LENGUAJE C: DATOS ESTRUCTURADOS

- Tienen una estructura interna, con elementos que no son un únicos.
- **Vectores y matrices (*arrays*)**: colección ordenada de objetos **del mismo tipo** que comparten el mismo nombre (identificador). Los elementos individuales del array se identifican utilizando un **índice** que indica su posición y que permite su acceso. En función de su dimensión se los denomina:
  - **Vector** = una sola dimensión.
  - **Matriz** = dos (**tabla** con filas y columnas) o mas dimensiones.
- **Cadenas de caracteres**: array de tipo char. Recordar que la dimensión siempre debe tener en cuenta que termina en '\0'. Librería para su manejo = **string.h**.
- **Estructuras** (también llamadas **registros**): permite agrupar con un mismo nombre datos de igual o **distinto tipo** según la necesidad del usuario. Se crea un nuevo tipo de dato que puede ser utilizado como los datos simples, incluso formando arreglos de estructuras.

VECTORES

- La **declaración** de una variable de tipo array se realiza indicando el tipo de dato que almacena, su nombre y la dimensión (cantidad de datos) entre corchetes:

```
int  iVec[100];          /* Un vector de 100 enteros (del 0 al 99) */
float  fValor[40] ;      /* Un vector de 40 flotantes  (del 0 al 39) */
```

- Los índices de un array comienzan por el elemento **0** y terminan en el ***n*-1**.

- Los elementos se pueden inicializar al momento de la declaración:

```
float  fValor[3] = {2.45, -1.34, 2.11};     int iValor[ ] = {-2, 368, 17, -28, 99};
```

- A cada elemento se accede mediante su índice:

```
int val[ ] = {0, 1, 2, 3, 4, 5};  ⟶
val[0] = 23;  ⟶
val[3] = val[0] + 5;  ⟶
for(int i=0; i<6; i++)
{
      printf("%d", val[i]);        /* imprime todos los valores de val[] */
}
```

0	1	2	3	4	5
23	1	2	3	4	5
23	1	2	28	4	5
val[0]	val[1]	val[2]	val[3]	val[4]	val[5]

Ej. 01: vector

ARRAYS – ARREGLOS

- Se debe tener cuidado con el manejo de los índices porque las variables se guardan en posiciones sucesivas de memoria.

```
#include <stdio.h>
int main()
{
    char num[] = {'4','8','5','6','3','2'};        // char quedan en bytes contiguos
    int aux=13;                                    // inicializo con un valor
    int i;
    num[6] = 100;                  // índice > dimensión
```

34	38	35	36	33	32	0D	00
num[0]	num[1]	num[2]	num[3]	num[4]	num[5]	aux	

ARRAYS – ARREGLOS

- Se debe tener cuidado con el manejo de los índices porque las variables se guardan en posiciones sucesivas de memoria.

```
#include <stdio.h>
int main()
{
    char num[] = {'4','8','5','6','3','2'};        // char quedan en bytes contiguos
    int aux=13;                                    // inicializo con un valor
    int i;
    num[6] = 100;                  // índice > dimensión ==> INVADE MEMORIA!!!
    printf("\nCambio el valor de 'aux' sin saberlo, ahora vale %d\n\n", aux);
    for(i=0; i<=6; i++)            // se imprimen mas valores que la dimensión !!!
       printf("numeros[%i] = %i \n", i, num[i]);
    return 0;
}
```

Ej. 2: vector_memoria

34	38	35	36	33	32	~~D~~ 64	0
num[0]	num[1]	num[2]	num[3]	num[4]	num[5]	aux	

num[6] num[7]

ARRAYS

- **No se puede trabajar con un arreglo como si fuera un un bloque**.
- Los datos de los arrays se deben procesar **elemento por elemento**.
- **No** se puede **asignar** un array completo a otro, se produce un error de compilación.
- **No** se pueden **comparar directamente** arrays completos, lo que se compara son las direcciones del primer elemento de cada uno.
- **No** se puede **imprimir directamente** un arreglo completo, se imprime la dirección del primer elemento.
- **No** se puede **mostrar el valor de todos los elementos** del array en forma conjunta. Se debe proceder elemento por elemento.
- Los elementos del arreglo se deben **leer de uno en uno**.

