Anexo I

Lenguaje C.

Guion-resumen

1. Introducción

El lenguaje de programación C ha sido utilizado durante muchos años y hoy en día sigue siendo un lenguaje a tener en cuenta. Ha sido utilizado para infinidad de utilidades como compiladores, sistemas operativos, bases de datos, etc. Este detalle de su uso general es quizá una de las diferencias importantes con respecto a otros lenguajes de programación cuyo uso está más relegado a una tarea en concreto. Dicho de otra forma, estamos ante un lenguaje de propósito general. Muchas son sus ventajas: portabilidad, compatibilidad, flexibilidad, etc.

El lenguaje C nació en 1972 en los Laboratorios Bell de la mano de Dennis Ritchie. Tuvo dos grandes revisiones en los años 1989 (C89) y 1999 (C99), este último conocido como el estándar ANSI C es objeto de esta oposición.

Entre las características más generales de este lenguaje destacaremos a modo de primer paso las siguientes:

— Permite la manipulación de bits, bytes y direcciones.

— No lleva a cabo comprobación de errores en tiempo de ejecución. Por supuesto sí lo lleva en tiempo de compilación y de "linkado".

— Existe un conjunto de palabras clave que luego contaremos que en C89 son 32 y en C99 ha sido ampliado en 5 más. No obstante, los diferentes compiladores de este lenguaje pueden añadir más.

— No se permite la creación de funciones dentro de funciones.

— Los bloques de código van encerrados entre llaves.

— Un programa puede estar formado por uno o varios archivos que se compilan cada uno de ellos por separado. Cada uno de esos archivos ha de llevar extensión ***.c**.

— Existen bibliotecas de subrutinas precreadas que podemos incluir en nuestro programa. Estas bibliotecas se encuentran en archivos con extensión ***.h**.

— Las instrucciones terminan en **;**.

— El concepto de verdadero toma el valor 1 y el de falso el 0. Aunque diremos que generalmente cualquier valor diferente de 0 es tomado como verdadero.

El formato general de un programa en C viene dado por el siguiente esquema:

ARCHIVO1.C	ARCHIVO2.C	.....	ARCHIVON.C
declaraciones globales **main**( ) { variables locales; sentencias; } funcion_1_1( ) { variables locales sentencias } ..... funcion_1_n(){ variables locales; sentencias; }	declaraciones globales { funcion_2_1( ) variables locales; sentencias; } ..... funcion_2_n(){ variables locales; sentencias; }		declaraciones globales funcion_n_1( ) { variables locales; sentencias; } ..... funcion_n_n(){ variables locales; sentencias; }

Pudiendo estar todo el código del programa en un solo archivo o en diferentes archivos que luego han de ser **compilados y linkados.**

Los programas en C están formados por una serie de líneas de código que se ejecutan sucesivamente. Todos los programas se dividen en bloques, estos bloques vienen dados por la combinación de "{" de apertura del bloque y "}" de finalización del bloque. La ejecución del programa siempre comienza en la función main(). Esta función es la encargada de llamar a las demás.

Los bloques están formados por instrucciones. En C todas las instrucciones acaban con un ";". Hay varios tipos de instrucciones que iremos estudiando a lo largo de este tema.

- **La función main**

Si el programa es pequeño es probable que la mayor parte del programa se halle dentro de la función main. Cuando el programa comienza a tener un tamaño mayor conviene dejar para la función main solo el cuerpo del programa. Al principio de la función main colocaremos todas las rutinas de inicialización. Se suele procesar en la función main las posibles contraseñas que queramos darle al programa y los mensajes de bienvenida. Si el programa admite parámetros en la línea de órdenes, la función main debe procesarlos, ya que la función main tiene normalmente acceso a la línea de argumentos con que fue ejecutado el programa. Es esta función la primera que se ejecuta y debe ser la que llame a las demás, aunque luego otra función llame sucesivamente a otras funciones. Al terminar la llamada a la función llamante, el control del programa vuelve a main. También debe ser la última en ejecutarse.

2. Definición de las variables

Las variables permiten guardar información. Los principales tipos de datos son los datos numéricos, los caracteres y las cadenas de caracteres. Comencemos creando un programa con una variable numérica entera, "x":

```
#include <stdio.h>
int x = 1;
main() {
    printf(«x vale %d\n», x);
}
```

El valor de las variables es, como su propio nombre indica, variable. Podemos alterar su valor en cualquier punto del programa. La forma más sencilla de hacerlo es mediante una sentencia de asignación. Para asignar un nombre a una variable se escribe su identificador seguido de un = y el nuevo valor. Este tipo de sentencia es el segundo más importante. Así:

```
#include <stdio.h>
int i = 1;
main() {
    printf(«el antiguo valor de i es %d\n», i);
    i = 2;
    printf(«el nuevo es %d\n», i);
}
```

- **Palabras reservadas**

El lenguaje C está formado por un conjunto pequeño de palabras clave. En el estándar C89 hay 32 palabras claves:

auto	break	enum	extern	union	unsigned	void	volatile	while			
float	for	goto	if	int	long	register	case	default	do	double	else
return	short	signed	sizeof	static	struct	typedef	char	const	continue	switch	

A este conjunto de palabras se les denomina palabras reservadas. En el estándar C99 se le han añadido cinco más.

- **Duración de las variables**

Las variables pueden ser de dos tipos: estáticas y dinámicas. Las **estáticas** se crean al principio del programa y duran mientras el programa se ejecute.

Las variables son **dinámicas** si son creadas dentro de una función. Su existencia está ligada a la existencia de la función. Se crean cuando la función es llamada y se destruyen cuando la función o subrutina devuelve el control a la rutina que la llamó. Las variables estáticas se utilizan para almacenar valores que se van a necesitar a lo largo de todo el programa. Las variables dinámicas se suelen utilizar para guardar resultados intermedios en los cálculos de las funciones. Como regla general, una variable es estática cuando se crea fuera de una función y es dinámica cuando se crea dentro de una función.

- **Definición de una variable**

Cuando se define una variable, lo que realmente estamos haciendo es reservando en memoria un espacio para poder almacenar los valores que pueda ir tomando dicha variable a lo largo de la ejecución del programa. La sintaxis que se debe de seguir en C para definir una variable viene dada por:

[Clase_almacenamiento] [modificadores] [tipo_dato] [identificador];

Todas las variables han de ser declaradas antes de ser usadas. Veamos en detalle cada uno de los elementos que forman esta sintaxis:

2.1. Nombres de identificadores

Son los nombres usados para referirse a las variables, funciones, etiquetas y otros elementos del programa. Son nombres construidos por una secuencia de letras, dígitos y el carácter subrayado. Las **reglas** que han de seguir los identificadores son:

— Pueden contener letras, dígitos y el carácter subrayado "_", pero obligatoriamente deben de comenzar por un carácter alfabético o el "_". Es decir, el primer carácter debe ser una letra o un símbolo de subrayado, los caracteres siguientes pueden ser letras, números o símbolos de subrayado.

— No está permitido el uso de espacios en blanco.

— La longitud del identificador vienen dada por el compilador que se utilice (generalmente está en torno a los 32 caracteres).

— Las letras mayúsculas y minúsculas son interpretadas como diferentes.

— No puede ser una palabra reservada.

Ejemplos:

Correcto: x, xx, xxx33333, cont, x_3, _x3, cuenta23, patatas_jamon.

Incorrecto: 1x, hola!, patatas jamon, xxxñ#xxxx

2.2. Tipos de datos

Existen cinco tipos de datos básicos; todos los demás tipos de datos se basan en alguno de estos tipos. El tamaño de estos tipos pueden variar dependiendo del procesador y del compilador utilizado; por ejemplo, los datos de tipo int pueden ocupar 16 o 32 bits dependiendo del procesador que tengamos:

TIPO	BITS	RANGO
char	8	0 a 255
int	16	-32.768 a 32.767
float	32	3,4 E -38 a 3,4 E +38
double	64	1,7 E -308 a 1,7 E +308
void	0	sin valor

Nota: Void *se usa para declarar funciones que no devuelven ningún valor o para declarar funciones sin parámetros.*

2.3. Modificadores de tipos de datos

— **signed:** con signo.

— **unsigned:** sin signo.

— **long:** usado en int hace que estos tengan un tamaño de 32 bits. Usado en double hace que estos tengan un tamaño de 80.

— **short:** usado en int hace que estos tengan un tamaño de 16 bits.

Los modificadores **signed, unsigned, long y short** se pueden aplicar a los tipos base entero y carácter. Sin embargo, **long** también se puede aplicar a **double**.

