Introducción al C
Ampliación de Informática

Belarmino Pulido
Dpto. Informática

Índice general

Capítulo 1

Introducción

- Lenguaje de medio nivel (entre ensamblador y un lenguaje de alto nivel).
- Muy sencillo, con 28 palabras clave.
- Portátil hasta cierto punto, sólo se necesitaría recompilar.
- Incorpora una librería estándar de funciones.
- Permite construirse las propias funciones o estructuras de datos.
- Utilizado fundamentalmente para la programación de sistemas, puesto que o forma parte o interfiere directamente con el sistema operativo del ordenador. Además genera un código que se ejecuta tan rápido como el ensamblador y está pensado como un lenguaje para programadores.
- Es un lenguaje estructurado: uso de bloques de programación, funciones.
- El C es un lenguaje compilado: fuente $\longrightarrow$ objeto $\longrightarrow$ ejecutable. El proceso de compilación es más lento, frente al uso de un intérprete, pero una vez compilado, el programa se ejecuta mucho más rápidamente.

Capítulo 2

Estructura de un programa

Un programa en C no es más que una colección de funciones, que una vez definidas se usan en el programa principal de una forma determinada.

Las funciones no son más que una colección de sentencias que realizan una tarea determinada. Podrán tener cualquier nombre, excepto **main** que es el nombre de la función principal, que se encarga de gestionar el resto. Podríamos mirarla como el programa principal.

Las funciones, para distinguirlas del resto de los objetos que aparecen en el programa, irán acompañadas, tras su nombre, de paréntesis. Podemos decir, que en general, tanto **main** como cualquier otra función, sigue en C el siguiente patrón:

nombre_funcion (lista_de_argumentos)
lista_argumentos, declaración

{ Llave abierta que indica el comienzo de la función
⋮
cuerpo de la función
⋮
} Llave cerrada que indica el fin de la función.

En C existe una sentencia específica **return** para devolver un valor de un tipo determinado. En caso de que no aparezca, el compilador de C determina que la ejecución de la función termina cuando se alcanza la llave de cierre.

2.1. La función main

Primera función en ejecutarse, esté donde esté. Por cuestiones de claridad siempre suele ponerse al principio, y todas las funciones que se utilizan se ponen debajo. La única diferencia de **main** con otras funciones es que cuando se encuentra el procesador la llave de cierre termina el problema, en vez de devolver el control al programa principal.

2.2. Los argumentos de una función

Las funciones pueden tener cero o más argumentos, separados por comas. De tal manera que cuando se define la función se ponen los parámetros formales, que contendrán la información que se les pase cuando se llame a la función. Hay que asegurarse que los parámetros formales se correspondan en número, tipo y posición con los reales.

El C es un lenguaje robusto y hace muy pocas comprobaciones de tipos, por lo que lo más probable es que haga algo, aunque no sea lo que nosotros queremos si nos equivocamos en los tipos. C tiende a utilizar posiciones de memoria, más que el concepto de variable, con lo cual ante unos parámetros de tipos distintos a los esperados el resultado de su utilización puede ser totalmente inesperado.

Ejemplo:

```
mul (x, y) /* funci\'on mul */
int x, y;  /* x e y se declaran como variables enteras */
{
  return (x*y); /* Devuelve el producto de ambos argumentos */
}
```

Ejemplo: Utilización de la función mul en un programa.

```
main()
{
   int x, y, j, k;

   x = 1;  y = 2;
   p = mul(x,y);
   printf ("%d",p); /* escribe el valor de p en decimal */
```

```
   j = 234; k = 10;
   p = mul (k, j);
   printf("%d",p);
}

/* Aqui vendria la definicion de la funcion mul */
```

2.2.1. Puntos y coma, llaves, comentarios y posición

El punto y coma es un terminador de una entidad lógica.
Las llaves permiten definir bloques de sentencias agrupadas.
El C no reconoce el carácter de nueva línea como terminador, por tanto podemos extendernos lo que queramos. Se utiliza por motivos de claridad simplemente.
Por tanto las siguientes expresiones son equivalentes:

```
x = y;
y = y + 1;
mul(x,y);
```

Es equivalente a:

```
x = y;  y = y + 1;          mul(x,y);
```

El formato de las líneas es libre en C, por lo que puede utilizarse indentación o no. Se recomienda utilizar siempre indentación para hacer más claros los programas.

