TEMA 10
DATOS ESTRUCTURADOS

10.1. Matrices

10.2. Strings

10.3. Estructuras

10.4. Enumerados

10.5. Punteros

10.1. Matrices

Una matriz es una colección de variables del mismo tipo, denominadas por un nombre común

```
int var0, var1, var2;
```

EQUIVALENTE A

```
int vars[3];
```

A cada elemento específico del array se accede con un índice

```
int var0, var1, var2;
```

EQUIVALENTE A

```
int vars[3];
```

Ahora podemos tratar múltiples datos de una manera más sencilla

```
for (i=0; i<3; i++) vars[i] = (i+1)*2;
```

Dimensiones de una matriz

- Una matriz puede tener una o varias dimensiones

```
int una_dim[10];
char dos_dim[10][15], tres_dim[5][10][4];
```

  Las matrices uni-dimensionales son llamadas *vectores*

- Disposición de una matriz en memoria

```
int dos_dim[3][4];
```

[0][0]	$\cdots$	[0][3]	[1][0]	$\cdots$	[2][2]	[2][3]

[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]
[2][0]	[2][1]	[2][2]	[2][3]

Definición de una matriz

- Según el momento en el que se asigna el tamaño:
  - Arrays estáticos: Se definen en tiempo de compilación
  - Arrays dinámicos: Se definen en tiempo de ejecución

- C no comprueba que los limites del array sean desbordados

```
int i, vect[10];
for (i=0; i<10; i++) vect[i+1] = vect[i];
```

  - Sobrepasar los límites del array puede implicar:
    * Sobreescribir otras variables
    * Corromper la estructura del programa

Ejemplo de uso

TABLA DE MULTIPLICAR

```
int i, j;
int mult[10][10];
for (i=0; i<10; i++)
  for (j=0; j<10; j++)
    mult[i][j] = i*j;
```

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9
0	2	4	6	8	10	12	14	16	18
0	3	6	9	12	15	18	21	24	27
0	4	8	12	16	20	24	28	32	36
0	5	10	15	20	25	30	35	40	45
0	6	12	18	24	30	36	42	48	54
0	7	14	21	28	35	42	49	56	63
0	8	16	24	32	40	48	56	64	72
0	9	18	27	36	45	54	63	72	81

Inicialización de matrices

- C permite inicializar matrices en el momento de su declaración

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int i[2][3] = {0, 1, 2, 10, 11, 12};
char error[20] = "Fichero no encontrado.\n";
//longitud=24
```

- C permite no indicar el tamaño de una matriz inicializada

```
int i[ ][3] = { 0, 1, 2, 10, 11, 12};
char error[ ] = "Fichero no encontrado.\n";
```

Paso de arrays a funciones

- Paso por variable (o valor)
  - Definición de la función

```
float Mayor(float vector[120]) {...} // O:
float Mayor(float vector[]) {...}
```

  - Llamada a la función EJEMPLO

```
float lista_notas[120], mejor_nota;
mejor_nota = Mayor (lista_notas);
```

- Paso por referencia (o puntero)
  - Definición de la función

```
float Mayor(float  *vector) { ... }
```

  - Llamada a la función EJEMPLO

```
float lista_notas[120], mejor_nota;
mejor_nota = Mayor (&lista_notas[0]);
```

10.2. Strings

- No existe el tipo cadena de caracteres. Estas se implementan con vectores (acaban en un carácter nulo \0)

  ```
  'Esto es una constante de cadena de caracteres'
  ```

  El carácter nulo está implícito y lo añade el compilador

- Las funciones de tratamiento de cadenas se encuentran en la librería *string*

  ```
  char *strcpy ( char *s1, const char *s2 );
  //Copia s2 en s1
  char *strcat ( char *s1, const char *s2 );
  //Concatena s2 a s1
  int strlen ( const char *s1 );
  //Devuelve la longitud de s1
  int strcmp ( const char *s1, const char *s2 );
  //Compara alfabéticamente
  ```

Strings. Ejemplo

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

void main(void)
{
 char c1[80], c2[80], c3[80];
 gets(c1); //(get string c1)
 gets(c2); //Lee una cadena y almacena en c2
 printf("La 1ª cadena mide %d.\n", strlen(c1));
 printf("La 2ª cadena mide %d.\n", strlen(c2));
 if (strcmp(c1,c2)==0) printf("Y son iguales\n");
     else printf("Pero son distintas cadenas\n");
 strcpy(c3,c1); //Copia en c3 la 1ª cadena leída
 strcat(c3,c2); //Pega c2 detrás de c3
 printf("Las cadenas leídas forman %s\n",c3); (A)
 printf("Las cadenas leídas forman ");        (B)
 puts(c3); printf("\n"); //Muestra una cadena (C)
 //La sentencia (A) equivale a (B) + (C)
}
```