TIPO	BITS	RANGO
char	8	-128 a 127
unsigned char	8	0 a 255
signed char	8	-128 a 127
int	16	-32.768 a 32.767
unsigned int	16	0 a 65.535
signed int	16	-32.768 a 32.767
short int	16	-32.768 a 32.767
unsigned short int	16	0 a 65.535
signed short int	16	-32.768 a 32767
long int	32	-2147483648 a 2147483647
signed long int	32	-2147483648 a 2147483647
float	32	3,4 E -38 a 3,4 E +38
double	64	1,7 E -308 a 1,7 E +308
long double	80	1.7E-308 a 1.7E+308 o 3.4E-4932 a 1.1E+4932

2.4. Modificadores de acceso

Se utilizan para controlar la forma en que se modifican las variables. Existen dos modificadores de acceso:

— **const.** Las variables de tipo **const** no pueden ser cambiadas durante la ejecución del programa. Por ejemplo: *const int x*;

— **volatile**. A través de este modificador se le indica al compilador que el contenido de la variable puede ser modificado por elementos ajenos al programa a lo largo de su ejecución. Por ejemplo: *volatile float hora*:

2.5. Clase de almacenamiento

• **Alcance de las variables**

Otra característica de las variables es su alcance. El alcance se refiere a los lugares de un programa en los que podemos utilizar una determinada variable. Distinguiremos así dos tipos principales de variables: globales y locales. Una variable es global cuando es accesible desde todo el programa, y es local

cuando solo puede acceder a ella la función que la creó. También hay una norma general para el alcance de las variables: una variable es global cuando se define fuera de una función y es local cuando se define dentro de una función o de un bloque. Dentro de las variables globales hay dos tipos: las que son accesibles por todos los ficheros que componen nuestro programa y las que son accesibles solo por todas las funciones que componen un fichero. Esto es debido a que normalmente los programas en C se fragmentan en módulos más pequeños, que son mas fáciles de manejar y depurar. Por ello, hay veces que interesa que una variable sea accesible desde todos los módulos, y otras solo que sea accesible por las funciones que componen un determinado módulo. Por defecto, todas las variables globales que se creen son accesibles por todos los ficheros que componen nuestro programa.

Existen, por tanto, tres sitios donde se pueden declarar variables: dentro de las funciones (variables locales), en la definición de parámetros de funciones (parámetros formales) y fuera de todas las funciones (variables globales). Normalmente en todo programa en C hay una sección de variables globales. En las variables globales almacenaremos datos que deben ser accesibles a todo el programa. Cuando el programa es pequeño, por ejemplo si consta de un solo fichero, por comodidad se suelen definir todas las variables como globales.

A través de la clase de almacenamiento se especifica la forma en la que se almacenara la variable. Existen cuatro especificadores de almacenamiento, que son:

— **auto**. Es la opción por defecto de tal forma que la variable se considera local siempre y cuando esté definida dentro de una función.

— **extern**. Con este especificador indicamos que la variable esta declarada con un enlace externo en algún otro lugar del programa. Su empleo más típico es cuando el archivo consta de dos o más archivos y en alguno de esos archivos deseamos que enlace con la variable global definida en otro de los archivos.

— **static**. En una variable local el especificador static hace que la variable mantenga su valor en las sucesivas llamadas a la función. Tienen memoria asignada durante toda la ejecución del programa. Su valor es recordado incluso si la función donde está definida acaba y se vuelve a llamar más tarde. En una variable global el especificador static hace que esa variable solo sea conocida en el archivo donde ha sido declarada.

— **register**. El especificador register pide al compilador que mantenga el valor de una variable con ese especificador de forma que se permita el acceso más rápido a la misma. Esto significa colocarla en un registro de la CPU. Solo se puede aplicar a variables locales y a los parámetros formales de una función.

3. Comentarios

Un comentario es una línea que se incluye en el programa, cuya misión consiste en aclarar la función de una parte concreta del programa a otro lector,

o incluso al mismo programador. En C hay dos formas de incluir estos comentarios. La primera es incluir el texto que sirve de comentario al principio de la sección, entre dos símbolos especiales: el /* o principio de comentario y el */ o fin de comentario. Todo el texto que se incluya entre ellos el compilador lo ignora, incluyendo los saltos de línea. Por ejemplo:

```
/* Hola, que tal,
estoy dentro de un comentario */
```

El otro tipo de comentarios se suele usar para señalar una determinada línea del programa. Para ello escribimos el comentario a la derecha de la línea a comenta con "//". Por ejemplo:

```
printf("HOLA, HOLITA"); // imprime HOLA, HOLITA.
```

4. Operadores

4.1. Clasificación de los operadores

Los operadores en C se pueden clasificar de la siguiente forma:

A) Aritméticos

- \- resta
- \+ suma
- * producto
- / división
- % módulo (resto de la división entera)
- — decrementar
- ++ incrementar

B) Relacionales

- > mayor que
- >= mayor o igual que
- < menor que
- <= menor o igual que
- == igual
- != distinto

C) Lógicos

- && y (Conjunción)
- || o (Disyunción)
- ! no (Negación)

D) El operador ?

Exp 1 ? Exp 2 : Exp 3

Se evalúa exp1 si es cierto se evalúa exp2 y toma ese valor para la expresión. Si exp1 es falso evalúa exp3 tomando su valor para la expresión.

Ejemplo:

x=5;

y=x>2 ? 100 : 200;

Resultado: 100

E) Operadores a nivel de bits

- & y
- | o
- ^ o exclusivo
- ~ complemento a 1
- >> desplazamiento a la derecha
- << desplazamiento a la izquierda

F) Operador de molde

Se utiliza para convertir el tipo de dato de un operando. Anteponiendo al operando el tipo entre paréntesis. Ejemplo (int) 3.2;

G) Los operadores de punteros & y *

- & devuelve la dirección de memoria del operando.

  Ejemplo: m=&cont; coloca en m la dirección de memoria de la variable cont.

- * devuelve el valor de la variable ubicada en la dirección que se especifica.

  Ejemplo: q=*m; coloca el valor de cont en q.

H) Operador de asignación

- = Asignación

Forma general: nombre_variable = expresion;

I) Operador de tamaño

- sizeof devuelve el tamaño de una variable o de un tipo de dato.

Es un operador monario que devuelve la longitud, en bytes, de la variable o del especificador de tipo al que precede. El nombre del tipo debe ir entre paréntesis.

Ejemplo: float f;

printf («%f»,sizeof f); Mostrara 4

printf («%d», sizeof (int)); Mostrara 2

J) Otros operadores

Existen otros operadores como son:

- El "**.**" Que permite acceder a elementos individuales en las estructuras y en las uniones.
- El "**->**" que opera igual que el anterior pero trabajando a través de punteros.
- El operador "**,**" que permite encadenar varias expresiones.
- El operador "**[]**" cuyo uso fundamental es en el trabajo con "arrays".
- El operador "**( )**" para aumentar la precedencia de las operaciones a realizar.

K) Abreviaturas en C

- += ; x=x+10 ———— x+=10
- -=; x=x-10 ———— x-=10
- *=; x=x*10 ———— x*=10
- /= x=x/10 ———— x/=10

4.2. Orden de prioridad de los operadores

( ) [ ] -> .

! ~ ++ — (tipo) * (punteros) & sizeof

* / %

+ -

<< >>

< <= > >=

== !=

&

^

|

&&

||

?:

= += -= *= /=

,

Los operadores monarios y "?" tienen prioridad de derecha a izquierda. Los demás de de izquierda a derecha y de arriba abajo.

5. Caracteres especiales (barra invertida)

\n Nueva línea	\f Salto de página
\t Tabulación horizontal	\\ Barra invertida
\b Espacio atrás	\' Comilla simple
\r Retorno de carro	\» Comilla doble

6. Instrucciones

En el lenguaje C, al igual que en la mayoría de los lenguajes de programación, existen los siguientes tipos de instrucciones:

- Selección, condicionales o de control.

- Repetitivas o bucles.
- Salto.

En las sentencias de control y repetitivas su ámbito incluye el bloque que se encierre entre "{ }"; no obstante, si este se omite, su ámbito es única y exclusivamente la siguiente sentencia a ejecutar. Por tanto, cuando necesitamos ejecutar varias sentencias que dependen de una condición o queramos que estas se repitan, utilizaremos la sentencia de tipo bloque de sentencias " { }".