2.2.2. La función printf

Función predefinida que muestra por pantalla el contenido de sus argumentos.

printf (" %d", 123);

En general tendrá la forma:

printf (çadena de control", lista de argumentos);

La cadena de control especifica el formato en que deben aparecer los argumentos, así como el número de argumentos que se le van a pasar. Esto sería equivalente a los WRITE con formato de Fortran.

Dentro de la cadena de control pueden aparecer:

- Caracteres que aparecerán como tales.
- Comandos de formato, que especifican cómo aparecen los argumentos.

Códigos de control en printf

código	**Formato**
%c	carácter sencillo
%d	decimal
%e	notación científica
%f	decimal en punto flotante
%g	utiliza el más corto de los dos anteriores
%o	octal
%x	hexadecimal
%s	cadena de caracteres
%u	decimal sin signo

Capítulo 3

Estructuras elementales: variables, constantes, expresiones y operadores

3.1. Variables

Pueden estar nombradas por uno o varios caracteres. Dependiendo de los compiladores se aceptarán nombres más o menos largos, pero al menos se aceptan seis distintos. Se puede incluir el carácter de subrayado. Pueden comenzar su nombre por una letra (más recomendable) o bien por el carácter de subrayado.

Se diferencian las letras mayúsculas de las minúsculas, por tanto la variable *importe* e *IMPORTE* son distintas.

Se puede dar cualquier nombre a una variable excepto: una palabra clave de C o el nombre de una función de C. **En C es obligatorio definir todas las variables antes de utilizarlas**, a diferencia por ejemplo de Fortran que asignaba tipos por defecto a algunas variables.

Lista de palabras clave en C. Notar el uso de minúsculas

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

En C también se distinguen variables **globales** conocidas por todas las funciones y las variables **locales** que son conocidas sólo dentro de su misma función. Las variables locales existen mientras la función es usada. Se crean cuando se llama a la función y se destruyen cuando se termina.

Cuando se definen variables se dice primero a qué tipo pertenecen y luego se les da el nombre. En C existen 7 tipos de datos predefinidos, que luego se pueden ampliar utilizando los registros, que en C se conocen como estructuras o *struct*.

En la tabla siguiente aparecen los 7 tipos predefinidos con su rango y longitud en bits para la mayor parte de los microprocesadores. Antes de utilizar esta información convendría contrastarla con el manual de C que se utilice. Notar también que en C los caracteres se manipulan y consideran como un entero sin signo.

Tipos de datos predefinidos en C y sus palabras clave asociadas

Tipos de datos	**Palabra clave C**	**Bits**	**Rango**
carácter	*char*	8	0 a 255
entero	*int*	16	-32768 a 32767
entero corto	*short int*	8	-128 a 127
entero largo	*long int*	32	-4294967296 a 4294967295
entero sin signo	*unsigned int*	16	0 a 65535
coma flotante	*float*	32	6 dígitos precisión
coma flotante con doble precisión	*double*	64	12 dígitos de precisión

Vamos a ver ahora un ejemplo de definiciones de variables dentro de una función , variables locales, y de definiciones fuera de la función main, que hacen a las variables globales.

```
ejemplo()
```

```
{
  int cuenta;
  char c;
  float valor;
  short int punt;
  long int eof_contador;
  double desembolso;
  unsigned int u;
  .
  .  /* Cuerpo de la funcion */
  .
}

int cuenta_linea;
main()
{
.. /* Definicion de la funcion */
}
```

3.2. Algunos tipos de variables especiales

Se pueden declarar variables que sean **static** lo que las hace existir durante toda la ejecución del programa. Estas se diferencian de las variables globales en que mantienen sus valores entre llamadas sucesivas a funciones. Tener en cuenta que las variables de tipo **static** sólo son conocidas dentro de las funciones en las que son declaradas.

Se puede utilizar el modificador **register** en enteros y caracteres. Así ocupan un registro en la CPU y no un lugar en Memoria Principal, con lo cual el acceso a la variable es mucho más rápido. Sólo puede aplicarse este modificador a las variables locales de una función y a los parámetros formales de la misma. La rapidez de acceso y modificación de estas variables las hace especialmente idóneas para las variables contador de los bucles. Recordar que el número de variables register permitidas en cada procesador está limitado. Si se utilizan más de las autorizadas la consecuencia es que las variables no se almacenan en la CPU si no en la memoria.

La sentencia **extern** permite utilizar una variable que ha sido declarada en otro sitio. Cuando se utilizan varios ficheros para un mismo programa, se puede declarar en un fichero una variable se declara global y puede así ser utilizado por otros ficheros.

