UNIDAD 2 – ESTRUCTURAS DE DATOS

ÍNDICE

Las estructuras de datos del lenguaje C son más simples que las que ofrece Java porque no existe el concepto de “clase” ni de “objeto”. C ofrece tipos de datos básicos y dos construcciones para crear datos más complejos. El control de acceso a datos que ofrece Java (métodos y campos privados, públicos y protegidos) no existe en C. Las variables son globales, locales a un fichero, o locales a un bloque de código.

1. **Tipos de datos básicos**

C ofrece tres tipos de datos básicos:

- Números enteros definidos con la palabra clave int
- Letras o caracteres definidos con la palabra clave char
  Números reales o en coma flotante definidos con las palabras claves float o double

1.1. Enteros

Se definen con “int” y admiten de forma opcional dos prefijos modificadores:

- “short” y “long”: Modifica el tamaño en bits del entero. Existen por tanto tres tipos de enteros: “int”, “short int” (que se puede abreviar como “short”), y “long int” (que se puede abreviar como “long”).

El lenguaje C no define tamaños fijos para sus tipos de datos básicos. Lo único que garantiza es que un short int tiene un tamaño menor o igual que un int y este a su vez un tamaño menor o igual a un long int. Esta característica del lenguaje

ha complicado la creación de programas que sean compatibles entre varias plataformas.

- “unsigned”: define un número natural (mayor o igual a cero).

1.2 Letras y cadenas

Las variables de tipo letra se declaran como “char”. Para referirse a una letra se rodea de comillas simples: 'M'. Como las letras se representan internamente como números, el lenguaje C permite realizar operaciones aritméticas como 'M' + 25.

Las cadenas de texto o strings son simplemente tablas de “char”. Las funciones de biblioteca para manipular estas cadenas asumen que **el último byte tiene valor cero**. Las cadenas de texto se escriben en el programa rodeadas de dobles comillas y contienen el valor cero al final. A continuación se muestran dos definiciones:

```
#define SIZE 6
char a = 'A';
char b[SIZE] = "hello";
```

1.3.- Números reales

Los números reales se definen con “float” o “double”. La diferencia entre ambas es la precisión que ofrece su representación interna. Hay un número infinito de reales, pero se representan con un número finito de bits. A mayor número de bits, mayor número de reales se representan, y por tanto, mayor precisión. Los reales definidos con “double” tienen un tamaño doble a los definidos con “float”. Al igual que en el caso de los enteros, el tamaño de estas representaciones varía de una plataforma a otra.

Algunas plataformas ofrecen números reales con tamaño mayor al “double” que se definen como “long double”. Los tamaños típicos para los tipos “float”, “double” y “long double” son 4, 8 y 12 bytes respectivamente. A continuación, se muestran varias definiciones de números reales.

```
float a = 3.5;
double b = -5.4e-12;
long double c = 3.54e320;
```

1.4 Tablas

Las tablas en C son prácticamente idénticas a las de Java, con el tamaño entre corchetes a continuación del nombre. Al igual que en Java, los índices de la tabla comienzan por cero. A continuación, se muestran algunos ejemplos:

```
#define SIZE_TABLE 100
#define  SIZE_SHORT 5
#define SIZE_LONG 3
#define SIZE_NAME 10
```

```
int table[SIZE_TABLE];
short st[SIZE_SHORT] = { 1, 2, 3, 4, 5 };
long lt[SIZE_LONG] = { 20, 30, 40};
char name[SIZE_NAME];
```

Los elementos de la tabla se acceden con el nombre de la tabla seguido del índice entre corchetes.

Una de las diferencias entre C y Java es que el acceso a una tabla en C no se verifica. Cuando se ejecuta un programa en Java si se accede a una tabla con un índice incorrecto, se genera una excepción de tipo “ArrayIndexOutOfBounds”. Estas comprobaciones no se hacen nunca en C (a no ser que se escriban explícitamente en el programa). Si se accede a una tabla con un índice incorrecto se manipulan datos en una zona de memoria incorrecta y el programa continua su ejecución.

