indicamos el puntero del archivo, aunque tambien se puede ingresar el valor "stdout" donde imprime en consola, con el único fin de tener una perspectiva del resultado de impresión.

Texto Estructurado Variable

La manipulación de archivos de texto estructurado variable implica el uso de formatos flexibles que permiten que los campos de los registros tengan longitudes diferentes.

Para delimitar los campos se utilizan “|” (o el que queramos), y el formato no se le debe especificar la N cantidad de espacios (exceptuando los flotantes).

Para todos los ejemplos se hará uso del siguiente código inicial:



```
1 #define SEPARADOR '|'  
2  
3 typedef struct  
4 {  
5     int leg;  
6     char nombre[MAXNOM];  
7     float prom;  
8 } ESTS;
```

• Escritura de Archivo

Para esto, hacemos uso de la función **fprintf** y un **formato** de texto. El uso de estructuras es casi obligatorio, donde vamos a hacer uso de sus datos para combinarlos con el formato al momento de hacer uso del fprintf, recorriendo uno por uno. Ejemplo:

```
● ● ●

1 #define FORMATO "%d|%s|%.2f\n"
2
3 int crearEstudiantesVariable(const char* nomArch)
4 {
5     size_t i;
6     ESTS est[] =
7     {
8         {10,"Maria Pia",2.2},
9         {20,"Juan Domingo",5.5},
10        {30,"Mariana",1}
11    };
12    FILE* arch = fopen(nomArch, "wt");
13
14    if(!arch)
15        return 0;
16
17    for(i = 0 ; i < sizeof(est)/sizeof(*est) ; i++)
18        fprintf(arch,FORMATO, est[i].leg, est[i].nombre ,est[i].prom);
19
20    fclose(arch);
21    return 1;
22 }
```

- **Lectura Archivo**

Para la lectura tenemos 3 maneras, forma automática (no recomendado), manual y directo desde el archivo. Se utiliza la función sscanf:

```
int sscanf(const char *str, const char *format, ...);
```

Lee datos de una cadena de caracteres. La función analiza la cadena proporcionada y extrae los valores según el formato especificado. Las variables a ingresar luego del formato deben ser siempre punteros. Tiene la misma sintaxis que printf. Debe ser combinado con la función fgets para guardar la línea en una cadena y con sscanf analizar esa línea.

Si se quiere analizar un espacio flotante se debe tener cuidado en el formato y no declarar la parte decimal, solo la entera (opcional).

- Lectura Manual

Para este proceso, utilizamos la función `strrchr`, que busca la última aparición de un carácter específico en una cadena, en este caso, el separador de campos.

1. Encontrar el ultimo salto de línea obtenido al leer la línea del archivo, una vez obtenido se lo reemplaza por el carácter nulo para evitar conflictos.
2. La primera lectura de un campo comenzamos leyendo desde el ultimo separador hasta el carácter nulo ('\0') para extraer la información del campo correspondiente.
3. Después de leer el campo, reemplazamos el separador por el carácter nulo. Para así leer el siguiente campo.
4. Repitiendo procedimiento para cada campo hasta procesar toda la estructura.
5. Por último, cuando ya no quedan más separados se utiliza el inicio de la cadena para leer el contenido restante.

Estructura														
Legajo				Nombre					Promedio					
0	0	3	2		H	O	L	A		5	.	2	\n	\0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Dirigirse al ultimo salto de linea con `[strrchr(str, '\n') = 13]`

0	0	3	2		H	O	L	A		5	.	2	\n	\0
													↑ Dir	

Remplazar el salto por el carácter nulo `[(str+13) = '\0']`

0	0	3	2		H	O	L	A		5	.	2	\0	\0
													↑ Dir	

Nos posicionamos en el ultimo separador `[strrchr(str, '|') = 9]` y realizamos lectura promedio

0	0	3	2		H	O	L	A		5	.	2	\0	\0
													↑ Dir	↑ Apartir de aca leemos

Remplazar el separador por el carácter nulo `[(str+9) = '\0']`

0	0	3	2		H	O	L	A	\0	5	.	2	\0	\0
													↑ Dir	

Nos posicionamos en el ultimo separador `[strrchr(str, '|') = 4]` y realizamos lectura nombre

0	0	3	2		H	O	L	A	\0	5	.	2	\0	\0
													↑ Dir	↑ Apartir de aca leemos

Remplazar el separador por el carácter nulo `[(str+4) = '\0']`

0	0	3	2	\0	H	O	L	A		5	.	2	\0	\0
													↑ Dir	

Nos posicionamos al inicio de la cadena y realizamos lectura legajo

0	0	3	2	\0	H	O	L	A		5	.	2	\0	\0
													↑ Apartir de aca leemos	

Este tipo de función se suele llamar **trozar**.

