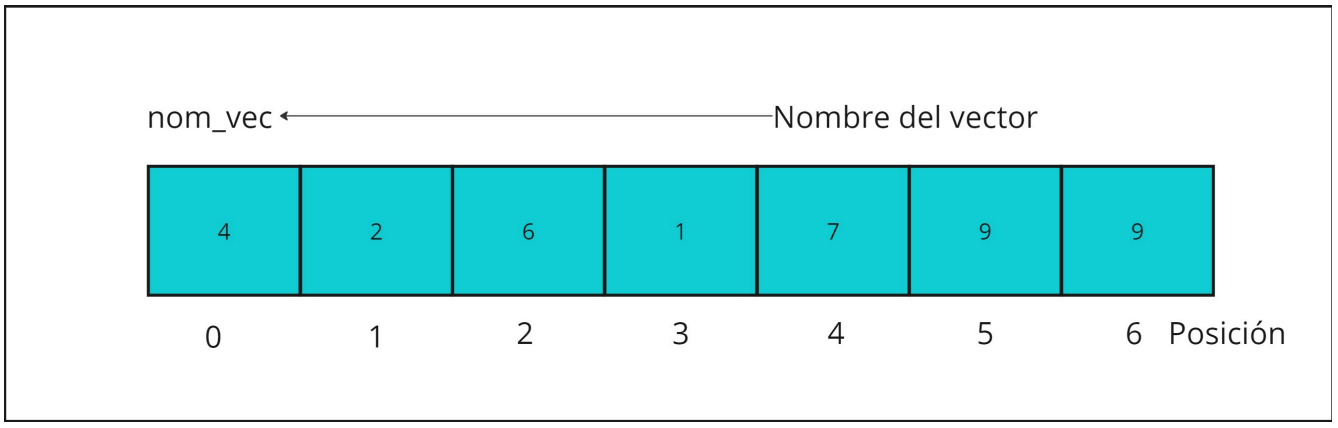
#C #github

---

2. Vectores:

Un **VECTOR** es C en un grupo de posiciones en memoria que están relacionadas entre sí porque tienen el **mismo nombre** y son del **mismo tipo.** Para referirse a una de estas posiciones o **elemento** del vector, se tiene que **especificar el nombre y el número de posición del elemento** particular dentro del mismo.



En la imagen ilustrativa el vector sería `nom_vec[7]`
`nom_vec` es un vector de enteros que tiene **7 elementos**, cualquiera de éstos elementos pueden ser referenciados escribiendo el nombre del vector y seguido entre `[]` la posición del elemento, EJ: `nom_vec[3]` que hace referencia al valor de la posición 3 del vector, o sea `6`.
El **primer elemento** de cualquier vector es el elemento de posición `0`.
El elemento de orden `i` se conoce como `nom_vec[i - 1]`. El numero que se encuentra dentro del `[]` se lo denomina **subíndice** y tiene que ser un **entero** o **una expresión entera**.
Si quisiéramos asignar dentro de un variable `x` el valor de un elemento de un arreglo modificado por una operación aritmética, sería algo así:
`x = nom_vec[3] * 4;`

2.1 ¿Cómo se declaran los vectores en C?

Como se mostró antes, para declarar un vector en C se tiene que **especificar el nombre del vector, el tipo y la cantidad de elementos que va a tener**.

¿Es obligatorio indicar la cantidad de elementos que va a tener un vector?

Cuando recién comenzamos a trabajar con C si tenemos que indicarlo, pero existen métodos para poder tener una lista de elementos que no tiene definido la cantidad de elementos, que vemos más adelante.
Es necesario indicar la cantidad de elemento porque C necesita reservar en memoria cuando se compilar el programa un bloque contiguo de momería para cada elemento que vaya a estar

dentro el arreglo.
¿Qué pasa si no declaramos el tamaño y solo ponemos `nom_vec[]` *y después en el resto de código le pedimos agregar elementos?* El programa compilado va a tener lo se denomina un **comportamiento indefinido**, donde capaz los elementos se sobrescriben en direcciones de memoria que no debería, errores de segmentación y datos corruptos. El programa tendría un problema fatal.

