

# Programación 1

Tema 6. Tipos de datos estructurados: Arrays

**Grado en Ingeniería Informática**

# Objetivos / Competencias

2

1. Comprender la diferencia entre tipos de datos simples y estructurados
2. Conocer los tipos de datos estructurados array unidimensional y bidimensional
3. Aprender a manejar arrays de una y dos dimensiones en lenguaje C

1. Tipos de datos estructurados
2. El tipo *array*
3. Arrays unidimensionales
4. Arrays bidimensionales
5. Definición de tipos con `typedef`
6. Fuentes de información

# Recordatorio: Tipos de datos Simples

4

- ▣ Todas las variables con las que hemos trabajado hasta ahora son de tipo simple
- ▣ Una variable de tipo simple solamente puede contener un valor cada vez
- ▣ Ejemplo: si **x** es de tipo entero, podemos asignar a **x** un sólo valor cada vez:

```
x= 7;
```

```
x = 10;
```

```
cin >> x;
```

```
...
```

# Tipos de datos estructurados

5

- ▣ Una variable de tipo estructurado consiste en una colección de datos de tipos simples
- ▣ Un tipo estructurado puede almacenar más de un elemento (valor) a la vez

## ■ Tipo Array

- Todos los elementos que almacena una variable de tipo array deben ser del mismo tipo

## ■ Tipo Registro

- una variable de tipo registro puede almacenar elementos de distinto tipo
- ▣ Ejemplo: consideremos una variable **z** que almacenará los números premiados en la bonoloto. Por lo tanto almacenaremos 6 valores cada vez

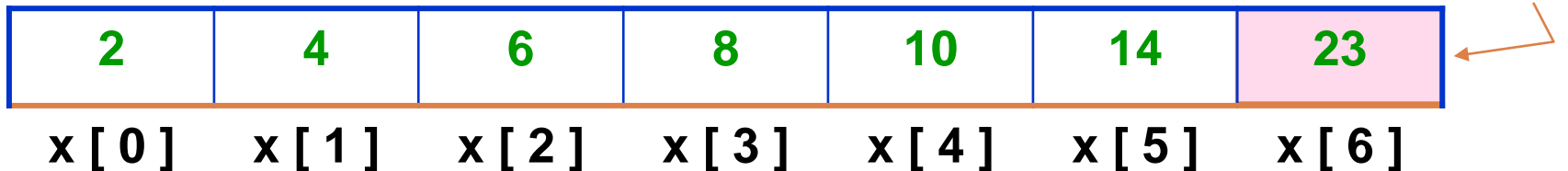
**z** = (1, 4, 6, 24, 13, 2);

**z** = (3, 9, 12, 15, 23, 27);

# El tipo *array*

6

- ❑ Es una estructura de datos en la que se almacena una colección de datos finita, homogénea y ordenada de elementos
  - **Finita**: debe determinarse cuál será el número máximo de elementos que podrán almacenarse en el array
  - **Homogénea**: todos los elementos deben ser del mismo tipo
  - **Ordenada**: se puede determinar cuál es el n-ésimo elemento del array
- ❑ Para referirse a un determinado elemento de un array se deberá utilizar un **índice** (encerrado entre corchetes) que especifique su posición relativa en el array



En lenguaje C, el primer elemento del array se encuentra en la **posición** (índice) **cero**

# Clasificación de los arrays

7

- Los arrays se clasifican según su número de dimensiones en:
  - Unidimensionales** (vectores)
  - Bidimensionales** (matrices)
  - Multidimensionales**, tres o más dimensiones
- La dimensión de un array es el número de índices utilizados para referenciar a uno cualquiera de sus elementos

Elemento 1
Elemento 2
Elemento 3
.....
Elemento n

**Array**  
unidimensional

Elemento 1,1	.....	Elemento 1,n
Elemento 2,1	.....	Elemento 2,n
Elemento 3,1	.....	Elemento 3,n
.....	.....	.....
Elemento m,1	.....	Elemento m,n