3.3. Los arrays o vectores

Podemos tener en C vectores de cualquier tipo de dato básico. Siendo su sintaxis la siguiente:

tipo nombre-variable [*numero-elementos*]

Importante:

- El primer elemento del vector tendrá índice 0 y el último el número de elementos menos 1.
- C no comprueba que se exceda el número de elementos de un vector.
- En el caso de definir una cadena de caracteres como un vector, luego puede accederse de forma individual a cada uno de ellos como si fuese un carácter.

Ejemplos:

```
/* Definiciones */
int q[10];
char c[10];
float f[20];

/* Utilizacion */
main()
{
     char cadena[80];  /* Array de 80 caracteres */

     gets(cadena);   /* rutina de C que lee una cadena por teclado */
     printf (cadena) /* Imprime la cadena; tambien podria utilizarse
                        printf ("%s",cadena); o bien puts(cadena)  */
     /* Impresion de caracteres de forma individual */
     printf ("%c %c %c", cadena[0], cadena[1], cadena[2]);

}
```

3.4. Expresiones y operadores

3.4.1. Sentencias de asignación

nombre_ variable = expresión

3.4.2. Conversión de tipos en asignaciones

En C cuando en una asignación hay tipos distintos a izquierda y derecha lo que se hace es transformar la expresión del lado derecho de la asignación para que quede del tipo del lado izquierdo. Se seguirán las siguientes normas para calibrar la pérdida de información en las conversiones:

Tipo obtenido	**Tipo de la expresión**	**Pérdida posible de información**
char	short int	signo
char	int	8 bits mayor peso
char	long int	24 bits mayor peso
short int	int	8 bits mayor peso
short int	long int	24 bits mayor peso
int	long int	16 bits mayor peso
int	float	parte fraccionaria y posiblemente más precisión
float	double	resultado redondeado

3.4.3. Inicialización de variables y arrays

A la mayoría de las variables se les puede dar un valor al declararlas:

tipo nombre_ variable = constante;

Ejemplos:

```
char ca = 'a';
int primero = 0;
float balance = 123.45;
```

Casos especiales:

- Variables globales y static se inicializan al principio del programa. Se supone que se inicializan a 0 por defecto. Sólo se supone.
- Variables locales y register se inicializan cuando se crea la función. No se inicializan por defecto.
- Conclusión: inicializar siempre para evitar problemas.
- Los arrays globales también se pueden inicializar, pero hay que tener en cuenta que C siempre finaliza una array con el carácter nulo '\0'. Por tanto cuando se declaran los arrays hay que dejar siempre un carácter más para que el compilador añada el de terminación.
- Los arrays locales no pueden inicializarse, pero los locales static si.

Ejemplos:

```
char cdn[8] = "cadenas";
char cdn[8] = {'c','a','d','e','n','a','s','\0'}
```

3.4.4. Constantes

Tipo de Dato	**Ejemplos**
char	'a' '\n' '9'
int	1 123 21000 -234
long int	35000 -34
short int	10 -12 90
unsigned int	10000 987
float	123.23 4.3e-3
double	123.23 123123123 -0.9876324

3.4.5. Constantes carácter con barra invertida

Existen caracteres especiales que no pueden introducirse directamente. A ellos se les asigna una combinación de teclas que incluyen el carácter '\', como por ejemplo:

```
'\n' : nueva linea
'\0' : terminador nulo
'\b' : espacio
'\f' : form feed
'\r' : retorno de carro
```

```
'\t' : tabulacion
'\'' : comilla simple
```

De esta manera pueden introducirse caracteres especiales como constantes.

3.4.6. Operadores

Operador Aritmético	Acción
–	resta, también menos unario
+	suma
*	multiplicación
/	división
%	resto de la división entera
– –	decremento: x– – = – – x ⟶ x = x - 1
++	incremento: : x++ = ++x ⟶ x = x + 1

Comentarios

- La división entre enteros o carácter produce un entero. Sin resto.
- La operación módulo sólo se puede efectuar entre enteros.
- Los operadores ++ y – – se evalúan de forma diferente si están a la izquierda o a la derecha de su operador. Si están a la izquierda se evalúa la operación antes de hacer la asignación. Si están a la derecha se hace la operación de asignación y luego se realiza el incremento o decremento.
- Para entender los conceptos de operadores lógicos y relacionales hay que tener en cuenta que en C se considera falso el valor 0, mientras que es cierto cualquier valor distinto de 0. Además el resultado de evaluar un operador lógico será 1 para significar cierto y 0 para denotar falso.
- Los operadores relacionales y lógicos tienen menos precedencia a la hora de evaluarse que los aritméticos.

