# Tema4programacion.pdf

Julieta_G_V

Programación I

1º Grado en Ciberseguridad e Inteligencia Artificial

Escuela Técnica Superior de Ingeniería Informática
Universidad de Málaga

# Tema 4 programación

## Tipos de datos

El tipo de datos define las características de los objetos.

Los **tipos de datos simples** definen conjuntos de valores formados por elementos indivisibles y ordenados (int, double, char, bool...).

Los **tipos de datos compuestos** definen conjuntos de valores formados por elementos compuestos que se pueden manipular como un todo o a partir de sus componentes individuales. Son:

- **struct (estructura o registro)** : define un nuevo tipo como una agrupación de varios componentes (que pueden ser de distintos tipos).
- **array** : define un nuevo tipo como una colección de múltiples elementos del mismo tipo, con acceso parametrizado y adecuado para procesamiento iterativo.
- **cadenas de caracteres** : es una especie de array pero de tipo char, es decir, con caracteres.

## Struct

Éste es un tipo definido por el programador, sus componentes se denominan campos y su ámbito de visibilidad se restringe a la propia estructura definida. Se puede acceder a cada componente mediante su identificador.

- **Definición de estructura y alias**

```
typedef struct n
ombre_del_nuevo_
tipo{
    Tipo1 campo_
1;
    Tipo2 campo_
2;
    ...
    TipoN campo_
```

- **Definición de estructura**

```
struct nombre_del_nuevo_tipo{
    Tipo1 campo_1;
    Tipo2 campo_2;
    ...
    TipoN campo_n;
};
// HAY que poner punto y coma al fin
al
```

```
n;
} nombre_del_nue
vo_tipo;
// HAY que poner
punto y coma al
final
```

```
// ejemplo
typedef struct f
echa{
    int dia;
    int mes;
    int anio;
} fecha;
```

```
// ejemplo
struct fecha{
    int dia;
    int mes;
    int anio;
};
```

- **Paso por valor y paso de punteros**

  El paso por valor de estructuras conlleva una alta sobrecarga, ya que se debe duplicar la memoria y copiar el contenido. Sin embargo, el paso de punteros constante de estructuras permite realizar la transferencia de información de entrada de forma eficiente. Por ello, todos los parámetros de entrada de estructuras se realizarán mediante el **paso de punteros constante** y todos los parámetros de salida o entrada/salida de estructuras se realizarán mediante el **paso de punteros**.

```
void leer_fecha(struct Fecha* f) // ⇑ Paso de puntero
{
        printf("Introduce dia mes y año: ");
        scanf(" %d %d %d", &f->dia, &f->mes, &f->anio);
}

void mostrar_fecha(const struct Fecha* f) // ⇓ Paso de p
untero constante
{
        printf(" %02d/ %02d/ %04d\n", f->dia, f->mes, f-
>anio);
}
```

# ¿DÍA DE CLASES infinitas?


WRIGLEY'S Orbit

masca y fluye

# Programación I

## Banco de apuntes de la

WUOLAH

**Comparte estos flyers en tu clase y consigue más dinero y recompensas**

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

```
int main()
{
        struct Fecha f1;
        leer_fecha(&f1);
        mostrar_fecha(&f1);
}
```

Para acceder a un determinado componente, se nombra el objeto seguido por el identificador del campo correspondiente, ambos separados por un **punto**. Si la variable es un puntero a una estructura, para acceder a un determinado componente se nombra la variable seguido por el identificador del campo correspondiente, ambos separados por una **flecha** (″- >″).

```
void mostrar_fecha(const struct Fecha* f)
{
        printf(" %02d/ %02d/ %04d\n", f->dia, f->mes, f-
>anyo);
        printf(" %02d/ %02d/ %04d\n", (*f).dia, (*f).me
s, (*f).anyo);
}

int main()
{
        struct Fecha f = { 24, 7, 2018 };
        mostrar_fecha(&f);
}
```

## Arrays

Éste es un tipo definido por el programador. El número de elementos de la colección debe ser constante y se define en el tiempo de compilación. Se puede acceder a cada elemento de la colección de forma parametrizada. Además, se puede especificar el tipo array en la propia definición de las constantes, variables y parámetros, especificando el número de elementos entre corchetes.

```
int practicas[NUMERO_ELEMENTOS] ;
```

“typedef” también permite definir un alias asociado al tipo array.

```
// tipo-base alias número-de-elementos
typedef int Numeros[NUMERO_ELEMENTOS] ;
Numeros practicas; // variable de tipo Numeros (ARRAY de IN
T)
```

En la definición de un tipo array intervienen:

- Tipo base: tipo de los elementos que constituyen el array (int, double, char...).
- El número de elementos que forman la colección, que se definen con un NOMBRE en “enum”.

```
enum{
        NUMERO_ELEMENTOS=5
};
```

Podemos definir constantes, variables y parámetros del tipo array.