6.1. Las instrucciones de control

A) La sentencia if

La primera sentencia de control es la sentencia if. Admite dos tipos de sintaxis:

```
if (expresion1)
    sentencia1;
```

o:

```
if (expresion1)
    sentencia1;
else
    sentencia2;
```

o:

```
if (expresion) {
    .....
    .....
}
else {
    .....
    .....
}
```

o:

```
if (expresion1) {
    .....
    .....
}
else if(expresion2){
    .....
```

```
        .....
}
...
....
else if(expresionn){
        .....
        .....
}
else {
        .....
        .....
}
```

o:

```
if (expresion1) {
        .....
        .....
}
else if(expresion2){
        if (expresion) {
        .....
        .....
        }
        else {
        .....
        .....
        }
}
else {
        .....
        .....
}
```

Sirve para bifurcar en un punto de programa. La sentencia if permite tomar decisiones al programa. En su primera forma, la sentencia1 solo se ejecuta si el resultado de evaluar la expresion1 es verdadero (distinto de cero). En la segunda forma tenemos dos posibilidades: si al evaluar la expresion1 el

resultado es verdadero se ejecuta la sentencia1, pero si el resultado es falso se ejecuta la sentencia2. En cualquier caso solo una de las dos sentencias se ejecuta.

Tras evaluarse la expresión if y ejecutarse la sentencia adecuada, el programa continua con la línea siguiente a la de la última sentencia del if. Para la sentencia if vale como expresión cualquier expresión valida en C, incluso las asignaciones y llamadas a funciones.

Cuando hay dos if anidados y a continuación hay un else, este else pertenece al último "if". Así, en el caso anterior, el primer else corresponde al segundo if. Si queremos que un else pertenezca al primer if de un if anidado deberemos encerrar al segundo entre paréntesis. Por ejemplo:

```
if (num > 0) {
    if (num == 1) puts(«num es igual a 1»);
}
else
    puts(«num es menor que 0»);
```

B) La sentencia switch

```
switch (variable) {
    case cte1 :
        ..........
        ..........
        break;
    case cte2 :
        ..........
        ..........
        break;
    ..............
    ..............
    default :
        ..........
        ..........
}
```

Switch solo puede comprobar la igualdad. Esta sentencia sirve para agrupar varias sentencias if en una sola, en el caso particular en el que una variable es comparada a diferentes valores, **todos ellos constantes.** Primero se evalúa la expresión de control. Se compara con la expresión de la primera etiqueta case. Si son iguales se ejecuta la sentencia1. Luego se vuelve a comparar la expresión de control con

la etiqueta del segundo case. De nuevo, si son iguales se ejecuta la sentencia2. Se repite el proceso hasta agotar todas las etiquetas case. Si al llegar a la etiqueta default no se ha ejecutado ninguna otra sentencia. Esta es la acción por defecto. La etiqueta default es opcional.

6.2. Bucles

Un bucle es un conjunto de sentencias que se ejecutan repetidamente hasta que se alcanza una condición de fin de bucle o condición de salida.

A) El bucle for

La sintaxis del bucle for es:

for (inicio; control; incremento o decremento)
sentencia;

Este bucle se utiliza para realizar una acción un número determinado de veces. Está compuesto de tres expresiones: la de inicio, la de control y la de incremento, y las sentencias que, si son varias, deben de ir dentro de un bloque ("{ }").

Primero se ejecuta la expresión de inicio. Normalmente, esta es una expresión de asignación a una variable, que le da un valor inicial. Luego se comprueba la expresión de control. Si esta expresión es verdadera, se ejecuta la sentencia o el grupo de sentencias. Si la expresión es falsa, el bucle finaliza. Tras ejecutarse la sentencia se evalúa la expresión de incremento. A continuación se vuelve al segundo paso. El bucle finaliza cuando la expresión de control es falsa.

B) El bucle while

Su sintaxis es la siguiente:

while (expresion)
sentencia;

El bucle while comienza por evaluar la expresión. Si es cierta se ejecutan las sentencias que, si son varias, deben ir dentro de un bloque ("{ }"). Entonces se vuelve a evaluar la expresion1. De nuevo si es verdadera se vuelve a ejecutar la sentencia. Este proceso continua hasta que el resultado de evaluar la expresión es falso.

C) El bucle do / while

La sintaxis de este bucle es:

do
sentencia;
while (expresion);

o:

```
do {
    .....
    .....
} while (condicion);
```

Su funcionamiento es análogo el del bucle while, salvo que la expresión de control se evalúa al final del bucle. Esto nos garantiza que el bucle do-while se ejecuta al menos una vez.

Podemos incluir dentro del bucle un grupo de sentencias como siempre dentro de un bloque ("{ }"); no obstante, este es el único bucle en el que no serían obligatorias llaves para encerrar el grupo de sentencias. Por ejemplo:

```
int i = 5;
do
    printf(«numero %d\n», i);
    —i;
while (i >= 0);
```

6.3. Sentencias de salto

A) La sentencia break

Tiene dos usos:

- Para finalizar un case en una sentencia switch.
- Para forzar la terminación inmediata de un bucle.

Ejemplo:

```
#include<stdio.h>
int main() {
  int i;
  for(i=0;i<=10;i++){
      if(i==3){
      /*break acaba con el for*/
          break;
      }
      printf(«Hola %d\n»,i);
  }
  return(0);
}
```

B) La sentencia exit

Para salir de un programa anticipadamente. Da lugar a la terminación inmediata del programa, forzando la vuelta al sistema operativo. Usa el archivo de cabecera stdlib.h.

Ejemplo:

```
#include<stdio.h>
#include<stdlib.h>
int main() {
  int num;
  printf(«introduce un numero\n»);
  scanf(«%d»,&num);
  if (num==2)    {
        /*al introducir el 2, exit obliga a terminar el
        programa sin ejecutar las sentencias posteriores*/
        exit(0);
  }
  printf(«Hola\n»);
return(0);
}
```

C) La sentencia continue

Hace comenzar la iteración siguiente del bucle, saltando así la secuencia de instrucciones comprendida entre el **continue** y el fin del bucle.

Ejemplo:

```
#include<stdio.h>
int main(){
  int i;
  for(i=1;i<=100;i++)     {
        if((i-7)%10==0 || (i-3)%10==0)
              continue;
        printf(«%d\n»,i);
  }
  return(0);
}
```

D) La sentencia goto

Esta sentencia está totalmente desaconsejada, y su función es la de permitir el salto de un punto del programa a otro a través de un marcador.

- **Diferencias entre las sentencias break y continue en un bucle**

Hay veces en que interesa romper un bucle en una determinada posición, para ejecutar una nueva pasada del bucle o para finalizar su ejecución. Para realizar estos dos tipos de salto disponemos de dos sentencias: la sentencia break y la sentencia continue.

La sentencia break rompe la ejecución de un bucle o bloque de instrucciones y continúa en la instrucción que siga al bucle o bloque. Por ejemplo:

```
int a = 10;
while (a—>=1) {
      if (a ==· 1)
            break;
      printf(«%d\n», a);
}
printf("fuera del bucle");
```

La sentencia continue rompe la ejecución habitual del bucle y procede a evaluar de nuevo la expresión del bucle. Actúa como si se saltase al final del bloque de un bucle. Por ejemplo:

```
int a = 1;
while (a++ < 10) {
      if (a==7)
            continue;
      printf(«%d\n», a);
}
```

7. Arrays

Podemos definir los arrays o tablas de C como una colección de datos del mismo tipo que se denominan o referencian por un mismo nombre común y que son almacenados en posiciones de memoria físicamente contiguas, donde la dirección de memoria más baja corresponde al primer elemento y la más alta al último elemento. Los arrays y los punteros que serán estudiados en la siguiente sección están íntimamente relacionados.

Dentro de los arrays tenemos los siguientes **tipos**:

- Arrays unidimensionales.
- Arrays bidimensionales.
- Arrays multidimensionales.
- Arrays de caracteres.
- Arrays indeterminados.

Un array tiene una dimensión y un tamaño. La dimensión viene dada por el número de pares de "[ ]" que este tenga y el tamaño (no capacidad que ocupa) por la multiplicación de los enteros que se introduce dentro de los corchetes, de tal forma que int x[2][3] tendrá por dimensión dos y por tamaño 2*3. Todos los arrays tienen el 0 como índice de su primer elemento y como último uno menos de su tamaño y los demás tienen la numeración consecutiva.

Ejemplo. char c [10]; array de caracteres de dimensión uno que tiene 10 elementos, desde c[0] hasta c[9].

7.1. Declaración de un array

Para crear un array se sigue la misma sintaxis que para la creación de variables, algo normal si pensamos en que realmente son colecciones de variables, de tal forma que para declarar un array de n elementos de un cierto tipo se introduce la línea:

modificadores tipo identificador [n];

Donde n es una constante de tamaño fijo. Si el array es estático o global, el compilador crea el espacio para la matriz al principio del programa. Si es de tipo automático, reservar el espacio en la pila de datos. Como todos los tipos de datos, un array se puede inicializar. Si el array es estático, por defecto cada elemento se inicializa a 0. Si es dinámico, los valores de cada elemento no están definidos y antes de usarlos los debemos inicializar. Para inicializar un array en el momento de su creación añadiremos tras el identificador y los corchetes de tamaño un = y la serie de valores (tipo nombre_array [tamaño] = {lista de valores};). Cada valor debe ser una constante válida para el tipo de datos del array, y cada valor ir separado del valor precedente mediante una coma. Para abrir y cerrar la serie de valores usaremos las llaves. Por ejemplo:

int x [4]={0,1,2,3 };

char y[] = { 't', 'a', 'i', '\0'};

Los arrays de carácteres que contienen cadenas permiten una inicialización de la forma:

char nombre_array [tamaño]=»cadena»;

Se añade automáticamente el terminador nulo al final de la cadena.