**Array**  
bidimensional

Elemento 1,1,2	.....	Elemento 1,n,2
Elemento 1,1,1	.....	Elemento 1,n,1
Elemento 2,1,1	.....	Elemento 2,n,1
Elemento 3,1,1	.....	Elemento 3,n,1
.....	.....	.....
Elemento m,1,1	.....	Elemento m,n,1

**Array**  
multidimensional

# Arrays unidimensionales

8

- Un array de una dimensión es un tipo de datos estructurado cuyos elementos se almacenan en posiciones contiguas de memoria, a cada una de las cuales se puede acceder directamente mediante un índice.
- Supongamos que queremos almacenar la nota del examen de Programación 1 de 50 estudiantes, por lo que necesitaremos:
  1. Reservar 50 posiciones de memoria
  2. Dar un nombre al array
  3. Asociar una posición en el array a cada uno de los 50 estudiantes
  4. Asignar las puntuaciones a cada una de dichas posiciones

		Posiciones del array	Valores almacenados	Direcciones de memoria
Nombre del array ↓ <b>calificaciones</b>	{	calificaciones[0]	7.50	X
		calificaciones[1]	4.75	X + 1
		calificaciones[2]	5.25	X + 2
		...	...	...
		calificaciones[49]	6.00	X + 49



# Declaración de un *array*

9

- ▣ Para poder utilizar una variable de tipo *array* (unidimensional) primero tenemos que declararla

- ▣ Sintaxis:

```
tipo_elementos nombre_array [num_elem] ;
```

- **tipo\_elementos**: indica el tipo de cada elemento del array; todos los elementos son del mismo tipo
  - **nombre\_array**: indica el nombre del array; puede ser cualquier identificador válido
  - **[num\_elem]** : indica el número **máximo** de elementos del array; debe ser un valor constante numérico entero
- ▣ Ejemplo: `float calificaciones[50];`

# Inicialización y acceso a un array

10

- Al igual que cualquier otro tipo de variable, antes de utilizar un array debemos inicializar su contenido
- Una posible forma de inicializar un array es accediendo a cada uno de sus componentes utilizando un bucle y asignarles un valor
- Para acceder a una posición de un array utilizamos la siguiente sintaxis:

```
nombre_array [indice] ;
```

Acceso a la calificación del alumno que ocupa la posición 5 en el array: `calificaciones[4];`



Utilizar valores de índices **fuera del rango** comprendido por el tamaño del array provoca errores en la ejecución de nuestro programa

# Ejemplo 1 de inicialización de *arrays*

11

- Si se conocen los valores que toman las componentes del array al definirlo, podemos definir y asignar valores simultáneamente:

```
// ejemplo inicialización array  
#include <iostream>  
using namespace std;  
  
main () {  
    int vectorA [4] = {1, 5, 3, 9};  
    int vectorB [] = {1, 5, 3, 9};  
    int vectorC [10] = {1, 5, 3, 9};  
}
```

Toma el número de valores como tamaño del vector

Es posible inicializar de manera parcial el array

# Ejemplo 2 de inicialización de un array

12


- Podemos inicializar un array haciendo que el usuario introduzca los datos por teclado, de la siguiente forma:

```
// ejemplo inicialización array
#include <iostream>
using namespace std;

void inicializar_Array(float calificaciones[ ]);

main () {
    float calificaciones[50];

    inicializar_Array(calificaciones);
}
```



```
// procedimiento para inicializar el array
void inicializar_Array(float calificaciones[ ])
{
    int i;

    for ( i=0 ; i < 50 ; i++ ) {
        cout << "Introduce la calificación " << i << ":";
        cin >> calificaciones[i];
    }
}
```



En lenguaje C, el paso de parámetros de los arrays siempre es **por referencia**



En lenguaje C, las funciones **no pueden devolver** un tipo array. Para modificar un array, ha de ser pasado como parámetro