Ejemplo:

```
1 int lecEstsVar(const char* nomArch)
2 {
3     FILE* arch = fopen(nomArch,"rt");
4     ESTS est;
5     char linea[MAXLINE];
6
7     if(!arch)
8         return 0;
9
10    while(fgets(linea,MAXLINE,arch) && trozarEstVar(&est,linea))
11        printf("%d - %s - %.2f\n",est.leg,est.nombre,est.prom);
12
13    fclose(arch);
14    return 1;
15 }
```

```
1 int trozarEstVar(ESTS* est, char *linea)
2 {
3     char *dirSeparador;
4
5     // Eliminar salto de linea
6     dirSeparador = strrchr(linea, '\n');
7     *dirSeparador = '\0';
8
9     // Lectura Promedio
10    dirSeparador = strrchr(linea, SEPARADOR);
11    if (!dirSeparador)
12        return 0;
13    sscanf(dirSeparador + 1,"%f",&(est->prom));
14    *dirSeparador = '\0';
15
16     // Lectura Nombre
17     dirSeparador = strrchr(linea, SEPARADOR);
18     if (!dirSeparador)
19         return 0;
20     strcpy(est->nombre,dirSeparador+1);
21     *dirSeparador = '\0';
22
23     // Lectura Legajo
24     sscanf(linea,"%d",&(est->leg));
25
26     return 1;
27 }
```

- Problemática de Cadenas

Para la lectura de cadenas dentro de la estructura es un caso especial debido a los saltos (de línea o espacio), ya que la lectura finaliza cuando los detecta.

Para poder solucionarlo debemos hacer uso de método que permite realizar la lectura de texto hasta encontrar un carácter específico (en este caso un separador). La manera más sencilla es utilizando expresiones regulares, donde remplazamos el formato "%s" por:

%s => %[^|]

1. % - Indica que estamos usando un especificador de formato
2. [] - Define un conjunto de caracteres permitidos o no para ser leídos.
3. ^ - Significa "todo excepto" el conjunto de caracteres que sigue
4. | - Es el carácter que se está excluyendo, en este caso, la barra vertical |.

- Lectura Automática

La lectura automática es más carismática con el programador, pero tiene el inconveniente de que es inestable. Razón por la que depende según la comisión su prohibición, por lo que debe ser consultado su utilización con el profesor.

Para hacer uso de esto expandimos los argumentos recibidos de la función sscanf, donde metemos el formato sin procesos adicionales con toda su estructura como argumento. Ejemplo:

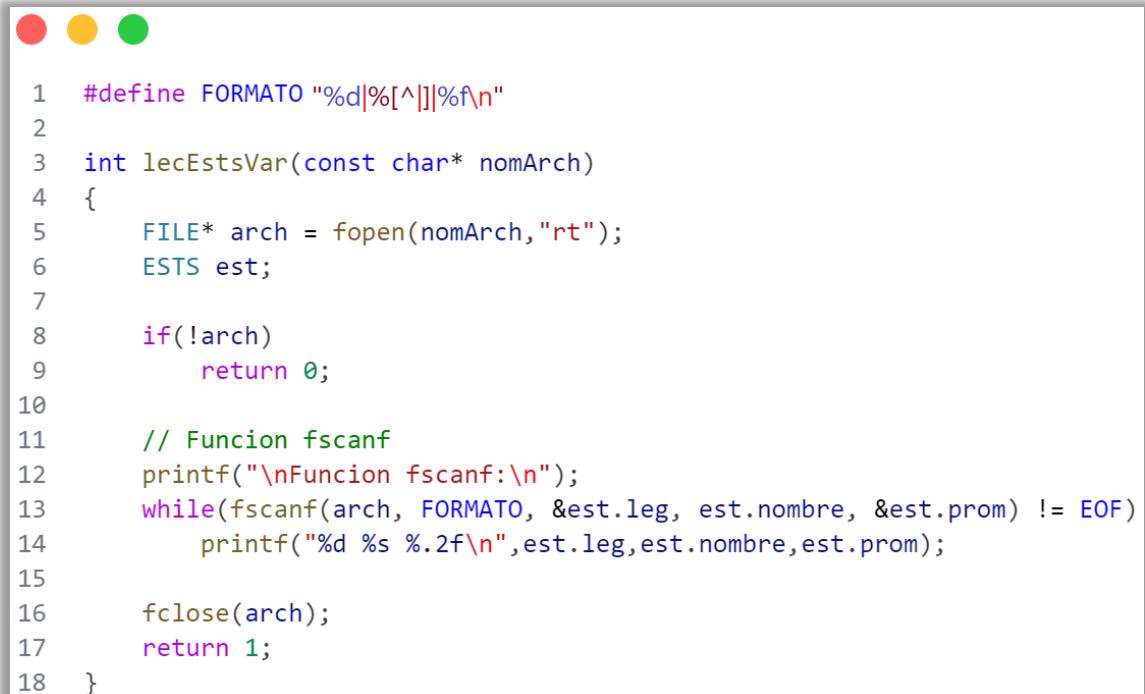
```
1 #define FORMATO "%d%[^|]|%f\n"
2
3 int lecturaEstudiantesVariable(const char* nomArch)
4 {
5     FILE* arch = fopen(nomArch, "rt");
6     ESTS est;
7     char linea[MAXLINE];
8
9     if(!arch)
10         return 0;
11
12     while(fgets(linea, MAXLINE, arch))
13     {
14         // Utilización del sscanf pero a mayor escala
15         sscanf(linea, FORMATO, &est.leg, est.nombre, &est.prom);
16         printf("%d %s %.2f\n", est.leg, est.nombre, est.prom);
17     }
18
19     fclose(arch);
20     return 1;
21 }
```