No podemos dar un número de valores mayor al tamaño del array, pero si podemos dar menos de los necesarios. El compilador siempre rellenar los demás con ceros. El compilador siempre asigna el primer valor al primer elemento del array, y los demás los asignan consecutivamente. Como siempre, acabaremos la línea con un ";".

7.2. Arrays multidimensionales

En C se pueden construir arrays de arrays, es decir, tipos de arrays cuyos elementos son a su vez arrays. Dado que ahora necesitaremos un índice para situarnos dentro del array principal y otro más para movernos dentro de cada uno de los nuevos, diremos que los arrays de arrays poseen dos dimensiones. A un array de dos dimensiones se le suele llamar **matriz**, y a un array de una dimensión, **vector**.

Para crear una matriz de enteros se hace de modo análogo a cuando se crea un array, salvo que ahora se añade el nuevo índice entre corchetes. Por ejemplo:

int matriz[8][9];

Declara una matriz de 8 filas por 9 columnas, o 9 por 8 columnas, según queramos representar. La elección de cuál índice representa las filas y cuál las columnas es arbitrario. Podemos usar la norma habitual en matemáticas: el de la izquierda representa filas y el de la derecha columnas.

Es decir, en el caso particular de los arrays bidimensionales se declaran utilizando la siguiente forma general:

tipo nombre_array [tamaño 2ª dim] [tamaño 1ª dim];

Ejemplo ——–> int x [10][20];

Los arrays multidimensionales siguen la misma sintaxis, pero en vez de ser de dos dimensiones, son de la dimensión que nosotros le indiquemos.

7.3. Cadenas de caracteres

Hay un tipo de arrays de especial importancia; las cadenas de caracteres.

Una cadena de caracteres es un array de caracteres que acaba con el carácter nulo. En C siempre las cadenas de caracteres acaban con este carácter. Esto se hace así por dos motivos: el tamaño de la cadena no tiene un límite prefijado: puede ser tan grande como lo permita la memoria. Las operaciones de manipulación de cadenas de caracteres se simplifican bastante. Para inicializar una cadena de caracteres basta crear un array de caracteres, en el que no necesitamos definir el tamaño e inicializarlo con la cadena de caracteres entrecomillada. Observar que el compilador siempre añade un carácter nulo al final, por lo que el tamaño del array es una unidad mayor del aparente. Por ejemplo:

char cadena[ ] = «hola, holita»

Los caracteres especiales como el tabulador \t y el retorno de carro \r se almacenan como un único carácter. El carácter nulo está representado por un 0.

7.4. Acceso a los miembros de un array

Para usar un elemento de un array se utiliza el identificador y el número de orden del elemento. Al primer elemento siempre le corresponde el número 0. Así

printf («%d», vector[0])

imprimiría el contenido del primer elemento del array que definimos antes, que lo habíamos inicializado a 0.

En el lenguaje C no se hace ningún control acerca de si intentamos leer un número de elemento mayor que el último número del array. Esto es lo que se llama sobrepasar el límite, y el compilador deja al programador la tarea de preocuparse por los límites del array. Si los sobrepasamos pueden ocurrir resultados imprevisibles.

7.5. Arrays indeterminados

El tamaño de los arrays es siempre constante y se especifica al crearlo. Hay dos formas de especificar el tipo índice: dándoselo explícitamente al compilador o haciéndolo implícitamente. Para dar un tamaño al array simplemente indicamos el número de elementos entre los corchetes. El otro modo consiste en hacer que sea el compilador el que decida el tamaño. Esto se hace cuando en la creación del array le damos una lista de valores iniciales. En este caso, si omitimos el tamaño, el compilador lo ajusta según el número de elementos que le demos para inicializar el array. Por ejemplo:

int vector[] = { 1, 2, 3, 4, 5, 6 };

Ejemplo de programa:

```
#include<stdio.h>
int main(){
    int datos[10][10],i,j;
    //se carga la matriz con la suma de filas y columnas, se visualiza
    for(i=0;i<10;i++)    {
        for(j=0;j<10;j++)                {
            datos[i][j]=i+j;
            printf(«%d\t»,datos[i][j]);
        }
        printf(«\n»);
    }
      return(0);
}
```

8. Punteros

8.1. Definición de un puntero

Un puntero es una variable que contiene como valor una **dirección de memoria de otra variable**. Un puntero es un nuevo tipo de datos, que no contiene un dato en sí, sino que contiene la dirección donde podemos encontrar el dato. Decimos que un puntero «apunta» a un dato, pudiendo alterar dicho dato a través del puntero.

Para poder usar punteros y direcciones de datos vamos a introducir dos nuevos operadores. El primero es el operador puntero("*"), que nos permite definir las variables como punteros y también acceder a los datos. El otro nuevo operador, el operador dirección ("&"), que nos permite obtener la dirección en la que se haya ubicada una variable en la memoria. El operador dirección es el complementario al operador puntero.

Todo puntero tiene asociado un tipo de datos que es conveniente que coincida con el tipo de dato de la variable a la cual va a apuntar. Un puntero se define igual que una variable normal, salvo que delante del identificador colocaremos un asterisco. Ejemplo: char *c; /* puntero a entero */.

Normalmente, al definir un puntero, lo solemos inicializar para que apunte a algún dato. Disponemos de tres formas de inicializar un puntero:

— Inicializarlo con la dirección de una variable que ya existe en memoria. Ejemplo: char *x = &c;

— Asignarle el contenido de otro puntero que ya esta, inicializado: char *x = &c; char *y = x;

— Inicializarlo con cualquier expresión constante que devuelva un valor.

Siempre se debe inicializar el puntero antes de usarlo. Una vez que el puntero apunta a un objeto o dato en la memoria podemos emplear el puntero para acceder al dato. A este proceso se la llama **desreferenciar el puntero**, debido a que es una operación inversa a obtener la dirección de una variable. Para desreferenciar un puntero se utiliza el operador puntero. Para acceder al dato al que apunta el puntero basta colocar el asterisco * delante del identificador. Como norma de buena escritura no se deja ningún espacio entre el * y el identificador, aunque el compilador lo acepte. Un puntero desreferenciado se comporta como una variable normal. Por ejemplo:

*int x = 100; int *punt = &entero; printf(«%d %d \n», *punt, x);*

Un uso habitual de los punteros es para recorrer los arrays. Para ello, basta con que el puntero apunte al primer elemento del array y luego, usando la aritmética de los punteros, podemos a acceder a todos los elementos del array a través del puntero. Ejemplo:

*int x[ ] ={ 0,1, 2, 3, 4, 5}; int *punt = x;*

En este momento el puntero apunta al primer miembro del array (no se usa el & para indicar el array). Ejemplo: printf(«%d\n», *(punt+2)); // imprimiría 2.

8.2. Arrays y punteros

Un nombre de array sin índice es un puntero al primer elemento del array.

Ejemplo:

```
//Estas sentencias son idénticas:
char p[10]; - p
- &p[0];
int *p, i[10];
p=i; // ambas sentencias ponen el valor 100 en el sexto elemento de i.
i[5]=100;
*(p+5)=100;

int a[10][10];
a=&a[0][0];
a[0][4]=*((*a)+4);

char cad[80], *p1;
p1 = cad;
cad[4] o *(p1+4); // Para acceder al quinto elemento de cada.
```

8.3. Indirección múltiple punteros a punteros

Consiste en que un puntero contiene la dirección de otro puntero que a su vez apunta a una variable.

Ejemplo: int x; int *y=&x; int **z=&y;

Ejemplo:

```
main() {
    int x, *p, **q;
    x=10;
    p=&x;
    q=&p;
    printf(«%d»,**q); /* imprime el valor de x */
    return 0;
}
```

Cuando trabajamos con cadenas de caracteres podemos recurrir a la inialización del puntero a la cadena directamente. Ejemplo: *char *cad="hola, holita"*;

8.4. Arrays de punteros

Se pueden crear arrays formados por punteros.