STRING: ASIGNACION DE VALORES

- En la declaración:

```
char  cadena1[5] = "hola";
char  cadena2[ ] = "hola";
char  cadena3[5] = { 'h', 'o', 'l', 'a', '\0' };
char  cadena4[ ] = { 'h', 'o', 'l', 'a', '\0' };
```

- **No se puede hacer** lo mismo si no es en la declaración:

```
cadena = "hola"          // error en la compilación
cadena1 = cadena2        // error en la compilación
```

- **Se puede asignar elemento por elemento**, sin olvidar el **carácter nulo**:

```
cadena [0] = 'h';
cadena [1] = 'o';
cadena [2] = 'l';
cadena [3] = 'a';
cadena [4] = '\0';
```

Ej. 03: vector_ingreso

- Existe una librería dedicada al manejo de cadena de caracteres = **string.h**

MATRICES BIDIMENSIONALES

- La **declaración** se realiza indicando el tipo de dato que se almacenan, el nombre del arreglo y ambas dimensiones entre corchetes. Cada elemento se accede a través de 2 índices: **fila** y **columna**.

  tipo_datos *nombre*[número_filas][número_columnas];

- Los índices comienzan por el elemento **0** y terminan en el ***n*-1**.

Ejemplos:

```
int  iMatriz[10][10];      /* matriz de 100 enteros (10 filas y 10 columnas)  */
float  fValor[4][6];       /* matriz de 24 flotantes  (4 filas y 6 columnas) */
```

columna → 0 1 2 3 4 5

fila ↓

0	[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]	fValor[0][]
1	[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]	fValor[1][]
2	[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]	fValor[2][]
3	[3][0]	[3][1]	[3][2]	[3][3]	[3][4]	[3][5]	fValor[3][]
	fValor[][0]	fValor[][1]	fValor[][2]	fValor[][3]	fValor[][4]	fValor[][5]	

MATRICES BIDIMENSIONALES

- Los elementos de un array pueden ser de cualquier tipo, incluido otro array.
- En C una matriz de dos dimensiones puede considerarse como un vector cuyos elementos son a su vez vectores.

  Ejemplo: int pantalla [800] [600];

  Puede verse como un vector de 800 elementos, en los cuales cada uno de ellos es un vector de 600 elementos. El contenido de los mismos es un entero.

- No pueden imprimirse ó asignarse todos los valores de los elementos como si fuera un bloque, debe recorrerse la matriz elemento por elemento:

```
float matriz [filas] [columnas];
for (int i=0; i<filas; i++)
    for (int j=0; j<columnas; j++)
        printf("%f", matriz[i][j]);        /* imprime el valor de cada elemento */
ó       scanf("%f", &matriz[i][j]);        /* ingresa cada uno de los valores */
```

MATRICES BIDIMENSIONALES

- Los elementos se pueden inicializar al momento de la declaración:

```
int  iValor[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10,  11, 12} };
```

- **No** se pueden dejar ambas dimensiones sin definir, **sólo la primera**.

```
int  iValor[][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10,  11, 12} };    // permitido
int iValor[][]={{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10,  11, 12}};  // error de compilación
```

- A cada elemento se accede mediante su índice: (imprimir o ingresar)

```
scanf("%d",  &iValor[2][3] );
printf("Valor de la fila 2, columna 3 = ", iValor[2][3]);
```

- El almacenamiento se realiza en posiciones consecutivas de memoria:

```
short int  iMat[2][3]={{416,243,85},{268,600,9}};  int  iNum= 99;  char c='A';
```

A0	01	03	0F	55	00	0C	10	58	02	09	00	63	00	00	00	41	…
[0][0]		[0][1]		[0][2]		[1][0]		[1][1]		[1][2]		iNum				'A'	

- Si se utilizan índices mayores a los declarados se produce **invasión de memoria**.

