Forma general de un programa en C

```
Declaraciones globales Declaraciones de fn
main( )
{
        variables locales sentencias
}
...
...
fn ( )
{
        .....
}
```

Nombre de identificadores

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

```
int x,y;
char ch;
int main ( )
{ {
x=120; x=y/10;

. . . . . .
} }
```

Archivo 2

```
extern int x,y;
extern char ch;
void func1( )
```

Variables 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 caracteres esto significa colocarla en un registro de la CPU.
Sólo se puede aplicar a variables locales y a los parámetros formales 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 = expresión;

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 tendrán valores desconocidos antes de que se lleve a cabo su primera asignación.

Constantes
#define nombre valor
Ejemplo de constantes
#define num 9
#define nombre "Matias"

Caracteres 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 Verdadero 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) *(lo apuntado por)**

Sizeof

Es un operador unario que devuelve la longitud, en bytes, de la variable o del especificador de tipo al que precede.
Ejemplo: float 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 parámetros)
{
        .....
        .....
        Return (variable de retorno)
}
```

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);
int main ( )
{
        int t=10;
        printf ("%d %d",cuad(t),t);
        return 0;
}

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

Salida es << 100 10 >>

Llamada por referencia

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

```
int main()
{
        int x,y;
        inter (&x,&y);
        return 0;
}
inter (int *x,int *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 caracteres 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:

```
int main ( ) /*Si una función recibe un array unidimensional, se puede declarar el parámetro formal como un puntero, como un array delimitado o como un array no delimitado.*/
{
        int i[10];
        func1 (i);
        return 0;
}
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 caracteres 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. \0

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 caracteres 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);
int 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:

```
int 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:

```
int main ( ) /* cambio de mayúscula / minúscula */
{
        char car;
        do { car=getche( );
                if (islower(car)) putchar (toupper (car));
                else putchar (tolower (car));
        } while (car=!'.');
        return 0;
}
```

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:

```
int 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);

Duración de las variables.

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

Las variables son dinámicas si son creadas dentro de una función. Su existencia está ligada a la existencia de la función. Se crean cuando la función es llamada y se destruyen cuando la función o subrutina devuelve el control a la rutina que la llamó.

Las variables estáticas se utilizan para almacenar valores que se van a necesitar a lo largo de todo el programa.

Las variables dinámicas se suelen utilizar para guardar resultados intermedios en los cálculos de las funciones.

Como regla general una variable es estática cuando se crea fuera de una función y es dinámica cuando se crea dentro de una función.

Por ejemplo en el siguiente programa :

```
#include <stdio.h>
int numero1 = 1;
int main ()
{
        int numero2 = 2;
        printf("%d, %d\n", numero1, numero2);
        return 0;
}
```

hemos creado la variable estática 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 función main() es la que ocupa todo el tiempo de ejecución, por lo que no apreciaremos diferencia en el uso de ambas.

Alcance de las variables

Otra característica de las variables es su alcance. El alcance se refiere a los lugares de un programa en los que podemos utilizar una determinada variable. Distinguiremos así dos tipos principales de variables: globales y locales. Una variable es global cuando es accesible desde todo el programa, y es local cuando solo puede acceder a ella la función que la 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 el 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 más 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 cambiará estas características.
El primer modificador es la palabra clave static. Cuando a una variable local se le añade el modificador static pasa de ser dinámica a ser estática. Así la duración de la variable se amplía a la duración del programa completo. Observar que una variable estática solo se crea una vez, al principio del programa, por lo que la inicialización solo se produce una vez para una variable estática.
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 extern. Este modificador se usa cuando una variable que se creó en otro modulo se quiere usar en un módulo. 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;
int main ( )
{
        printf("%d\n", numero);
        return 0;
}
```

es un programa en que declaramos la variable externa número, 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 dinámica (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:

```
int main ( )
{
        auto numero = 1;
        printf("%d\n", numero);
        return 0;
}
```

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 máquina, 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 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 dinámica a ser estática. Así la duración de la variable se amplía a la duración del programa completo. Observar que una variable estática solo se crea una vez, al principio del programa, por lo que la inicialización solo se produce una vez para una variable estática.
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 extern. Este modificador se usa cuando una variable que se creó en otro modulo se quiere usar en un módulo. 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;
int main ( )
{
        printf("%d\n", numero);
        return 0;
}
```

es un programa en que declaramos la variable externa número, 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 dinámica (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:

```
int main ( )
{
        auto numero = 1;
        printf("%d\n", numero);
        return 0;
}
```

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 máquina, 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 más 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 así, 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.

Punteros

Cuando queramos pasar un dato a una función normalmente pasamos una copia del dato. Esto es sencillo de hacer y rápido, 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 sí, 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 está 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 está inicializado:

```
char *p = &p1;
char *p2 = p; /* p ya está inicializado */
```

- inicializarlo con cualquier expresión constante que devuelva un lvalue.

Las más 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 sí 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 método 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 cómo 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 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++);
```

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 está bastante extendido en la programación de sistemas.

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 qué tipo de dato se está 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.

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 dinámica. 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 estáticas 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 dinámica (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 dinámica. 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 está 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 dinámica para un arreglo comenzaremos asignando el espacio con malloc.
Por ejemplo:

```
#include <stdlib.h>
int 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);
        return 0;
}
```

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 está 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.

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 está 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 más 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 mas 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. Así la línea:

```
printf("hola\n");
```

es equivalente a la escritura en el fichero stdout:

```
fprintf(stdout, "hola\n");
```

A los ficheros stdin y stdout normalmente 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 redirección (> envía la salida estándar de un programa a un fichero), de tubería (| conecta la salida estándar de un programa con la entrada estándar de otro) y la utilidad cat, que envía un fichero a la salida estándar.

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

Para ver en qué 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 esta 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 más 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 estén 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 esté 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 existiera 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 preprocesador 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 implementación, 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.

Webgrafía y Licencia

◆ Textos tomados, corregidos y modificados de diferentes páginas de Internet, tutoriales y documentos.

◆ Autor:

Matías E. García

Prof. & Tec. en Informática Aplicada
www.profmatiasgarcia.com.ar
info@profmatiasgarcia.com.ar