- Lectura Directa al Archivo

Esta función nos permite ahorrar 2 pasos (fgets y sscanf) en uno solo mediante la función fscanf:

```
int fscanf(FILE *archivo, const char *formato, ...);
```

se utiliza para leer datos de un archivo, de acuerdo al formato especificado. Su comportamiento es similar a la función fscanf solo que recibe un archivo, no una cadena. Ejemplo:



```
1 #define FORMATO "%d|%[^]|%f\n"
2
3 int lecEstsVar(const char* nomArch)
4 {
5     FILE* arch = fopen(nomArch, "rt");
6     ESTS est;
7
8     if(!arch)
9         return 0;
10
11    // Funcion fscanf
12    printf("\nFuncion fscanf:\n");
13    while(fscanf(arch, FORMATO, &est.leg, est.nombre, &est.prom) != EOF)
14        printf("%d %s %.2f\n", est.leg, est.nombre, est.prom);
15
16    fclose(arch);
17    return 1;
18 }
```

Texto Estructurado Fijo

La manipulación de texto estructurado fijo implica laburar con un formato en el que cada campo ocupa un número predefinido de bytes. Esto permite que los datos estén alineados uniformemente y facilita tanto la lectura como la escritura, ya que se puede acceder a cada campo en una posición fija dentro de cada línea o registro.

Se deben utilizar formatos fijos donde N es el tamaño del mismo.
Ejemplo: "%Nd %Ns" (recomendado guardarlo en macro). Si vamos a guardar una cadena, en el formato el espacio de bytes debe ser siempre menor al espacio que ocupa la cadena original, ejemplo: si la cadena ocupa 25 bytes, entonces en el formato debe ser siempre menor a 25 (24, 23, 22, etc...).

Para todos los ejemplos se hará uso del siguiente código inicial:

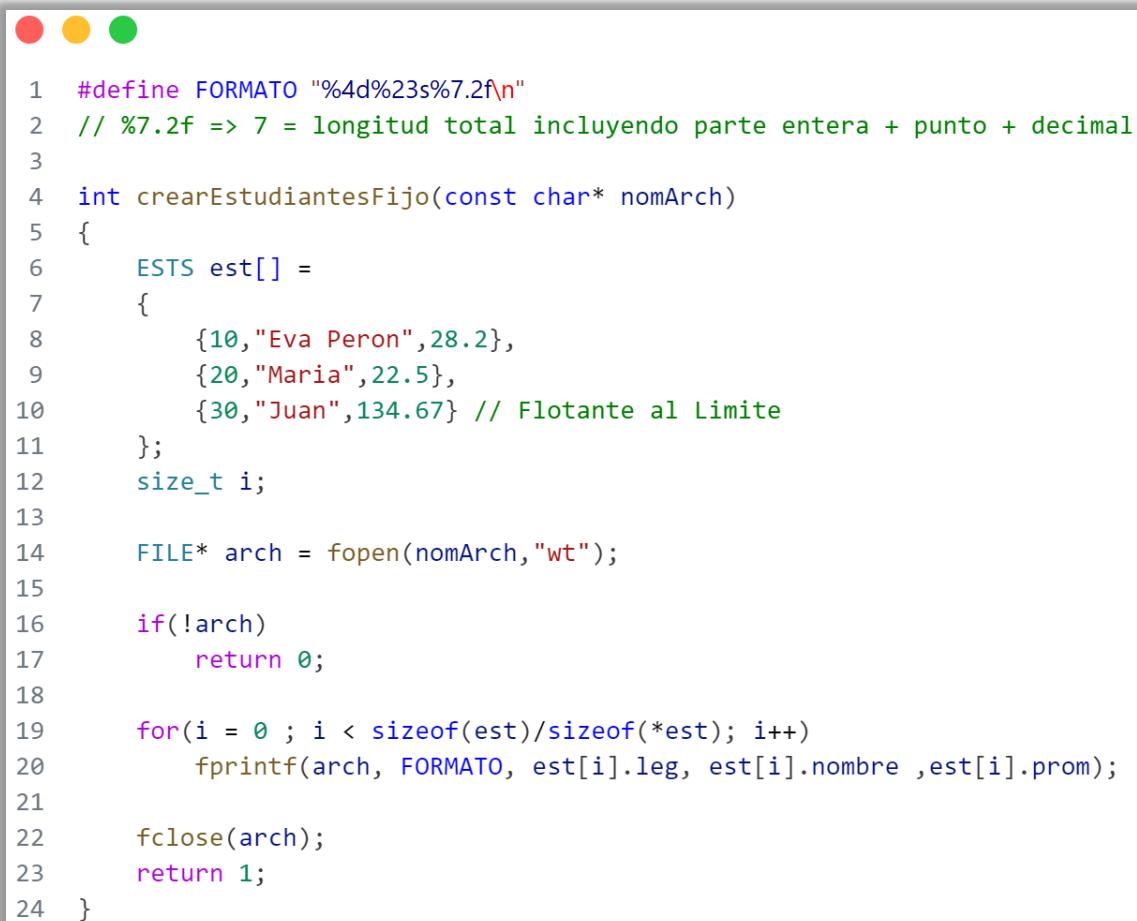
```
1 #define TAM_LEG 4
2 #define TAM_NOM 23
3 #define TAM_PRO 7
4 // total = 4 + 23 + 7 = 34
5
6 typedef struct
7 {
8     int leg;
9     char nombre[MAXNOM];
10    float prom;
11 } ESTS;
```