Operador Relacional	Acción
>	mayor que
>=	mayor o igual que
<	menor que
<=	menor o igual que
==	igual
!=	distinto

Operador lógico	Acción
&&	AND
||	OR
!	NOT

Ejemplo:

```
10 > 5 && ! (10 < 9) || 3 <= 4
   0   && !    0     ||   1
   0   &&      1     ||   1
       0             ||   1
                     1
```

Precedencia entre los operadores lógicos y relacionales:

```
maxima    !
          > >= < <=
          == !=
          &&
minima    ||
```

3.4.7. Operadores de bits

Las operaciones con bits se refieren a la comprobación, colocación o desplazamiento de los bits actuales de una variable entera o carácter. Estas operaciones no pueden realizarse ni con los tipos **float** ni **double**. Estas operaciones suelen realizarse cuando se comunican programas con dispositivos físicos.

Operador sobre bit	Acción
&	AND
|	OR
^	XOR
~	complemento a uno
>>	desplazamiento a la derecha
<<	desplazamiento a la izquierda

Nota:
Las operaciones entre bits se suelen utilizar para cambiar los valores de determinados bits de un operando, no para evaluar expresiones a cierto o a falso.

Los operadores de *desplazamiento* >> y << mueven todos los bits de una

variable a la izquierda y a la derecha, según se especifique.

variable >> número de posiciones en bits
variable << número de posiciones en bits

Nota:
Los desplazamientos no son rotaciones, esto es, cuando se desplazan los bits por un extremo se van completando con ceros por el extremo contrario.
Estas operaciones suelen utilizarse para realizar conversiones rápidas o para realizar operaciones rápidas de multiplicación o división en binario.

Ejemplo:

char x;	**x después de cada ejecución**	**Valor de x**
x=7;	0 0 0 0 0 1 1 1	7
x<<1;	0 0 0 0 1 1 1 0	14
x<<3;	0 1 1 1 0 0 0 0	112
x<<2;	1 1 0 0 0 0 0 0	192
x>>1;	0 1 1 0 0 0 0 0	96
x>>2;	0 0 0 1 1 0 0 0	24

3.4.8. Los operadores de puntero & y *

En C, un puntero nos permite acceder a la dirección de memoria que ocupa una variable. Lo cual puede ser muy útil en ciertos tipos de rutinas. Sin embargo, se utilizan para dos funciones fundamentales:

- Referencia rápida a los elementos de un array.
- Las funciones pueden modificar los parámetros que recibe.

Los operadores especiales que permiten la existencia de punteros son:
&: Operador unario que devuelve la dirección de memoria del operando. Por ejemplo:
m = & cont
Coloca en m la dirección de memoria que ocupa la variable cont, que está relacionada con la memoria del ordenador, y no con el valor que tome la variable cont.

En el caso en que cont tuviera el valor 1996 almacenado en la posición de memoria 2002, el resultado de la asignación $m = \&cont$ pone en m el valor 2002.

*****: Devuelve el contenido de la posición de memoria que se le pasa como operando. Si seguimos con el ejemplo anterior, el resultado de realizar: $q = *m$ es que q vale 1996, puesto que es el contenido de la posición de memoria 2002, que es donde estaba almacenada cont que valía 1996.

Nota:
Los operadores de puntero **&**, ***** comparte símbolos con los operadores AND binario y de la multiplicación aritmético. Los operadores de puntero tienen mayor precedencia que los anteriores.

En C, las variables que van a contener direcciones de memoria se llaman **punteros**, y en su declaración hay que indicar al compilador que van a contener direcciones, incluyendo ***** delante de su identificador.
*char *ca;*
Estas declaraciones de variables pueden aparecer mezcladas con las de otras variables del mismo tipo que no sean punteros:

*int x, *y, cont;*

Ejemplo:

```
main()   /* Asignacion con * y & */
{
     int destino, origen;
     int *m;

     origen = 10;
     m = &origen;
     destino = *m;
}
```

En este programa, destino contiene al final el contenido de origen, o sea, 10.