# Búsqueda lineal de un elemento en un array

13

Recorremos el array desde la primera posición accediendo a posiciones consecutivas hasta encontrar el elemento buscado

```
// Búsqueda lineal de un elemento  
// función para buscar un elemento "elem" en un array con TAM_MAX elementos  
// Devuelve la posición de "elem" en el array si lo encuentra o -1 si no lo encuentra  
int Busqueda_Lineal(int nom_array[], int elem)  
{  
    int pos;  
    bool encontrado;  
  
    pos = 0;  
    encontrado = false;  
    // terminamos la búsqueda si se alcanza el final del array o si se ha encontrado el elemento  
    while ( pos < TAM_MAX && ! encontrado) {  
        if (nom_array[pos] == elem)  
            encontrado = true;  
        else  
            pos = pos + 1;  
    }  
    if (! encontrado)  
        pos = -1;  
  
    return(pos);  
}
```

# Búsqueda binaria de un elemento en un array

14

Si los elementos del array están ORDENADOS podemos utilizar la **búsqueda binaria (dicotómica)**: reducimos la búsqueda dividiendo en mitades, de forma que se va acotando el intervalo de búsqueda dependiendo del valor a buscar

*// Búsqueda binaria de un elemento en un array con TAM\_MAX elementos ordenados de forma creciente*

```
int Busqueda_Binaria(int nom_array[], int elem) {
```

```
    int    pos_inicio, pos_fin, pos_media;
```

```
    bool encontrado;
```

```
    // [pos_inicio, pos_fin] = intervalo actual de búsqueda
```

```
    pos_inicio = 0; // primera posición del array
```

```
    pos_fin = TAM_MAX - 1; // última posición del array
```

```
    encontrado = false;
```

```
    while ( pos_inicio <= pos_fin  &&  ! encontrado) {
```

```
        pos_media = (pos_inicio + pos_fin) / 2; // posición intermedia del array
```

```
        if (elem == nom_array[pos_media]) // elemento encontrado en la posición pos_media
```

```
            encontrado = true;
```

```
        else if (elem > nom_array[pos_media] )
```

```
            pos_inicio = pos_media + 1; // el elemento hay que buscarlo en la mitad superior
```

```
        else
```

```
            pos_fin = pos_media - 1; // el elemento hay que buscarlo en la mitad inferior
```

```
    }
```

```
    if (! encontrado)
```

```
        pos_media = -1;
```

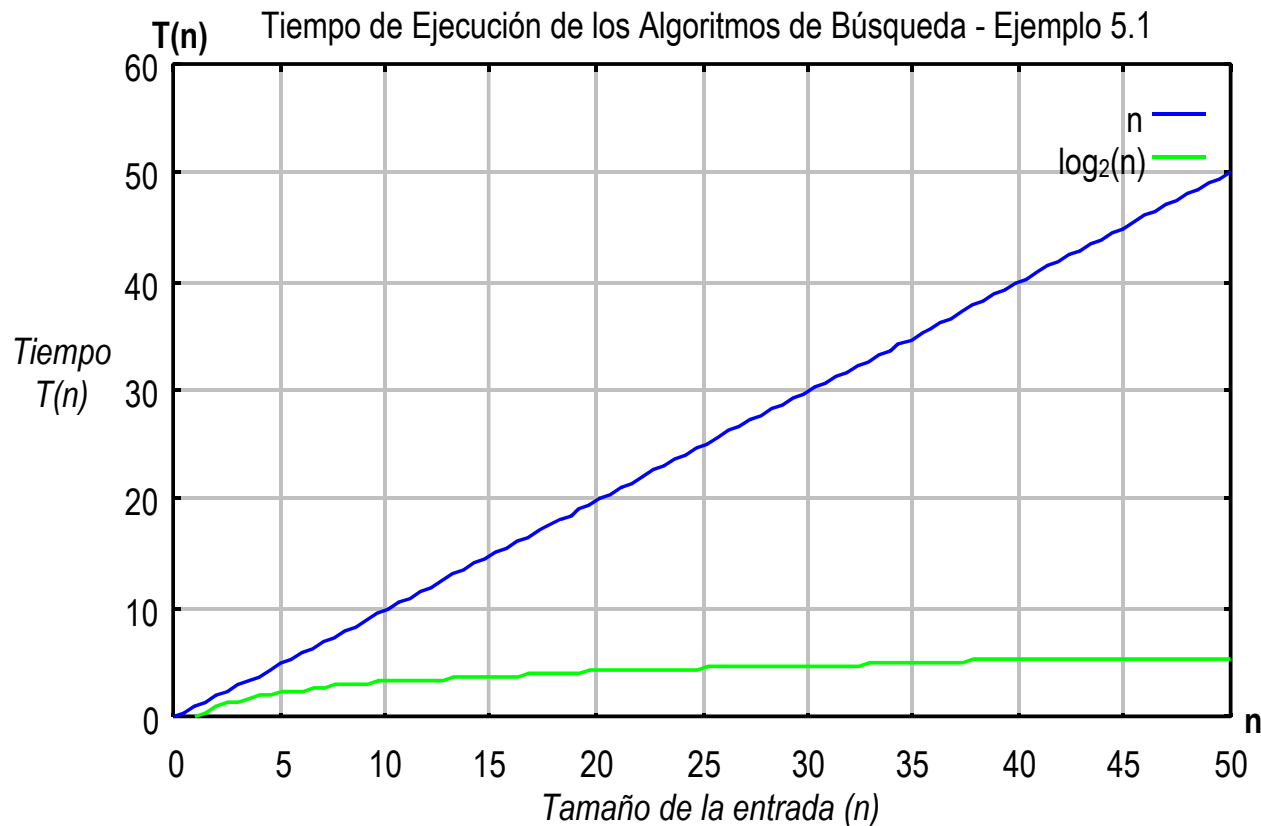
```
    return(pos_media);
```

```
}
```