Volviendo a como se define un vector en C, el siguiente es un ejemplo:

```
#inlcude <stdio.h>

int main(){
        int nom_vec[5] = {4,6,3,2,6} // Se declaran los elementos del arreglo.
        int i;

        // Mostrar los elementos del vector:

        for (i=0; i<5; i++){
                printf("El elemento de posición %d es el %d!", i, nom_vec[i]);
        }

        return 0;
}
```

2.2 Operaciones con vectores:

Con los vectores se puede:

- **Cargar.**
- **Recorre**
- **Mostrar.**
- **Buscar un elemento.**
- **Desplazar.**
- **Intercambiar.**
- **Ordenar.**

2.2.1 Cargar elementos:

Cargar un vector de n1 elementos, siendo n1 a lo sumo 50:
Para cargar un vector se usa una instrucción repetitiva, por ejemplo `for`.
Si no se sabe exactamente la cantidad de elementos que va a tener el vector, lo declaramos con la cantidad máxima de elementos posible. En este caso 50:
`int vec[50];`

La cantidad de elementos que va a tener la ingresamos por teclado. Después validamos que no se esté superando la cantidad máxima.

```
do {

        printf("Ingrese la cantidad real de elementos");

        scanf("%d", &n1);

        } while (n1 > 50 || ni <1);
```

El código funcional completo sería:

```
#include <stdio.h>

int main(){

        int vec[50];
        int n1, i;

        do {

                printf("Ingrese la cantidad real de elementos");

                scanf("%d", &n1);

        } while (n1 > 50 || n1 <1);

        for (i=0; i < n1; i++){

        printf("Ingrese el componente %d", i);
        scanf("%d%, &vec[i]);
        }
}
```

2.2.2 Recorrer el vector y sus elementos:

Se usa generalmente un ciclo repetitivo `for` para recorrer el vector y sus elementos.
Por ejemplo su queremos sumar el total de los elementos del vector, el código sería:

```
for (i = 0; i < n1; i++){


```

```
        sum += vec[i];
}
```

Obviamente, en el código completo se definió la variable `sum` como `int` y `sum = 0`.

2.2.3 Mostrar los elementos de un vector:

Para imprimir en pantalla también se suele usar el `for` porque hay que recorrerlo para mostrar sus elementos:

```
for (i=0; i < n1; i++){
        printf("vec[%d] = %d \n", i, vec[i]);
}
```

O también se puede hacer usando tabulaciones:

```
...
printf("%8s %13s \n", "elemento", "valor");
for (i=0; i <n1; i++){
        printf("%8d %13d \n", i, vec[i]);
}
```

2.2.4 Buscar un elemento dentro de un vector:

Como es posible que podamos encontrar el elemento que estamos buscando antes de terminar de recorrer todo el vector, no es práctico hacerlo con un `for`. Es preferible usar en este caso un `while` ya que nos permite comparar cada elemento con una condición y cortar la búsqueda una vez que lo encontremos.
La condición del `while` tiene que ser doble, ya que el elemento que buscamos podría también no estar dentro del vector. Tendría que haber una condición por si lo encuentra y otra para ver si la posición no llegó al último elemento.

Por ejemplo, para decir en qué posición está el elemento cuyo valor es 10:

```
// Resto de código

n=0;
while (vec[n] !=  10 && n < n1){

        n++;
}
if (n == n1){
        printf("El valor 10 no se encuentra!");
} else {
```

```
        printf("El valor 10 está en la %d posición", n);
}
```

Una ve que el código sale del `while` doble, tenemos que preguntar por cuál de las dos condiciones salió y responder según corresponda.

2.2.5 Desplazar los elementos de un vector:

Para **desplazar los elementos** de un vector, podemos hacerlo hacia **adelante** (a la **izquierda**) o hacia **atrás** (a la **derecha**).

Si lo hacemos hacia **adelante** un lugar, **se pierde el primer valor**, ya que en la posición `0` quedaría el valor que estaba en la posición `1`, y así sucesivamente hasta llegar a la última posición que se quiere desplazar. Sin embargo, el valor que está en la **última posición** se **repite** en la posición anterior, lo que significa que en muchos casos se debe aclarar qué hacer con esa posición. Por ejemplo, se podría **reemplazar** el valor por `0`.

```
...
for (i==1; 1<=posfinal; i++){

        vec[i-1] = vec[i];
} // Si pide que remplace por 0 se pondría:
        vec[posfinal] = 0;
```

El `for` **empieza en** `1` porque se asigna a la posición `i - 1`, y si `i` valiese `0`, **quedaría en la posición** `-1`, que **NO EXISTE**. Si el desplazamiento es hasta la **posición final**, se copia el valor de dicha posición.

Si el desplazamiento es de más de un lugar, solo varía en cuánto le **restemos a** `i` y en el **valor inicial** del `for`.