A continuación se introduce una tabla resumen de las precedencias entre los operadores en C:

máxima	() [ ] – ⟶ .
	! ~ ++ – – – (tipo) * & sizeof
	* / %
	+ –
	<< >>
	< <= > >=
	== !=
	&
	~
	|
	&&
	||
	?:
mínima	= += –= *= /=

3.5. Expresiones

3.5.1. Conversión de tipos en las expresiones

Si se mezclan operandos de distinto tipo se convierten a uno de ellos, siguiendo las siguientes normas:

- **char** y **short int** pasan a **int**. Los **float** a **double**.
- Para las parejas de operandos, si uno es **double**, el otro se convierte a **double**. En caso contrario, si uno de los operandos es **long**, el otro se convierte en **long**. O si uno de los operandos es **unsigned** el otro es **unsigned**.

Cuando se encuentra una expresión se van aplicando estas reglas a cada par de operandos, hasta que se ha aplicado a toda la expresión.

3.5.2. Moldes

Utilizando este operador unario, se convierte el tipo de una expresión en el tipo que nosotros especificamos mediante el molde. Tiene la siguiente sintaxis:

(tipo) expresión

Ejemplo:

```
main() /* Imprime i e i/2 con parte fraccionaria */
{
  int i;

  for (i=1; i<=100; ++i)
     printf("%d /2 es: %f\n",i, (float) i /2);
}
```

3.5.3. Espaciado y paréntesis

En C se pueden utilizar paréntesis y espaciado sin que afecte a la semántica del programa. Se recomienda utilizarlo para aumentar la legibilidad del programa.

3.5.4. Abreviaturas en C

En el caso de los operadores aritméticos se pueden escribir las expresiones de forma abreviada. Por ejemplo:

variable = variable operador expresión
pasa a ser
variable operador = expresión

```
x = x + 1                    x += 1
x = x - 1                    x -= 1
x = x * 2                    x *= 2
x = x / 2                    x /= 2
```

Aunque esta notación no es obligatoria es ampliamente utilizada, por lo que se recomienda su utilización con el fin de familiarizarse con su aparición en los programas.

Capítulo 4

Estructuras, uniones y tipos definidos del usuario

Además de los tipos de datos básicos, se pueden crear registros con las sentencias **struct** y **union**. Además también se pueden renombrar tipos de datos existentes utilizando *typedef*.

El formato general de una declaración **struct** es:

```
struct [nombre_estructura] {
    campo_1;
    campo_2;
    .
    .
    .
} [variable_estructura];
```

El formato general de una declaración **union** es:

```
union [nombre_union] {
    campo_1;
    campo_2;
    .
    .
    .
} [variable_union];
```

La diferencia entre una unión y una estructura o registro es que la definición de unión implica que todos los tipos de datos que aparecen separados

por punto y coma ocupan la misma posición de memoria. Se utiliza cuando se quiere que una misma variable pueda contener datos de distinto tipo, o cuando se quiere utilizar una misma variable desde distintos puntos de vista.

Ejemplos:

```
struct dir {
   char nombre[30];
   char calle[40];
   char ciudad[20];
   char prov[20];
   unsigned long int DP;
};
```

Si queremos definir una variable:

```
struct dir ainfo;
```

Cuando al mismo tiempo se definen variables aprovechando la definición de la estructura, o cuando sólo vamos a tener una variable de ese tipo y no queremos definir el tipo, las sentencias serían de la forma:

```
struct dir {
   char nombre[30];
   char calle[40];
   char ciudad[20];
   char prov[20];
   unsigned long int DP;
} ainfo, binfo, cinfo;

struct {
   char nombre[30];
   char calle[40];
   char ciudad[20];
   char prov[20];
   unsigned long int DP;
} dinfo;
```

Para referirnos a los elementos de una estructura se utiliza la notación:

nombre_estructura.nombre_campo

Por ejemplo:

```
ainfo.CP = 33180;
printf ("%d\n",ainfo.DP);
```

Si lo que queremos es tener un vector de estructuras, tendremos que definir:

```
struct dir ainfo[100];

printf("%d\n",ainfo[i].DP);

struct x {
    int a[10][10];
    float b;
} y;
```

En el caso de las uniones, la definición de una unión nos permite usar la posición de memoria para almacenar diferentes informaciones. Cuando se crea la unión se reserva memoria para almacenar la mayor cantidad posible de información.

```
union u {
    int i;
    char ca;
};

union u cnvt;
```