10.3. Estructuras

- Una estructura es una agrupación de variables bajo un nombre común. Permite mantener junta información relacionada
- Declaración de la estructura: Los campos se declaran dentro de la estructura de la forma habitual. Las variables pueden declararse tras la declaración de la estructura:

```
  struct lector
{ char nombre[80],
       telefono[20],
       direccion[100];
  int  codigo,
       edad,
       prestamos;} Rafa, Maria;

  struct libro
{ char titulo[80],
       Autor[80],
       editorial[100];
  int  paginas,
       prestado_a;} libro1, libro2;
```

Estructuras. Utilización

... ó pueden declararse después:

```
struct lector Pepe, x, y, socios[1000];
//declara 1003 lectores
struct libro bib[10000]; // declara 10000 libros
```

- Una estructura completa se comporta como un tipo atómico.

```
struct libro libro_perdido;
libro_perdido = bib[3156];
```

- Se accede a los campos con la notación *punto*

```
if (nuevo_socio.edad > 26)
printf("No puedo extender su carnet joven");
lector23.prestamos++;
```

Estructuras. Ejemplo de uso

```
void Prestar(struct lector lec, struct libro lib)
{if (lib.prestado)
            printf("El libro no está disponible");
     else if (lec.prestamos>=3)
            printf("ya tiene demasiados libros");
              else {lib.prestado = lec.codigo;
                    lec.prestamos++;}}

Prestar(Pepe,bib[211]);
    //Gestiona un prestamo a Pepe
```

10.4. Enumerados

- Una enumeración es un conjunto de constantes enteras, $\{0, 1, 2, 3, 4, \ldots\}$, con nombre que especifica todos los valores que ese campo puede tener

- Declaración de la enumeración. Las variables pueden declararse tras la declaración de la enumeración:

```
enum estilo
    {aventuras, amor, cficcion, historia}
     cat1,cat2,z,t;//declara 4 estilos
```

- Declaración de las variables después:

```
enum estilo x,mis_gustos,y;//declara 3 estilos
```

Enumerados. Ejemplo

```
struct lector
{char nombre[80],
      telefono[20],
      direccion[100];
 int  codigo,
      edad,
      prestamos
 enum estilo pref;}

struct libro
{char titulo[80],
      Autor[80],
      editorial[100];
 int  paginas,
      prestado_a;
 enum estilo tipo;}
```

Tipos definidos por el usuario

C permite definir explícitamente un nuevo tipo de dato con la instrucción *typedef*

```
typedef struct lector
{char nombre[80],
      telefono[20],
      direccion[100];
 int  codigo,
      edad,
      prestamos
 enum estilo pref;}

lector lec1,lec2,lec3;//define 3 lectores
```

Ejemplo de utilización:

```
void Listado_personalizado(lector l)
{int lib;
 for(lib=0; lib<total_libros; lib++)
     if (bib[lib].tipo == l.pref)
          printf("%s\n",bib[lib].titulo);}

Listado_personalizado(lec3);
```

10.5. Punteros

El valor de cada variable está almacenado en una *dirección* determinada de la memoria

Un puntero es una variable cuyo contenido es la dirección de alguna variable

Diremos que un puntero "apunta" a una variable si su contenido es la dirección de esa variable

Normalmente, un puntero ocupa 4 bytes de memoria

Debe definirse en base al tipo de la variable a la que apunta

Ejemplo:

int *puntero

puntero es (una variable que apuntará a la dirección de / un puntero a) una variable de tipo entero

Operadores de Dirección e Indirección

El lenguaje C dispone del *operador de dirección*, &, que permite obtener la dirección de la variable sobre la que se aplica

También se dispone del *operador de indirección*, *, que permite acceder al contenido de la zona de memoria a la que apunta el puntero sobre el cual aplicamos dicho operador

Ejemplo:

```
int i, j, *p;    // p es un puntero

p=&i;            // p apunta a la dirección de i

*p=10;           // i toma el valor 10

p=&j;            // p apunta a la dirección de j

*p=11;           // j toma el valor 11
```

Operaciones legales con punteros

Ni las constantes ni las expresiones tienen dirección (no se les puede aplicar &)

No se permiten las asignaciones directas entre punteros que apuntan a distintos tipos de variables (existe un tipo indefinido de puntero , **void**, que funcionan como comodín para las asignaciones independientemente de tipos).

Ejemplo:

```
int *p;

double *q;

void *r;

p=q;        // ilegal

r=q;        // legal

p=r;        // legal
```

Aritmética de los punteros I

Un puntero contiene información no solo de la dirección en memoria de la variable a al que apuntan sino también de su tipo.