Ejs. 4 y 5: matriz y matriz_invade

MATRICES MULTIDIMENSIONALES

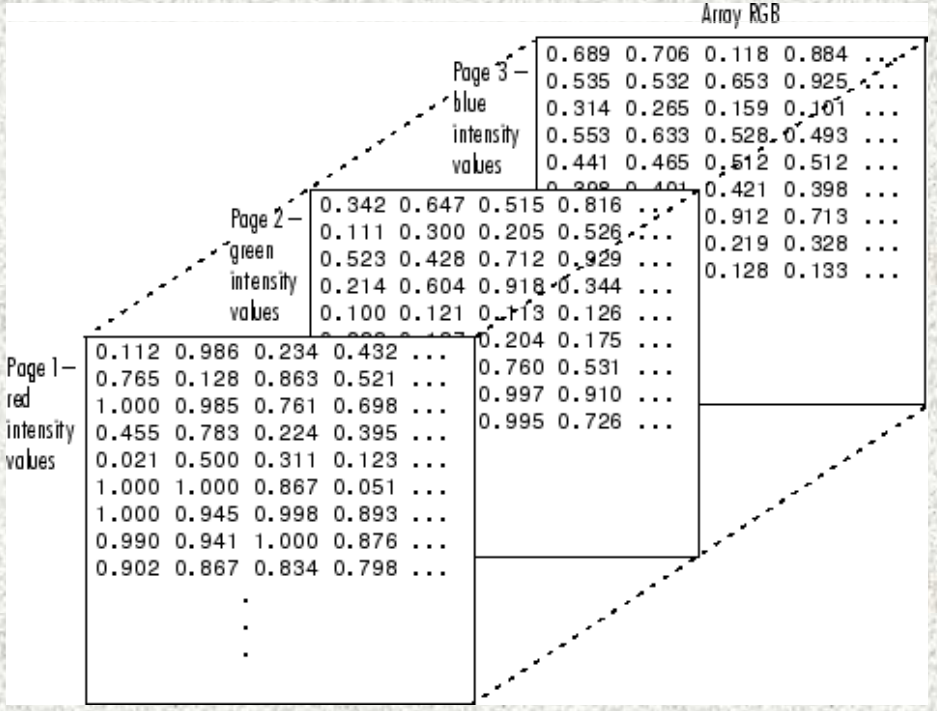
- Los arreglos pueden tener más dimensiones, considerando un índice adicional por cada dimensión que se requiere. Una matriz de 3 dimensiones puede considerarse como una 2D donde cada elemento es un vector con tantos componentes como indica el tercer índice. Se procede análogamente para más dimensiones.

- En una pantalla color, se puede considerar que la primer dimensión corresponde a uno de los colores básicos (RGB = Red Green Blue) y las 2 siguientes representan la posición del pixel (fila, columna).

  Ejemplo: color[1][10][20] = 0.5; fija el color verde en la mitad de su máxima intensidad en x=10 y=20.

  Para poner ese pixel en rojo:

  ```
  color[0][10][20] = 1.0;   // rojo
  color[1][10][20] = 0.0;   // verde
  color[2][10][20] = 0.0;   // azul
  ```



MATRICES MULTIDIMENSIONALES

- Los elementos se pueden inicializar al momento de la declaración:

1. /** almacena los valores en forma consecutiva comenzando por la última dimensión **/
```
short  valores[2][3][4] = {100,101,102,103, /**/104,105,106,107,  /**/108,109,110,111,
                           200,201,202,203, /**/204,205,206,207, /**/208,209,210,211};
```

2. /** almacena los primeros valores de cada fila, cada '{' abre una dimensión **/
```
short  valores[2][3][4] = { { { 100},                    // fila 1 sólo 1 valor
                              { 110, 111},               // fila 2 sólo 2 valores
                              { 120, 121, 122, 123} },   // fila 3 completa
                            { { 200, 201, 202, 203},     // segunda página
                              { 210},
                              { 220, 221, 222, 223} } };
```

3. /** almacena sólo valores específicos (útil si hay muchos nulos) **/
```
short  valores[2][3][4] = { { [0][0]=100, [1][1]=105, [2][2]=110 },    // página 0
                            { { 200, 201, 202, 203},       // página 1, fila 0
                              { 210},                      // página 1, fila 1
                              { 220, 221, 222, 223} } };   // página 1, fila 2
```

Ej. 6: matriz_3d

VECTORES COMO PARAMETROS

- Todos los arrays se pasan siempre por **referencia** (la función puede modificar los valores). Cuando se pasa un vector a una función lo que se está pasando es la dirección de su primer elemento.