4.1. La función sizeof

Dado que el tamaño de los distintos tipos de datos variará de unas máquinas a otras, se dispone de la función **sizeof** que devuelve el tamaño de la variable que se pase como argumento, sea esta del tipo que sea:

```
char ca;
int i;
float f;

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

Si lo que queremos es utilizar la sentencia **typedef** para renombrar tipos de datos y que nos sea más sencilla la programación:

```
typedef float balance;

balance deuda;

typedef struct cliente {
     float credito;
     int deuda;
     char nombre[40];
};
cliente clist[NUMCLIENTES]; /* Define un array de estructuras del
                               tipo cliente, que es una estructura */
```

Con este método la programación es más sencilla, pero no se están definiendo realmente nuevos tipos de datos.

Capítulo 5

Estructuras de control: condicionales y esquemas iterativos

5.1. Esquemas condicionales: if y switch

C contiene dos tipos de esquemas condicionales, **if** que corresponde con la estructura **si-entonces-sino** y **switch** que corresponde con la estructura **segun**.

5.1.1. IF-ELSE

```
if (prueba-condicion) sentencia1
else sentencia2

if (prueba-condicion)
{
   sentencia1;
   sentencia2;
   ...
}
else {
   sentencia1;
   sentencia2;
   ...
}
```

Ejemplo:

```
main() /* Programa que comprueba si leo un numero determinado */
{
   int n = 123;
   int correcto;

   correcto = getnum();

   if (correcto == n) printf ("**** Correcto ****");
   else printf (" !!!!Equivocado !!!!!");
}
```

En C la sentencia **else** se asocia a la sentencia **if** más cercana que no tenga asociada una sentencia else.

Ejemplo:

```
main() /* Programa que comprueba si leo un numero determinado */
{
   int n = 123;
   int correcto;

   correcto = getnum();

   if (correcto == n) {
       printf ("**** Correcto ****");
       printf ("%d era el numero \n");
   }
   else
      if (correcto > n) printf ("!!!!  Equivocado !!!!! Demasiado alto");
      else printf ("!!!!  Equivocado !!!!!  Demasiado bajo");
}
```

5.1.2. El operador ?

Este operador nos permite realizar de forma sencilla y rápida la misma labor que realiza un condicional **if-then-else**.

```
main() /* Programa que comprueba si leo un numero determinado */
```

```
{
   int n = 123;
   int correcto;

   correcto = getnum();

   if (correcto == n) {
      printf ("**** Correcto ****");
      printf ("%d era el numero \n");
   }
   else
      correcto > n ?  printf ("alto") :  printf ("Bajo");
}
```

5.1.3. SWITCH

Es equivalente al esquema condicional generalizado: **segun-hacer-en-otro-caso**.

```
    switch(variable) {
       case constante1:  sentencia;
       case constante1:  sentencia;
       case constante1:  sentencia;
       ...
       default: sentencia;
    }
```

Ejemplo:

```
menu()
{
    char ca;

    printf("1.  Comprobacion\n");
    printf("2.  Correccion\n");
    printf("3.  Muestra\n");
    printf("Otra tecla terminar\n");
    printf("\n\n Introduzca su elecci\'on\n");

    ca = getchar();
```

```
    switch(ca) {
       case '1':  func_comprobacion(); break;
       case '2':  func_correccion(); break;
       case '3':  func_muestra(); break;
       default:  printf("Termino");
    }
}
```

La presencia del **break** en la estructura **switch** es debido a que en C el hecho de que se verifique la comparación *ca = 'x'* no indica que sólo se ejecuta la sentencia asociada a ese **case**. **break** termina la ejecución del **switch**.

5.2. Esquemas iterativos: for, while y do/while

5.2.1. for

Realiza en C las acciones asociadas a un esquema **para**. Su sintaxis, tanto para ejecutar una sentencia como para ejecutar un grupo de sentencias es:

for (inicialización; condición; incremento)

sentencia;

for (inicialización; condición; incremento) {

sentencia;
sentencia;
...

}

Ejemplo:

```
for (x=100; x != 65; x = x - 5)
{
   z = sqrt(x);
   printf ("La raiz cuadrada de %d es %f \n", x, z);
}
```

En C la prueba de que la condición se cumpla se realiza antes de entrar al bucle, con lo cual pueden darse casos en que no se ejecuten ninguna vez las sentencias del bucle.

Además en C pueden darse varias sentencias de inicialización separadas por comas. Lo mismo puede hacerse para las sentencias de incremento o decremento.

Ejemplo:

```
for (x = 0, y = 100; x < y; ++x, --y)
    printf ("%d %d\n", x, y);
```

En C se permite que alguna de las partes de control no estén presentes. Simplemente tendríamos que respetar su posición y seguir poniendo los puntos y coma. Esto se refiere fundamentalmente a la ausencia de inicialización o de parte de incremento, que podrían estar presentes fuera o dentro del bucle, respectivamente. Aunque esta opción se puede realizar, está prohibido como norma general en esta asignatura.

