Lenguaje de Programacion C I

Forma general de un programa en C

Declaraciones globales

main( )

{

variables locales

sentencias

}

f1( )

{

.....

}

...

...

fn ( )

{

.....

}

Nombre de indentificadores

Son los nombres usados para referirse a las variables, funciones, etiquetas y otros objetos definidos por el usuario.
La longitud de un identificador en Turbo C puede variar entre 1 y 32 caracteres. El primer carácter debe ser una letra o un símbolo de subrayado, los caracteres siguientes pueden ser letras, números o símbolos de subrayado.

Correcto −−−−−> cont, cuenta23, balance_total

Incorrecto −−−−−> 1cont, hola!, balance...total

En C las mayúsculas y las minúsculas se tratan como distintas.

Tipos de datos

Existen cinco tipos de datos atómicos:

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

(*) El **void** se usa para declarar funciones que no devuelven ningún valor o para declarar funciones sin parámetros.

Modificadores de tipos

signed

unsigned

long

short

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	64	1,7 E −308 a 1,7 E +308

Modificadores de acceso

Las variables de tipo **const** no pueden ser cambiadas durante la ejecución del programa. Por ejemplo,

const int a;

Declaración de variables

Todas las variables han de ser declaradas antes de ser usadas. Forma general:

tipo lista_de_variables; int i,j,l;

short int si;

Existen 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).

Variables externas

Si una función situada en un fichero fuente desea utilizar una variable de este tipo declarada en otro fichero, la debe declarar (o mejor dicho referenciar) con la palabra **extern**.

Archivo 1 Archivo 2

int x,y; extern int x,y;

char ch; extern char ch;

main ( ) void func1( )

{ {

x=120; x=y/10;

. . . . . .

} }

Variable estáticas (static)

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. Ejemplo:

```
series (void)

{

static int num;

num=num+23;

return (num);

}
```

Variables registro

El especificador register pide a Turbo C que mantenga el valor de una variable con ese especificador de forma que se permita el acceso más rápido a la misma. Para enteros y carácteres esto significa colocarla en un registro de la CPU.

Sólo se puede aplicar a variables locales y a los parámetros fomales de una función.

Son ideales para el control de bucles.

```
pot_ent (int m, register int e)

{

register int temp;

temp=1;

for ( ; e; e−−) temp *=m;

return (temp);

}
```

Sentencias de asignación

Forma general: nombre_variable = expresion;

-

Abreviaturas en C

x=x+10 <----------> x+=10

x=x−10 <----------> x−=10

Conversión de tipos

Se da cuando se mezclan variables de un tipo con variables de otro tipo.

El valor de la derecha de la asignación se convierte al tipo del lado izquierdo.

Puede haber pérdida de los bits más significativos en un caso como: short = long

Inicialización de variables

Tipo nombre_variable = constante;

char c='a';

int primero=0;

float balance=123.23;

Todas las variables globales se inicializan a cero sino se especifica otro valor inicial. Las variables locales y register tendran valores desconocidos antes de que se lleve a cabo su primera asignación.

Constantes

Tipo dato Ejemplo de constantes

char 'a' '\n' '9'

int 1 123 −234

float 123.23

Una constante de tipo cadena de caracteres está constituida por una secuencia de caracteres entre comillas dobles "Hola".

Carácteres con barra invertida

\n Nueva línea

\t Tabulación horizontal

\b Espacio atrás

\r Retorno de carro

\f Salto de página

\\ Barra invertida

\' Comilla simple

\" Comilla doble

Operadores

En C hay tres clases de operadores: aritméticos, relacionales y lógicos, y a nivel de bits.

Aritméticos

− resta

+ suma

* producto

/ división

% módulo (resto de la división entera)

−− decrementar

++ incrementar

x=10; x=10;

y=++x; y=x++;

y=11 y=10

Relacionales

En C cierto es cualquier valor distinto de cero. Falso es cero.

> mayor que

>= mayor o igual que

< menor que

<= menor o igual que

== igual

!= distinto

Lógicos

&& y

|| o

! no

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=10:

y=x>9 ? 100 : 200 −−−−−−−−> y = 100

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

& (la dirección de)

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

Ejemplo: q=*m; coloca el valor de cont en q. *(en la dirección)

Sizeof

Es un operador monario que devuelve la longitud, en bytes, de la variable o del especificador de tipo al que precede.

Ejemplo: flat f;

printf ("%f",sizeof f); Mostrara 4

printf ("%d", sizeof (int)); Mostrara 2

El nombre del tipo debe ir entre paréntesis.

ESTRUCTURAS CONDICIONALES

If

if (expresion) {

.....

.....

}

else {

.....

.....

}

Switch

switch (variable) {

case cte1 :

.....

.....

break;

case cte2 :

.....

.....

break;

.....

.....

default :

.....

.....

}

Switch sólo puede comprobar la igualdad.

BUCLES

For

for (inicialización; condición; incremento) sentencia

inicialización −−−−> asignación

condición −−−−> expresión relacional

Ejemplo: for (x=1; x<=100; x++) printf ("%d",x); Imprime los números del 1 al 100

While

while (condición) sentencia;

Ejemplo: while (c!='A') c=getchar( );

Do / While

Analiza la condición al final.

do {

.....

.....

} while (condición);

Break

Tiene dos usos:

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

Exit

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

Ejemplo: #include <stdlib.h>

main (void)

{

if (!tarjeta_color( )) exit(1);

jugar( );

}

-

Continue

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

do {

scanf("%d",&num);

if (x<0) continue;

printf("%d",x);

} while (x!=100);

Funciones

tipo nombre_funcion (lista de parametros)

{

.....

.....

}

tipo, especifica el tipo de valor que devuelve la sentencia return de la función.

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 ( ) cuad (int x)

{ {

int t=10; x=x*x;

printf ("%d %d",cuad(t),t); return(x);

return 0; }

}

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 (int *x,int *y)

inter (&x,&y); {

int temp;

temp=*x;

*x=*y;

*y=temp;

}

Arrays

Todos los arrays tienen el 0 como índice de su primer elemento.

char p [10]; array de carácteres que tiene 10 elementos, desde p[0] hasta p[9].

Para pasar arrays unidimensionales a funciones, en la llamada a la función se pone el nombre del array sin índice.

Ejemplo:

main ( ) Si una función recibe un array unidimensional, se puede

{ declarar el parámetro formal como un puntero, como un

int i[10]; array delimitado o como un array no delimitado.

func1 (i);

}

func1 (int *x) /puntero/

func1 (int x[10]) /array delimitado/

func1 (int x[ ]) /array no delimitado/

Inicialización de arrays

Forma general de inicialización de un array:

tipo nombre_array [tamaño] = {lista de valores};

lista de valores, es una lista de constantes separadas por comas, cuyo tipo es compatible con el tipo del array. La primera constante se coloca en la primera posición del array, la segunda constante en la segunda posición y así sucesivamente.

Ejemplo: int i[10]={1,2,3,4,5,6,7,8,9,10};

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.

Ejemplo:

char cad[5]="hola"; equivalentes char cad[5]={'h','o','l','a','\o'};

Es posible que C calcule automáticamente las dimensiones de los arrays utilizando arrays indeterminados. Si en la inicialización no se especifica el tamaño el compilador crea un array suficientemente grande para contener todos los inicializadores presentes.

char e1[ ]="error de lectura \n";

-

Cadenas

Aunque C no define un tipo cadena, estas se definen como un array de carácteres de cualquier longitud que termina en un carácter nulo ('\0').

Array que contenga 10 caracteres: char s[11];

Una constante de cadena es una lista de caracteres encerrada entre dobles comillas.

Funciones de manejo de cadenas

Archivo de cabecera **string.h**

char ***strcpy** (char *s1, const char *s2); copia la cadena apuntada por s2 en la apuntada por s1. Devuelve s1.

char ***strcat** (char *s1, consta char *s2); concatena la cadena apuntada por s2 en la apuntada por s1, devuelve s1.

int **strlen** (const char *s1); devuelve la longitud de la cadena apuntada por s1.

int **strcmp** (const char *s1, const char *s2); compara s1 y s2, devuelve 0 si con iguales, mayor que cero si s1>s2 y menor que cero si s1<s2. Las comparaciones se hacen alfabéticamente.

Arrays Bidimensionales

Se declaran utilizando la siguiente forma general:

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

Ejemplo −−−−−> int d [10][20];

Cuando se utiliza un array bidimensional como argumento de una función realmente sólo se pasa un puntero

al primer elemento, pero la función que recibe el array tiene que definir al menos la longitud de la primera dimensión para que el compilador sepa la longitud de cada fila.

Ejemplo: función que recibe un array bidimensional de dimensiones 5,10 se declara así:

```
func1 (int x[ ][10])

{

.....

}
```

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;
```

Esto también se puede aplicar con los arrays de dos o más direcciones.

```
int a[10][10];

a=&a[0][0];

a[0][4]=*((*a)+4);
```

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(1000);
```

Estructuras

La forma general de una definición de estructura es:

```
struct etiqueta {

tipo nombre_variable;

tipo nombre_variable;

.....

.....

} variables _de_estructura
```

Ejemplo:

```
struct dir {

char nombre[30];

char calle[40];

char ciudad[20];

char estado[3];

unsigned long int codigo;

} info_dir;
```

A los elementos individuales de la estructura se hace referencia utilizando . (punto).

Ejemplo:

```
info_dir.codigo = 12345;
```

Forma general es: nombre_estructura.elemento

Una estructura puede inicializarse igual que los vectores:

```
struct familia {

char apellido[10];
```

```
char nombrePadre[10];

char nombreMadre[10];

int numerohijos;

} fam1={"Garcia","Juan","Maria",7};
```

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;
```

Paso de estructuras a funciones

Cuando se utiliza una estructura como argumento de una función, se pasa la estructura íntegra mediante el uso del método estándar de llamada por valor.

Ejemplo:

```
struct tipo_estructura {

int a,b;

char c;

};

void f1 (struct tipo_estructura param);

main ( )

{

struct tipo_estructura arg;

arg.a = 1000;

f1(arg);

return 0;
```

```
}

void f1 (struct tipo_estructura param)

{

printf ("%d",param.a);

}
```

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 p)

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 −>

p −> balance

Tipo enumerado

enum identificador {lista de constantes simbólicas};

Ejemplo: enum arcoiris {rojo, amarillo, verde, azul, blanco};

(realmente asigna rojo=0, amarillo=1, ...)

printf ("%d %d", rojo, verde); imprime 0 2 en pantalla

Podemos especificar el valor de uno o más símbolos utilizando un inicializador. Lo hacemos siguiendo el símbolo con un signo igual y un valor entero.

enum moneda {penique, niquel, diez_centavos, cuarto=100, medio_dolar, dolar};

Los valores son: penique 0, niquel 1, diez_centavos 2, cuarto 100, medio_dolar 101, dolar 102

-

Punteros

int x=5, y=6;

int *px, *py;

px=py; copia el contenido de py sobre px, de modo que px apuntará al mismo objeto que

apunta py.

*px=*py; copia el objeto apuntado por py a la dirección apuntada por px.

px=&x; px apunta a x.

py=0; hace que py apunte a nada (NULL).

px++; apunta al elemento siguiente sobre el que apuntaba inicialmente

Se puede sumar o restar enteros a y de punteros.

p1=p1+9; p1 apunta al noveno elemento del tipo p1 que está más allá del elemento al

que apunta actualmente.

Punteros y arrays

char cad[80], *p1;

p1 = cad p1 ha sido asignado a la dirección del primer elemento del array

cad.

Para acceder al quinto elemento de cad se escribe:

cad[4] o *(p1+4)

Arrays de punteros

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]

Punteros a punteros

puntero −−−−−−−−−−> variable Indirección simple

puntero −−−−−−−−−> puntero −−−−−−−−−> variable Indirección múltiple

float **balancenuevo;

balancenuevo no es un puntero a un número en coma flotante, sino un puntero a un puntero a float.

Ejemplo:

```
main(void)

{

int x, *p, **q;

x=10;

p=&x;

q=&p;

printf("%d",**q); /* imprime el valor de x */

return 0;

}
```

E/S por consola

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);

Ejemplo:

```
main ( ) /* cambio de mayúscula / minúscula */

{

char car;

do {

car=getche( );

if (islower(car)) putchar (toupper (car));

else putchar (tolower (car));

} while (car=!'.')