- **Acceso a los componentes:** un determinado elemento de un array podrá utilizarse en cualquier lugar en que resulten válidas las variables de su mismo tipo base. Para acceder al iésimo elemento de un array v, se le nombra como v[ i ], donde i pertenece a { 0...NUMERO_ELEMENTOS - 1}.

```
enum {
        NELMS = 5
} ;

int main()
{
        int v[NELMS];
        for (int i = 0; i < NELMS; ++i) {
                v[i] = 2 * i;
// v: 0 2 4 6 8
        }
        int primer_elemento = v[0];
// 0
        int ultimo_elemento = v[NELMS - 1];
```


```
// 8
}
```

En el paso de parámetros, el array pierde su tamaño y automáticamente se convierte en un puntero al primer elemento del array. El número de elementos del array se debe pasar como parámetro (VLA). Además, todos los parámetros de salida o entrada/salida de arrays se realizarán mediante el **paso de arrays de longitud variable (VLA)** y todos los parámetros de entrada de arrays se realizarán mediante el **paso de arrays de longitud variable (VLA) constante**.

También es posible definir estructuras con componentes de tipo array. Se aplica el paso de parámetros de estructuras.

- **Operaciones con arrays:**
  - **memset(dest, valor, szbytes)** copia el valor (byte) especificado, a la zona de memoria apuntada por dest, tantos bytes como indica sznbytes.
  - **memmove(dest, org, szbytes)** copia de la zona de memoria apuntada por org, a la zona de memoria apuntada por dest, tantos bytes como indica sznbytes. Las áreas de memoria pueden estar solapadas.
  - **memcpy(dest, org, szbytes)** copia de la zona de memoria apuntada por org, a la zona de memoria apuntada por dest, tantos bytes como indica sznbytes. Las áreas de memoria NO pueden estar solapadas.
- **Arrays incompletos:** hay situaciones en las que debemos almacenar una lista de elementos, donde el número de elementos puede variar durante la ejecución del programa, pero nunca sobrepasará un determinado límite máximo. Normalmente, la opción más adecuada para gestionar esta estructura de datos suele ser definir un tipo estructura que contenga el número de elementos válidos que actualmente contiene la lista y un array, del tamaño adecuado al límite máximo de la lista, que almacene los elementos consecutivamente al principio.

```
enum {
MAX_ELEMENTOS = 100,
};

typedef struct Lista {
```

```
        int nelms; // número de elementos almacenados en
el array
        int elementos[MAX_ELEMENTOS]; // los elementos s
e almacenan en el array
}Lista;
```

## Arrays multidimensionales

El tipo base de un array puede ser tanto simple como compuesto, por lo que puede ser otro array, dando lugar a arrays con múltiples dimensiones.

```
void leer_matriz(int nfils, int ncols, int matriz[nfils][nc
ols])
{
        printf("Introduce %d x %d números:\n", nfils, ncol
s);
        for (int f = 0; f < nfils; ++f) {
                for (int c = 0; c < ncols; ++c) {
                scanf(" %d", &matriz[f][c]);
                }
        }
}
```

```
void mostrar_matriz(int nfils, int ncols, const int matriz
[nfils][ncols])
{
        for (int f = 0; f < nfils; ++f) {
                for (int c = 0; c < ncols; ++c) {
                        printf("%3d ", matriz[f][c]);
                }
                printf("\n");
        }
}
```

Para buscar un elemento en un array 2D:

```
void buscar2d(int nfils, int ncols, const int matriz[nfils]
[ncols], int x, int* fil, int* col)
```

```
{
        *fil = - 1;
        *col = - 1;
        bool encontrado = false;
        for (int f = 0; (f < nfils) && ( ! encontrado); ++
f) {
                for (int c = 0; (c < ncols) && ( ! encontra
do); ++c) {
                        if (x == matriz[f][c]) {
                                *fil = f;
                                *col = c;
                                encontrado = true;
                        }
                }
        }
} // En el momento en el que se encuentra el numero, el buc
le termina. Además, el bucle dice la posición del elemento
con *fil y *col.
```