- **Escritura de Archivo**

Para esto, hacemos uso de la función fprintf y un formato de texto. El uso de estructuras es casi obligatorio, donde vamos a hacer uso de sus datos para combinarlos con el formato al momento de hacer uso del fprintf, recorriendo uno por uno.

Se debe tener cuidado al momento de imprimir un flotante debido a que la parte entera en el formato representa la longitud total permitida, incluyendo: entero + punto + decimales. Ejemplo: %5.2f => 2 enteros + 1 punto + 2 decimales, ese es el rango permitido. Se debe tener cuidado porque un uso incorrecto superpone otros campos.

Ejemplo:



```
1 #define FORMATO "%4d%23s%7.2f\n"
2 // %7.2f => 7 = longitud total incluyendo parte entera + punto + decimal
3
4 int crearEstudiantesFijo(const char* nomArch)
5 {
6     ESTS est[] =
7     {
8         {10,"Eva Peron",28.2},
9         {20,"Maria",22.5},
10        {30,"Juan",134.67} // Flotante al Limite
11    };
12    size_t i;
13
14    FILE* arch = fopen(nomArch,"wt");
15
16    if(!arch)
17        return 0;
18
19    for(i = 0 ; i < sizeof(est)/sizeof(*est); i++)
20        fprintf(arch, FORMATO, est[i].leg, est[i].nombre ,est[i].prom);
21
22    fclose(arch);
23    return 1;
24 }
```

- **Lectura de Archivo**

Se utiliza la función sscanf:

```
int sscanf(const char *str, const char *format, ...);
```

Lee datos de una cadena de caracteres. La función analiza la cadena proporcionada y extrae los valores según el formato especificado, los valores a ingresar luego del formato deben ser siempre punteros. Tiene la misma sintaxis que printf. Debe ser combinado con la función fgets para guardar la línea en una cadena y con sscanf analizar esa línea.

Si se quiere analizar un espacio flotante se debe tener cuidado en el formato y no declarar la parte decimal, solo la entera (opcional).

- Lectura Manual

La lectura del formato fijo es muy similar a la del formato variable, con la diferencia de que no se utilizan separadores ni funciones como strrchr.

1. Dirigirse al salto de línea que se encuentra en la posición anterior del carácter nulo (como los campos son fijos, tan solo debemos sumar esas longitudes para ir al salto). Una vez en el salto de línea, remplazarlo por el carácter nulo.
2. Continuando con la lectura, se resta desde la posición del carácter nulo (nuevo) el tamaño del campo para ir al principio de este y a partir de ahí realizamos la lectura del mismo.
3. Repetimos este procedimiento para cada campo, avanzando desde el último hasta el primero hasta que llegamos al inicio de la cadena.
4. Finalizando, la lectura del primer campo se realiza con la dirección original de la cadena.

Estructura Valores											
Legajo				Nombre				Promedio			
0	0	3	2	H	O	L	A	5	.	2	\n \0
0	1	2	3	4	5	6	7	8	9	10	11 12

Dirigirse al final de la linea coincidente al tamaño del mismo("\n")

0	0	3	2	H	O	L	A	5	.	2	\n \0
↑ Dirección Pos											

Remplazamos el salto por el carácter nulo (str[11] = '\0')

0	0	3	2	H	O	L	A	5	.	2	\0 \0
↑ Apartir de aca leemos											

Restamos al dir el tamaño del ultimo argumento, en este caso -3

0	0	3	2	H	O	L	A	5	.	2	\0 \0
↑ Apartir de aca leemos											

Remplazamos el inicio del campo por el carácter nulo (str[8] = '\0')

0	0	3	2	H	O	L	A	\0	.	2	\0 \0
↑ Apartir de aca leemos											

Restamos al dir el tamaño del ultimo argumento, en este caso -4

0	0	3	2	H	O	L	A	\0	.	2	\0 \0
↑ Apartir de aca leemos											

Remplazamos el inicio del campo por el carácter nulo (str[4] = '\0')