}
```

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[80];

gets (cad);

printf ("La longitud es %d", strlen (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);
```

-

E/S por consola con formato

printf ( ) El prototipo de printf ( ) es:

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

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.

%c un único caracter

%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

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.

scanf ( )

Su prototipo es:

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

Ejemplo:

scanf ("%d",&cuenta);

Lenguaje de Programacion C II

Para familiarizarnos con el lenguaje C lo mejor es comenzar haciendo pequeños programas. El objetivo ser dar una primera impresión de como trabaja el C, sin profundizar demasiado en todas sus características. En esta primera parte se abordar los métodos que son comunes a todos los lenguajes de programación estructurada.

Los programas en C est n formados por una serie de líneas de código que se ejecutan sucesivamente. Todos los programas se dividen en bloques. Cada bloque de código se encierra en funciones. La ejecución del programa siempre comienza en la función main(). Esta función es la encargada de llamar a las demás. Para escribir la función main se coloca al principio de la función el siguiente código:

main()

{

Luego escribimos todas las líneas de código. A cada línea de código le llamaremos sentencia. Después de cada sentencia escribiremos un ';'. En C todas las sentencias acaban con un . Hay varios tipos de sentencia. Las más comunes la llamada a una función. Cuando hace esto el programa llama a la función. Entonces se ejecuta el código de la función. Cuando la función finaliza la ejecución devuelve el control a la función main(). Las funciones son muy difíciles de reconocer, pues a continuación de su nombre van los paréntesis.

Para acabar el código de una función siempre escribiremos al principio de una nueva línea una llave de cierre '}'.

Por ejemplo:

#include <stdio.h>

main()

```
{

printf("hola, mundo\n");

}
```

En este programa la función main consta de una sola sentencia, que es la llamada a la función printf(). La función printf imprime en la salida habitual, que generalmente es el terminal en el que trabajamos, el mensaje que le demos. El mensaje que queremos imprimir es hola, mundo, seguido de un avance del cursor al principio de la línea siguiente. El mensaje lo debemos encerrar entre comillas ", para que el compilador sepa cual es su longitud.

La función printf() es una función que pertenece a la librería estándar del C. Esta librería tiene un conjunto muy amplio de funciones listas para usar, lo que nos evitar el trabajo de usarlas. Para usar esta librería debemos avisar al compilador. Por ello incluimos como primera línea del programa la línea #include <stdio.h>.

En esta librería hay un montón de funciones útiles. Para comenzar daremos una lista de alguna de ellas, con una pequeña explicación de lo que hacen, aunque no nos extenderemos en su uso. Cuando comencemos a usarlas rápidamente nos daremos cuenta de su uso. Estas funciones son:

printf(mensaje) imprime un mensaje en el terminal

putchar(carácter) escribe un carácter en el terminal

getchar() recoge un carácter del terminal

scanf(variables) lee variables del terminal

gets() lee una línea del terminal

De estas funciones, printf() y scanf() son las mas complicadas de usar, y las que admiten m s posibilidades de funcionamiento. De echo son unas de las mas complicadas de la librería estándar.

Estructura de un programa en C.

Un programa en C es simplemente un fichero de caracteres que contiene un conjunto de instrucciones que un programa especial, el compilador o traductor, se encargar de transformar en un programa que la computadora pueda ejecutar. Un programa normalmente suele estar compuesto de tres partes:

−la sección de variables, que especifica los datos y sus tipos que vamos a manejar a lo largo del programa.

la función principal, que se suele llamar "main", que ser la que defina la estructura del programa.

las funciones o subrutinas auxiliares, que son llamados por la rutina principal, la función main. Estas subrutinas se suelen colocar después de la función main.

Cuando la envergadura del programa es grande se suele fragmentar el programa en varias partes, incluyendo cada parte en un fichero separado. El lenguaje C define el m todo que debemos seguir para separar las diferentes partes del programa. Normalmente colocaremos en cada fichero todas las subrutinas y funciones que se encarguen de una tarea del programa. As¡ la función main ir "llamando" a las subrutinas a medida que las vaya necesitando.

Un primer vistazo a la programación estructurada: las funciones

Como hemos visto en C los programas constan de una rutina principal, de nombre main y una serie de subrutinas asociadas. En C las rutinas se crean mediante una función. Una función es un fragmento de código que realiza una tarea. En C las funciones siempre tienen un nombre, que viene dado por un identificador. Por ejemplo main es el identificador de la función main, que es la rutina principal de todo programa en C. Para escribir la función main solo tenemos que colocar al principio de una línea su identificador, seguido de los caracteres '(' y ')'. Es costumbre entre los programadores de C de escribir el primer paréntesis pegado al identificador de la función, ya que facilita enormemente su clasificación como función. En realidad esto no es necesario ya que el compilador salta los espacios automáticamente. A continuación del ultimo paréntesis se escribe el carácter de abrir llaves '{' . También es costumbre entre los programadores el escribir esta llave en la línea siguiente, y no a continuación del paréntesis, aunque al compilador le da igual, y no protestar si introducimos o quitamos espacios en blanco. Recordemos que decíamos que el C es un lenguaje de formato libre. A continuación de la llave se escribe el código de la función. Por último, al principio de una nueva línea debemos incluir otra llave '}', que servir de cierre de nuestra función. Un ejemplo de función main puede ser:

```
main()

{

printf("Hola, mundo\n");

}
```

Esta función casi constituye un programa completo, salvo que al intentar compilarlo el compilador se puede quejar y decirnos que no conoce el

identificador printf. Si repasamos nuestra lista de palabras reservadas veremos que no aparecen ni la palabra main ni la palabra printf, sin embargo el sistema sólo se quejar probablemente de la palabra printf. Esto se debe a que la palabra main suele estar predefinida como función en la mayoría de los sistemas en C. La función printf simplemente imprime en la salida del sistema (generalmente la pantalla del usuario) la frase que le demos. La función printf forma parte de lo que se conoce como librería estándar del C. Con casi todos los compiladores se suele incluir un conjunto de funciones predefinidas que realizan las funciones mas usuales de los lenguajes de programación: entrada y salida de datos por pantalla, creación y manipulación de ficheros y otras muchas funciones. En esto el lenguaje C es muy potente, pues la librería estándar es muy amplia.

Para avisar al compilador que vamos a usar la función printf simplemente debemos añadir una línea al principio de nuestro programa, quedando ahora el programa como sigue:

```
#include <stdio.h>

main()

{

printf("Hola, mundo\n");

}
```

Este es ya un programa completo en C, uno de los m s simples que se pueden crear. Simplemente escribe un mensaje de salida en nuestro terminal. El significado de sus partes se ir viendo m s adelante.

El código de un programa en C

El C es un lenguaje de formato libre. Esto quiere decir que consiste en que un programa est formado por comandos que est n separados por espacios en blanco y también. Para ello el C considera como espacios en blanco, no sólo el carácter blanco ' ', sino también el carácter de tabulador '\t' y el carácter de nueva línea '\n' o '\r'. El numero de espacios que empleemos no varía el significado del programa. Aunque al compilador del lenguaje le da igual el número de espacios en blanco que insertemos entre los comandos, por motivos de legibilidad seguiremos una serie de normas. La primera de ellas hace referencia a los 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 incluyamos entre ellos el compilador lo ignorar , incluyendo los saltos de línea. Por ejemplo si una sección del programa se encarga de ofrecer los resultados finales del programa, podríamos incluir en el código el siguiente comentario:

/* esta sección se encarga de imprimir los datos en la impresora asignada */

Y aquí¡ iría el resto del programa.

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. Por ejemplo, si en una línea aparece el comando gets(), podríamos poner a la derecha en un comentario lo que hace:

gets(linea); /* recoge una línea del teclado */

Este tipo de comentario es el que usaremos en muchas de la explicaciones de ahora en adelante.

La sección de 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 sólo fichero, por comodidad se suelen definir todas las variables como globales. Esto tiene como ventaja que no deberemos definir el tipo de las variables en las funciones auxiliares, ya que ser n directamente visibles para ellas. Como inconveniente tenemos que al ser las variables globales accesibles por todas las funciones podremos modificarlas por error, por lo que pueden aparecer errores difíciles de depurar. La mejor estrategia es dejar como variables globales sólo las estrictamente necesarias, definiendo en cada función las variables que necesitemos.

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 sólo el cuerpo del programa. Al principio de la función main colocaremos todas las rutinas de inicialización. Si vamos a manejar algún fichero a lo largo del programa la función main es un buen lugar para abrirlo. También se suele procesar en la función main los mensajes de bienvenida, en los que se muestra el nombre del autor y demás información. Si el programa admite par metros 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.

Las variables

El siguiente paso para programar en C consiste en presentar las variables. Las variables permiten guardar información. Los principales tipos de datos son los datos numéricos, los caracteres y las cadenas de caracteres. En principio situaremos las variables al principio del programa.

Para utilizar una variable situaremos antes de la función main el nombre de la variable precedido de su tipo. En principio vamos a usar tres tipos de variables: los enteros, cuyo especificador de tipos es la palabra reservada int, los caracteres, cuya palabra reservada es char, y las cadenas de caracteres, cuyo especificador de tipo es char *. Los enteros son cualquier número positivo o negativo sin decimal, de digamos "tamaño pequeño". Los caracteres son letras, números, signos de puntuación y algún carácter especial, como el fin de línea y la campanilla. Las cadenas de caracteres son un conjunto de caracteres, de cualquier longitud encerado entre comillas dobles".

Comencemos creando un programa con una variable numérica, numero1:

```
#include <stdio.h>

int numero1 = 1;

main()

{

printf("numero1 vale %d\n", numero1);

}
```

Como siempre, haremos el programa siguiendo la estructura que hemos seguido. Observar que al crear la variable le hemos colocado un = 1. Esto permite almacenar un 1 en nuestra variable. Luego dentro de la función main salo tenemos que llamar a printf() para imprimir nuestra variable. Aquí¡ ya empezamos a ver complicaciones en el uso de printf(). Printf coge nuestra cadena de caracteres numero1 vale %d\n. El %d indica que printf debe insertar en el mensaje el valor de la variable que a continuación le demos, en nuestro caso numero1. El carácter \n es un carácter especial, el de nueva línea, y le dice a printf() que debe avanzar el cursor al principio de la línea siguiente.

El valor de las variables es, como su propio nombre indica, variable. Podemos alterar su valor en cualquier punto del programa. La forma mas 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 m s importante. As¡:

```
#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);
```

```
}
```

El lado derecho de una asignación es una expresión. Una expresión es otro tipo de sentencia, quizá s el tercero m s importante. Una expresión es una sentencia que al ser evaluada devuelve un valor. Los tipos de expresiones m s frecuentes son las operaciones aritméticas: suma, resta, multiplicaciones y divisiones. Para escribir una expresión aritmética usaremos una notación similar a la usada en las calculadoras, incluyendo el uso de paréntesis. Por ejemplo:

```
#include<stdio.h>

int i;

main()

{

i = 2 + 2;

printf(" 2 + 2 = %d\n" , i);

}
```

Otros ejemplos de operaciones aritméticas son:

```
i = 2 * 2 + 6;

i = 2 * (2+ 6);

i = 1 − 4 / 2;
```

Observar que el * y el / tienen mayor "precedencia" que el + y el −, es decir al evaluar la expresión se evalúa antes un * y un − que un + o un −.

El tema de la evaluación de expresiones es mas complejo, y lo dejaremos para un capítulo mas avanzado.

Además contamos con otro tipo de expresiones, el resultado de llamar a una función. Lo veremos mejor con el tipo de variable carácter. Las variables de tipo carácter se emplean de modo análogo a las de tipo entero, salvo que ahora almacenan un carácter. En el siguiente ejemplo crearemos una variable y después de imprimirla la asignaremos el resultado de una expresión del tipo llamada a función. Utilizaremos la función getchar() que devuelve el siguiente carácter disponible desde el terminal.

```
#include <stdio.h>

char c = 'a' ;

main()

{

putchar(c);

putchar('\n');
```

```
printf("introduce un carácter por el terminal\n");

c = getchar();

printf("el primer carácter que has ");

printf("introducido es: %c\n", c);

}
```

Observar que el modificador para imprimir un carácter con printf() es %c.

Si sustituimos este modificador por el %d se imprimir el código ASCII del carácter introducido.