Ejemplo: Array de punteros a enteros: *int *x [10];*

Para asignar la dirección de una variable entera llamada var al tercer elemento del array de punteros, se escribe:

```
x[2]=&var;
```

Para encontrar el valor de *var*:

```
*x[2];
```

9. Cadenas de caracteres

Como habíamos definido, una cadena de caracteres es un array de caracteres cuyo ultimo carácter es el nulo '\0'. Para definir una cadena de caracteres basta definir un array de caracteres del tamaño conveniente, dejando espacio para el carácter nulo. Por ejemplo:

```
char cad[ ] = «Hola»;
```

La mayoría de las funciones de cadenas de la librería estándar comienzan con el prefijo str y se hayan definidas en el fichero de cabecera <string.h>. Las funciones más importantes de esta librería son:

- **strlen(cad).** Devuelve el tamaño de una cadena de caracteres, sin incluir el carácter nulo de terminación.
- **strcpy(s1,s2).** Copia la cadena s2 en la cadena s1, incluyendo el carácter de terminación y devuelve un puntero a s1. Los dos parámetros que necesita son punteros a caracteres y devuelve un puntero a caracteres.
- **strcat(s1, s2).** Copia la cadena s2 al final de la cadena s1. Para ello busca el carácter de terminación de s1 y a partir de allí va colocando sucesivamente los caracteres de s2, incluyendo el carácter de terminación.
- **strchr(cad, c).** Busca el carácter c a lo largo de la cadena cad. Si lo encuentra devuelve un puntero a la primera posición del carácter. Si falla la búsqueda devuelve un puntero nulo. La función tiene dos par metros, el puntero a la cadena en la que buscar el carácter y el carácter a buscar.
- **strcmp(s1, s2).** Devuelve 0 si s1 y s2 son iguales, menor que 0 si s1<s2 y mayor que 0 si s1>s2.

Aquí tenemos las declaraciones de las funciones anteriores:

char ***strcpy*** *(char *s1, const char *s2);.*

char ***strcat*** *(char *s1, consta char *s2);*

int ***strlen*** *(const char *s1);*

int ***strcmp*** *(const char *s1, const char *s2);*

10. Estructuras, uniones, enumeraciones y typedef

10.1. Estructuras

Una estructura es un tipo de datos compuesto por un grupo de datos, cada uno de los cuales puede ser de un tipo distinto. A cada componente de la estructura se le llama campo.

A) Definición de una estructura

Para la definición de estructuras C dispone de la palabra reservada struct. Para crear una estructura primero comenzamos por definir el tipo de estructura. Para ello se procede de manera parecida a la definición de una variable.

La forma general de una definición de estructura es:

```
struct etiqueta {
        tipo nombre_variable;
        tipo nombre_variable;
        .......
        ........
} variables _de_estructura;
```

Ejemplo:

```
struct estru {
        int x;
        char cad[10];
        float sueldo ;
        double real;
        double imaginario;
};
```

Una vez que hemos definido un tipo de estructura ya podemos definir variables estructuras de dicho tipo. Esto se hace de una forma análoga a la definición de variables normales, esto es, se pone la palabra reservada struct, el identificador del tipo de estructura y el identificador de la nueva estructura. Por ejemplo:

```
struct estru xxx;
```

También se puede definir una variable estructura a la vez que se define el tipo de estructura. Ejemplo:

```
struct estru{
    double x ;
    double y;
} z1, z2;
```

Podemos definir variables estructuras sin tipo específico. Ejemplo:

```
struct {
    int x;
    char y[14];
    int z;
} zorro;
```

A los elementos individuales de la estructura se hace referencia utilizando . (punto). Forma general es: nombre_estructura.elemento

Ejemplo:

```
zorro.z = 12345;
```

B) Arrays de estructuras

Se define primero la estructura y luego se declara una variable array de dicho tipo. Ejemplo:

```
struct dir info_dir [100];
```

Para acceder a una determinada estructura se indexa el nombre de la estructura:

```
info_dir [2].codigo = 12345;
```

C) Punteros a estructuras

Declaración: struct dir * pruntero_dir;

Existen dos usos principales de los punteros a estructuras:

- Para pasar la dirección de una estructura a una función.
- Para crear listas enlazadas y otras estructuras de datos dinámicas.

Para encontrar la dirección de una variable de estructura se coloca & antes del nombre de la estructura. Ejemplo:

```
struct bal {
    float balance;
    char nombre[80];
} persona;
struct bal *p;
p = &persona; //coloca la dirección de la estructura persona en el puntero.
```

No podemos usar el operador punto para acceder a un elemento de la estructura a través del puntero a la estructura. Debemos utilizar el operador flecha "->". Ejemplo: p -> balance.

10.2. Uniones

Una unión es un tipo de datos formado por un campo capaz de almacenar un solo dato pero de diferentes tipos. Dependiendo de las necesidades del programa el campo adopta uno de los tipos admitidos para la unión. Para definir uniones el C utiliza la palabra reservada unión. La definición y el acceso al campo de la unión es análogo al de una estructura. Al definir una variable de tipo unión el compilador reserva espacio para el tipo que mayor espacio ocupe en la memoria. Siempre hay que tener en cuenta que solo se puede tener almacenado un dato a la vez en la variable. En C es responsabilidad del programador conocer qué tipo de dato se esta guardando en cada momento en la unión.

Para definir una unión seguimos la misma sintaxis que para las estructuras. Ejemplo:

```
union un {
    int num1;
    float num2;
} var_union;
```

Define una unión en la que el campo puede ser de tipo entero o de tipo número con coma flotante.

10.3. Enumeraciones

Una enumeración es un conjunto de constantes enteras con nombre. Su sintaxis es:

enum identificador {lista de constantes simbólicas} variables_enumeracion;

Ejemplo: *enum enum { x, y , z=10, t} var_enum*;

En la enumeración anterior cuando var_enum tome el valor x valdra 0, cuando tome el valor y valdrá 1, cuando tome el valor z valdrá 10 y cuando tome el valor t valdrá 11.

10.4. Typedef

Sirve para definir nuevos nombres para los tipos de datos. Ejemplo:

```
typedef float real;
real x;  //estaríamos declarando la x como tipo float
```

11. Funciones

Una función es un módulo o parte de una aplicación informática que forma un bloque o unidad de código. Es el lugar en donde se produce la actividad del lenguaje C. Gracias al empleo de funciones se facilita el mantenimiento y el uso futuro de ellas. El lenguaje C contiene muchas funciones ya predefinidas en bibliotecas (librerías) tanto del sistema como también del programador que puede insertar las suyas. Por tanto, una función es una rutina o conjunto de sentencias que realiza una determinada labor. Las funciones admiten argumentos, que son datos que le pasan a la función las sentencias que la llaman.

- **Definición de una función**

La sintaxis base de toda funcion es:

```
tipo_devuelto nombre_funcion (lista de parámetros) {
    .....
    /* bloque de código */
    .....
}
```

— **tipo_devuelto** especifica el tipo de valor que devuelve la sentencia return de la función.

— **identificador** es el nombre de la función. Debe ser un identificador válido.

— **lista_de_argumentos** es una lista de variables, separadas por comas, que conforman los datos que le pasamos a la función.

La lista de argumentos es opcional. Un ejemplo es la función main(), que en principio no tiene argumentos.

Cuando el programa al ejecutarse alcanza la llave de cierre '}' de la función, esta finaliza y devuelve el control al punto del programa que la llamó.

Para obligar a la función a retornar un determinado valor se utiliza la sentencia return, seguida del valor a retornar. Los tipos de datos escalares son los punteros, tipos numéricos y el tipo carácter. En C no se pueden devolver arrays ni estructuras.

- **Paso de parámetros a una función**

Utilizando la lista de argumentos podemos pasar parámetros a una función. En la lista de parámetros se suele colocar un conjunto de identificadores, separados por comas, que representan cada uno de ellos a uno de los parámetros de la función. El orden de los parámetros es importante. Para llamar a la función habrá que colocar los parámetros en el orden en que la función los espera. Cada parámetro puede tener un tipo diferente. Para declarar el tipo de los parámetros añadiremos entre el paréntesis ')' y la llave '{' una lista de declaraciones, similar a una lista de declaraciones de variables. Ejemplo:

Archivo1.c:

```
#include<ctype.h>
void imp_may(char *cadena) {
   int i;
   for(i=0;cadena[i];i++)
        cadena[i]=toupper(cadena[i]);
}
```

Archivo2.c:

```
#include<stdio.h>
#include<ctype.h>
void imp_may(char *cadena);
int main() {
   char x[80];
   printf(«introduce una cadena de caracteres\n»);
   gets(x);
   imp_may(x);//llamada a la f por referencia
   printf(«\n\nla cadena nueva es\n»);
   printf(«%s\n\n»,x);
   return(0);
}
```

- **Paso de parámetros por valor y por referencia**

En los lenguajes de programación estructurada hay dos formas de pasar variables a una función: por referencia o por valor. Cuando la variable se pasa por referencia la función puede acceder a la variable original. En C todos los parámetros se pasan por valor. La función recibe una copia de los parámetros y variables y no puede acceder a las variables originales. Cualquier modificación que efectuemos sobre un parámetro no se refleja en la variable original. En determinadas ocasiones necesitaremos alterar el valor de la variable que le pasamos a una función. Para ello en el C se emplea el mecanismo de los punteros.