0	0	3	2	\0	O	L	A	\0	.	2	\0 \0
↑ Apartir de aca leemos											

Leemos apartir del inicio de la cadena

0	0	3	2	\0	O	L	A	\0	.	2	\0 \0
↑ Apartir de aca leemos											

Ejemplo:

```
1 int lecEstsFijo(const char* nomArch)
2 {
3     FILE* arch = fopen(nomArch,"rt");
4     ESTS est;
5     char linea[MAXLINE];
6
7     if(!arch)
8         return 0;
9
10    while(fgets(linea,MAXLINE,arch) && trozarEstFijo(&est,linea))
11        printf("%d - %s - %.2f\n",est.leg,est.nombre,est.prom);
12
13    fclose(arch);
14    return 1;
15 }
```

```
1 int trozarEstFijo(ESTS* est, char *linea)
2 {
3     size_t tamano = TAM_LEG + TAM_NOM + TAM_PRO;
4     char *dirCampo = linea + tamano;
5
6     // Se comprueba que ambos tamaños sean iguales
7     if(tamano + 1 != strlen(linea))
8         return 0;
9
10    // Eliminamos salto de linea
11    *dirCampo = '\0';
12
13    // Lectura Promedio
14    dirCampo -= TAM_PRO; // 7
15    sscanf(dirCampo,"%f",&est->prom);
16    *dirCampo = '\0';
17
18    // Lectura Nombre
19    dirCampo -= TAM_NOM; // 23
20    strcpy(est->nombre,dirCampo);
21    *dirCampo = '\0';
22
23    // Lectura Legajo
24    sscanf(linea,"%d",&est->leg);
25
26    return 1;
27 }
```

- Problemática de Cadenas

Para la lectura de cadenas dentro de la estructura es un caso especial debido a los saltos (de línea o espacio), ya que la lectura finaliza cuando los detecta.

Para poder solucionarlo debemos hacer uso de método que permite realizar la lectura de texto hasta encontrar un carácter específico (en este caso un separador). La manera más sencilla es utilizando expresiones regulares, donde remplazamos el formato "%s" por:

%Ns => %N [^\\n]

1. N - Tamaño de la cadena
2. % - Indica que estamos usando un especificador de formato
3. [] - Define un conjunto de caracteres permitidos o no para ser leídos.
4. ^ - Significa "todo excepto" el conjunto de caracteres que sigue
5. \\n - El carácter de nueva línea (\\n), que indica el final de una línea de texto.

- Lectura Automática

La lectura automática es más carismática con el programador, pero tiene el inconveniente de que es inestable. Razón por la que depende según la comisión su prohibición, por lo que debe ser consultado su utilización con el profesor.

Para hacer uso de esto expandimos los argumentos recibidos de la función sscanf, donde metemos el formato sin procesos adicionales con toda su estructura como argumento. Ejemplo:

```
1 #define FORM "%4d%23[^\\n]%7f\\n"
2
3 int lecEstsFijo(const char* nomArch)
4 {
5     FILE* arch = fopen(nomArch, "rt");
6     ESTS est;
7     char linea[MAXLINE];
8
9     if(!arch)
10         return 0;
11
12     while(fgets(linea, MAXLINE, arch))
13     {
14         sscanf(linea, FORM, &est.leg, &est.nombre, &est.prom);
15         printf("%d - %s - %.2f", est.leg, est.nombre, est.prom);
16     }
17
18     fclose(arch);
19     return 1;
20 }
```

- Lectura Directa al Archivo

Esta función nos permite ahorrar 2 pasos (fgets y sscanf) en uno solo mediante la función fscanf:

```
int fscanf(FILE *archivo, const char *formato, ...);
```

se utiliza para leer datos de un archivo, de acuerdo al formato especificado. Su comportamiento es similar a la función fscanf solo que recibe un archivo, no una cadena. Ejemplo:

```
1 #define FORMATO "%4d%23[^\\n]%-7f\\n"
2
3 int lecEstsFijo(const char* nomArch)
4 {
5     FILE* arch = fopen(nomArch, "rt");
6     ESTS est;
7
8     if(!arch)
9         return 0;
10
11    // Funcion fscanf
12    printf("\nFuncion fscanf:\n");
13    while(fscanf(arch, FORMATO, &est.leg, est.nombre, &est.prom) != EOF)
14        printf("%d %s %.2f\\n", est.leg, est.nombre, est.prom);
15
16    fclose(arch);
17    return 1;
18 }
```

Funciones de Tipo de dato Universal

- **Escritura**

- **Lectura**

Para esto hacemos uso únicamente de lectura de línea y la almacenamos en una cadena (**fgets**), esta se encontrará en memoria dinámica con un tamaño determinado el por usuario (**malloc**). Al recorrer cada línea y guardarla en cadena, realizamos una acción enviada como argumento.

```
1 int lecArch(char* nomArch, size_t tamano, char *tipoLec, void accion(void*))  
2 {  
3     FILE* arch = fopen(nomArch,tipoLec);  
4     char *linea = malloc(tamano);  
5  
6     // Verificamos la direccion dinamica  
7     if(!linea)  
8         return 0;  
9  
10    // Verificamos la apertura del archivo  
11    if(!arch)  
12    {  
13        free(linea);  
14        return 0;  
15    }  
16  
17    // Iteramos por cada linea del archivo con fgets  
18    while(fgets(linea, MAXLINE,arch))  
19        accion(linea); // Accion generica  
20  
21    fclose(arch);  
22    free(linea);  
23    return 1;  
24 }
```

Manipulación Archivos Binarios

Las funciones deben ser de uso global.