Aquí ya podemos observar una característica muy importante del C, que es la equivalencia de tipos. El tipo de datos entero y el tipo de datos carácter son intercambiables, es decir, si una expresión espera un entero y le damos un carácter y le damos un entero, el compilador promocionar el carácter a entero. Esto siempre es posible. El proceso inverso también se puede hacer. Si a una variable de tipo carácter y le damos un entero el compilador tratar de meterlo como pueda. Si el entero es pequeño esto se hará sin problemas. Si el entero es grande, el compilador truncar el entero, es decir, le cortar un trozo. El proceso de truncado se ve mejor usando representación binaria. Los caracteres se suelen representar mediante bytes, es decir 8 bits. Los enteros suelen ser dos o cuatro bytes. El compilador para pasar de un carácter a entero simplemente le añade un byte a cero. Para pasar de entero a carácter el compilador se olvida de todos los bytes salvo el de menor orden. Por eso la conversión entre tipos funciona la mayoría de las ocasiones.

Asi la función getchar() devuelve un entero, no un carácter. Esto se hace as¡ por otro motivo un poco m s oscuro, relacionado con el fin de fichero, que explicaremos al tratar el tema de los ficheros. No obstante, la función getchar() funciona sin complicaciones. As¡ mismo la función putchar() admite un carácter o un entero, pero siempre los imprime como carácter. Cadenas de caracteres y arreglos.

Las cadenas de caracteres nos permiten manejar texto con facilidad. Ya hemos visto que la función printf nos permite sacar una cadena por pantalla. Esto nos hace pensar que las cadenas de caracteres pueden servir para almacenar tiras de caracteres, como nombres o texto. Sin embargo poseen una pega: son constantes, por lo que no se pueden alterar.

Para almacenar grupos de caracteres utilizaremos los arreglos de caracteres.

Identificadores y palabras reservadas.

El lenguaje C est formado por un conjunto pequeño de palabras clave o comandos y una serie de operadores. Hay cerca de 40 palabras claves, frente a las 150 del BASIC o 200 que poseen otros lenguajes, como el COBOL y el PASCAL. Estas palabras son:

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

A este conjunto de palabras se les denomina "palabras reservadas".

Cuando en el C creemos nuestras propias subrutinas y conjuntos de datos les deberemos poner nombres. Para nombrar las funciones (subrutinas) y variables (datos) utilizaremos los identificadores. Un identificador es un conjunto de caracteres alfanuméricos (letras y números) que comienzan siempre por una letra. Para formar un identificador usaremos indiferentemente mayúsculas y minúsculas, números y también el carácter '_'

(subrayado), aunque este carácter tiene un significado especial, por lo que en principio debemos evitar su uso. Como identificador vale cualquier secuencia de letras y números, de cualquier longitud, pero con dos limitaciones:

−no podemos empezar con un número

−las palabras reservadas no se pueden usar como identificador, aunque un

identificador puede comenzar con una palabra reservada.

De aquí la denominación de palabras reservadas. Ejemplos de identificadores validos serían:

hola

hola127

printf

whiledo

Variables

Una variable es un lugar donde se puede almacenar temporalmente un dato. En C las variables tienen un nombre que las identifica, y sirve para hacer referencia a ellas. También tienen un tipo, que es el tipo de datos que puede almacenar. Por ejemplo podemos tener una variable de tipo entero de nombre num. Observar que para dar un nombre a una variable tenemos que usar un identificador. Para crear una variable en un lugar determinado del un programa escribiremos primero el tipo de variable y luego el identificador con el que queremos nombrar la variable, seguido todo de un ';'. Por ejemplo:

int numero; /* crea la variable numero de */

/* tipo entero */

char caracter1; /* crea una variable de tipo caracter1 */

las variables se pueden inicializar, es decir, darles un valor inicial, en el momento de creación. Para ello detrás del identificador ponemos el caracter1 '=' seguido del valor inicial. Los valores iniciales pueden ser cualquier constante v lida para el tipo de variable que creemos. Por ejemplo:

int numero = 0; /*crea la variable entera numero */

/* y la inicializa a 0*/

char c = 'a'; /* crea una variable caracter1 y */

/* la inicializa a 'a' */

Duración de las variables.

Las variables pueden ser de dos tipos: est ticas y din micas. Las est ticas se crean al principio del programa y duran mientras el programa se ejecute.

Las variables son din micas si son creadas dentro de una función. Su existencia est 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 est ticas se utilizan para almacenar valores que se van a necesitar a lo largo de todo el programa. las variables din micas se suelen utilizar para guardar resultados intermedios en los cálculos de las funciones.

Como regla general una variable es est tica cuando se crea fuera de una función y es din mica cuando se crea dentro de una función.

Por ejemplo en el siguiente programa :

```
#include <stdio.h>

int numero1 = 1;

main()

{

int numero2 = 2;

printf("%d, %d\n", numero1, numero2);

}
```

hemos creado la variable est tica numero1, que dura todo el programa, y la variable numero2, que dura sólo mientras se ejecuta la función main(). En este programa tan pequeño , la funcion main() es la que ocupa todo el tiempo de ejecución, por lo que no apreciaremos diferencia en el uso de ambas, aunque m s adelante si se ver su uso.

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 as¡ 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 creo. 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. En nuestro ejemplo anterior numero1 es una variable global y numero2 es una variable local.

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 m s pequeños, que son mas fáciles de manejar y depurar. Por ello hay veces que nos interesar que una variable sea accesible desde todos los módulos, y otras solo queremos que sea accesible por las funciones que componen un determinado modulo. Por defecto todas las variables globales que creemos son accesibles por todos los ficheros que componen nuestro programa.

Modificadores de tipo.

Podemos fácilmente modificar el alcance y la duración de una variable que tiene asignado por defecto: Esto es una operación muy común y útil. Para hacerlo antepondremos al tipo de la variable un modificador, que es

una palabra reservada, que cambiar estas característica.

El primer modificador es la palabra clave static. Cuando a una variable local se le añade el modificador static pasa de ser din mica a ser est tica. As¡ la duración de la variable se amplía a la duración del programa completo. Observar que una variable est tica solo se crea una vez, al principio del programa, por lo que la inicialización solo se produce una vez para una variable est tica.

Además el modificador static tiene otro uso. Si añadimos este modificador a una variable global, definida fuera de una función, entonces modificamos su alcance: pasa de tener alcance global a todos los ficheros del programa a ser solo accesible por las funciones del fichero en el que se crea.

Otro modificador usual es externa. Este modificador se usa cuando una variable que se creo en otro modulo se quiere usar en un modulo. Cuando añadimos a la variable este modificador el compilador queda advertido de que la variable ya existe en otro modulo, por lo que el compilador no tiene que crearla, sino simplemente usarla. Entonces a este tipo de proceso se le llama declaración de tipo de variable. Por ejemplo:

```
extern int numero;

main()

{

printf("%d\n", numero);

}
```

es un programa en que declaramos la variable externa numero, que habremos creado en otro modulo.

Una diferencia muy importante entre una definición y una declaración es que en la definición no se reserva espacio en la memoria para la variable, y en la definición si se crea.

Hay otros modificadores que no son tan usuales: auto, volatile y register. El modificador auto indica que una variable local es din mica (en la terminología del C automática). Observar que por defecto las variables locales a una función son automáticas, por lo que no se usa. Sin embargo todos los compiladores la reconocen y no protestan si la usamos. Por ejemplo:

```
main()

{

auto numero = 1;

printf("%d\n", numero);

}
```

crea una variable automática entera. Si quitamos auto el programa no se diferencia.

El modificador register se usa m s a menudo, sobre todo en la llamada "programación de sistemas". Recordemos que el C fue creado para este tipo de programación. Este modificador le pide al compilador que almacene la variable en un registro de la maquina, que es el lugar m s eficiente para guardar las variables. Esto se hace porque el trabajo con los registros del procesador es mucho m s r pido que el trabajo con la memoria

central. Hay dos detalles importantes: normalmente no hay muchos registros libres en el procesador, pues el compilador los usa para otros propósitos. Entonces el modificador register es mas bien un ruego que una orden. Otro aspecto es que muchos compiladores realizan trabajos de optimización, que son modificaciones en el código que generan que hace trabajar al programa m s deprisa. Aun as¡, en rutinas críticas, que deben ejecutarse deprisa se suele usar.

El modificador volatile se usa para decirle que el contenido de la variable puede ser modificado en cualquier momento desde el exterior de nuestro programa, sin que podamos evitarlo. Lo que hace el modificador es instruir al compilador para que lea de nuevo el valor de la variable de la memoria cuando tenga que usarlo. Este modificador evita que el compilador no genere código para optimizar la variable. Evidentemente el uso de volatile excluye el uso de register y viceversa.

Ambos modificadores, register y volátiles se emplean de manera análoga al modificador auto.

Tipos de datos en C

Un programa normalmente lo que hace es procesar un conjunto de datos para obtener alguna conclusión de ellos. Pos ejemplo un programa de estadísticas recoge una serie de datos y elabora gráficas que nos ayudan a tomar decisiones. En C los datos de una aplicación se representa haciendo uso de un conjunto de datos predefinidos en el lenguaje.

Distinguiremos para empezar tres tipos de datos fundamentales: caracteres, enteros y cadenas de caracteres (en ingles "strings"). El tipo de datos carácter consiste de un único carácter y se suele representar por su carácter en código ASCII situado entre apóstrofes. Por ejemplo:

'p' /* la letra p minúscula */

'1' /* el numero 1 */

' ' /* el carácter en blanco */

Hay otras formas de representar caracteres, que se emplean cuando es un carácter que no se puede introducir directamente desde el teclado. Para ello debemos conocer su código ASCII. Para representar el carácter de numero ascii 27, (el código para el carácter ESCAPE), basta colocar el numero ascii en el sistema octal precedido de la barra atrás, y todo ello entre apóstrofes, tal y como hacemos para los demás caracteres:

'\27**' /* representa el código ESCAPE, de ASCII 27 */

En C hay algunos caracteres especiales que se usan frecuentemente. Estos caracteres tiene una representación especial. Algunos de ellos son:

'\n' /* carácter nueva línea */

'\r' /* retorno de carro (carriage return) */

'\t' /* tabulador */

El siguiente tipo de datos es el entero. Consiste en un número sin parte decimal, aunque puede tener signo. Generalmente con el tipo de datos entero no representamos números muy grandes. son ejemplos:

0

124

−2000

El tipo de datos entero nos permitir hacer operaciones aritméticas, como la suma y la multiplicación. El tipo de datos entero es quizás el m s importante de todos los tipos de datos, y muchas veces es el tipo d datos por defecto, es decir, cuando el compilador se encuentre con un dato y no le hayamos dicho cual es su tipo supondrá que es un entero. Esto se ve claramente con las funciones, ya que siempre devuelven un valor, que por defecto es un entero.

El último tipo de datos importante en C es la cadena de caracteres. Est formada por un conjunto de caracteres encerrados entre comillas. Podemos usar todos los caracteres del conjunto ASCII, incluso los especiales. Los caracteres normales se incluyen entre las comillas tal cual, sin necesidad de apóstrofes, y los especiales se incluyen utilizando la representación del C. Por ejemplo:

"Hola, mundo\n"

En este ejemplo observamos la cadena de caracteres "Hola, mundo", a la que hemos añadido un carácter de retorno de carro al final. El motivo de ello es que m s adelante cuando la imprimamos el carácter de retorno de carro \n actuar como un comando que obligar al cursor a avanzar una línea y situarse al principio de la siguiente.

Los tipos de datos carácter, entero y cadena no son los únicos que posee el lenguaje C, pero si los mas importantes. Como avance podemos decir que también posee el tipo de datos en coma flotante, que es el que usan las calculadoras científicas, varios tipos da datos enteros, de diferente tamaño, el tipo de datos entero sin signo, un tipo de datos científico de mayor precisión, y otros tipos de datos que sirven para fabricarnos un tipo de datos a medida de nuestra necesidad: arreglos, estructuras y uniones. Pero esto es un tema m s avanzado.

Tipos de enteros.