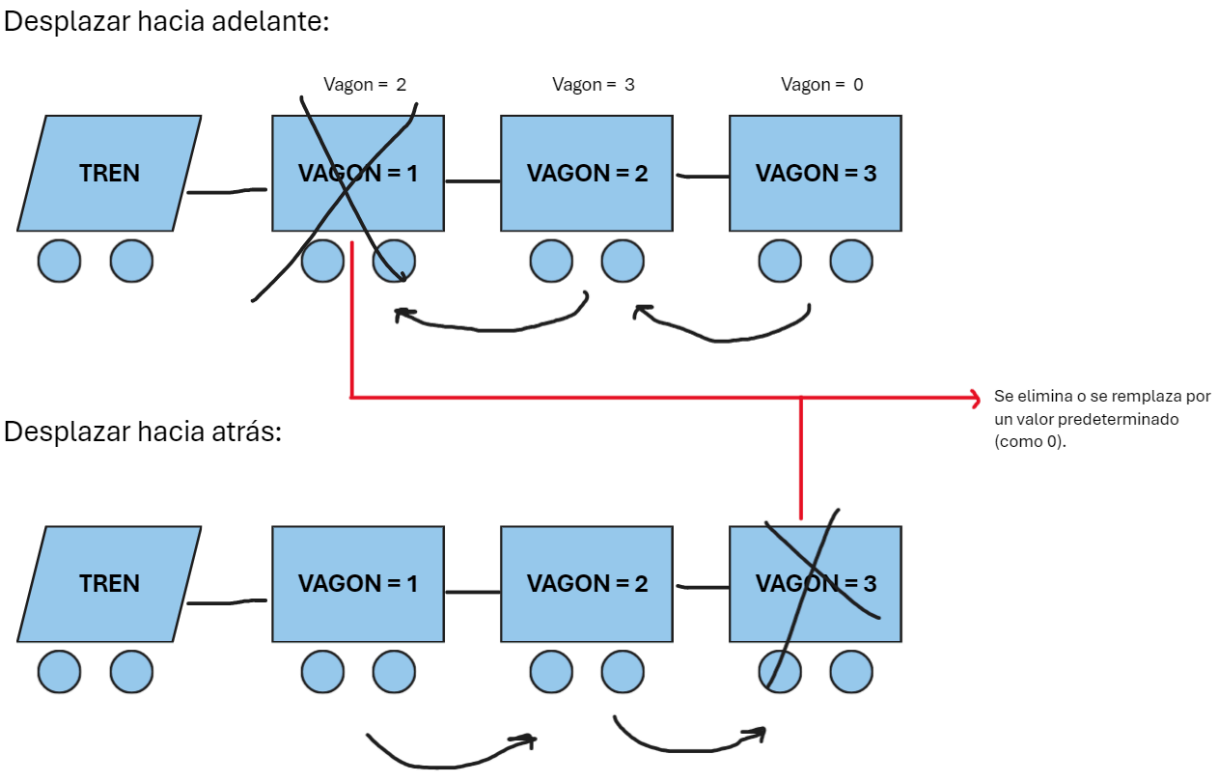
Si lo hacemos hacia **atrás un lugar**, se **pierde el último valor**, ya que en la posición `posfinal` quedará lo que estaba en la **posición anterior** a esa, y así sucesivamente hasta la **primera posición** que se quiere reemplazar. Pero el valor que está en la **primera posición** se **repite** en la siguiente, por lo que muchas veces se debe aclarar qué hacer con esa posición, como por ejemplo, **reemplazarlo por** `0`.

En este caso, hay que tener cuidado con el `for`, porque si incrementamos la variable, es decir, si asignamos desde la **primera posición** hasta la **última**, **perderíamos todos los valores** y acabaríamos repitiendo el primer valor a lo largo de todo el intervalo. Para **EVITAR ESTO**, tenemos que recorrer el `for` **de atrás hacia adelante**. Entonces:

```
for (i = posfinal -1; i>= posinicial; i--){
        vec[i + 1] = vec [i]
}
```

```
// Si pide que remplace por 0 se pondría:
        vec[posinicial] = 0
```

El `for` comienza en `posfinal -1` porque se le asigna a la posición `i + 1` y si `i` valiese `posfinal` quedaría en una posición más que el máximo del vector que NO EXISTE.



2.2.6 Intercambiar los valores de los elementos de un vector:

Para **intercambiar dos elementos** de un vector, se deben conocer las dos posiciones que se quieren intercambiar. Supongamos que queremos intercambiar `pos1` con `pos2`.