# Búsqueda (coste temporal)

15

- Búsqueda lineal: tiempo de ejecución lineal
- Búsqueda binaria: tiempo de ejecución logarítmico



# Cadenas de caracteres

16

- ❑ Una cadena de caracteres (**string**) es una secuencia finita de caracteres consecutivos
- ❑ Usaremos **arrays de caracteres**
- ❑ En un array de caracteres podremos almacenar
  - ❑ Palabras
  - ❑ Frases
  - ❑ Nombres de persona, nombres de ciudades, ...
  - ❑ Códigos alfanuméricos
  - ❑ etc.



En lenguaje C++ existe el tipo string, aunque en P1 no lo usaremos



# Representación de cadenas de caracteres en C

17

- En lenguaje C, una cadena de caracteres se escribe entre dobles comillas

“hola”

- En lenguaje C, todas las cadenas de caracteres deben finalizar con el carácter nulo ‘\0’, que debe almacenarse en el array a continuación del último carácter de la cadena

‘h’	‘o’	‘l’	‘a’	‘\0’					
0	1	2	3	4	5	6	7	8	9

- La cadena “hola”
  - se ha almacenado en un array de caracteres de tamaño 10
  - está formada por 4 caracteres (tiene longitud 4) pero ocupa en memoria el espacio de 5 caracteres (porque se almacena también el carácter ‘\0’)

```
char cad[10] = “hola”;
```

# Funciones de C para manejar cadenas de caracteres

18

Función	Descripción	Uso
<b>cin.getline</b> (cadena, TAMAÑO)	lectura de una cadena de caracteres por teclado hasta fin de línea o el tamaño máximo especificado por el parámetro <i>TAMAÑO</i> (entero positivo). La secuencia de caracteres leída se almacena en el parámetro <i>cadena</i> (array de caracteres de al menos un tamaño determinado por el parámetro <i>TAMAÑO</i> )	Como procedimiento
<b>strcpy</b> (cadena_destino, cadena_origen)	copia de cadenas. Copia el contenido de <i>cadena_origen</i> en <i>cadena_destino</i>	Como procedimiento
<b>strcmp</b> (cadena1, cadena2)	comparación alfabética de cadenas <u>si</u> <i>cadena1</i> < <i>cadena2</i> <u>entonces</u> devuelve un número < 0 <u>si</u> <i>cadena1</i> == <i>cadena2</i> <u>entonces</u> devuelve 0 <u>si</u> <i>cadena1</i> > <i>cadena2</i> <u>entonces</u> devuelve un número > 0	Como función
<b>strlen</b> (cadena)	devuelve un tipo <i>int</i> que indica la longitud de la cadena de caracteres especificada como parámetro, es decir, el número de caracteres válidos de dicho array (hasta el carácter especial de fin de cadena '\0', sin incluir éste)	Como función