Tras este acceso incorrecto pueden suceder dos cosas. La primera es que la memoria a la que ha accedido por error esté fuera de los límites del programa. En este caso la ejecución termina de manera abrupta y en el intérprete de comandos se muestra el mensaje “segmentation fault”. La otra posibilidad es que se acceda a otro lugar dentro de los datos del programa. Esta situación seguramente producirá un error cuyos síntomas sean difíciles de relacionar con el acceso incorrecto.

Ejemplo del uso de tablas

Ejercicio 1 – Tablas de enteros.

En este primer ejercicio se define una tabla con unos valores iniciales y se muestra su contenido.

```
#include <stdio.h>
int main()
{
        int tabla[5]={3,2,5,67,89};
        int i;

        for(i=0;i<5;i++)
  {
        printf("%d \n",tabla[i]);
  }
return 0;
}
```

Ejercicio 2

En este segundo ejercicio se introducen los valores a partir de un bucle.

```
#include <stdio.h>
int main()
{
```

```
        int tabla[5];
        int n,i;
   for(i=0;i<5;i++)
   {
        printf("Elemento ");
        scanf("%d",&n);
        tabla[i]=n;
        }
        for(i=0;i<5;i++)
   {
        printf("%d \n",tabla[i]);
   }
return 0;
}
```

Ejercicio 3. Obtener el número de elementos de una tabla

```
#include <stdio.h>
int main()
{
        int tabla1[5]={3,2,5,67,89};
        int i;

   int l1=sizeof(tabla1)/sizeof(tabla1[0]);
        printf("%d \n",l1);
return 0;
}
```

Observa que el número de elementos es el tamaño en bytes de la tabla dividido por el tamaño en bytes de un elemento de la tabla, en este caso 4 bytes ya que es un entero.

Ejercicio 4 –

En este ejercicio se compara el tamaño de dos tablas indicando si es el mismo o diferente

```
#include <stdio.h>
int main()
{
        int tabla1[5]={3,2,5,67,89};
        int tabla2[6]={13,62,45,3,9,4};

    int l1=sizeof(tabla1)/sizeof(tabla1[0]);
    int l2=sizeof(tabla2)/sizeof(tabla2[0]);
    if (l1 == l2){
                printf("%s \n","Tienen el mismo tamaño");}
                else {
                printf("%s \n","Tienen diferente tamaño");}

return 0;
}
```

Ejercicio 5 – Sumar dos tablas.

```
#include <stdio.h>
int main()
{
        int tabla1[5]={3,2,5,67,89};
        int tabla2[5]={13,45,3,9,4};
        int tablas[5];
        int i;
        for(i=0;i<5;i++){
        tablas[i]=tabla1[i] + tabla2[i];
        printf("%d \n",tablas[i]);
        }
return 0;
}
```

Ejercicio 6 – rotar a la izquierda los elementos de una tabla

```
#include <stdio.h>
int main()
{
        int tabla[5]={3,2,5,67,89};
        int i,p;
        p=tabla[0];
        for(i=0;i<4;i++){
        tabla[i]=tabla[i+1];
        }
        tabla[4]=p;
        for(i=0;i<5;i++){
        printf("%d \n",tabla[i]);}
return 0;
}
```

Ejercicio 7.- Array de caracteres.

```
#include <stdio.h>
int main()
{
        char cadena[]="Hola Mundo";
        printf("%s \n",cadena);
return 0;
}
```

Ejercicio 8.- Longitud de una cadena.

En este ejercicio se define una cadena de caracteres y se obtiene su longitud.

```
#include <stdio.h>

int main()

{

        char cadena[]="Hola Mundo";

        int l=sizeof(cadena)/sizeof(cadena[0]);

        printf("%s  %d \n","El numero de caracteres es ", l);
```

```
return 0;

}
```

Observa que el resultado es 11 ya que se cuenta el final de la cadena.

Ejercicio 9. Mostrar el contenido de la cadena carácter a carácter.

```
#include <stdio.h>
int main()
{
	char cadena[]="Hola Mundo";
	int l=sizeof(cadena)/sizeof(cadena[0]);
	int i;
	for (i=0;i<l-1;i++){
		printf("%c",cadena[i]);
	}
return 0;
}
```

Observa que el bucle se recorre desde 0 hasta l-1 para evitar mostrar el delimitador de la cadena.