- **Escritura**

```
size_t fwrite(void *vec, size_t longitud, size_t tamanio, FILE *arch);
```

Hacemos uso principal de esta función fwrite. Retorna la cantidad de registros escritos. Recibe como parámetros:

1. *vec* – Puntero del vector
2. *longitud* – Cantidad de elementos del vector
3. *tamanio* – Tamaño de cada elemento
4. *arch* – Puntero del archivo

Ejemplo:

```
1 int crearArchBin(char *nombArch, void *vec, size_t longitud, size_t tamanio)
2 {
3     FILE *arch = fopen(nombArch, "wb");
4
5     if(!arch)
6         return 0;
7
8     // Escribe en el archivo todo a la vez
9     fwrite(vec, tamanio, longitud, arch);
10
11    fclose(arch);
12    return 1;
13 }
```

- **Lectura**

```
size_t fread(void *ptr, size_t tamanio, size_t cantidad, FILE *arch);
```

Hacemos uso principal de esta función fread. Retorna la cantidad de registros leídos. Recibe como parámetros:

1. *ptr* – Puntero de la dirección a guardar registro
2. *tamanio* – Tamaño del registro
3. *cantidad* – Cantidad de registros a leer
4. *arch* – Puntero del archivo

Ejemplo:

```
1 int leerArchBin(char *nombArch, const size_t tamanio, void accion(void*))  
2 {  
3     FILE *arch;  
4     // memoria dinamica para hacerlo de uso global  
5     void *buffer = malloc(tamanio);  
6  
7     // Verificar Memoria Dinamica  
8     if(!buffer)  
9         return 0;  
10  
11    arch = fopen(nombArch,"rb");  
12  
13    // Verificar Archivo  
14    if(!arch)  
15    {  
16        free(buffer);  
17        return 0;  
18    }  
19  
20    // En cada iteración transpasa al siguiente registro  
21    while(fread(buffer, tamanio, 1, arch) == 1)  
22        accion(buffer);  
23  
24    // Cerrar  
25    free(buffer);  
26    fclose(arch);  
27    return 1;  
28 }
```

Algoritmo Fusión Merge

- **Definición**

Este algoritmo es utilizado para funcionar/combinar 2 o más archivos o vectores en 1 solo, siempre que ambos cumplan ciertas características ambos archivos:

- Deben tener un campo en común, como un ID (identificador único)
- Deben estar ordenado por el campo por ID
- Deben compartir características similares para facilitar la fusión

Para ilustrar cómo es el algoritmo, se utilizarán estos archivos como ejemplo:

Productos.dat		
COD	DESCRIPCION	STOCK
00	Higo	100
04	Banana	100
07	Pera	50
09	Manzana	70
14	Durazno	90
16	Sandía	90

Movimientos.dat	
COD	CANTIDAD
01	20
07	20
07	-10
08	40
12	60
12	-10
15	80
15	40

<= Kiwi
<= Pera
<= Pera
<= Uva
<= Naranja
<= Naranja
<= Mandarina
<= Mandarina

- Productos.dat:
Información de productos como código, descripción y stock
- Movimientos.dat:
Registra movimientos de stock para los productos, especificando el código y la cantidad.

• Funcionamiento del Algoritmo

El objetivo es combinar ambos archivos en uno solo, donde se actualice el stock de los productos según los movimientos indicados y crear nuevos productos no registrados. Para ello, el proceso se divide en los siguientes pasos:

1. Ingresar Valores Principales (nombre archivos)

2. Realizar la Actualización o Creación de Registros

En cada iteración, se comparan los ID de los productos y los movimientos. Existen tres posibilidades para cada comparación:

- Producto = Movimiento:

Se actualiza el stock sumando la cantidad del movimiento al producto.

- Producto < Movimiento:

El producto no tiene un movimiento asociado, por lo que solo se almacena en el archivo auxiliar.

- Producto > Movimiento:

Se trata de un nuevo producto (existente en el archivo de movimientos, pero no en el de productos), que debe ser registrado en el archivo auxiliar con los datos del movimiento.

3. Se Llega al Final de un Archivo

Todos los cambios (actualizaciones y nuevos productos) se van almacenando en un archivo auxiliar hasta que se procesen ambos archivos completamente. Se puede llegar al final de un archivo u otro, cuando esto ocurre se debe decidir que hacer:

- Producto fin-> Movimiento Sigue o Finaliza algoritmo

Si se llega al final del archivo de productos, pero aún quedan movimientos, se agregarán los nuevos productos.