# Ejemplos arrays unidimensionales (I)

19

Procedimiento que imprime por pantalla el contenido de un array de elementos de tipo double

```
// imprime por pantalla los elementos de  
// un array de tipo double  
void print_Array(double a[], int len)  
{  
    int i;  
    for (i=0; i < len; i++)  
        cout << "[" << i << "] = " << a[i] << endl;  
}
```

Función que calcula la media de notas de alumnos

```
// disponemos de "len" notas de tipo float  
float calcula_Media(float a[], int len)  
{  
    int i;  
    float suma;  
  
    suma = 0.0;  
    for (i=0; i < len; i++)  
        suma = suma + a[i];  
  
    // suponemos len > 0  
    return(suma / len);  
}
```

# Ejemplos arrays unidimensionales (II)

20

Dado un array de enteros, mover todos sus elementos una posición a la derecha. El desplazamiento será circular, es decir, el último elemento pasará a ser el primero

```
void mover_En_Circular (int v[ ])
{
    int i, ult;

    // guardar el valor de la última posición de la tabla
    ult = v[LMAX-1];

    // mover todos los elementos una posición a la derecha, excepto el último
    for (i=LMAX-1; i > 0; i--)
        v[i] = v[i-1];

    // actualizar la primera posición con el valor que teníamos en la última
    v[0] = ult;
}
```

# Ejemplos arrays unidimensionales (III)

21

Dado un array de enteros, devolver el mayor valor, el número de ocurrencias de dicho valor, y la posición de la primera y última aparición en la que se encuentra almacenada

```
void Ocurrencias(int v[ ], int &mayor, int &num_ocur, int &pos_pri, int &pos_ult)
{
    int i;

    mayor = v[0]; // inicialmente el número mayor será el que está en la primera posición
    num_ocur = 1;
    pos_pri = 0;
    pos_ult = 0;

    // recorrer la tabla: desde la segunda posición hasta la posición final (constante LMAX)
    for (i=1; i < LMAX; i++) {
        if (v[i] > mayor) { // encontramos un nuevo número mayor
            mayor = v[i];
            num_ocur = 1;
            pos_pri = i;
            pos_ult = i;
        }
        else if (v[i] == mayor) {
            // encontramos una nueva ocurrencia del número mayor hasta el momento
            num_ocur = num_ocur + 1;
            pos_ult = i;
        }
    }
}
```

# Algoritmos de ordenación de arrays

22

- Es interesante y habitual la operación de ordenación en un array
  - Ejemplo: mantener ordenado nuestro vector de calificaciones para poder consultar rápidamente las cinco mejores notas. Para ello tendríamos que ordenar nuestro vector de mayor a menor (en orden decreciente) y acceder a las cinco primeras posiciones del vector
- Existen muchos algoritmos para ordenar los elementos de un array.

# Ordenación por intercambio

23

```
void burbuja(int v[], int n) {  
    int aux, i, j;  
  
    for (i = 1; i < n; i++)  
        for (j = n-1; j >= i; j--)  
            if (v[j-1] > v[j]) {  
                aux = v[j-1];  
                v[j-1] = v[j];  
                v[j] = aux;  
            }  
}
```

# Ejemplo de ordenación de arrays (I)

24

- La ordenación por **inserción directa** puede compararse con la ordenación de una mano de cartas



- Cada vez que cogemos una carta la insertamos en su posición correcta entre las que ya tenemos ordenadas en la mano

- La inserción divide el array en dos partes:

- La primera (representa las cartas que tenemos en la mano) está ordenada y crece en tamaño a medida que avanza la ordenación
- La segunda (representa las cartas que vamos añadiendo que están en la mesa) está sin ordenar, y contiene los elementos que vamos a ir insertando en la primera parte del array. Esta segunda parte va decreciendo a medida que avanza la ordenación



# Ordenación por inserción

25

```
void insercionDirecta(int s[],int n){  
    int i,j,aux;  
  
    for (i = 1; i < n; i++) {  
        aux = s[i];  
        j = i - 1;  
        while (j >= 0 && s[j] > aux) {  
            s[j+1] = s[j];  
            j--;  
        }  
        s[j+1] = aux;  
    }  
}
```