```
x = 0;
for (; x < 10; )
{
   printf ("%d", x);
   ++x;
}
```

Incluso utilizando la sentencia **break** mencionada anteriormente se podría llegar a prescindir de la condición de terminación del bucle. Cuando se necesite alguna estructura de control de programa de este tipo se recurrirá a los bucles while o do/while.

También está permitido no realizar ninguna acción. Esto nos permitiría no hacer nada durante la fase de comprobaciones. Esto es, se emplea como retardo.

5.2.2. while

Realiza en C el esquema iterativo **mientras**. Su sintaxis es la siguiente:

while *(condicion) sentencia;*

while *(condicion) { sentencia; sentencia; ... }*

Las sentencias se realizan siempre que condicion sea distinta de CERO. Esto es, siempre que al evaluar la condición se obtenga una expresión que es distinta de cero, que es el equivalente al falso en C, se considera que se ha de realizar la o las sentencias asociadas.

Lo mismo que el bucle **for**, **while** comprueba que se cumpla la condición antes de ejecutar por primera vez la o las sentencias.

```
espera_ca()
{
   char ca;

   ca = 0; /* Inicializa ca */

   while(ca != 'A') ca = getchar();
}
```

Aquí tampoco es necesario que haya sentencias dentro del bucle. Por ejemplo, el bucle anterior podría realizarse de la siguiente manera:

```
espera_ca()
{
   char ca;

   ca = 0; /* Inicializa ca */

   while((ca = getchar()) != 'A') ;
}
```

En este caso, debido a la prioridad que otorgan los paréntesis, se lee primero un carácter con la función **getchar**, se asigna su valor a la variable **ca** y a continuación se comprueba que esa variable sea distinta de 'A'.

5.2.3. do/while

Al igual que el esquema **repetir**, este esquema iterativo comprueba la condición de terminación al final, permitiendo que se realice su sentencia asociada al menos una vez. Su sintaxis es la siguiente:

do *sentencia* **while** *(condicion) ;*

do *{ sentencia; sentencia; ... }* **while** *(condicion);*

Por ejemplo:

```
do {
   num = getnum();
} while (num < 100);
```

Uno de los usos más frecuentres de esta estructura sea cuando se despliega un menú en el que se pide que se seleccione una opción, y que ha de hacerse al menos una vez y hasta que al seleccionar una tecla haya que abandonar el bucle.

Ejemplo:

```
menu()
{
    char ca;

    do {
       printf("1.  Comprobacion\n");
       printf("2.  Correccion\n");
       printf("3.  Muestra\n");
       printf("Otra tecla terminar\n");
       printf("\n\n Introduzca su elecci\'on\n");

       ca = getchar();

       switch(ca) {
         case '1':  func_comprobacion(); break;
         case '2':  func_correccion(); break;
         case '3':  func_muestra(); break;
         default:  printf("Termino");
       }
      } while (ca != 0);
}
```

5.2.4. Terminación de bucles con break y exit

Cuando en el interior de un bucle nos encontramos con una sentencia **break** se produce la terminación del bucle. Es decir, se dejan de ejecutar las sentencias que vendrían a continuación y se considera que se cumple la condición de terminación.

En el caso de bucles anidados, la sentencia **break** hace que se abandone el bucle más interno.

```
for (t=0; t < 100; t+=1) {
  contador = 1;
  do {
     printf ("%d", contador);
```

```
     contador++;
     if (contador == 10) break;
  } while (1);  /* Esto ser\'{\i}a un bucle infinito */
}
```

Sin embargo, si nos encontramos dentro de un bucle con la sentencia **continue**, lo que ocurre es que dejamos de ejecutar las sentencias que queden en esa iteración y nos situamos otra vez al principio del bucle para realizar de nuevo la comprobación. Es algo así como dejar todo lo que queda y volver a comprobar la condición.