El tipo de datos entero admite varias variantes, que le permiten representar cantidades m s grandes. Aunque el tamaño de los enteros depende de la implementacion con la que estemos trabajando, suele ser habitual que tenga un tamaño de 16 bits. Por ello el entero m s grande que se puede representar suele estar en torno al 32000. Cuando queremos alcanzar un número mas alto y no nos importa el signo le podemos pedir al compilador que el tipo de datos sea "sin signo". Esto se hace con la palabra reservada unsigned. As¡ el tipo de datos entero sin signo se comporta como un nuevo tipo de datos. Podemos definir variables del tipo unsigned int, al igual que lo hacíamos con enteros normales. Normalmente cuando nos referimos al tipo de datos sin signo nos estamos refiriendo al tipo entero sin signo.

Por ello cuando definamos un entero sin signo no hara falta escribir la palabra reservada int. Un ejemplo de definición con inicialización de un unsigned es:

unsigned u = 40000;

Podríamos también haber escrito unsigned int, pero no suele ser habitual. Si trabajamos con enteros de 16 bits, el número 40000 no cabria en un entero normal.

Otro tipo de datos útil es el entero largo, que análogamente al entero sin signo se representa con la palabra reservada long. Las palabras long y unsigned son modificadores de tipo, ya que actúan sobre uno de los tipos incorporados. Por ejemplo:

long largo = 100000;

define una variable de tipo entero largo. Hay m s modificadores de tipo. El modificador short hace referencia a un entero corto, de pequeño tamaño. Hay que tener en cuenta que el tamaño de cada tipo de datos depende del sistema.

En C estándar un entero corto tiene al menos 16 bits. Muchas veces el entero corto y el entero coinciden. En los sistemas basados en microprocesadores de 32 bits suele ser frecuente encontrar enteros de 32 bits de anchos, por lo que no coinciden el entero corto y el entero.

Además también podemos definir enteros cortos y largos sin signo. Por ejemplo:

unsigned short int u = 1;

unsigned long l = 1000000;

El signo de los caracteres.

Dependiendo del sistema el tipo de datos carácter puede tener signo o no.

Normalmente es una cuestión que no posee relevancia, pero por si necesitásemos que el tipo de datos carácter lleve o no signo, podemos aplicarle dos modificadores: unsigned para caracteres sin signo, y signed para caracteres con signo.

Tipos de datos en coma flotante.

Para representar números decimales el C posee dos tipos de datos: números en coma flotante con simple y doble precisión. El tipo de datos float corresponde al número de simple precisión. El tipo double representa a los de doble precisión.**********

Expresiones

Las expresiones son sentencias que tras realizar determinada una acción devuelven un resultado. En C la mayoría de las expresiones devuelven un entero. Hay tres tipos de expresiones:

−valores por la izquierda o lvalue (del ingles left value). Al evaluarlas devuelven la dirección de un cierto dato, que nos permitir acceder a l para inicializarlo, modificarlo, etc. Se llaman as¡ porque son las expresiones que normalmente se colocan en el lado izquierdo del = en una expresión de asignación. Suelen ser nombres de variables, elementos de un arreglo, etc.

−Valores por la derecha o rvalue (del ingles right value). Al evaluarlas obtenemos un dato de cierto tipo. Normalmente son las expresiones que se colocan a la derecha del = en la asignación. Ejemplos de ellos son los contenidos de las variables, los datos contenidos en los arreglos y los valores constantes.

−llamada a una función. Si la función no es de tipo void, al llamarla devolver un dato. Este dato puede ser usado como una expresión del tipo valor por la derecha.

Tipos de expresiones

El primer tipo a considerar dada su importancia es la asignación. La sintaxis de una asignación es

lvalue = rvalue;

Para evaluar una asignación el compilador evalúa el lado derecho. El dato que obtiene es entonces cargado en

la dirección que resulta de evaluar el lado izquierdo. Por ejemplo:

i = 1;

introduce el entero 1 en la dirección de la variable i. Como toda expresión la asignación devuelve un valor, que es el mismo que se carga en la variable. Esto puede ser usado para inicializar varias variables con el mismo valor. Por ejemplo, si tenemos tres variables enteras i, j y k:

i = j = k = 0;

las inicializa a las tres a 0. Observar que una misma variable puede estar en ambos lados de una asignación:

i = i + 1;

incrementa la variable i en una unidad.

El segundo tipo de expresiones son los operadores. Hay varios tipos de operadores:

−operadores aritméticos: Son el +, − , *, % y /, que realizan las operaciones aritméticas básicas. Estas expresiones tienen de sintaxis:

rvalue operador rvalue

Los operadores aritméticos son:

+ el operador de suma

− el operador de resta

* el la multiplicación

/ la división entera

% el resto de la división (operador modulo).

Por ejemplo:

i = a + 1;

j = b * 4 + 10 / 2;

−operadores de incremento y decremento. Sirven para incrementar una cierta variable. Admiten cuatro posibles combinaciones:

++lvalue incrementa el contenido de lvalue y devuelve el contenido nuevo

−−lvalue decrementa el contenido de lvalue y devuelve el contenido nuevo

lvalue++ incrementa el contenido de lvalue y devuelve el valor que contenía antes de incrementarlo.

lvalue−− decrementa el contenido de lvalue y devuelve el valor que contenía antes de decrementarlo.

Por ejemplo:

i = j++; /* carga en i el valor de j y */

/* luego incrementa j */

i = ++j; /* incrementa j y luego carga su valor en i */

Estos operadores son muy usados en las variables de los bucles y con los

punteros. El tipo de datos que devuelven es el del lvalue.

−operadores de comparación. Son:

< "menor que"

> "mayor que"

<= "menor o igual que"

>= "mayor o igual que"

== "igual que"

!= "no igual que"

La sintaxis para estas expresiones son:

rvalue operador rvalue

El valor que devuelve es de tipo entero: devuelve un 0 si el resultado de la comparación es falso y un valor distinto de 0 si el resultado de la comparación es verdadero. En C el cero se toma como valor falso, y cualquier valor diferente del cero es verdadero. El valor concreto empleado para representar el valor verdadero es irrelevante, y normalmente depende del sistema empleado. Cuando lo que comparamos son caracteres, se compara realmente su código ASCII. Por ejemplo:

1 > 2 devuelve un valor verdadero

1 == 1 devuelve un valor verdadero

'1' == 1 devuelve falso

'a' < 'b' es verdadero

Operadores lógicos

Realizan las operaciones lógicas habituales en el álgebra de Bool. Realizan el AND (Y lógico), el OR (O lógico) y el NOT (negación lógica). Son:

&& AND (Y lógico)

|| OR (O lógico)

! NOT (negación lógica

Los dos primeros son operadores binarios y el tercero es un operador unario. Su sintaxis es:

expresion1 && expresion2

expresion1 || expresion2

!expresion1

El resultado de evaluar la expresión AND es verdadero si ambos son verdaderos, y falso en caso contrario. El resultado de evaluar la expresión OR es verdadero si alguna o las dos expresiones es verdadera. Es falsa si las dos expresiones son falsas. El resultado de la expresión NOT es falso si la expresión es verdadera, y verdadero si la expresión es verdadera.

Para evaluar estos operadores se toma como verdadero un valor de la expresión distinto de 0 y como falso un valor 0.

Control del flujo del programa

En C las sentencias se ejecutan sucesivamente una tras otra. Esto define un camino o dirección según la cual se va desarrollado el programa. Sin embargo, habrá momentos en que el programa deba ejecutar determinadas partes dependiendo del estado en el que se halle el programa o de las variables externas. Esto permitir modificar el orden de la ejecución para adaptarse al estado del programa y bifurcar hacia nuevas subrutinas cuando se cumplan ciertas condiciones, que el programador fijar de antemano.

La sentencia if

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

if (expresi¢n1)

sentencia1;

o también:

if (expresi¢n1)

sentencia1;

else

sentencia2;

Esta sentencia es equivalente a la que poseen la mayoría de lenguajes de programación y sirve para bifurcar en un punto de programa. la sentencia if permite tomar decisiones al programa. En su primera forma la sentencia1 sólo se ejecuta si el resultado de evaluar la expresi¢n1 es verdadero (distinto de cero). En la segunda forma tenemos dos posibilidades: si al evaluar la expresi¢n1 el resultado es verdadero se ejecuta la sentencia1, pero si el resultado es falso se ejecuta la sentencia2. En cualquier caso sólo una de las dos sentencias se ejecuta. Por ejemplo:

if (numero1 == 1)

```
puts("la variable numero1 vale 1");

else

puts("la variable numero1 no vale 1");
```

Tras evaluarse la expresión if y ejecutarse la sentencia adecuada, el programa continua con la línea siguiente a la de la ultima sentencia del if. Para la sentencia if vale como expresión cualquier expresión v lida en C, incluso las asignaciones y llamadas a funciones. El caso en que la expresión es una asignación suele ser sorprendente, ya que en la mayoría de los lenguajes este tipo de expresiones no es valido. Como sentencia vale cualquier tipo de sentencia v lida en C, entre ellas la propia sentencia if. En este caso hablaremos de sentencias if anidadas. Por ejemplo:

```
if (num > 0)

if (num == 1)

puts("num es igual a 1")

else

puts("num es mayor que 1)

else

puts("num es menor que 1");
```

Cuando hay dos if anidados y a continuación hay un else, este else pertenece al ultimo if. As¡ 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");
```

Cuando necesitamos ejecutar varias sentencias que depende de un if, utilizaremos la sentencia de tipo bloque de sentencias. Un bloque de sentencias es un grupo de sentencias encerradas entre llaves { y }. Por ejemplo:

```
if (num >= 0) {

printf("num %d\n");
```

```
if (num == 0)

puts("num 0");

if (num >= 1)

puts("num mayor o igual a 1");

}
```

El bucle while

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. El bucle while es el tipo de bucle m s sencillo. En su modo m s simple se escribe:

```
while (expresi¢n1)

sentencia1;
```

El bucle while comienza por evaluar la expresi¢n1. Si es cierta se ejecuta la sentencia1. Entonces se vuelve a evaluar la expresi¢n1. De nuevo si es verdadera se vuelve a ejecutar la sentencia1. Este proceso continua hasta que el resultado de evaluar la expresión es falso. Por esto se le llama a esta expresión la condición de salida. Por ejemplo:

```
int variable = 10;

while (variable)

printf("la variable vale %d\n", variable−−);
```

En este caso se imprimir el valor de la variable hasta que se llegue a 1. Normalmente en las sentencias del bucle while se coloca alguna instrucción que modifique la expresión de control. Lo m s habitual es utilizar un bloque de sentencias en vez de una sentencia en vez de una sentencia única. Por ejemplo:

```
int variable = 10;

while (variable) {

printf("valor de la variable %d\n", variable);

printf("valor tras decrementar la variable %d\n", variable);

}
```

El bucle do−while

La sintaxis de este bucle es:

```
do
```

sentencia1;

while (expresi¢n1);

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. Es menos habitual que el bucle while.

Podemos incluir dentro del bucle un grupo de sentencias, en vez de la sentencia1. Este es el único bucle que no necesita llaves para encerrar un grupo de sentencias. Por ejemplo:

char c = '9';

do

printf("numero actual %c\n", c);

−−c; /* ahora la decrementamos como si fuera entera */

while (c >= '0');

El bucle for

La sintaxis del bucle for es:

for (inicio, control, incremento)

sentencia1;

Este bucle se utiliza para realizar una acción un número determinado de veces. Est compuesto de tres expresiones: la de inicio, la de control y la de incremento, y una sentencia. Su versión m s sencilla es:

for (i =0; i < 10; i++)

printf("i vale %d\n", i);

Esta versión del bucle imprime un mensaje en la pantalla mientras que no se alcance la condición de salida, i == 10.

El funcionamiento del bucle for es el siguiente:

−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.

Habitualmente lo que hace esta expresión es incrementar la variable de control.

−a continuación se vuelve al segundo paso. El bucle finaliza cuando la expresión de control es falsa.

En un bucle for podemos omitir la expresión de inicio, por ejemplo si sabemos que la variable ya esta inicializada:

```
int i = 0;

for ( ; i <= 10; ++i)

printf("%d\n", i);
```

También podemos omitir la expresión de incremento. Esto es habitual cuando la variable de control ya se modifica dentro del bucle. Por ejemplo:

```
int i= 10;

for ( ; i < 10; )

printf("i = %d\n", i++);
```

El bucle for es equivalente a un bucle while escrito del siguiente modo:

```
inicio;

while (control) {

sentencia1;

incremento;

}
```

Este bucle while puede servirnos para salir fácilmente de dudas al escribir un bucle for, ya que se ve claramente el orden de ejecución de las expresiones y sentencias dentro del bucle for. Como se ve, en cada pasada del bucle for se sigue el orden: evaluación de control, ejecución de sentencia1 y evaluación de incremento.

Las sentencias break y continue.

Hay veces en que interesa romper un bucle en una determinada posición, para ejecutar una nueva pasada del bucle o para finalizar su ejecución Esto suele ser habitual cuando el bucle tiene una gran complicación o cuando necesitamos salir "por las malas" de l. Esto ultimo suele ser frecuente cuando en el bucle se producen "condiciones de error". 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 continua en la instrucción que siga al bucle o bloque. Por ejemplo:

```
int a = 10;

while (1) {

if (a−− <= 1)
```

```
break;

printf("%d\n", a);

}
```

Aunque en apariencia este es un bucle sin fin, ya que la condición con while (1) siempre es cierta, este bucle se acabar cuando la variable a valga 1. El bucle simplemente decrementa la variable e imprime su valor.

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) {

printf("%d\n", a);

if (a==7)

continue;

}
```

La sentencia de selección múltiple switch

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, y que realiza acciones si coincide con ellos. Su

sintaxis es:

```
switch (control) {

case exp1: sent1; break;

case exp2: sent2; break;

default sent0; break;

}
```

Su sintaxis es m s complicada que la de anteriores bucles, ya que agrupa un mayor número de acciones y posibilidades en una sola sentencia. El modo de funcionamiento es el siguiente:

−primero se evalúa la expresión de control.