- Movimiento fin -> Producto Sigue

Si se llega al final del archivo de movimientos, pero aún hay productos, estos se copiarán tal cual al archivo auxiliar.

4. Reemplazar Archivo Original

Una vez finalizado el proceso, se reemplaza el archivo original de productos por el archivo auxiliar, removiendo y renombrando.

5. Actualización de Campos Restantes [opcional]

En caso de que existan otros archivos relacionados con los productos, estos pueden ser opcionalmente actualizados con los nuevos datos.

- **Resultado**

El archivo temporal resultante contendrá todos los productos, incluyendo aquellos con movimientos y los nuevos registros. Sin embargo, el archivo de movimientos no incluye los nombres de los nuevos productos:

Productos.tmp		
COD	DESCRIPCION	STOCK
00	Higo	100
01	Kiwi	20
04	Banana	100
07	Pera	60
08		40
09	Manzana	70
12		50
14	Durazno	90
15		80
16	Sandía	90

En este ejemplo, el archivo temporal "Productos.tmp" incluye todos los productos actualizados. Sin embargo, los nuevos productos no tienen nombres asignados. Para solucionar esto, es necesario utilizar un archivo adicional de nombres:

Nombres.dat	
COD	NOMBRE
01	Kiwi
07	Pera
08	Uva
12	Naranja
15	Mandarina

El archivo de nombres se procesará de forma similar al archivo de movimientos, y el resultado final será algo como esto:

Productos.tmp		
COD	DESCRIPCION	STOCK
00	Higo	100
01	Kiwi	20
04	Banana	100
07	Pera	60
08	Uva	40
09	Manzana	70
12	Naranja	50
14	Durazno	90
15	Mandarina	80
16	Sandía	90

• Ejemplo en Código

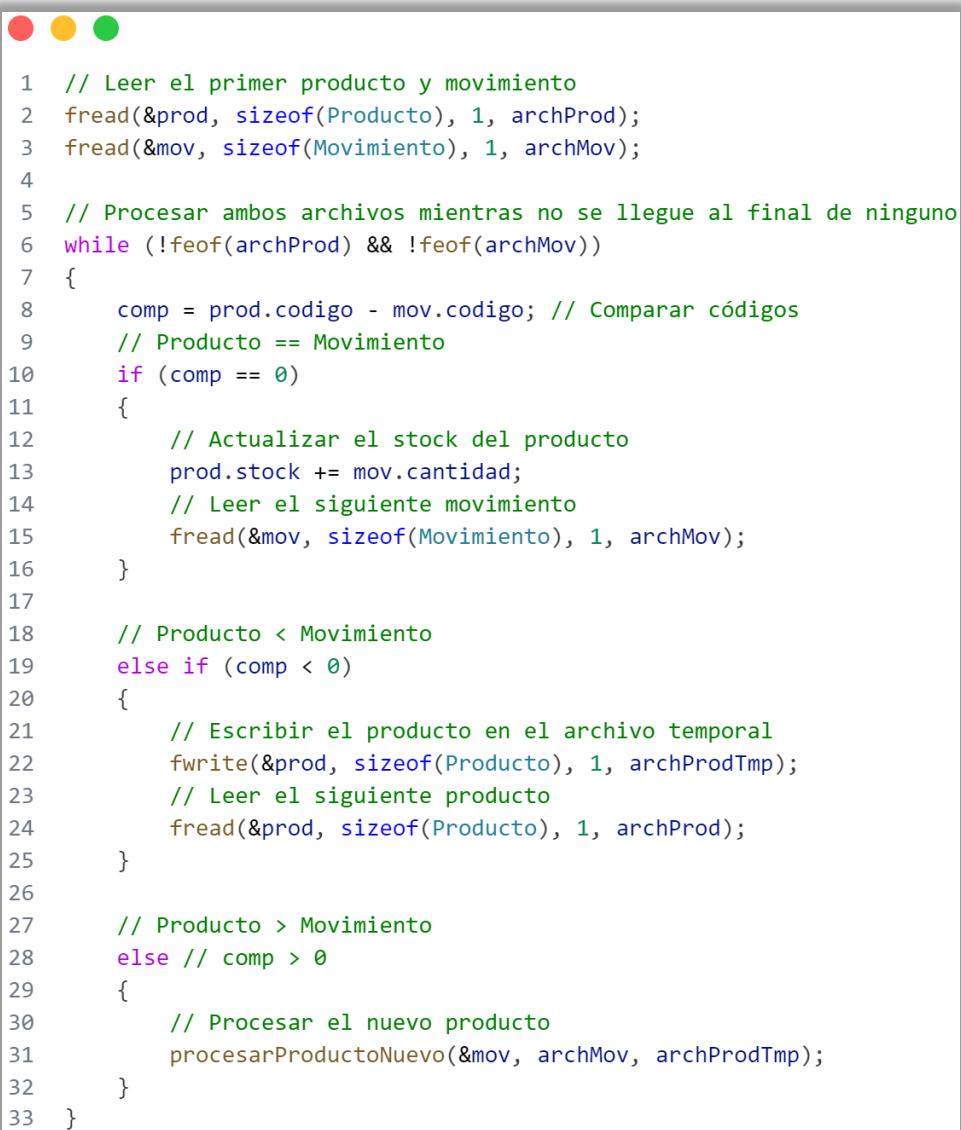
1. Ingresar Valores Principales

```
int actualizarStockProductos(const char* nomArchProds, const char* nomArchMov)
```