Sobre los punteros podremos:

- Sumar?
- Restar?
- Multiplicar¿
- Dividir¿

? significa que las unidades son los bytes de memoria que ocupa el tipo de las variables a las que apunta

Aritmética de los punteros II

Ejemplo:

int *p;

double *q;

p++; /* sumar 1 a p, implica aumentar 2 bytes en la dirección a la que apunta */

q--; /* disminuir 1 a q, implica disminuir 4 bytes en la dirección a la que apunta */

¿ significa que es una multiplicación y división entre constantes que suelen ser tamaño de dimensiones de matrices o tamaños de estructuras, y número de fila o de columna (se verá más adelante)

Relación Matrices ∼ Punteros

Sea el array así definido:

int array [8] [5];

Los arrays se almacenan en memoria por filas:

Fórmula de direccionamiento de una matriz:
"La posición en memoria del elemento array[7][2] será: posición (array[0][0]) + 7 * 5 + 2"

Ejemplo:

int vect[10], mat[3][5], *p;

p=&vect[0];

printf ("%d", *(p+2)); // imprime vect[2]

p=&mat[0][0];

printf ("%d", *(p+2)); // imprime mat[0][2]

printf ("%d", *(p+5)); // imprime mat[1][0]

printf ("%d", *(p+14)); // imprime mat[2][4]

Relación Matrices $\sim$ Punteros I

El nombre de una matriz uni-dimensional (vector) es un puntero que apunta a la dirección de memoria que contiene al primer elemento de la matriz

Ejemplo:

double v [10]; // v es un puntero a v[0]

double *p;

p=&v[0]; // p = v

Es decir, como v apunta a v[0] tenemos que *(v+4) es v[4]

Y recíprocamente: p[3] es lo mismo que v[3]

Resumiendo:

- *p $\equiv$ v[0] $\equiv$ *v $\equiv$ p[0]
- *(p+1) $\equiv$ v[1] $\equiv$ *(v+1) $\equiv$ p[1] ...

Relación Matrices $\sim$ Punteros II

Sea la siguiente declaración:

int mat [5] [3], **p, *q;

El nombre de esta matriz (mat) es un puntero al primer elemento de un vector de punteros (mat[ ]), *hay un vector de punteros con el mismo nombre que la matriz y el mismo número de elementos que filas de la matriz* cuyos elementos apuntan a los primeros elementos de cada fila de la matriz (fila de la matriz $\sim$ matriz unidimensional)

Así pues mat es un puntero a una variable de tipo puntero

Por tanto, mat es lo mismo que &mat[0] y mat[0] lo mismo que &mat[0][0]. Análogamente, mat[1] es &mat[1][0] ...

Relación Matrices $\sim$ Punteros III

Si hacemos ****p=mat;** tendremos:

- *p $\equiv$ mat[0]
- *(p+1) $\equiv$ mat[1]
- **p $\equiv$ mat[0][0]
- **(p+1) $\equiv$ mat[1][0]
- *(*(p+1)+1) $\equiv$ mat[1][1]

Relación Matrices $\sim$ Punteros IV

Si hacemos **q=&mat[0][0];** podremos acceder al elemento **mat[i][j]** así:

- *(q+j+i $\cdot$ 3)
- *(mat[i]+j)
- *(*(mat+i)+j)
- *(mat+i)[j]

Existe una única diferencia entre punteros y matrices, a saber, con los primeros se pueden definir estructuras equivalentes a matrices cuyas filas puedan tener distinto número de elementos.

Variables y Parámetros en las Funciones I

En la definición de cualquier función suelen aparecer variables de tres tipos:

- **auto** (por defecto) solo visibles para la propia función, se crean cada vez que se llama a la función

- **static** solo visibles para la propia función, conservan su valor entre distintas llamadas a la función

- **extern** son variables definidas fuera de la función, son visibles para la propia función

Llamaremos *argumentos formales* a los que aparecen en la primera línea de la definición de la función

Llamaremos *argumentos actuales* a los que aparecen en la llamada a la función desde el programa que la realiza

Variables y Parámetros en las Funciones II

El valor de retorno de una función (es único) no puede ser un array, aunque si un puntero o una estructura (que puede contener arrays como elementos miembros)

En la llamada a una función los argumentos actuales son evaluados y se pasan copias de los mismos a las variables que constituyen los argumentos formales de la función (también se transfiere el control de ejecución)

En consecuencia, los cambios que hagamos en estas variables no se transmiten a aquellas que se usaron en la llamada

El único valor que se puede transmitir al programa que realiza la llamada aparece tras algún *return*

Diremos en este caso que se han pasado los argumentos **POR VALOR**

Variables y Parámetros en las Funciones III

Sea la función que pretende permutar los valores de sus argumentos **x** e **y**:

```
void permutarmal (double x, double y)

{ double temp;

temp=x;

x=y;

y=temp; }
```

No funciona porque solo permuta los valores de las copias que se le pasa en cada llamada, sin embargo esta si:

```
void permutarbien (double *x, double *y)

{ double temp;

temp=*x;

*x=*y;

*y=temp; }
```

Variables y Parámetros en las Funciones IV

A la primera función (permutamal, paso de parámetros por valor), se le llama con el nombre de las variables que se pretenden permutar

A la segunda función (permutabien) se le pasa en la llamada, la dirección de las variables

Diremos en este último caso que se han pasado los argumentos **POR REFERENCIA**

• Ejemplo con arrays (función que multiplica una matriz cuadrada por otra uni-dimensional):

void mult (int n, double a[ ][10], double x[ ], double y[ ]) // posible declaración

void mult (int n, double (*a)[10], double *x, double *y) // otra posible declaración

void mult (int n, double *a[10], double *x, double *y) // declaración no válida