- **Llamada por valor**

Copia el valor de un argumento en el parámetro formal de la subrutina. Los cambios en los parámetros de la subrutina no afectan a las variables usadas en la llamada.

```
int cuad (int x);
main ( ) {
int t=10;
printf («%d %d»,cuad(t),t);
return 0;
}
```

```
cuad (int x)
{
x=x*x;
return(x);
}
```

Salida es "100 10"

- **Llamada por referencia**

Es posible causar una llamada por referencia pasando un puntero al argumento. Se pasa la dirección del argumento a la función, por tanto es posible cambiar el valor del argumento exterior de la función.

```
int x,y;
inter (&x,&y);
```

```
inter (int *x,int *y) {
int temp;
temp=*x;
*x=*y;
*y=temp;
}
```

- **Declaración y comprobación de tipos (Prototipo)**

Al igual que para las variables, cuando una función se va a usar en un programa antes del lugar donde se define, o cuando una función se define en otro fichero (funciones externas), la función se debe declarar.

La declaración de una función consiste en especificar el tipo de datos que va a retornar la función, el identificador de la función y el número de argumentos y su tipo. Una declaración típica de función es:

tipo_devuelto identificador(lista_de_argumentos_con_tipo);

Esto avisa al compilador de que la función ya existe, o que la vamos a definir después.

Ejemplo: void hola(int *x);

- **Funciones recursivas**

Una función se dice recursiva cuando se llama a sí misma.

Ejemplo:

```
int fact (int x)      {
    int factorial;
    if(x==1)
        return 1;
    factorial=x* fact(x-1);
    return factorial;
}
```

12. Memoria dinámica

- malloc (n) reserva una porción de memoria libre de n bytes y devuelve un puntero sobre el comienzo de dicho espacio.
- free (p) libera la memoria apuntada con el puntero p.

Ambas funciones utilizan el archivo de cabecera stdlib.h Si no hay suficiente memoria libre para satisfacer la petición, malloc ( ) devuelve un nulo.

Ejemplo:

```
char *p;
p=malloc(1000000);
```

13. Entrada y salida por consola (estándar)

Un programa en C se comunica con el usuario y con el sistema a través de las funciones de entrada y salida. Con estas funciones se pueden solicitar y enviar datos al terminal del usuario y a otros programas. Además, podemos

elegir entre enviar datos binarios o enviarlos como cadenas de texto. Las funciones de entrada y salida en C más habituales son las que forman parte de la llamada «librería estándar».

13.1. Entrada y salida de caracteres

En la librería estándar se definen las dos principales vías de comunicación de un programa en C: la entrada estándar y la salida estándar. Generalmente ambas están asociadas a nuestro terminal de manera que cuando se imprimen datos en la salida estándar los caracteres aparecen en el terminal, y cuando leemos caracteres de la entrada estándar los leemos del teclado del terminal. La entrada y salida estándar trabaja con caracteres (en modo carácter), con datos o números binarios. Es decir, todos los datos que enviemos a la salida estándar deben ser cadenas de caracteres. Por ello, para imprimir cualquier dato en la salida estándar primero deberemos convertirlo en texto, es decir, en cadenas de caracteres. Sin embargo esto lo haremos mediante las funciones de librería, que se encargan de realizar esta tarea eficientemente.

Comenzaremos con las dos funciones principales de salida de caracteres: **putchar()** y **getchar()**. La función putchar escribe un único carácter en la salida estándar. Su uso es sencillo y generalmente está implementada como una macro en la cabecera de la librería estándar. La función getchar() devuelve el carácter que se halle en la entrada estándar. Esta función tiene dos particularidades. La primera es que aunque se utiliza para obtener caracteres no devuelve un carácter, sino un entero. Esto se hace así ya que con un entero podemos representar tanto el conjunto de caracteres que cabe en el tipo carácter (normalmente el conjunto ASCII de caracteres) como el carácter EOF de fin de fichero. En UNIX es habitual representar los caracteres usando el código ASCII, tanto en su versión de 7 bits como en su versión ampliada a 8 bits. Estos caracteres se suelen representar como un entero que va del 0 al 127 o 256. El carácter EOF entonces es representado con un -1. Además, esto también lo aplicaremos cuando leamos los ficheros binarios byte a byte.

Una tercera función de caracteres que no es muy frecuente es la función **ungetchar()**. Con ella devolvemos al sistema el ultimo carácter que hemos leído con getchar(). No se puede llamar dos veces seguidas a ungetchar. Habitualmente, cuando leemos un conjunto de caracteres de la entrada estándar le pediremos que sean de un determinado tipo. Si, por ejemplo, queremos leer un dato numérico bastará con hacer un bucle que lea números (caracteres numéricos). El bucle normalmente terminará cuando el carácter leído no sea un número. La mejor forma de saber si el siguiente carácter es un número es leerlo. Pero al leerlo, si no es un número ya no estará disponible para futuras lecturas. Aquí es donde se usa ungetchar(). Una vez que hemos comprobado que no es un número lo devolvemos, y así estará listo para la siguiente lectura.

Visto esto podemos seguir con las funciones **gets()** y **puts()**. La función puts() simplemente imprime una cadena de caracteres en la salida estándar. Le debemos proporcionar la dirección donde encontrar la cadena de caracteres. Como ejemplo vamos a dar una implementación sencilla de esta función:

```
void putchar(char *p) {
        while (*p)
        putchar(*p++);
          }
```

realmente la función puts es más complicada, pues devuelve un EOF si ha ocurrido algún error.

Para imprimir datos de un modo más general C dispone de la función **printf()**, que se ocupa de la impresión formateada en la salida estándar.

La función printf() imprime los datos en la salida estándar según una cadena de control. Está definida en la cabecera estándar stdio.h como:

```
int printf(const char *formato, ...);
```

La función printf() tiene varias características peculiares. La primera es que es una función común número variable de argumentos. Normalmente a estas funciones se las llama variadic, y se reconocen porque incluyen en su línea de argumentos el símbolo de elipsis (tres puntos ...). Solo el primer parámetro es obligatorio, y es del tipo puntero constante a carácter. Esta cadena tiene dos funciones: imprimir un mensaje en la salida estándar y formatear los demás argumentos que se la pasan a la función para ser impresos como texto.

- **Funcionamiento de la función printf()**

Si llamamos a la función printf() simplemente con una cadena de caracteres esta función la imprime de modo parecido a como lo hace la función puts(). El prototipo de printf ( ) es:

```
int printf (const char *cad_fmt, ...);
```

Por ejemplo:

```
printf(«Hola, holita\n»);
```

Imprime la cadena «Hola, holita\n» en la salida estándar. Pero, además, la función printf es capaz de imprimir otros tipos de datos como variables numéricas en la salida estándar. Para ello debemos avisar a la función de que le pasamos como argumento una variable, ya que la función no tiene modo alguno de saber si le hemos pasado algún parámetro. El modo de hacerlo es insertando códigos de control en la cadena de formato. Estos códigos normalmente van precedidos del carácter %. Por ejemplo, el código %d representa enteros en formato decimal. Así, la forma de imprimir una variable entera en la salida estándar es:

```
printf(«esto es un entero: %d\n», 10);
```

Cuando printf() se encuentra el código %d en la cadena de formato lee el siguiente argumento de la función, que debe ser un entero, y lo convierte en su representación decimal como cadena de caracteres. La cadena que representa al número sustituye al código %d de la cadena de formato y se imprime la cadena resultante. Hay una gran variedad de códigos de control para formatear los diferentes tipos de datos. Los más importantes son:

- %c un único carácter.
- %d decimal.
- %i decimal.
- %e notación científica.
- %f decimal en coma flotante.
- %o octal.
- %s cadena de caracteres.
- %u decimales sin signo.
- %x hexadecimales.
- %% imprime un signo %.
- %p muestra un puntero.

La cadena de formato consiste en dos tipos de elementos: caracteres que se mostrarán en pantalla y órdenes de formato que empiezan con un signo de porcentaje y va seguido por el código del formato.

Las órdenes de formato pueden tener modificadores que especifiquen la longitud del campo, número de decimales y el ajuste a la izquierda.

Un entero situado entre % y el código de formato actúa como un especificador de longitud mínima de campo. Si se quiere rellenar con ceros, se pone un 0 antes del especificador de longitud de campo.

— %05 rellena con ceros un número con menos de 5 dígitos.

— %10.4f imprime un número de al menos diez caracteres con cuatro decimales.