Ejercicio 10.- Mostrar el contenido de una cadena usando la librería string.h

La librería string.h dispone de varias funciones de cadena que simplifican los códigos. Una de ellas es strlen(cadena) que devuelve la longitud de una cadena sin incluir el carácter final de fin de cadena. Si utilizamos esta función en el ejercicio anterior debemos recorrer el bucle hasta l.

```
#include <stdio.h>
#include <string.h>
int main()
{
	char cadena[]="Hola Mundo";
	int l=strlen(cadena);
	int i;
	for (i=0;i<l;i++){
		printf("%c",cadena[i]);
	}
return 0;
}
```

Ejercicio 11- Copiar cadenas.

Otra función particularmente interesante es strcpy que copia el contenido de una cadena en otra definida previamente.

#include <stdio.h>

```
#include <string.h>
int main()
{
	char origen[]="Hola Mundo";
	char destino[strlen(origen)+1];
```

```
        int l=strlen(origen);
        strcpy(destino,origen);
        printf("%s",destino);
return 0;
}
```

Observa que el tamaño de destino debe ser el resultado de aplicar strlen a origen más 1 ya que hay que reservar espacio para el fin de cadena.

Ejercicio 12.- Copiar parte de una cadena

La función strncpy (destino, fuente, n) copia los n primeros caracteres de la cadena fuente en la cadena destino.

```
#include <stdio.h>
#include <string.h>
int main()
{
        char origen[]="Hola Mundo";
        char destino[5];
        int l=strlen(origen);
        strncpy(destino,origen,5);
        printf("%s",destino);
return 0;
}
```

Ejercicio 13.- Invertir una cadena.

La función strrev(cadena) invierte el orden de una cadena.

```
#include <stdio.h>
#include <string.h>
int main()
{
        char cadena[]="abcdefg";
        printf ("%s", strrev(cadena));
return 0;
}
```

1.5 Tablas de múltiples dimensiones

C permite la definición de tablas de múltiples dimensiones escribiendo los diferentes tamaños rodeados de corchetes y concatenados. El acceso se realiza concatenando tantos índices como sea preciso rodeados de corchetes. Al igual que en el caso de las tablas unidimensionales, no se realiza ningún tipo de comprobación de los índices cuando se accede a un elemento. A continuación se muestra la definición de tablas de más de una dimensión.

```
#define MATRIX_A 100
#define MATRIX_B 30
#define COMMON_SIZE 10

int matrix[MATRIX_A][MATRIX_B];
```

```
long squarematrix[COMMON_SIZE][COMMON_SIZE];
char soup[COMMON_SIZE][COMMON_SIZE];
```

Ejercicio 14.- Array de cadenas

En este ejercicio se define un array de cadenas a partir de tablas bidimensionales

```
#include <stdio.h>
int main()
{
        char nombres[5][20]={"Antonio","Luis","Juan","Marcos","Maria"};
        int i;
        for(i=0;i<5;i++){
        printf("%s \n",nombres[i]);}
return 0;
}
```

Ejercicio 15. Crear una tabla de string

```
#include <stdio.h>
int main()
{
        char nombres[5][20];
        int i;
        for(i=0;i<5;i++){
        gets(nombres[i]);}
        for(i=0;i<5;i++){
        printf("%s \n",nombres[i]);}
return 0;
}
```

2.- Tamaño de los tipos de datos básicos

En C, el tamaño de los tipos de datos básicos puede variar de una plataforma a otra. Esta característica está detrás de buena parte de las críticas que recibe este lenguaje, pues de ella se derivan problemas de compatibilidad (una aplicación se comporta de forma diferente cuando se ejecuta en plataformas diferentes).

A modo de ejemplo, en la siguiente tabla se incluyen los tamaños de los tipos de datos para las plataformas Linux/Intel i686.