# Ordenación por selección

26

- **Paso 1:** buscar y seleccionar de entre todos los elementos que aún no estén ordenados el menor de ellos (si es un orden creciente)
- **Paso 2:** intercambiar las posiciones de ese elemento con el que está en el extremo izquierdo de los desordenados

# Ordenación por selección

27

```
void seleccion(int v[], int n) {  
    int aux, i, j, k;  
  
    for (k = 0; k < n-1; k++) {  
  
        i = k;  
        j = k+1;  
        while (j < n) {  
            if (v[j] < v[i])  
                i = j;  
            j = j+1;  
        }  
        aux = v[k];  
        v[k] = v[i];  
        v[i] = aux;  
    }  
}
```

# Arrays bidimensionales (Matrices)

28

- Se necesitan 2 índices para acceder a uno cualquiera de sus elementos
- Supongamos que queremos almacenar la nota del examen de 7 grupos de P1, cada uno de los cuales tiene 25 alumnos

nombre del array bidimensional **notasFP**

Grupo

**notasFP[1][2]**: Nota del grupo 1, alumno 2

Alumno

	0	1	2	...	24
0					
1			7.2		
2					
...					
5					
6					

# Declaración de un *array* bidimensional

29

- ▣ Para poder utilizar una variable de tipo *array* bidimensional, primero tenemos que declararla

- ▣ Sintaxis:

```
tipo nombre_array [n_elemF] [n_elemC];
```

- **tipo**: tipo de cada elemento del array; todos los elementos del array son del mismo tipo
- **nombre\_array**: nombre del array
- **[n\_elemF]** : número de "filas" del array (primera dimensión)
- **[n\_elemC]** : número de "columnas" del array (segunda dimensión)

# Inicialización y acceso a un *array* bidimensional

30

- ▣ Una posible forma de inicializar un array bidimensional es accediendo a cada uno de sus componentes utilizando dos bucles (uno para cada dimensión) y asignarles un valor
- ▣ Para acceder a una posición de un array bidimensional utilizamos la siguiente sintaxis:

`nombre [indiceF] [indiceC] ;`

- **nombre**: nombre del array
- **[indiceF]** : posición de la primera dimensión (**fila**) del array a la que queremos acceder; debe ser siempre un valor comprendido entre **0..n\_elemF-1**
- **[indiceC]** : indica la posición de la segunda dimensión (**columna**) del array a la que queremos acceder; debe ser siempre un valor comprendido entre **0..n\_elemC-1**

Ejemplos:

`notasFP [6] [24];` // denota la nota del alumno 24 del grupo 6

`notasFP [6];` // denota todas las notas del grupo 6 (array unidimensional asociado a la fila 6)

# Ejemplo 1 de inicialización de un *array* bidimensional

31

Si se conocen los valores, se puede inicializar de la siguiente forma:

```
// ejemplo inicialización de array bidimensional
#include <iostream>
using namespace std;

const int N_FILAS = 4;
const int N_COL = 2;

main () {
    float matDD [N_FILAS][N_COL] = { {3.6, 6.7},
                                       {2.9, 7.6},
                                       {8.9, 9.3},
                                       {1.9, 0.2},
                                       };

    int mat [][][N_COL] = { {3, 6},
                             {9, 7},
                             {8, 3},
                             {1, 0},
                             };
}
```

En lenguaje C, no es necesario especificar el tamaño de la primera dimensión de un array

## Ejemplo 2 de inicialización de un *array* bidimensional

32

- También podemos inicializar un array haciendo que el usuario introduzca los datos por teclado, de la siguiente forma:

```
// ejemplo inicialización de array bidimensional
#include <iostream>
using namespace std;

const int N_FILAS = 10;
const int N_COLUMNAS = 20;
void inicializar_Matriz(float matriz[ ][N_COLUMNAS]);

main () {
    float matriz[N_FILAS][N_COLUMNAS];
    inicializar_Matriz(matriz);
}
```

```
// procedimiento para inicializar la matriz
void inicializar_Matriz(float matriz[ ][N_COLUMNAS])
{
    int i, j;

    for ( i=0 ; i < N_FILAS ; i++ ) {
        cout << "Fila " << i << ":" << endl;
        for ( j=0; j < N_COLUMNAS; j++ ) {
            cout << "columna " << j << ":";
            cin >> matriz[i][j];
        }
    }
}
```