Si se aplica a cadenas o enteros el número que sigue al punto especifica la longitud máxima del campo.

— %5.7s imprime una cadena de al menos cinco caracteres y no más de siete.

Entre el código de alineación y el código de control podemos insertar un valor de anchura de campo que controla el ancho de la conversión. Por ejemplo:

printf(«:%3d:», 4); / imprime : 3: */*

También podemos especificar un valor que controle el número de dígitos decimales en un valor real. Este valor se coloca tras la anchura de campo precedido de un punto. Por ejemplo:

printf(«%.3f», 3.99999); / imprime 3.999 */*

Para cadenas de caracteres también podemos insertar un valor que permite escoger cuantos caracteres se imprimen de la cadena. Para ello daremos este valor tras un punto, al igual que hacemos para el valor de precisión. Por ejemplo:

printf(«%.4s\n», «Hola, holita\n»); / imprime Hola */*

13.2. La función scanf()

La función scanf() hace el trabajo inverso a la función printf(), es decir, examina la entrada estándar y carga valores en variables. Se define como:

int scanf(const char *formato, ...);

Esta función trabaja de un modo parecido a como lo hace printf(). Necesita una cadena que indica el formato de los datos que se deben leer. La cadena de formato no se imprime, sino que solo sirve para que scanf() determine el tipo de datos a leer. El resto de los argumentos deben ser punteros a las variables donde se deben almacenar los datos leídos. Por ejemplo:

scanf(«%d», &i);

Lee un entero en formato decimal y lo almacena en la variable i. Hay que tener cuidado de pasar siempre punteros a scanf(), por lo que para guardar datos en variables normales deberemos emplear el operador dirección &. Los códigos de control son análogos a los de printf, es decir, %d., %e, %s, ...

La función scanf() es bastante sensible a los errores. Si el usuario introduce los datos incorrectamente, la función scanf() simplemente falla.

Si queremos realizar una función de lectura más robusta podemos realizar lo siguiente:

— Leemos la entrada en un array de caracteres. Para ello, simplemente usaremos la función gets().

— Exploramos el array de caracteres manualmente paso a paso. Para ello, podemos usar la función sscanf().

La función **sscanf** se define como:

int sscanf(const char *s, const char *formato, ...);

Realiza una tarea parecida a scanf(), pero explorando la cadena apuntada por s en vez de la entrada estándar. De este modo podemos ir explorando la

cadena leída previamente con gets() paso a paso e informando al usuario del lugar donde ha cometido un error al introducir los datos.

13.3. Otras funciones para la entrada/salida estándar

- **getche ( ):** lee un carácter del teclado, espera hasta que se pulse una tecla y entonces devuelve su valor. El eco de la tecla pulsada aparece automáticamente en la pantalla. Requiere el archivo de cabecera "conio.h"
- **putcahr ( ):** imprime un carácter en la pantalla.

Los prototipos son:

int getche (void);

int putchar (int c);

Hay dos variaciones de getche( ):

— **getchar ( ):** función de entrada de caracteres definida por el ANSI C. El problema es que guarda en un buffer la entrada hasta que se pulsa la tecla INTRO.

— **getch ( ):** trabaja igual que getche( ) excepto que no muestra en la pantalla un eco del carácter introducido.

- **gets ( ) y puts ( ):** permiten leer y escribir cadenas de caracteres en la consola.

  gets ( ): lee una cadena de caracteres introducida por el teclado y la sitúa en la dirección apuntada por su argumento de tipo puntero a carácter. Su prototipo es:

  *char * gets (char *cad);*

  Ejemplo:

```
main ( ) {
    char cad[12];
    gets (cad);
    return (0);
}
```

  puts ( ): escribe su argumento de tipo cadena en la pantalla seguido de un carácter de salto de línea. Su prototipo es:

  *char * puts (const char *cad);*

14. Entrada y salida con ficheros

14.1. Apertura y cierre de un fichero

Para abrir un fichero, primero debemos crear una variable de tipo puntero a **FILE**. Este puntero permitirá realizar las operaciones necesarias sobre el fichero. Este puntero deberá apuntar a una estructura de tipo FILE. Estas estructuras son creadas por el sistema operativo al abrir un fichero. Para poder inicializar nuestro puntero a fichero bastará llamar a la función **fopen()**. Esta función intenta abrir un fichero. Si tiene éxito creará una estructura de tipo FILE y devuelve un puntero a FILE que apunta a la estructura creada. En caso de no poder abrir el fichero devuelve en puntero nulo. La función **fopen()** se define en la cabecera estándar stdio.h como:

*FILE *fopen(const char * filename, const char *modo);*

Necesita dos argumentos del tipo puntero a carácter. Cada uno de ellos debe apuntar a una cadena de caracteres. El primero indica el nombre del fichero a abrir. En UNIX y otros sistemas se puede especificar con el nombre del fichero el directorio donde se abrirá el fichero. El segundo indica el modo en el que se abrirá el fichero. Hay que tener cuidado en pasar un puntero a cadena de caracteres y no un solo carácter. Es fácil cometer la equivocación de pasar como segundo argumento un carácter 'r' en vez de la cadena «r». Los modos mas frecuentes de abrir un fichero son:

- «r» Abre un fichero de texto que existía previamente para lectura.
- «w» Crea un fichero de texto para escritura si no existe el fichero con el nombre especificado, o trunca (elimina el anterior y crea uno nuevo) un fichero anterior.
- «a» Crea un fichero de texto si no existe previamente o abre un fichero de texto que ya existía para añadir datos al final del fichero. Al abrir el fichero el puntero del fichero queda posicionado al final del fichero.
- «rb» Funciona igual que «r» pero abre o crea el fichero en modo binario.
- «wb» Análogo a «w» pero escribe en un fichero binario.
- «ab» Análogo a «a» pero añade datos a un fichero binario.
- «r+» Abre un fichero de texto ya existente para lectura y escritura.
- «w+» Abre un fichero de texto ya existente o crea uno nuevo para lectura y escritura.
- «a+» Abre un fichero de texto ya existente o crea un fichero nuevo para lectura y escritura. El indicador de posición del fichero queda posicionado al final del fichero.
- «r+b» o «rb+» Funciona igual que «r+» pero lee y escribe en un fichero binario.

- «w+b» o «wb+» Análogo a «w+» pero en modo binario.
- «a+b» o «ab+» Análogo a «a+» pero en modo binario.

Una llamada típica a la función fopen() es la siguiente:

```
FILE *fp;
if (( fp = fopen( «datos», « r»)) = = NULL)
perror( «No se puede abrir el fichero\n»);
```

Para cerrar un fichero basta llamar a la función **fclose** que se define en stdio.h como:

```
int fclose(FILE *fichero);
```

Su argumento es un puntero a una estructura FILE asociada a algún fichero abierto. Esta función devuelve o en caso de éxito y EOF en caso de error.

14.2. Lectura y escritura sobre un fichero

Para leer y escribir en un fichero en modo texto se usan funciones análogas a las de lectura y escritura de la entrada y salida estándar. La diferencia estriba en que siempre se deberá dar un puntero a FILE para indicar sobre qué fichero efectuaremos la operación, ya que se pueden tener simultáneamente abiertos varios ficheros. Las funciones que trabajan con ficheros tienen nombres parecidos a las funciones de entrada y salida estándar, pero comienzan con la letra f. Las más habituales son:

```
• int fprintf (FILE *fichero, const char *formato, ... );   /* trabaja igual que printf() sobre el fichero */
• int fscanf (FILE *fichero, const char *formato, ... );    /* trabaja igual que scanf() sobre el fichero */
• int fputs (const char *s, FILE *fichero );                /* escribe la cadena s en el fichero */
• int fputc (int c, FILE *fichero);                         /* escribe el carácter c en el fichero */
• int fgetc (FILE *fichero);                                /* lee un carácter del fichero */
• char *fgets( char *s, int n, FILE * fichero);             /* lee una línea del fichero */
```

Hay una equivalencia entre las funciones de lectura y escritura estándar y las funciones de lectura y escritura de ficheros. Normalmente las funciones de lectura y escritura estándar se definen en la cabecera estándar como macros. Así la línea:

```
printf(«hola\n»);
```

es equivalente a la escritura en el fichero stdout:

```
fprintf(stdout, «hola\n»);
```

A los ficheros “stdin” y “stdout” normalmente se accede con las funciones de lectura y escritura estándar. Estos ficheros son automáticamente abiertos y cerrados por el sistema. Para escribir en la salida de error estándar se deben usar las funciones de ficheros con el fichero “stderr”. Normalmente en UNIX se redi-

rige la salida de error estándar a la impresora. Esta salida de error es muy útil en los procesos por lotes y cuando se usan filtros. Un filtro es simplemente un programa que lee datos de la entrada estándar, los procesa y los envía a la salida estándar. Por ello, es conveniente que no se mezclen los mensajes de error con el resultado del proceso. Un ejemplo de filtro sería un programa que expande los caracteres de tabulación en espacios en blanco. Si el programa se llama convierte y se quiere procesar el fichero mifichero, se deberá escribir la línea:

cat mifichero | convierte > nuevofichero

Se han usado los mecanismos del UNIX de redirección (> envía la salida estándar de un programa a un fichero), de tubería (| conecta la salida estándar de un programa con la entrada estándar de otro) y la utilidad cat, que envía un fichero a la salida estándar.