- Declaración: se indica el tipo de los datos y el nombre del vector añadiendo [].

```
int suma_datos(int datos[]);        // no especifica el tamaño, nombre opcional
```

- Llamada: se pasa el nombre del vector sin indicar tamaño ni los corchetes.

```
int  datos[100];                    // reserva de memoria
  ⋮
suma = suma_datos(datos);           // llamada a la función
```

- Para indicar que la función no puede modificar el vector que se le está pasando como parámetro se utiliza el modificador **const:**

```
int suma_datos(const int datos[]);        // suma_datos no puede modificar datos
```

VECTORES COMO PARAMETROS

- Forma 1: utilizando #define para especificar el tamaño.

```
#define   TAM     5                          // tamaño del vector
void imprimir_vector(short int vector[]);    // DECLARACION FUNCIONES
void leer_vector(short int []);              // también es una forma válida, no tiene el nombre

int main()
{      ⋮
       short int  iValores[TAM];             // declaración  del vector
       leer_vector(iValores);                //  llamada a las funciones
       imprimir_vector(iValores);
         ⋮
}
void leer_vector(short int vector[])         // DEFINICION FUNCION
{      ⋮
      for(int cnt=0; cnt<TAM; cnt++)
         ⋮
}
void imprimir_vector(short int vector[])     // DEFINICION FUNCION
{      ⋮
}
```

Ej. 7a: vector_parametro

VECTORES COMO PARAMETROS

- Forma 2: utilizando otro parámetro para especificar la longitud.

```
void imprimir_vector(short int vector[], int longitud);    // DECLARACION FUNCIONES
void leer_vector(short int [], int);        // también es una forma válida, no tiene los nombres

int main()
{      ⋮
       short int  iValores[5];                          // declaración  del vector
       leer_vector(iValores, 5);                        //  LLAMADA A LAS FUNCIONES
       imprimir_vector(iValores, 5);
         ⋮
}
void leer_vector(short int vector[], int longitud)          // DEFINICION FUNCION
{      ⋮
      for(int cnt=0; cnt<longitud; cnt++)
         ⋮
}
void imprimir_vector(short int vector[], int longitud)    // DEFINICION FUNCION
{      ⋮
      for(int cnt=0; cnt<longitud; cnt++)
 }
```

Ejs. 7b y 8: vector_parametro y vector_const

MATRICES COMO PARAMETROS

- A diferencia de los vectores, es imprescindible indicar explícitamente el tamaño de cada una de las dimensiones, excepto el de la primera que se puede indicar o no.
- Declaración: se indica el tipo de los datos, el nombre del array y sus dimensiones.

```
int suma_datos(int datos2[10][20]);       // especifica el tamaño de filas y columnas
int suma_datos(int datos3[10][20][5]);    // especifica todos los tamaños
int suma_datos(int datos2[][20]);         // especifica sólo el tamaño de las columnas
int suma_datos(int datos3[][20][5]);      // se omite la primer dimensión
```

- Llamada: se pasa el nombre del array sin indicar tamaño ni los corchetes.

```
int  datos[100][100][2];                  // reserva de memoria
  ⋮
suma = suma_datos(datos);                 // llamada a la función
```

- Definición: se procede igual que con los vectores, o se utiliza #define (para cada dimensión) o se pasan los tamaños como parámetros adicionales.

Ejs. 9 y 10: matriz_parametro y matriz3d_parametro

ARREGLOS COMO PARAMETROS

- Recordar:

  - Para trabajar con funciones que reciben arrays como parámetros es necesario indicar explícitamente cada una de las dimensiones, excepto la primera.

  - Se presenta un problema cuando se requiere que el tamaño del arreglo varíe de acuerdo a las necesidades del programa que se está ejecutando. Cada dimensión es una variable que puede ser modificada por el usuario.

Solución: **manejo dinámico de memoria** ➔ **PUNTEROS**.