−A continuación 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. (* revisar si esto es cierto o la etiqueta default se ejecuta aun cuando se haya ejecutado otra *)Esta es la acción por defecto. La etiqueta default es opcional. Si no la ponemos el programa simplemente salta a la línea siguiente.

Hay que tener cuidado con un aspecto de este bucle. Cuando una expresión de una etiqueta case es igual a la sentencia de control, se ejecutan todas las sentencias que sigan hasta que se alcance una nueva etiqueta case, y luego se vuelve a comparar la expresión de control. Este mecanismo tiene una ventaja y un inconveniente. La ventaja es que no necesitamos encerrar entre llaves el grupo de sentencias a ejecutar para cada etiqueta. El inconveniente es que al agotar las sentencias de una determinada etiqueta la sentencia switch prosigue con la siguiente etiqueta case. Habitualmente lo que se pretende es que tras ejecutar el grupo de sentencias se finalice el switch. Para evitar el que se ejecuten m s sentencias habitualmente se acaba cada grupo de sentencias con una sentencia break. La sentencia break pasa entonces la ejecución a la siguiente línea de programa que prosiga al bucle switch.

También se permite poner etiquetas múltiples para un mismo grupo de sentencias. Si dejamos una etiqueta case sin sentencias a ejecutar entonces se asocia a la siguiente etiqueta. Esto es útil para ejecutar una misma acción para distintos valores de la expresión.

Funciones

Una función es una rutina o conjunto de sentencias que realiza una determinada labor. En C todas las funciones devuelven un valor, que por defecto es un entero. las funciones admiten argumentos, que son datos que le pasan a la función las sentencias que la llaman.

definición de una función.

la sintaxis habitual en la definición de una función es:

tipo identificador(lista_de_argumentos)

{

/* bloque de código */

}

Donde:

−tipo es el tipo de datos devuelto por la función

−identificador es el nombre de la función. Debe ser un identificador valido.

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

El tipo y la lista de argumentos son opcionales. Si omitimos el tipo, la función por defecto devolver un entero. Muchas veces el valor devuelto por la función es ignorado en el programa.

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

```
hola()

{

printf("hola\n");

}
```

que simplemente es una función que cuando es llamada imprime en pantalla un mensaje de saludo.

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.

retorno de valores

Cuando la función finaliza hemos dicho que se devuelve un valor. Este valor en principio no est definido, es decir, puede devolver cualquier cosa.

Para obligar a la función a retornar un determinado valor se utiliza la sentencia return, seguida del valor a retornar. Como todas las sentencias en C se debe acabar con un ';'. Por ejemplo:

```
lista()

{

return 1;

}
```

devuelve el entero 1 cada vez que es llamada. En C podemos devolver cualquier tipo de datos de los llamados escalares. Los tipos de datos escalares son los punteros, tipos numéricos y el tipo carácter. En C no se pueden devolver arreglos ni estructuras.

Paso de par metros a una función

Utilizando la lista de argumentos podemos pasar par metros a una función.

En la lista de par metros se suele colocar un conjunto de identificadores, separados por comas, que representar cada uno de ellos a uno de los par metros de la función. Observar que el orden de los par metros es importante. Para llamar a la función habrá que colocar los par metros en el orden en que la función los espera.

Cada par metro puede tener un tipo diferente. Para declarar el tipo de los par metros añadiremos entre el paréntesis ')' y la llave '{' una lista de declaraciones, similar a una lista de declaraciones de variables. Es habitual colocar cada par metro en una línea, tabulados hacia la derecha.

Asi:

```
imprime(numero, letra)

int numero;
```

```
char letra;

{

printf("%d, %c\n", numero, letra);

}
```

es una función que admite dos variables, una entera u otra de tipo carácter.

paso de par metros 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 función puede acceder a la variable original. Este enfoque es habitual en lenguajes como el Pascal.

En C sin embargo todos los par metros se pasan por valor. La función recibe una copia de los par metros y variables, y no puede acceder a las variables originales. Cualquier modificación que efectuemos sobre un par metro no se reflejar en la variable original. Esto hace que no podamos alterar el valor de la variable por equivocación.

Sin embargo, 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, que se ve mas adelante.

declaración y comprobación de tipos

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 s 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. Esto es obligatorio cuando vamos a usar una función que no devuelve un entero. Además en la declaración se puede especificar el número de argumentos y su tipo. Una declaración típica de función es:

```
tipo identificador( lista_de_argumentos_con_tipo );
```

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

La lista de argumentos con tipo difiere de la lista de argumentos antes presentada en que el tipo de cada argumento se coloca dentro de la lista, antes de su correspondiente identificador, como hacíamos en la definición de variables. Por ejemplo:

```
char print(int numero, int letra);
```

declara una función que devuelve un carácter y tiene dos par metros, un entero y un carácter.

La lista de argumentos permite al compilador hacer comprobación de tipos, ya que el tipo y numero de argumentos debe coincidir en la declaración, definición y llamada a una función.

Este tipo de especificación del tipo de argumentos también se puede emplear en la definición de las funciones, aunque lo contrario no es posible. Asi:

```
char print(int numero, int letra)
```

{

printf("%d, %c\c", numero, letra);

}

es otra definición v lida para la función print que hemos empleado.

Arreglo

Los arreglos son conjuntos de datos de un mismo tipo, el tipo base. A cada uno de los datos de un arreglo le llamaremos elemento de arreglo, y esta designado por un número. En C al primer elemento de un arreglo le corresponde siempre el número 0, y los demás tienen la numeración consecutiva.

Declaración de una matriz.

Para crear un arreglo de n elementos de un cierto tipo se introduce la línea:

Tipo identificador [n];

donde n es una constante de tamaño fijo. Si el arreglo es estático o global el compilador crea el espacio para la matriz al principio del programa, generalmente en el rea de datos. Si es de tipo automático, reservar el espacio en la pila de datos.

Como todos los tipos de datos , un arreglo se puede inicializar. Si el arreglo es estático, por defecto cada elemento se inicializa a 0. Si es din mico los valores de cada elemento no est n definidos y antes de usarlos los debemos inicializar. Para inicializar un arreglo en el momento de su creación añadiremos tras el identificador y los corchetes de tamaño un = y la serie de valores. Cada valor debe ser una constante v lida para el tipo de datos del arreglo, 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 vector [4]={0, 1,2,3 };

char hola[] = { 'h', 'o', 'l', 'a', '\0'};

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

Acceso a los miembros de un arreglo

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

printf ("%d", vector[0])

imprimiría el contenido del primer elemento del arreglo 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 ultimo número del arreglo. Esto es lo que llama sobrepasar el límite, y el compilador deja al programador la tarea de preocuparse por los límites del arreglo. Si los sobrepasamos, pueden ocurrir resultados imprevisibles

(normalmente modificaremos alguna otra variable del programa, o incluso bloquearemos el programa).

Tamaño de los arreglos

El tamaño de los arreglos es siempre constante y se especifica al crear el arreglo. Hay dos formas de especificar el tipo: índice dándoselo explícitamente al compilador o haciéndolo implícitamente. El primer modo es el ya señalado anteriormente . Para dar un tamaño al arreglo simplemente indicamos el número de elementos entre los corchetes. Este es el modo m s habitual de dar el tamaño, sobre todo si no se va a inicializar en el momento de su creación. El otro modo consiste en hacer que sea el compilador el que decida el tamaño. Esto se hace cuando en la creación del arreglo le damos una lista de valores iniciales . En este caso si omitimos el tamaño del arreglo el compilador ajusta el tamaño del arreglo según el número de elementos que le demos para inicializar el arreglo. Por ejemplo:

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

Este ultimo m todo es muy cómodo, sobre todo si el arreglo va a tener un tamaño pequeño y todos los valores iniciales son constantes.

Cadenas de caracteres

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

Una cadena de caracteres es un arreglo de caracteres que acaba con el carácter nulo. En C siempre las cadenas de caracteres acaban con este carácter. Esto se hace as¡ 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. El inconveniente es que para conocer el tamaño de la cadena normalmente necesitamos recorrerla con un bucle, aunque esto suele hacerse rápidamente.

Para inicializar una cadena de caracteres basta crear un arreglo 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 arreglo es una unidad mayor del aparente. Por ejemplo:

char cadena[] = "abracadabra" /* cadena de 12 caracteres*/

Los caracteres especiales como el tabualador \t y el retorno de carro \r se almacenan como un único carácter. El carácter nulo est representado por un 0. Esto nos permitir utilizar comparaciones con este carácter en los bucles que recorren cadenas de caracteres .

Arreglos multidimensionales.

En C se pueden construir arreglos de arreglos , es decir tipos de arreglos tipos de arreglos cuyos elementos son a su vez arreglos. Dado que ahora necesitaremos un ¡índice para situarnos dentro del arreglo principal y otro m s para movernos dentro de cada uno de los nuevos arreglos , diremos que los arreglos de arreglos poseen dos dimensiones. A un arreglo de dos dimensiones se le suele llamar matriz, y a un arreglo de una dimensión, vector. Las matrices son tipos de datos ampliamente usados en matemáticas. Normalmente diremos que un ¡índice representa a las filas de la matriz y otro a las columnas. Otro uso habitual de las matrices es para representar tablas de valores.

Para crear una matriz de enteros, es decir, un arreglo de arreglos de enteros, lo haremos de modo análogo a cuando creábamos un arreglo, salvo que ahora añadiremos 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 cual ¡índice representa las filas y cual las columnas es arbitrario. Podemos usar la norma habitual en matemáticas: el de la izquierda representa filas y el de la derecha columnas.

Acceso a los miembros de una matriz.

Para acceder a un miembro concreto de una matriz, siguiendo el convenio anterior, colocaremos su número de fila y de columna entre corchetes. Por ejemplo:

printf("%d\n", matriz[1][2]);

imprime el elemento correspondiente a la fila 1, columna 2.

Para inicializar las matrices disponemos ahora de dos formas:

−incluir todos los elementos de la matriz entre llaves, ordenados por filas y por columnas. Para ello se hace del siguiente modo: tras el signo igual en la definición abrimos una llave. A continuación colocamos cada columna de valores, todos ellos separados por sus comas. Cada columnas de valores debe ir encerrada entre llaves, separando una columna de otra por comas. Al final cerramos la llave que habíamos abierto al principio. Por ejemplo:

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

define una matriz de 2 filas con 3 columnas por fila.