En lenguaje C, no es necesario especificar el tamaño de la primera dimensión de un array en la declaración del módulo



# Ejemplos arrays bidimensionales (I)

33

- Dados 25 alumnos, de los que se conocen las notas de 7 asignaturas, calcular la nota media de las asignaturas para cada uno de los alumnos e imprimirlas por pantalla

```
#include <iostream>
using namespace std;

const int N_ALUMNOS = 25;
const int N_ASIGNATURAS = 7;
void imprime_Media_Alumnos(float notas[ ][N_ASIGNATURAS]);
```

```
main () {
    float notas [N_ALUMNOS][N_ASIGNATURAS];

    imprime_Media_Alumnos (notas);
}
```

// calcula la media de notas para cada alumno y las imprime por pantalla

```
void imprime_Media_Alumnos(float notas[ ][N_ASIGNATURAS]) {
    int i;

    for (int i=0; i< N_ALUMNOS; i++)
        cout << "El alumno " << i << " tiene de media " << calcula_Media(notas[i], N_ASIGNATURAS) << endl;
}
```

utilizamos la  
función de uno  
de los ejemplos  
anteriores



# Ejemplos arrays bidimensionales (II)

34

Dada una matriz cuadrada de enteros, imprimir en el siguiente orden, los elementos de la diagonal, los elementos del triángulo superior (por encima de la diagonal) y los del triángulo inferior (por debajo de la diagonal), todo ello con un recorrido por filas y columnas

```
// Versión que recorre tres veces la matriz
void Imprime_Matriz_3 (int  matriz[ ][LMAX])
{  int i, j;

    // Imprimir diagonal
    for (i=0; i < LMAX; i++) // recorrer filas
        for (j=0; j < LMAX; j++) // recorrer columnas
            if (i == j)
                cout << matriz[i][j];
    // Imprimir triángulo superior
    for (i=0; i < LMAX; i++)
        for (j=0; j < LMAX; j++)
            if (j > i)
                cout << matriz[i][j];
    // Imprimir triángulo inferior
    for (i=0; i < LMAX; i++)
        for (j=0; j < LMAX; j++)
            if (j < i)
                cout << matriz[i][j];
}
```

```
// Versión que recorre una sola vez la matriz
void Imprime_Matriz_1(int  matriz[ ][LMAX])
{  int i, j;

    // Imprimir diagonal
    for (i=0; i < LMAX; i++) // recorrer filas
        cout << matriz[i][i];

    // Imprimir triángulo superior
    for (i=0; i < LMAX-1; i++)
        for (j=i+1; j < LMAX; j++)
            cout << matriz[i][j];

    // Imprimir triángulo inferior
    for (i=1; i < LMAX; i++)
        for (j=0; j < i; j++)
            cout << matriz[i][j];
}
```

# Definición de tipos de Datos con `typedef`

35

- ❑ Se utiliza la palabra reservada **typedef** para que el usuario defina tipos de datos estructurados (tales como arrays y structs)
- ❑ Es útil crear nuevos tipos de datos para mejorar la legibilidad de los programas

*// Definición de tipos de datos*

```
typedef int    T_enteros[20];
```

```
typedef float  T_notas[50];
```

```
typedef char   T_Cadena[30];
```

```
typedef int    T_Matriz[3][3];
```

*// Declaración de variables*

```
T_notas    notas_FP1, notas_FP2;
```

```
T_Cadena   nom_alumno1, nom_alumno2;
```

```
T_Matriz   matriz1, matriz2;
```

*// Si no definimos tipos de datos,  
// la declaración de variables de tipo  
// array sería:*

```
float  notas_FP1[50];
```

```
float  notas_FP2[50];
```

```
char   nom_alumno1[30];
```

```
char   nom_alumno2[30];
```

```
int     matriz1[3][3];
```

```
int     matriz2[3][3];
```