Tabla 2.1. Tamaños de los tipos de datos en C en la plataforma Linux/Intel i686

Tipo	Tamaño (bytes)
char, unsigned char	1
short int, unsigned short int	2
int, unsigned int, long int, unsigned long int	4
float	4
double	8
long double	12

3.- Tipos de datos estructurados

C permite definir estructuras de datos que agrupan campos de otros tipos de datos. La sintaxis se muestra a continuación:

```
struct nombre_de_la_estructura
{
   tipo_1 nombre_del_campo1;
   tipo_2 nombre_del_campo2;
   ...
   tipo_N nombre_del_campoN;
};
```

La construcción anterior sólo define un nuevo tipo de datos, no se declara variable alguna. Es decir, la construcción anterior tiene la misma entidad que el tipo “int” o “float”. El nombre del nuevo tipo estructurado definido es “struct nombre_de_la_estructura”. Por ejemplo:

```
#define FIRST_SIZE 100
#define LAST_SIZE 200
#define CONTACTS_NUM 100

/* Definición de la estructura */
struct contactos
{
    char nombre[FIRST_SIZE];
    char apellidos[LAST_SIZE];
    unsigned int edad;
    unsigned int relefono;
};

/* Declaración de variables con esta estructura */
struct contactos persona1, persona2, familia[CONTACTS_NUM];
```

Las líneas 6 a 12 definen un nuevo tipo de datos estructurado que contiene cuatro campos, los dos primeros son tablas de letras y los dos últimos son enteros. A pesar de que estos campos tienen nombres y tamaños, hasta el

momento no se ha declarado ninguna variable. Es en la línea 15 en la que se sí se declaran tres variables de este nuevo tipo estructurado. La última de ellas es una tabla de 100 de estas estructuras. Asegúrate de que tienes clara la diferencia entre la “definición” de un tipo de datos y la “declaración” de variables de ese tipo. La siguiente figura muestra estos conceptos para una estructura y un tipo básico.

La definición de una estructura y la declaración de variables se pueden combinar en la misma construcción, pero es preferible separar ambas operaciones

El acceso a los campos de una variable estructurada se denota por el nombre de la variable seguido de un punto y del nombre del campo tal y como se muestra en el siguiente ejemplo.

Ejercicio 16

```
#include <stdio.h>
int main()
{
struct informacion
{
    char nombre[20];
    char apellido[20];
    unsigned int edad;

} ficha;
    printf("%s \n", "Nombre ");
    gets(ficha.nombre);
     printf("%s \n", "Apellido ");
    gets(ficha.apellido);
     printf("%s \n", "Edad ");
    scanf("%d", &ficha.edad);
    printf("%s, %s, %d", ficha.nombre, ficha.apellido, ficha.edad);
}
```

Ejercicio 17. En el siguiente ejemplo creamos un array de estructuras

```
int main()
{
struct informacion
{
    char nombre[20];
    char apellido[20];
    unsigned int edad;

} ficha[3];
int i;
for(i=0;i<3;i++)
  {
    printf("Alumno %d \n",i+1);
    printf("%s ", "Nombre ");
    scanf("%s",&ficha[i].nombre);
    printf("%s ", "Apellido ");
    scanf("%s",&ficha[i].apellido);
    printf("%s ", "Edad ");
    scanf("%d", &ficha[i].edad);
```

```
 }
for(i=0;i<3;i++)
 {
  printf("%s, %s, %d \n", ficha[i].nombre, ficha[i].apellido, ficha[i].edad);
 }
}
```

4.- Definición de sinónimos de tipos con typedef

C permite definir sinónimos para los tipos de datos mediante el operador typedef y la siguiente sintaxis:

typedef *tipo_de_datos_ya_definido* ***sinónimo***

Por ejemplo, la siguiente línea define entero como tipo sinónimo de int.

typedef int entero

En general, se usan cuando la declaración inicial puede ser confusa.