−incluir la lista completa de elementos de la matriz entre llaves, separados por comas, uno tras otro sin separar las columnas. Para colocar todos los valores correctamente necesitamos saber como accede el compilador a los elementos de una matriz. El compilador supone que en la memoria est n guardados los elementos ordenados por filas, es decir, est n juntos todos los datos correspondientes a una fila. As¡ la matriz la podíamos haber inicializado con:

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

y obtendremos el mismo resultado de antes. Vemos que los tres primeros elementos corresponden a la fila 1, y los tres siguientes son los de la fila 2. Cuando el compilador necesita acceder al elemento de fila i y columna j hace el siguiente calculo: multiplica (i − 1) por el número de columnas, y al resultado le suma (j − 1). Con ello accede a la matriz como si fuese de una sola dimensión. En nuestro caso el elemento matriz[1][3] ser el que ocupe el lugar 0 * 3 + 2, es decir, el tercer elemento de un arreglo unidimensional. Los dos unos que aparecen restando se deben a que el compilador empieza a contar los elementos de la matriz desde el 0.

Como cabe esperar se pueden formar arreglos de cualquier dimensión. Por ejemplo un arreglo de tres dimensiones podría ser:

tresdim[10][15][20];

Esto podría representar un conjunto de 10 tablas, cada una de 15 filas y 20 columnas por fila. Para inicializarla procederíamos de un modo análogo al caso bidimensional.

Punteros

Cuando queramos pasar un dato a una función normalmente pasamos una copia del dato. Esto es sencillo de hacer y r pido, siempre que no queramos modificar mediante la función el dato original o que el dato sea pequeño.

Otro enfoque consiste en decirle a la función donde encontrar los datos. Para ello le pasamos a la función una dirección. Con ella la función podré acceder a los datos utilizando un puntero. Un puntero es un nuevo tipo de datos, que no contiene un dato en si, si no 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.

Definición de un puntero.

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

Para definir un puntero lo primero que hay que tener en cuenta es que todo puntero tiene asociado un tipo de datos. Un puntero se define igual que una variable normal, salvo que delante del identificador colocaremos un asterisco. Por ejemplo:

char *pc; /*puntero a carácter */

char *pi; /* 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.

Para obtener la dirección en la que est ubicada una variable colocamos delante del identificador de la variable el operador dirección &. No se suele dejar espacios entre el signo & y el identificador. Por ejemplo:

char *p = &p1;

−asignarle el contenido de otro puntero que ya est inicializado:

char *p = &p1;

char *p2 = p; /* p ya est inicializado */

−inicializarlo con cualquier expresión constante que devuelva un lvalue.

Las mas frecuentes son una cadena de caracteres, el identificador de un arreglo, el identificador de una función, y otros muchos. Este es un detalle importante: los identificadores de funciones y de arreglos son en si mismos valores por la izquierda (lvalues), por lo que se pueden usar directamente para inicializar punteros.

Una forma adicional de inicializarlo es darle directamente una posición de memoria. Este m todo no es portable, ya que depende del sistema, pero suele ser muy útil en programación de sistemas, que es uno de los usos fundamentales del C.

Un error muy frecuente consiste en no inicializar el puntero antes de usarlo. Este error frecuentemente lo

localiza el compilador y avisa de ello.

Desreferenciacion de un puntero.

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 entero = 4, *p = &entero;

printf("%d %d \n", *p, entero);

Aritmética de punteros. Un uso habitual de los punteros es para recorrer los arreglos. En efecto, comencemos por crear un arreglo y un puntero al comienzo del arreglo.

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

int *p = arreglo;

En este momento el puntero apunta al primer miembro del arreglo. Podemos modificar fácilmente el primer miembro, por ejemplo:

*p = 5;

printf("%d\n", arreglo[0];

Ya que un puntero es una variable también, le podemos sumar una cantidad.

Sin embargo el resultado no se parece al que obtenemos con variables. Si a un puntero de tipo carácter le sumamos 1 o lo incrementamos, el contenido de la variable puntero es aumentado una unidad, con lo que el puntero a caracteres apuntaría al siguiente miembro del arreglo. En principio este es el efecto que necesitaremos.

Supongamos que tenemos ahora nuestro puntero a enteros, y apunta al principio del arreglo. Si nuestro sistema necesita dos bytes para representar los enteros, tras incrementar un puntero en una unidad veremos que el contenido de la variable puntero ha aumentado en dos unidades. Esto lo podemos ver utilizando la función printf con el modificador %p. En efecto:

printf("%p\n", p); /* usamos el puntero anterior */

++p;

printf("%p\n", p;

El resultado es que el puntero apunta ahora al siguiente elemento del arreglo. Este modo de incrementar el puntero es muy conveniente pues nos permite recorre un arreglo fácilmente. Para recorrer el arreglo sólo tendremos que crear un puntero apuntando al principio del arreglo e irlo incrementando mientras manipulamos los datos del arreglo. Por ejemplo, el siguiente bucle imprime todos los caracteres de una cadena:

```
char *p = "Hola, mundo.\n"; /* la cadena es un lvalue */

while (*p)

putchar(*p++);
```

Aquí observamos como se usa el operador incremento con los punteros. Ya que el operador puntero tiene mayor precedencia que el operador incremento, en la expresión *p++ primero se desreferencia el puntero, usándose en con la función putchar(), y luego se incrementa el puntero. Por eso no hacen falta los paréntesis con este operador. Si quisiésemos incrementar el carácter al que apunta el puntero, debemos encerrar entre paréntesis al operador puntero y al puntero. Por ejemplo:

```
char *p = "Hola, mundo.\n"; /* la cadena es un lvalue */

++(*p); /* Aparecerá una I */

while (*p)

putchar(*p++);
```

Cadenas de caracteres

La manipulación de cadenas de caracteres est implementado en la librería

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

```
char cadena[100] = "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 mas importantes de esta librería son:

```
size_t strlen( const char *s);
```

La función strlen() devuelve el tamaño de una cadena de caracteres, sin incluir el carácter nulo de terminación. Por ejemplo:

```
printf("numero de caracteres de la palabra hola = %d", strlen("hola");
```

Necesita un par metro de tipo puntero a carácter y devuelve un size_t, que suele est r definido como entero sin signo.

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

La función strcpy() copia la cadena s2 en la cadena s1, incluyendo el carácter de terminación y devuelve un puntero a s1. Los dos parametros que necesita son punteros a caracteres, y devuelve un puntero a caracteres.

Deberemos asegurarnos de que s1 tiene sitio para almacenar la cadena s2.

```
char *strcat(char *s1, const char *s2);
```

La función strcat() copia la cadena s2 al final de la cadena s1. Para ello busca el carácter de terminación de s1 y a partir de all¡ va colocando sucesivamente los caracteres de s2, incluyendo el carácter de terminación.

Los par metros que necesita y que devuelve son del mismo tipo que los de strcpy. Tampoco hace comprobaciones de que exista espacio para la cadena s2, ya que de ello debe asegurarse el programador.

char *strchr(const char *s, int c);

La función strchr() busca el carácter c a lo largo de la cadena s. 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.

Devuelve un puntero a caracteres.

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

La función strcmp() compara dos cadenas de caracteres. Para ello compara elementos sucesivos de ambas cadenas hasta que encuentra dos elementos diferentes. Si ambas cadenas son iguales la función devuelve un 0. Si el elemento diferente es menor en s1 entonces devuelve un número negativo, y si es mayor en s1 entonces devuelve un número positivo. para comparar los caracteres la función toma los caracteres como enteros sin signo. Al utilizar el código ASCII para representar las cadenas, los números tienen un código menor que las minúsculas, y stas que las mayúsculas. Esta función no es muy eficiente cuando se trata de ordenar cadenas en las que aparecen caracteres acentuados o eñes, pero es fácil construir una a medida.

char *strncat(char*s1, const char *s2, size_t n);

La función strncat() sirve para copiar un fragmento de la cadena s2 en al final de la cadena s1. Necesita tres par metros, dos punteros a caracteres y un número del tipo size_t, generalmente un unsigned. La función copia los primeros n caracteres de s2 al final de s1 y luego añade un carácter nulo al final de s1. La función devuelve un puntero a carácter que apunta a s1.

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. Las estructuras tiene su equivalente en otros lenguajes de programación, como el Pascal, en los registros. También se llaman registros a los grupos de datos en la terminología de las bases de datos.

Definición de una estructura

Para la definición de estructuras el C dispone de la palabra reservada estruct. 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, con algunas modificaciones. Primero colocamos la palabra reservada struct y luego el identificador que dar nombre al nuevo tipo de estructura. Luego abriremos llaves y comenzaremos a definir los campos de la estructura. Cada campo se define como una variable normal, es decir, dando su tipo y un identificador. Vale cualquier definición de tipo habitual, incluso punteros y estructuras. El identificador servir para designar a cada campo. Cada definición de campo acabar con un punto y coma, como siempre. Finalizaremos la definición de la estructura con una llave de cierre y un punto y coma. Por ejemplo:

struct fecha { /* para almacenar una fecha */

```
int dia;

char mes[14];

int anyo;

};

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 fecha fecha_de_hoy;

struct complejo x, y;
```

Una estructura también se puede inicializar. Para ello se dan los valores iniciales entre llaves, separados por comas, al igual que hacíamos son los arreglos. La novedad es que ahora cada dato puede tener un tipo diferente.

Por ejemplo:

```
struct fecha fecha_actual = { 12, "Enero", 1900};

struct complejo x = {1, 1};
```

Hay dos posibilidades m s en la definición de estructuras. La primera es definir una variable estructura a la vez que se define el tipo de estructura. Para ello hasta dar los identificadores de las nuevas variables estructuras después de la llave de cierre. Por ejemplo:

```
struct complejo{

double x ;

double y;

} z1, z2;
```

Además podemos definir variables estructuras sin tipo específico. Para ello basta omitir el identificador del tipo de estructura en la definición de la estructura, dando sólo el identificador de la variable estructura. De este modo la nueva variable va asociada al tipo creado. Por ejemplo:

```
struct {

int dia;
```

```
char mes[14];

int anyo;

} mi_aniversario;
```

Campos de bits.

Hay un nuevo tipo de datos que solo se puede usar con estructuras: el campo de bits. Un campo de bits se comporta igual que un entero sin signo, sólo que al definir el campo de bits se define el número de bits que lo compondrá . Por Ejemplo:

```
struct comida {

unsigned clase : 2; /* dos bites para el tipo */

unsigned temporada : 1; /* a 1 si es de temporada */

unsigned es_perecedero :1, es_congelado : 1;

};
```

Si el tamaño del campo de bits es 0 nos permite alinear el siguiente campo sobre un entero. Hay que tener en cuenta que la alineación de los campos de bits y de los demás campos la define el compilador.

Hay que tener en cuenta que no se puede obtener la dirección de un campo de bits. El C estándar define la macro offsetof() para calcular el desplazamiento de un campo de una estructura desde el principio de la misma.

El uso de campos de bits permite empaquetar información pequeña eficientemente dentro de una estructura. Su uso est bastante extendido en la programación de sistemas.

Paso de estructuras a las funciones.

En C estándar esta permitido pasar una estructura como argumento de una función, devolverla como resultado de una función, asi como asignar una estructura completa a una función. Esto no era posible en los primeros compiladores de C. No est permitido comparar estructuras completas. Por ejemplo:

```
struct complejo {

double x, y;

} z1, z2, suma;

struct complejo suma_complejos( struct complejo j1, struct complejo j2);

suma = suma_complejos(j1, j2); /*esta función se definiría aparte */

if (j1 > j2)

puts("es mayor j1"); /*esto no est  permitido */
```

Sin embargo esto no es lo m s eficiente. Como las estructuras tienen a menudo un tamaño considerable suele ser conveniente pasarlas a través de punteros. Para obtener la dirección donde se halla una estructura usaremos el operador dirección &, como con una variable normal. Para definir un puntero a una estructura usaremos el operador *. Por ejemplo:

```
struct complejo j;

struct complejo *pj = &j;
```

Acceso a los campos de una estructura.

Para acceder individualmente a cada campo de una estructura se usa el operador punto '.'. Para ello colocamos el operador de la estructura , un punto y el identificador del campo. Cada campo de una estructura designado mediante este operador se comporta como si de una variable del mismo tipo que el campo se tratase. Podemos realizar todas las operaciones habituales de las variables: asignación, uso de punteros, llamadas a funciones con el campo como par metro:

```
struct complejo z = {1,1};

printf("z vale %f, i%f\n", z.x, z.y);

z.x = z.y = 0;
```

Para acceder a los campos de una estructura a través de un puntero tenemos un nuevo operador, el operador puntero a campo −> (un guión seguido de un signo "mayor que"). Para acceder a un campo de una estructura a través de un puntero a ella basta poner el identificador del puntero, el nuevo operador puntero a campo y luego el identificador del campo. Por ejemplo:

```
struct complejo z = {1, 1};

struct complejo *pz = &z;

printf("%f, i%f\n", pz−>x, pz−>y);
```

Podríamos haber usado la notación habitual para punteros y el operador punto de acceso a campos, pero nos aparece un nuevo inconveniente: ya que el operador punto tiene mayor precedencia que el operador puntero tendríamos que encerrar el identificador y el operador puntero entre paréntesis, para luego aplicarles el operador punto. En efecto, las siguientes expresiones son equivalentes:

```
pz−>x = 1;

(*pz).x = 1;
```

Si omitimos los paréntesis se haría lo siguiente: pz sería considerado como una estructura, luego el operador punto daría el campo x de la estructura y el operador puntero intentaría acceder a donde apuntase el campo x. Vemos que hay dos errores, ya que ni pz es una estructura ni el campo x es un campo puntero. Esta expresión sería v lida si tuviésemos una estructura del tipo:

```
struct elemento {
```

```
int tipo;

char *nombre;

} uno = {1, "estructura"};

printf("%c\n",*uno.nombre); /*imprime la primera letra de nombre */
```

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 adoptar 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 sólo se puede tener almacenado un dato a la vez en la variable. En C es responsabilidad del programador el conocer que tipo de dato se est guardando en cada momento en la unión.

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

Por ejemplo:

```
unión dato_num {

int num1;

float num2;

} dato;
```

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

Las uniones normalmente se emplean como campos en las estructuras. Para llevar la cuenta del tipo de datos almacenado en la unión normalmente se reserva un campo en la estructura. Por ejemplo:

```
struct dato_num {

int tipo;

unión {

float simple;

double doble;

}dato;

};
```

Las uniones son especialmente útiles para la realización de registros de bases de datos, ya que permiten

almacenar información de diferentes tipos dentro de los registros. En programación de sistemas es usual encontrarlas dentro de las estructuras de datos de las rutinas, ya que permiten una gran flexibilidad a la hora de almacenar información.

Tipos de datos enumerados.

Gestión de la memoria

En C se pueden almacenar variables y estructuras de datos en tres lugares diferentes: la pila para las variables automáticas, la memoria global para las variables globales y la memoria din mica. La pila es una sección de la memoria que es gestionada automáticamente por el compilador y es donde se almacenan las variables locales. El compilador crea el espacio para la variable en tiempo de ejecución y libera el espacio ocupado cuando la variable deja de usarse (cuando salimos del ámbito o función en que se declaró). Por eso reciben el calificativo de automáticas.

Las variables est ticas globales se almacenan en la sección de la memoria llamada memoria global. El compilador también se encarga de proporcionarles espacio a estas variables, pero lo hace en tiempo de compilación, por lo que el espacio queda reservado durante toda la ejecución del programa.

Generalmente los valores iniciales de las variables globales son asignados en tiempo de compilación.

El ultimo tipo de memoria es el almacenamiento libre (free store en ingles) conocido habitualmente como la memoria din mica (heap en ingles). El compilador no se hace cargo de ella, sino que es el programador el que solicita su uso a través de funciones predefinidas que se encargan de manejar la memoria din mica. Estas funciones son malloc, calloc, realloc y free, y se definen en el fichero de cabecera <stdlib.h>. Estas funciones generalmente trabajan solicitando directamente porciones de memoria al sistema operativo. Su uso es muy sencillo y generalmente se usan a través de punteros.

La función malloc() nos permite solicitar memoria al sistema. Posee un único argumento: el tamaño del espacio que queremos reservar. En C el tamaño de un tipo de datos es el número de caracteres que ocupa en la memoria, ya que el tipo carácter tiene un tamaño de un byte generalmente.

El tamaño de un tipo de datos también se puede calcular con el operador sizeof. La función malloc se define como:

void * malloc(size_t longitud);

y su nombre viene de memory alloc (asignación de memoria, en ingles).

Necesita un par metro, que es la longitud del espacio que vamos a reserva. Este par metro es del tipo size_t, que es el tipo empleado en c para medir tamaños de tipos. Normalmente se suele definir size_t como unsigned, y el valor longitud representa el número de caracteres que se asignan. La función malloc devuelve un puntero del tipo puntero a caracteres al espacio de memoria asignado. Si el sistema no puede proporcionar la memoria pedida devuelve un puntero nulo. El espacio que devuelven las funciones malloc, c alloc y realloc no est inicializado, por lo que lo que debe inicializarlo el programador.

Para liberar el espacio asignado con malloc basta llamar a la función free, (liberar en ingles). Esta función se define como:

void free(void *ptr);

y su argumento es el puntero que devolvió malloc. No devuelve ningún valor y funciona igualmente con los

punteros que devuelven calloc y realloc.

La función calloc funciona de modo análogo a malloc salvo que tiene un par metro adicional, numelem, que le permite especificar el número de objetos a asignar. Se define como:

void *calloc(size_t numelem, size_t longitud);

Como la función malloc devuelve un puntero al espacio de memoria asignado y un puntero null si no ha sido posible asignar el espacio. Normalmente se utiliza para asignar espacio para un grupo de datos del mismo tipo.

La función realloc nos permite modificar el tamaño del espacio asignado con malloc o calloc. Se define como:

void *realloc(void *ptr, size_t longitud);

El primer argumento ptr es un puntero a un espacio previamente asignado con calloc o malloc. El segundo es la nueva longitud que le queremos dar.

Devuelve un puntero al nuevo espacio asignado o un puntero nulo si falló la asignación. Además la función copia el contenido del antiguo espacio en el nuevo al comienzo de este. Esto siempre lo puede hacer se la longitud del anterior espacio asignado es menor que la nueva longitud solicitada. El espacio sobrante en este caso no se inicializa. Si el espacio solicitado es de menor tamaño que el anterior se copia sólo la parte que quepa, siempre desde el principio, al comienzo del nuevo espacio asignado.

Uso de las funciones de asignación de memoria.

Para reservar espacio en la memoria din mica para un arreglo comenzaremos asignando el espacio con malloc. Por ejemplo:

```
#include <stdlib.h>

main()

{

char *p;

int i;

p = (char *) malloc(1000);

for (i = 0; i < 1000; ++i)

p[i] = getchar();

for (i = 999; i >= 0; −−i)

putchar(p[i]);

free(p);

}
```

Este programa lee los primeros 1000 caracteres de la entrada estándar y los imprime en orden inverso. Se puede ver el uso de malloc y de free. En la línea de llamada a malloc hemos introducido una variante: hemos forzado una conversión del tipo devuelto por malloc. Esto lo hemos hecho porque conviene acostumbrarse a asignar a los punteros otros punteros del mismo tipo. Aunque en C estándar se puede asignar un puntero void a cualquier puntero, es conveniente realizar el moldeado del tipo, ya que en C++, el lenguaje C avanzado, esto no est permitido. Muchos compiladores nos avisar n si intentamos asignar un puntero void a otro puntero, aunque nos permiten compilar el programa sin problemas. Los compiladores de C++ directamente paran la compilación. Ya que no nos cuesta mucho hacer el moldeado, es buena costumbre el hacerlo.

Funciones variadic

En C se permite definir funciones con un número variable de argumentos. Son las llamadas funciones variadic. El ejemplo m s común de función variadic es la función printf(). En C estándar se define un m todo para crear funciones variadic. Para ello se proporciona la cabecera stdarg.h que incluye los tipos y macros que permiten crear dichas funciones.

Declaración de funciones variadic.

Una función variadic se declara igual que las demás funciones salvo que en su lista de argumentos aparece en ultimo lugar el símbolo de elipsis (tres puntos). Puede tener otros argumentos adicionales, pero el ultimo siempre es una elipsis. Un ejemplo de declaración variadic es la función printf(), que se declara como:

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

En esta declaración observamos que la función printf() necesita al menos un argumento, que debe ser del tipo puntero a carácter y luego un número variable de argumentos. Como la función no conoce a priori el número de argumentos debemos diseñar algún m todo de decirle el número de argumentos.

Hay dos m todos sencillos. El primero, que es el usado en printf(), es suministrarle a la función la información sobre el número de argumentos en uno de los argumentos. La cadena de formato pasada como primer argumento a printf() le indica los argumentos que van a continuación y el tipo de los mismos.

Otro método usual es utilizar como ultimo argumento uno con significado especial. Por ejemplo en una función que calcule el mayor de los argumentos podemos colocar como ultimo argumento el número 0, o en una función en la que se hagan operaciones con punteros el puntero nulo.

Definición de funciones variadic

Para definir una función variadic debemos seguir los siguientes pasos:

−incluimos como ultimo argumento de la lista de argumentos la elipsis.

−en la lista de variables declaramos una variable de tipo va_list. Este tipo, al igual que las demás macros necesarias est declarado en stdarg.h, por lo que nos debemos asegurar de incluir dicha cabecera.

−ejecutar la macro va_start antes de comenzar a leer los argumentos. La macro va_start necesita dos argumentos. El primero es de tipo va_list, y debe ser la variable que hemos declarado anteriormente. El segundo es el nombre del ultimo argumento que se declara en la lista de argumentos. Hay algunas restricciones en el tipo de este argumento. No es seguro utilizar como ultimo argumento un arreglo, un float o algún tipo que cambie al ser promocionado. Es seguro usar un puntero y un entero normal.

−para in leyendo los argumentos restantes se va ejecutando la macro va_arg sucesivamente. Esta macro va

leyendo de la lista de argumentos cada argumento, según el tipo que le proporcionemos. Necesita dos argumentos: el primero es la variable de tipo va_list que habíamos definido y el segundo es un tipo, no una variable. Devuelve el valor del tipo que hemos solicitado.

−cuando hayamos leído todos los argumentos debemos ejecutar la macro va_end, cuyo único argumento es la variable de tipo va_list que hemos definido.

Como ejemplo crearemos una función que imprime todas las cadenas de caracteres que le pasemos como argumentos en la salida estándar:

```
#include <stdarg.h>

void prints(char *s, ...)

{

char *p;

va_list arg;

va_start(arg, s);

puts(s);

while ((p = va_arg(arg, char *)) != NULL)

puts(s);

va_end(arg);

}
```

Entrada y salida 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 m s habituales son las que forman parte de la llamada "librería estándar". Originalmente esta librería fue implementada para atender las necesidades del sistema operativo UNIX, aunque es habitual encontrarlas en cualquier compilador de C, incluso en sistemas que difieren bastante del UNIX, como son los entornos gráficos y de ventanas.

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 est n ambas 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 en sencillo y generalmente est 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 as¡ 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. El porqué queda m s claro al explicar el uso de 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 num rico bastar con hacer un bucle que lea números (caracteres numéricos). El bucle normalmente terminar cuando el carácter leído no sea un número. La mejor forma de saber si el siguiente carácter es un número es leerlo. Pero a leerlo, si no es un número ya no estar disponible para futuras lecturas. Aquí¡ es desde se usa ungetchar(). Una vez que hemos comprobado que no es un número lo devolvemos, y as¡ estar listo para la siguiente lectura.

Visto esto podemos seguir con las funciones gets() y puts(). La función puts() simplemente se 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 implementacion sencilla de esta función:

```
void putchar(char *p)

{

while (*p)

putchar(*p++);