- **Difuminar matriz:** difuminar una matriz es realizar la media de un número con sus vecinos, que serán 3, 5 u 8 dependiendo de la posición del propio elemento (esquina, lateral o centro) y asignar esa media a la posición del elemento.

```
#include <stdio.h>
#include<stdbool.h>

enum {
        NUMFILS = 3,
        NUMCOLS = 4,
};

void leer_matriz(int nfils, int ncols, int matriz[nfils]
[ncols]){
        printf("Introduce %d x %d números:\n", nfils, nc
ols);
        for (int f = 0; f < nfils; ++f) {
                for (int c = 0; c < ncols; ++c) {
```

```
                        scanf(" %d", &matriz[f][c]);
                }
        }
}

bool es_valido(int nfils, int ncols, int f, int c, int f
f, int cc){
        return (0 <= f+ff && f+ff < nfils) && (0 <= c+cc
&& c+cc < ncols);
}
// && (vabs(ff)+vabs(cc) == 2); // -> selecciona solo lo
s vecinos de las esquinas
// && (vabs(ff)+vabs(cc) == 1); // -> selecciona solo lo
s vecinos horizontal y vertical
// && (vabs(ff)+vabs(cc) > 0); // -> selecciona solo los
vecinos (no central)

double media_vecinos(int nfils, int ncols, const int mat
riz[nfils][ncols], int f, int c){
        int nvec = 0;
        int suma = 0;
        for (int ff = -1; ff <= +1; ++ff) {
                for (int cc = -1; cc <= +1; ++cc) {
                        if (es_valido(nfils, ncols, f,
c, ff, cc)) {
                                suma += matriz[f+ff][c+c
c];
                                ++nvec;
                        }
                }
        }
        return (nvec > 0) ? (double)suma / (double)nvec
: 0;
}

void difuminar_matriz(int nfils, int ncols, const int ma
triz[nfils][ncols], int nuevo[nfils][ncols]){
        for (int f = 0; f < nfils; ++f) {
```

```
                for (int c = 0; c < ncols; ++c) {
                        nuevo[f][c] = media_vecinos(nfil
s, ncols, matriz, f, c);
                }
        }
}

void mostrar_matriz(int nfils, int ncols, const int matr
iz[nfils][ncols]){
        for (int f = 0; f < nfils; ++f) {
                for (int c = 0; c < ncols; ++c) {
                        printf("%3d ", matriz[f][c]);
                }
                printf("\n");
        }
}

int main()
{
        int matriz[NUMFILS][NUMCOLS];
        int nuevo[NUMFILS][NUMCOLS];
        leer_matriz(NUMFILS, NUMCOLS, matriz);
        difuminar_matriz(NUMFILS, NUMCOLS, matriz, nuev
o);
        mostrar_matriz(NUMFILS, NUMCOLS, nuevo);
}
```

- **Multiplicación de matrices**

- Multiplicación de Matrices: producto de 2 matrices (de máximo 10x10 elementos).

```c
#include <stdio.h>
#include <assert.h>

enum {
     MAX = 10,
};

struct Matriz {
    int nfils;
    int ncols;
    double elm[MAX][MAX];
};

void escribir_matriz(const struct Matriz* m)
{
    for (int f = 0; f < m->nfils; ++f) {
        for (int c = 0; c < m->ncols; ++c) {
            printf("%6.2lg ", m->elm[f][c]);
        }
        printf("\n");
    }
}
```

- Eliminar elementos de forma ordenada

**Eliminar elemento ordenado**

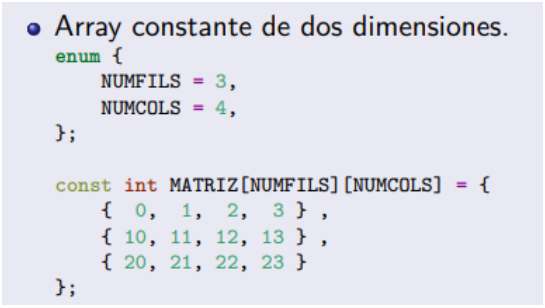
```c
void eliminar_ord(struct Lista* lista, int valor, bool* ok)
{
    int pos = buscar(lista, valor);
    if (0 <= pos && pos < lista->nelms) {
        for (int i = pos+1; i < lista->nelms; ++i) {
            lista->elm[i-1] = lista->elm[i];
        }
        --lista->nelms;
        *ok = true;
    } else {
        *ok = false;
    }
}
```

- **Eliminar todos los elementos**

```c
void eliminar_todos(struct Lista* lista, int valor)
{
        int j = 0;
        for (int i = 0; i < lista->nelms; ++i) {
                if ( lista->elm[i] != valor ) {
//lo que se quiere eliminar es el valor, los elementos d
istintos de valor se sobreescriben y no se pierden
                lista->elm[j] = lista->elm[i];
                ++j;
                }
```

```
        }
        lista->nelms = j;
}
```