Tenemos una función que actualiza el stock de productos a partir de un archivo de productos y otro de movimientos.

No es necesario mostrar la implementación entera, pero si se dirá que se realizamos una apertura de archivos con 'archProd', 'archMov' y 'archProdTmp', y que se tiene las variables iniciales de prod (struct Prod), mov (struct mov) y comp (int).

2. Realizar la Actualización o Creación de Registros



```
1 // Leer el primer producto y movimiento
2 fread(&prod, sizeof(Producto), 1, archProd);
3 fread(&mov, sizeof(Movimiento), 1, archMov);
4
5 // Procesar ambos archivos mientras no se llegue al final de ninguno
6 while (!feof(archProd) && !feof(archMov))
7 {
8     comp = prod.codigo - mov.codigo; // Comparar códigos
9     // Producto == Movimiento
10    if (comp == 0)
11    {
12        // Actualizar el stock del producto
13        prod.stock += mov.cantidad;
14        // Leer el siguiente movimiento
15        fread(&mov, sizeof(Movimiento), 1, archMov);
16    }
17
18    // Producto < Movimiento
19    else if (comp < 0)
20    {
21        // Escribir el producto en el archivo temporal
22        fwrite(&prod, sizeof(Producto), 1, archProdTmp);
23        // Leer el siguiente producto
24        fread(&prod, sizeof(Producto), 1, archProd);
25    }
26
27    // Producto > Movimiento
28    else // comp > 0
29    {
30        // Procesar el nuevo producto
31        procesarProductoNuevo(&mov, archMov, archProdTmp);
32    }
33 }
```

```

1 // Función para procesar un nuevo producto a partir de un movimiento
2 void procesarProductoNuevo(Movimiento* mov, FILE* archMov, FILE* archProdTmp)
3 {
4     // Crear un nuevo producto con los datos del movimiento
5     Producto prodNuevo;
6     prodNuevo.codigo = mov->codigo; // Asignar el código del movimiento
7     prodNuevo.stock = mov->cantidad; // Inicializar el stock con la cantidad del movimiento
8
9     // Leer el siguiente movimiento
10    fread(mov, sizeof(Movimiento), 1, archMov);
11
12    // Mientras haya movimientos y el código sea igual al del nuevo producto
13    while (!feof(archMov) && prodNuevo.codigo == mov->codigo)
14    {
15        // Acumular la cantidad en stock
16        prodNuevo.stock += mov->cantidad;
17        fread(mov, sizeof(Movimiento), 1, archMov); // Leer el siguiente movimiento
18    }
19
20    // Escribir el nuevo producto en el archivo temporal
21    fwrite(&prodNuevo, sizeof(Producto), 1, archProdTmp);
22 }
23

```

3. Se Llega al Final de un Archivo

Luego de finalizar el bucle de verificación si se llegó al final de los archivos, se ejecuta:

```

1 // Escribir productos restantes que no tuvieron movimientos
2 while (!feof(archProd))
3 {
4     fwrite(&prod, sizeof(Producto), 1, archProdTmp);
5     fread(&prod, sizeof(Producto), 1, archProd); // Leer el siguiente producto
6 }
7
8 // Procesar los movimientos restantes que no tuvieron productos
9 while (!feof(archMov))
10 {
11     procesarProductoNuevo(&mov, archMov, archProdTmp);
12 }

```

4. Remplazar Archivo Original

```

1 // Reemplazar el archivo original de productos con el archivo temporal
2 remove(nomArchProds); // Eliminar el archivo original
3 rename("Productos.tmp", nomArchProds); // Renombrar el archivo temporal

```

Luego se cierran los archivos.

Unidad 6: Recursividad

Extra: Uso de Librería SDL

- **Instalación en CodeBlocks**

<http://libsdl.org/release/SDL2-devel-2.0.14-mingw.tar.gz>

Pasos a seguir:

Settings -> Compiler

1. Toolchain executables -> Additional paths -> Le agregamos la carpeta "bin"
2. Search directories -> Compiler -> Le agregamos la carpeta "include"
3. Search directories -> Linker -> Le agregamos la carpeta "lib"
4. Linker settings -> Other linker options -> Copiar el siguiente comando: `-lmingw32 -lSDL2main -lSDL2`

- **Funcionamiento**

<https://lazyfoo.net/tutorials/SDL/>