}
```

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

Para imprimir datos de un modo m s general el 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. Est 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 ...). Sólo el primer par metro 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 la función fprintf la imprime de modo parecido a como lo hace la función puts(). Por ejemplo:

printf("Hola, mundo\n");

imprime la cadena "Hola, mundo\n" en la salida estándar. Pero además la función printf es capaz de imprimir otros tipos de datos como variables num ricas en la salida estándar. Para ello debemos avisar a la función printf() de que le pasamos como argumento una variable, ya que la función no tiene modo alguno de saber si le hemos pasado algún par metro. 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. As¡ 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 m s importantes son:

−para imprimir caracteres y cadenas de caracteres:

%c imprime un carácter

%s imprime una cadena de caracteres.

Esta código permite imprimir cadenas sin que printf() mire si la cadena contiene posibles códigos de

control.

%% imprime el carácter %

%p imprime la dirección donde apunta un puntero.

El tipo de dato impreso depende de la implementacion. Se suele usar en la depuración de programas o sistemas

−para imprimir enteros

%d imprime un entero en su representación decimal

%u imprime un entero sin signo en su

representación decimal

%x imprime un entero en su representación hexadecimal

%o imprime un entero en su representación octal

%ld imprime un entero largo en su representación decimal

%hd imprime un entero corto en su representación hexadecimal.

Recordar que los enteros cortos se pasan como enteros

−para imprimir números reales (con decimales)

%f imprime un valor del tipo doble precisión como su valor real en simple precisión.

Recordar que los floats se pasan como doubles

%e imprime un valor double en sus representaciones doble precisión. La cadena generada es del tipo

+−ddd.ddd e+−ddd

Estos códigos de control pueden ser en la mayoría de los casos completados con códigos de alineación y de signos. Los códigos de alineación se colocan entre el signo % y el código de control. Los más frecuentes son:

− para justificar por la izquierda una conversión

+ añade un signo + a los valores positivos

' ' (espacio) añade un espacio a los valores con signo que no tengan signo m s o menos

para añadir el prefijo octal 0 en una conversión a octal, o el prefijo 0x en una conversión a hexadecimal

0 rellena con ceros antecedentes en una conversión cuando se ha especificado en ancho que es mayor que el resultado de la conversión

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 controla 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, mundo\n"); /* imprime Hola */

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 sólo 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 m s robusta podemos realizar lo siguiente:

−leemos la entrada en un arreglo de caracteres. Para ello simplemente usaremos la función gets()

−exploramos el arreglo 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, ...);

y 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.

La función scanf salta los espacios precedentes cuando se lee un entero. Si la función no ha terminado de leer los datos pedidos espera a leer una nueva línea de la entrada estándar. Esto lo hace as¡ porque para efectos de formato la función scanf() coincidirá al carácter de nueva línea y al carácter de tabulador como un espacio en blanco, y la función scanf() salta los espacios en blanco. Este efecto de nuevo se puede evitar con el procedimiento anteriormente descrito.Ficheros

El sistema de ficheros habitual en C estándar est enfocado al sistema operativo UNIX, aunque otros sistemas operativos ofrecen con los compiladores de C una librería de funciones que emula en mayor o menor grado a la implementacion UNIX.

El sistema de ficheros de UNIX

El sistema UNIX organiza el sistema de ficheros en directorios. Cada directorio puede contener ficheros y otros directorios. Cada archivo o subdirectorio est identificado por un nombre y una extensión opcional. Es sistema UNIX considera a los dispositivos también como archivos, de manera que se utilizan las mismas funciones para escribir y leer de los dispositivos que las empleadas con ficheros habituales.

Los archivos se manejan con la librería estándar del C mediante las estructuras de archivo FILE. Al ejecutar un programa en UNIX el sistema provee automáticamente tres archivos al programa: la entrada estándar, la salida estándar y la salida de error estándar. El usuario no necesita realizar ninguna operación previa de apertura para emplearlos. De hecho las funciones de entrada y salida estándar envían y recogen datos a través de estos ficheros.

Apertura y cierre de un fichero.

Para abrir un fichero primero debemos crear una variable de tipo puntero a FILE. Este puntero permitir realizar las operaciones necesarias sobre el fichero. Este puntero deber 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 bastar llamar a la función fopen(). Esta función intenta abrir un fichero. Si tiene éxito crear 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 abrir el fichero. El segundo indica el modo en el que se abrir 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 m s 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 a

"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 an

"r+b" ¢ "rb+" Funciona igual que "r+" pero lee y escribe en un fichero binario.

"w+b" ¢ "wb+" Análogo a "w+" pero en modo binario.

"a+b" ¢ "ab+" Análogo a "a+" pero en modo binario.

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

FILE *fp;

if ((fp = fopen("mifichero", " r")) = = NULL)

perror("No puedo abrir el fichero mifichero\n");

/* imprime un mensaje de error */

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 0 en caso de éxito y EOF en caso de error.

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 deberemos dar un puntero a FILE para indicar sobre que fichero efectuaremos la operación, ya que podemos tener simultáneamente abiertos varios ficheros. Las funciones que trabajar con ficheros tienen nombres parecidos a las funciones de entrada y salida estándar, pero comienzan con la letra f. Las m s 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 cabecea estándar como macros. As¡ 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 accederemos 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 deberemos usar las funciones de ficheros con el fichero stderr. Normalmente en UNIX se redirige 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 queremos procesar el fichero mifichero, debemos escribir la línea:

cat mifichero | convierte > nuevofichero

Hemos usado los mecanismos del UNIX de redireccion (> 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.

Lectura y escritura de datos binarios.

Para leer y escribir grupos de datos binarios, como por ejemplo arreglos 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 ultimo argumento un conjunto de datos y lo almacena en el arreglo apuntado por p. Debemos 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 debemos abrir el fichero y cerrarlo después de usarlas. Por ejemplo para leer un arreglo de 100 enteros:

int arreglo[100];

FILE *fp;

fp = fopen("mifichero", "rb");

fread(arreglo, 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 usaremos 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 debemos 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 est n en la memoria.

Operaciones especiales con los ficheros.

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

int feof(FILE *fichero);

Esta función devuelve un 0 si no se ha alcanzado el fin de fichero y un valor distinto de 0 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 0 si ha ocurrido algún error en las operaciones con el fichero y un 0 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);

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 ultimo modo deberemos usar un valor de desplazamiento igual o menor que 0.

Para ver en que posición se halla el puntero del fichero podemos 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 est

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, 0L, SEEK_SET) y luego limpia el indicador de error.

Entrada y salida con tampón.

Cuando la librería estándar abre un fichero le asocia un tampón (buffer) intermedio, que permite agilizar las lecturas y escrituras. As¡ cada vez que se lee un dato del fichero primero se leen datos hasta llenar el tampón, y luego se van leyendo del tampón a medida que se solicitan. De este modo se accede al sistema de ficheros m s eficientemente, ya que para leer un grupo de datos sólo se efectúan unas pocas lecturas del fichero. Este proceso se repite para la escritura de datos. Cuando se envían a un fichero se van almacenando temporalmente en el tampón y cuando se llena el tampón se escribe su contenido en el fichero. Esto plantea varios problemas. El primero es que siempre se debe vaciar el tampón cuando se cierra el fichero. Esto lo hace automáticamente la función fclose(). Sin embargo las demás funciones no lo hacen. Por ello si estamos escribiendo datos en un fichero de lectura y escritura y queremos leer datos, primero debemos vaciar el tampón, para que los datos que leamos est n actualizados. Para ello la librería estándar proporciona la función fflush(), que se define en stdio.h como:

int fflush(FILE *fichero);

Esta función devuelve un 0 si tiene éxito y EOF en caso de error. Además si fichero es un puntero nulo entonces fflush() vacía los tampones de todos los fichero abiertos para escritura.

Para alterar el tamaño del tampón de un fichero podemos llamar a la función setvbuf() inmediatamente después de abrir el fichero. Esta función se define en stdio.h como:

int setvbuf(FILE *flujo, char *buffer, int modo, size_t longitud);

El argumento longitud fija el tamaño del tampón. Es argumento nos indica el tipo de tampón elegido. Hay tres tipos: tampón de línea, tampón completo y sin tampón. Para especificar estos tres tipos de tampones se definen en stdio.h las macros _IOFBF (que indica tampón completo), _IOLBF (que indica tampón de línea) y _IONBF (que indica que el fichero no tiene tampón). El argumento buffer apunta al lugar donde queremos que est el tampón. Si pasamos como argumento buffer un puntero nulo el sistema se encargar de reservar el lugar del tampón y lo liberar al cerrar el fichero.

Podemos asignar a un fichero un tamaño de tampón grande para que el sistema realice menor número de operaciones de lectura y escritura. si el fichero es interactivo(por ejemplo un terminal) quizás nos ser m s útil ajustar el tampón al modo de línea o incluido eliminar el tampón. Deberemos probar diferentes valores y modos para as¡ determinar el mejor tampón a usar.

Operaciones misceláneas 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 0. 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().

El Preprocesador de C

El preprocesado 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.

El preprocesado puede ser de dos tipos: externo (lo realiza un programa adicional) o interno(se preprocesa y compila a la vez. En UNIX el preprocesado es externo, ya que lo hace el programa cpp, que es ejecutado automáticamente por el compilador cc. Es bastante instructivo preprocesar un fichero y revisar el código fuente resultante.

Directivas de preprocesado.

Para realizar las diferentes acciones que admite el preprocesado disponemos 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. La m s sencilla de las 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 implementacion, pero suele estar situado en el directorio del compilador. El Preprocesador y el compilador ya conocen donde 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 sólo 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.

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 EOF −1

#define SEEK_SET 0

#define BUFSIZ 512

Otro uso muy normal en los ficheros de cabecera emplear un símbolo definido con #define como bandera (selector para condiciones), utilizándolo con la directiva #ifdef.

Una macro definida con #define no se puede redefinir a menos que la siguiente definición coincida con la primera exactamente. Para redefinir una macro primero debemos eliminarla con la directiva #undef. Debemos acompañar a la directiva #undef con el nombre de la macro.

Estilo de programación: normas de indentacion.

Definición de las normas de indentacion que se usar n a lo largo del curso en los programas en C. Se seguir n normas muy parecidas a las empleadas habitualmente por otros autores en libros recientes de programación, con algunas variaciones propias.

Estilo y disposición general del programa.

Cada archivo contendrá en primer lugar un comentario donde se explicar brevemente el cometido del mismo. Por ejemplo:

/*

* HOLA.C Este programa imprime en mensage de saludo en la salida

* estándar.

*/

A continuación ir una lista de las cabeceras que se incluir n en l:

#include <stdio.h>

#include <stdlib.h>

#include "hola.h"

A continuación la lista de variables globales:

int i = 0,j,numero = 0,num2;

char c,temp;

A continuación la lista de declaración de funciones, si no se usa un fichero de cabecera para este propósito.

void impmensage(char *mensage);

FILE *pidefichero(char *mensage);

A continuación la función main() y luego las demás.

main(argc, argv)

int argc;

char *argv[];

{

lista de variables locales;

lista de sentencias;

}

Escritura de funciones

Se comienza en la primera columna. Primero va el tipo de la función. Luego el nombre y la lista de argumentos. Entre el nombre y el paréntesis de la lista de argumentos no se deja espacios en blanco. Detraes de cada coma que separa los argumentos se deja un espacio en blanco. Al final se cierra el paréntesis.

A continuación va la declaración de los tipos de los argumentos. Cada declaración de tipo de argumento va en una línea, precedida de un tabulador.

A continuación se abren las llaves de la función. Luego el cuerpo de la función, separando la lista de variables y la lista de declaración de las funciones de las demás sentencias. Todas ellas tienen como mínimo un tabulador a la izquierda, no pudiendo empezar en la primera columna. Se finaliza con la llave de cierre.

char *funci¢n1(arg1, arg2, arg3)

int arg1;

char *arg2;

struct *sutipo arg3;

{

int i,

j;

char *puntero;

setencia1;

.

.

sentencian;

}

Declaración de variables

Se tratar n de colocar por tipos en columnas, procurando que coincidan las comas y los puntos y comas:

char carct1,c;

Podremos incluir una breve explicación de su uso:

char temp; /* variable temporal */

Bucle while

Si consta de una sola sentencia se separa la línea de la expresión de la línea de la sentencia. Se deja un espacio en blanco entre la palabra while y el paréntesis de la expresión:

while (expresión)

sentencia;

Si el bucle consta de un bloque de sentencias se añade la llave de comienzo de bloque detraes de la expresión, en la misma línea. Las diferentes líneas del bloque las colocaremos debajo de la línea de la expresión, colocándoles un tabulador al principio de cada línea. Por ultimo, a la misma altura que las sentencias del bloque, colocaremos la llave de cierre:

while (expresión) { /* comentario muy breve */

sentencia1;

sentencia2;

}

Sentencia if/else

Tras la palabra if, un espacio en blanco y la expresión entre paréntesis.

La palabra else, si la hay, va a la misma altura que la palabra if.

Como norma general seguiremos las mismas convenciones que para el bucle while en el asunto de los bloques y las sentencias, tanto para la para if como para else:

```
if (expresión)

sentencia1;

else

sentecia2;
```

o también:

```
if (expresión) {

sentencia1;

sentencia2;

}

else {

sentencia3;

sentencia4;

}
```