Por ejemplo.

```
#include <stdio.h>
int main(void)
{
    int notas;
    notas=100;
    return 0;
}
```

EL siguiente códifo define una variable entera y le asigna un valor. Podemos modificar el código anterior de la siguiente forma.

```
int main()
{
    typedef int nota_alumno_t;
    nota_alumno_t notas;
    notas=100;
    return -1;
}
```

Ambas secciones de código hacen lo mismo: crean un tipo int (notas) y le dan un valor de 100. El método para hacer esto en la segunda sección hace que sea más legible porque la declaración typedef hace que nota_alumno_t signifique lo mismo que int. En este ejemplo, la variable notas guarda las "notas" de un estudiante, así que definir notas como una variable de tipo nota_alumno_t le da al nombre de esa variable un contexto.

Si definimos

typedef int entero;

Estamos creando una especie de alias “entero” de la variable int. Después queda más claro escribir

entero edad1,edad2;

Este mismo planteamiento lo podemos aplicar a las estructuras

```
typedef struct {
    int codigo;
    char descripcion[41];
    float precio;
} tproducto;
```

Ahora podemos escribir

```
tproducto pro1,pro2;
```

Otro camino para declarar los alias con estructuras es

```
struct producto{
    int codigo;
    char descripcion[41];
    float precio;
};

typedef struct producto tproducto;
tproducto pro1,pro2;
```

5.- Conversión de tipos.

C dispone de varias funciones para convertir tipos. Podemos distinguir entre conversión entre números y conversión numero-cadena.

5.1 Conversión entre tipos de números.

El casting o simplemente cast nos permite hacer una conversión explícita de un tipo de dato a otro, a criterio del programador siempre y cuando estos tipos sean compatibles.

Este cast se realiza a través de un operador de conversión de tipos (type casting operator) y es un recurso a tener en cuenta ya que hay situaciones en que nos puede resultar de gran utilidad.

Hacer uso de un cast es tan sencillo como poner (tipo de dato) delante de la expresión o variable a convertir.

Veamos un ejemplo:

Declaramos una variable de tipo int con un identificador tan creativo como "a" y le realizamos diferentes cast a a para mostrarlo como si fuera un float, un double y un char en un printf. Lo que obtendríamos en pantalla sería lo siguiente:

```
#include <stdio.h>
int main() {
   int a=65;
```

```
    printf("%d, %f, %c",a,(float)a,(char)a);
        return 0;
}
```

El resultado sería…

65, 65.000000, A

El primer valor impreso es obvio, a es un entero y su valor es 65. En el segundo caso se convierte a float y se muestre con 6 decimales. En el tercer caso interpretamos el valor 65 como código ascii y por tanto se muestra el carácter cuyo código ascii es 65, es decir, la A.

5.2 Conversión de cadenas.

La función atoi() convierte una cadena al entero correspondiente siempre que esto sea posible.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int a="65";
    int numero=atoi(a);
    printf("%d",numero);
        return 0;
}
```

La variable numero es entera y tiene el valor de la cadena a convertida a entero. El mismo resultado se obtendría con

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char a[2]="65";
    int numero=atoi(a);
    printf("%d",numero);
        return 0;
}
```

En general, esta función se utiliza cuando hay operaciones aritméticas que implican el uso de estas variables, por ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char a[2]="65";
    char b[2]="15";
    int numero1=atoi(a);
    int numero2=atoi(b);
    float resultado=numero1/numero2;
    printf("%f",resultado);
        return 0;
}
```

Siendo el resultado 4,000000.

La función itoa() realiza la operación inversa, convierte un entero a cadena. Su sintaxis es

char * itoa (int value, char * str, int base);

Podemos verlo major en el siguiente ejemplo

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int numero=65;
   char cadena[10];
   itoa(numero,cadena,10);
      printf("%s",cadena);
      return 0;
}
```

En los ejemplos anterior hemos trabajado con cadenas y enteros. Podemos hacer lo mismo con otros tipos de números, por ejemplo long y float.

```
long int atol ( const char * str ); //Convierte una cadena a entero doble
```

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   char numero[2]="65";
   long int res=atol(numero);

      printf("%ld",res);
      return 0;
}
```

La función atof convierte cadena a flotante.

```
double atof (const char* str);
```

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   char numero[10]="65,25";
   float res=atof(numero);

      printf("%f",res);
      return 0;
}
```