- **Definir una matriz como constante**




## Cadenas de caracteres (strings)

Las cadenas de caracteres representan una secuencia de caracteres (palabras, textos...). Estas cadenas se almacenan en arrays de tipo char y terminan con el carácter nulo '\0', que también cuenta como elemento dentro del string. Al ser un array un puntero, el string también se pude representar con * en vez de con [...].

```
enum{
    MAXCARS=63+1,                           //63 caracteres m
ás el caráter terminador
}

void leer(int max, char nombre[max])
{
        printf("Introduce un nombre: ");
        scanf(" %63[^\n]", nombre);
```

- **Operaciones con cadenas de caracteres**

  Para leer o imprimir un string:

```
printf("%s", nombre);        //imprime el string complet
o, hasta '\0'
printf("%45s", nombre);      //imprime 45 caracteres del
string
```

```
scanf(" %63[^\n]", nombre)   //escanea hasta 63 caracter
es o hasta un salto de línea
scanf(" %25s", nombre);      //escanea como máx 25 carac
teres o hasta leer un ESPACIO o SALTO DE LINEA
```

Funciones de la biblioteca #include <string.h> :

```
int longitud = strlen(nombre);     //devuelve el número
de elementos de un string sin incluir ‘\0’

strcpy(texto_destino, texto_origen);       //copia un te
xto de origen a destino sin tamaño máximo
strncpy(texto_destino, texto_origen, MAXCARS);
texto_destino[MAXCARS-1]='\0';               //copia el
string de origen a destino de como máximo MAXCARS

int res = strcmp(texto1, texto2); // compara letras hast
a terminador >> si ==0; los dos strings son iguales; si
no los son, !=0
int res = strncmp(texto1, texto2, maxcars); // compara l
etras como máximo maxcars
```

Conversión entre los tipos básicos:

```
int transformar(int max, char texto[max])       //transf
orma un string de números a int
{
    int num=0;
    for(int i=0; i<max && texto[i]!='\0'; i++){
        num=num*10+(texto[i]-'0');
    }
    return num;
}
//si queremos de char a double, se pondría double num
```

- **Ejercicios:**

- Diseñe una función que devuelva verdadero si la palabra recibida como parámetro es "palíndromo" y falso en caso contrario.

```
bool es_palindromo(int maxcars, const char palabra[maxcars])
{
    bool ok = false;
    int longitud = strlen(palabra);
    if (longitud > 0) {
        int i = 0;
        int j = longitud - 1;
        while ((i < j) && (palabra[i] == palabra[j])) {
            ++i;
            --j;
        }
        ok = i >= j;
    }
    return ok;
}
```

Una palabra es palíndroma si se lee igual en ambos sentidos.

Las cadenas de caracteres también pueden ser definidas como constantes, y de hecho así debe ser cuando son parámetros de entrada: const char nombre[max].

## Almacenamiento persistente de datos: ficheros

El fichero de entrada es aquel que el programa recibe y lee. El fichero de salida es aquel que el programa envía. Hay tanto que abrir como cerrar los ficheros si los queremos utilizar.

- **Abrir ficheros (de entrada y de salida)**

```
FILE* f_ent = fopen(nombre, "r");     // la "r" especifi
ca "read"
FILE* f_sal = fopen(nombre2, "w");    // la "w" especifi
ca "write"

//nombre y nombre2 son strings introducidos como parámet
ros de entrada en el subprograma
```

- **Cerrar ficheros (de entrada y de salida)**

```
fclose(f_ent);
fclose(f_sal);
```

- **Lectura e impresión de ficheros**

```
fscanf(f_ent, "%c", &caracter);
fprintf(f_sal, "%c", caracter);
```

- **Operaciones con ficheros**

```
feof(f_ent) / !feof(f_ent) ; //indica que es el final de
l fichero o que no lo es
ferror(f_ent)==0 / ferror(f_ent)!=0 ; //indica que no se
han producido errores o que sí

char linea[max];
fgets(linea, max, f_ent)!=NULL; //lee una línea del fich
ero f_ent hasta \n
```

```
bool ok=true;
while(fscanf(f_ent, "%c", &caracter)==1 && ferror(f_ent)
==0){
        .
        .
        ok=true;
}
if(!feof(f_ent)){  //si ha salido del bucle y non es el
final del archivo, ha habido errores
    ok=false;
}
```

```
FILE* f_ent = fopen(nombre, "r");
if(f_ent!=NULL){
        FILE* f_sal = fopen(nombre2, "w");
        if(f_sal!=NULL){
                .
                .
        }
}
```