14.3. Lectura y escritura de datos binarios

Para leer y escribir grupos de datos binarios, como por ejemplo arrays y estructuras, la librería estándar provee dos funciones: fread() y fwrite().

Se declaran en stdio.h como:

```
size_t fread(void *p, size_t longitud, size_t numelem, FILE *fichero);
size_t fwrite(void *p, size_t longitud, size_t numelem, FILE *fichero);
```

La función fread() lee del fichero pasado como último argumento un conjunto de datos y lo almacena en el array apuntado por p. Se debe especificar en longitud la longitud del tipo de datos a leer y en numelem el número de datos a leer. La función fwrite() se comporta igual que fread() pero escribe los datos desde la posición apuntada por p en el fichero dado. Como siempre, para usar estas funciones, se debe abrir el fichero y cerrarlo después de usarlas. Por ejemplo, para leer un array de 100 enteros:

```
int array[100];
FILE *fp;
fp = fopen(«mifichero», «rb»);
fread(array, sizeof(int), 100, fp);
fclose(fp);
```

Estas funciones devuelven el número de elementos leídos. Para comprobar si ha ocurrido un error en la lectura o escritura se usará la función ferror (FILE *fichero), que simplemente devuelve un valor distinto de 0 si ha ocurrido un error al leer o escribir el fichero pasado como argumento.

Al escribir datos binarios en un fichero se deben tener en cuenta consideraciones de portabilidad. Esto es debido a que el orden en que se almacenan los bytes que componen cada tipo de datos en la memoria puede variar de unos sistemas a otros, y las funciones fread() y fwrite() los leen y escriben según estén en la memoria.

14.4. Operaciones especiales con los ficheros

Para comprobar si se ha alcanzado el fin de fichero, por ejemplo cuando se lee un fichero binario con fread(), se puede emplear la función **feof**(), que se define en stdio.h como:

*int feof (FILE *fichero);*

Esta función devuelve un o si no se ha alcanzado el fin de fichero y un valor distinto de o si se alcanzó el fin de fichero.

Para comprobar si ha ocurrido un error en la lectura o escritura de datos en un fichero disponemos de la función **ferror**, que se declara en stdio.h como:

*int ferror (FILE *fichero);*

Esta función devuelve un valor distinto de o si ha ocurrido algún error en las operaciones con el fichero y un o en caso contrario. Estas dos funciones trabajan leyendo los indicadores de fin de fichero y error de la estructura FILE asociada a cada fichero. Podemos limpiar ambos indicadores utilizando la función **clearerr**(), que se define en stdio.h como:

*void clearerr (FILE *fichero);*

14.5. Posicionamiento del indicador de posición del fichero

Cuando se manejan ficheros de acceso aleatorio se necesita poder colocar el indicador de posición del fichero en algún punto determinado del fichero.

Para mover el puntero del fichero la librería estándar proporciona la función fseek(), que se define en stdio.h como:

*int fseek (FILE *fichero, long desplazamiento, int modo);*

La función devuelve un 0 si ha tenido éxito y un valor diferente en caso de error. El argumento desplazamiento señala el número de caracteres que hay que desplazar el indicador de posición. Puede ser positivo o negativo, o incluso 0, ya que hay tres modos diferentes de desplazar el indicador de posición. Estos modos se indican con el argumento modo. En stdio.h se definen tres macros que dan los posibles modos. La macro SEEK_SET desplaza al indicador de posición desde el comienzo del fichero. La macro SEK_CUR desplaza el indicador de posición desde la posición actual y, la macro SEEK_END desplaza al indicador de posición desde el final del fichero. Para este último modo se debe usar un valor de desplazamiento igual o menor que 0.

Para ver en qué posición se halla el puntero del fichero se puede usar la función ftell(), que se define en stdio.h como:

*long ftell (FILE *fichero);*

Para un fichero binario ftell() devuelve el número de bytes que está desplazado el indicador de posición del fichero desde el comienzo del fichero.

Además, para llevar el indicador de posición al comienzo del fichero, tenemos la función rewind(), que se define en stdio.h como:

*void rewind(FILE * fichero);*

Esta función simplemente llama a fseek (fichero, oL, SEEK_SET) y luego limpia el indicador de error.

14.6. Otras operaciones con ficheros

La librería estándar proporciona algunas funciones adicionales para manejar ficheros. Por ejemplo, la función **remove**(), que se define en stdio.h como:

*int remove(const char *nombrefichero);*

Esta función elimina el fichero de nombre nombrefichero. Conviene cerrar el fichero antes de eliminarlo. También disponemos de una función para renombrar el fichero, la función **rename**(), definida en stdio.h como:

*int rename(const char *antiguo, const char *nuevo);*

intenta renombrar al fichero de nombre antiguo. Si tiene éxito devuelve un o. Hay que asegurarse antes de que no existía un fichero de nombre nuevo.

Otra función para abrir ficheros es **freopen**(), que se define en stdio.h como:

*FILE *freopen(const char *nombre, const char *modo, FILE *fichero);*

Esta función cierra el fichero pasado como tercer argumento y lo abre con el nuevo nombre y modo especificado. Devuelve un puntero a FILE que apunta al nuevo fichero abierto, o un puntero nulo en caso de error, tal y como lo hace fopen().

15. El preprocesador

El preprocesador en el lenguaje C es una herramienta que convierte el programa fuente desarrollado con extensión *.c en un fichero compilable con extensión *.l siendo este el primer paso que se efectúa en la fase de compilación de un programa C con el fin de obtener el programa objeto con extensión *.obj. El preprocesador utiliza sentencias llamadas directivas de compilación que son órdenes dirigidas al compilador. Estas sentencias utilizan el símbolo # como carácter específico antes de la sentencia.

Es una parte de la compilación en la que se hacen algunas tareas sencillas. Las fundamentales son:

- Supresión de comentarios.

- Expansión de macros.
- Inclusión del código de las cabeceras.
- Conversión de las secuencias de escape en caracteres dentro de cadenas de caracteres y de constantes de tipo carácter.

15.1. Directivas de preprocesado

Para realizar las diferentes acciones que admite el preprocesado se dispone de una serie de directivas de preprocesado, que son como comandos que instruyen al preprocesador para realizar las expansiones. Todas las directivas del preprocesador comienzan con el carácter # seguida del nombre de comando. El signo # debe estar al comienzo de una línea, para que el preprocesador lo pueda reconocer.

- **Directiva include (inclusión de ficheros)**

Una de esas directivas es #include. Esta directiva debe ir seguida de un nombre de fichero. El nombre debe ir entrecomillado o encerrado entre signos de mayor y menor. Lo que hace el preprocesador es sustituir la línea donde se halla la directiva por el fichero indicado. Por ejemplo:

#include <stdio.h>

#include «stdio.h»

La diferencia entre encerrar el nombre del fichero entre comillas o entre signos de mayor y menor es que al buscar el fichero con las comillas la búsqueda se hace desde el directorio actual, mientras que entre signos de mayor y menor la búsqueda se hace en un directorio especial. Este directorio varía con la implementación, pero suele estar situado en el directorio del compilador. El preprocesador y el compilador ya conocen dónde se ubica el directorio. Todas las cabeceras estándar se hallan en ese directorio.

Se puede incluir cualquier tipo de fichero fuente, pero lo habitual es incluir solo ficheros de cabecera.

Hay que tener en cuenta que el fichero incluido es preprocesado. Esto permite expandir algunos tipos de macros y ajustar la cabecera al sistema mediante las directivas de preprocesado. Para ello se suelen usar macros que actúan como banderas.

- **Directiva define (definición de macros)**

En C una macro es un identificador que el preprocesador sustituye por un conjunto de caracteres. Para definir una macro se dispone de la directiva #define. Su sintaxis es:

#define identificador conjunto de caracteres

Se utiliza habitualmente en los ficheros de cabecera para definir valores y constantes. Por ejemplo:

```
#define x 1
```

- **Directivas condicionales**

  Las directivas de compilación condicional son: #if, #endif, #else, #elif, #ifdef y #ifndef.

- **Otras directivas**

  Existen otras directivas como son: #undef, #line, #pragma y #error.