Si hacemos esto: `vec[pos2] = vec[pos1];`, el valor en `pos2` se cambia correctamente, pero **el valor original de** `pos2` **se pierde**, por lo que **no puede intercambiar nada** y `pos1` **se queda con su valor original**.

Para **INTERCAMBIAR CORRECTAMENTE**, es necesario declarar una **VARIABLE AUXILIAR** para guardar en ella el valor de `pos2` **antes** de hacer la asignación. Entonces, el código sería:

```
aux = vec[pos2]l
vec[pos2] = vec[pos1];
vec[pos1] = =aux;
```

2.2.7 Ordenar los elementos de un vector:

Para ordenar un vector sobre sí mismo hay varios métodos. Hay uno que se llama **BRUBUJEO**, funciona así:
Ordenar un vector de n1 elementos en forma ascendente.

```
...
int i,j,aux;

for (i=0; i<n1 -1; i++){
        for (j = i + 1; j < n1; j++){

                if (vec[i] > vec[j]){

                        aux = vec[i];
                        vec[i] = vec[j];
                        vec[j] = aux;
                }
        }
}
```

2.3 Argumentos de funciones con Vectores:

Para pasar un **vector como argumento** en la llamada de una función, es necesario **especificar el nombre del vector sin los corchetes**.

Ejemplo:

Si se declara `int vec[10];`, la llamada a la función sería: `x = maximo(vec, n1);`. Acá, se pasa `vec`, que es el **nombre del vector**, y `n1`, que es la cantidad de **elementos reales**. **Se pasa el VECTOR y el TAMAÑO a la función.**

En **C**, los vectores se pasan a las funciones mediante **SIMULACIÓN DE LLAMADAS POR REFERENCIA**. Esto significa que las funciones pueden **modificar los valores** de los elementos en los vectores originales.

El **nombre del vector** representa la **dirección del primer elemento**.

Si no se desea pasar el vector completo, sino un **elemento específico del vector**, se tiene que pasar **por valor**, es decir, indicando el **nombre del vector** seguido del **subíndice** o **posición del elemento**. Ejemplo: `pepe(vec[3]);`.

En el prototipo y desarrollo de la función, si un argumento es un vector, se debe **especificar con su tipo, nombre y los corchetes**. Para que una función reciba un vector, la lista de parámetros debe indicar que se recibirá un vector, por ejemplo: `int maximo(int vec[], int n1)` o `int maximo(int vec[], int)`. **No es necesario poner el tamaño del vector.**

EJ:

```
#include <stdio.h>

void cargar(int x[], int n){
```

```
        int i;
        for (i=0; i < n; i++){
                printf("\n Ingrese el elemento %d del vector",i);
                scanf("%d", &x[i]);
        }
        return;
}

void mostrar(int x[], int n){
        int i;
        printf("ELEMENTO VALOR");
        for (i=0; i < n; i++){
                printf("%8d %8d", i, x[i]);
        }
        return;
}

int main(){
        int nom_vec[5], tam=5; // Declaramos el vector y otra variable que
indique el tamaño

        cargar(nom_vec, tam);

        mostrar(nom_vec, tam);

        return 0;
}
```

---

2. Funciones:

Una ***FUNCIÓN*** es un **pequeño programa, un trozo de código que está dentro de un programa**. Las funciones tienen varias sentencias bajo un solo nombre y existen de forma autónoma, cada una con su propio ámbito. La división del código en funciones permite que se puedan **reutilizar** tanto en el programa actual como en otros.

Para reutilizar una función dentro de un programa, solo hay que hacer un "**llamado**" a esa función. Si se agrupan funciones en bibliotecas, otros programas pueden acceder a ellas y usarlas.

En **C**, las funciones no pueden anidarse, es decir, **no se puede declarar una función dentro de otra función**.

SINTAXIS:
`TIPO_DE_RETORNO NOMBRE (TIPO 1 ARG 1, TIPO 2 ARG2....)`

Una **estructura de función es:**

```
tipo_de_retorno nombre(lista de parámetros){

        cuerpo de la función
        return expresión;
}
```

2.1 Resultados de una función:

Una función puede devolver un única valor. El resultado se muestra con una sentencia de `return;`.

2.2 Llamada a una función:

Las funciones para ejecutarse se tienen que "llamar" primero.
La función llamada que recibe el control del programa se ejecuta desde el principio y cuando termina.
Dentro del `return` lo que se ponga, el tipo de la expresión TIENE QUE SER EL MISMO QUE SE DEFINE COMO TIPO DE SALIDA O RETORNO en la definición de la función.
Una función NO NECESITA DEVOLVER UN VALOR, un `return` sin expresión hace que el control regrese al programa sin pasar ningún valor al programa principal. En este caso, las funciones que no regresan nada se las determina como `void`.

Ej:
Dados dos números, mostrar el mayor:

```
void maximo(int x, int y){

        if (x>y){
                printf("El valor maximos es %d", x);
        }
        else {
                        printf("El valor maximo es %d", y)
        }
        return;
}
```

Ej2:
Dado un número natural, mostrar su factorial:

```
long fac (int n){

        int i;
        long prod =1;
        if (n>1){
                for (i=2; i<n; i++){
                        prod *= i;
                }
        }
        return prod;
}
```

Ej3:
Calcularla suma de n números reales:

```
float suma(int n){

        int i;
        float sum = 0.0, x;
        printf("Introduce &d números reales: ", n);

        for (i=0; i<n; i++){
                scanf("%f", &x);
                sum += x;
        }

        return sum;
}
```

2.3 Hacer uso de las funciones en un programa:

Si se usan funciones dentro de un programa, éstas tienen que estar en 3 formas distintas:

- **La primera es el prototipo**.
- La segunda es la llamada de la función dentro del `main()`.
- Y la tercera es la definición y desarrollo.

La primera es la definición de la función, los prototipos tienen la cabecera de una función pero terminan con punto y coma. Se escriben normalmente al principio de un programa, antes de la definición del `main()`. Cuando una función se declara se da su nombre y la lista de sus parámetros.
Cuando se define significa que existe en un lugar en un programa donde existe la función desarrollada.
Los nombres de los argumentos en los prototipos no tienen significado, son independientes.

Los nombres de los argumentos formales o locales no tienen porqué coincidir con los de los reales.
El tipo de dato del argumento local tiene que ser igual que al del tipo de dato del argumento real que recibe desde el punto de llamada.

Ejemplo de prototipo:

```
#include <stdio.h>

long fac(int n);
void maximo(int x, int y);

int main(){

        ...
}
```

La segunda es la llamada a la función dentro del `main()`. Cuando se llamada a la función dentro del programa principal los argumentos se llaman argumentos reales, ya que definen la información real que se transfiere.
En la llamada los argumentos van si su tipo, porque fueron definidos al inicio del programa.
Los argumentos reales son variables que se usan en el programa o valores constantes.
La llamada a la función puede ser de 2 formas distintas, según retornen o no un valor.

1. Si la función no regresa ningún valor en la llamada solo se nombre la función.
2. Si la función regresa un valor, en la llamada tiene que haber una asignación del valor que regresa a otra variable del mismo tipo.

EJ:

```
#include <stdio.h>

void mostrar(int n, int res);

int main(){

        int n, res;
        ...
        mostrar(n, res);
        ...
}
```

Ej2:

```
#include <stdio.h>

int suma(int a, int b);

int main(){

        int n1, n2, c;
        ...
        c = suma(n1,n2);
}
```

En el prototipo se puede no poner el nombre de las variables, pero SI ES OBLIGATORIO EL TIPO Y RESPETAR EL ORDEN:
`int power(int, int);`
Si no se nombran parámetros, se toman como `int`:
`int power();`

La tercera es la definición y desarrollo. Al terminar el `int main()` se hace el desarrollo de la función con su definición, su cuerpo y su `return`.
La declaración prototipo y la definición de la función tienen que coincidir en cantidad de parámetros y formatos y el orden en que se enumeran. Los nombres de los parámetros no siempre coinciden, son optativos.
La función que se llama no puede alterar directamente una variable de la función que se hace la llamada, sólo puede modificar su copia temporal. Es decir, si entra una variable del programa principal y se modifica en la función, al volver no llega modificada al programa principal su la función es de tipo `void`.

Ej:
Escribir a 2 columnas las potencias de 2 y de 3 con exponentes 0 al 9:

```
#include <stdio.h>

int power(int m, int n); // Función prototipo, después se desarrolla.

int main(){

        int i, a, b;
        for (i=0; i<10; i++){
                if (i==0){
                        a = 1;
                        b = 1;

                }
                else {
                        a = power(2, i);
```

```
                        b = power(3, i);
                }

        printf("%3d %6d %6d \n", i, a, b);
        } // Fin del programa principal.


FALTA CODIGOOOOOOOOOOOOOO
}
```

Cada llamada pasa 2 argumentos a `power()` y cada vez regresa un entero `p`.