```
do {    /* Este bucle lee numeros y solo escribe los que sean positivos o 0 */
   x = getnum();
   if (x < 0) continue;
   printf ("%d", x);
} while (x != 100);
```

Existe también en C la función del sistema **exit** que en caso de ser ejecutada termina el programa. Es una forma abrupta de llegar al final del programa. En algunos sistemas exige un argumento, en otros no tiene o bien es opcional. Este argumento le diría al sistema operativo si la forma en que ha terminado el programa ha sido normal o se ha debido a algún fallo.

Capítulo 6

Funciones

Las funciones en C implementan las subrutinas o acciones del lenguaje algorítmico, y tienen la siguiente sintaxis:

nombre_funcion ([lista_de_argumentos])
[lista_argumentos, declaración]

{ Llave abierta que indica el comienzo de la función
⋮
cuerpo de la función;
⋮
} Llave cerrada que indica el fin de la función.

Las funciones en C siempre devuelven algún valor, que se puede especificar mediante la sentencia **return**. Si no se especifica, devolverán un valor entero y si no se devuelve ningún valor, por defecto retornará cero. Debido a esta característica del C, se pueden emplear funciones como parte de las expresiones:

```
x = potencia (y);
if (max(x,y) > 100) printf ("mayor que");
for (ca=getchar(); esdigito(ca);) ...;
```

Pero no pueden ser objeto de una asignación.

Además en C, el hecho de que una función devuelva un valor no implica que haya que asignárselo a ninguna variable. Simplemente se descarta.

```
main()
{
   int x, y , z;

   x = 10; y = 20;
   z = mul (x,y);            /* 1 */
   printf ("%d", mul(x,y));  /* 2 */
   mul (x,y);
}
```

6.1. Las variables en una función

Pueden ser de tres tipos:

1. Locales, de carácter dinámico, que se crean al comienzo de la función y que se destruyen cuando se termina la función.
2. Globales: que existen durante la ejecución de todo el programa.
3. Estáticas: declaradas dentro de la función y mantienen su valor entre llamadas a la misma función.

6.2. Argumentos

6.2.1. Llamadas por valor y por referencia

Llamada por valor: se copia el valor de la variable en el parámetro formal. Los cambios en los parámetros no afectan a las variables originales. Esta forma de pasar los argumentos es la opción por defecto en C.

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

Llamada por referencia: se pasan como argumentos las direcciones de los parámetros reales. Los cambios realizados afectan al parátro real. De esta

forma se puede modificar el contenido de una variable mediante una función, lo cual es imposible si se realiza un paso de parámetros por valor.

```
main()
{
   int x, y;
   x = 10; y = 20;

   intercambio(&x, &y);
   printf ("x = %d, y = %d", x, y);
}

intercambio(x,y)
int *x, *y;
{
    int temp;

    temp = *x;  /* Toma el valor de la direccion x */
    *x = *y;
    *y = temp;
}
```

6.3. Llamadas de funciones con arrays o vectores

El nombre de un vector, sin índice, puede utilizarse cuando se pasa como parámetro un vector a una función. Esto tiene como efecto que no se pase todo el vector, sino sólo la dirección del primer elemento. Lo cual nos obliga a declarar ese argumento, dentro de la función como un puntero.

En el caso en que se pase un elemento de un array a una función, utilizando su índice, C se comportará como si se le pasase un elemento cualquiera.

```
Ejemplo 1:
     /* Programa que toma e imprime 10 numeros */
     main()
     {
        int t[10], i;

        for (i=0; i<10; ++i) t[i] = getnum();
        visualiz(t);
     }
```

```
visualiz(num)
int *num;
{
   int i;

   for (i=0; i<10;i++) printf ("%d", num[i]);
}
```

Ejemplo 2:

```
 /* Programa que toma 10 numeros y los imprime, uno a uno */
 main()
{
   int t[10], i;

   for (i=0; i<10; ++i) t[i] = getnum();
   for (i=0; i<10; ++i) visualiz(t[i]);
}

visualiz(num)
int num;
{
   int i;

   printf ("%d", num);
}
```

Según nuestra función vaya a modificar o no sus argumentos, utilizaremos el paso por valor o por referencia:

```
   /* Programa que lee una cadena y la pasa a mayusculas */
   main()
   {
      char s[80];

      gets(s);
      imprim_may(s);
   }

   imprim_may (cadena)
   char *cadena;
   {
     register int t;
```

```
  for (t=0; cadena[t]; t++) {
    cadena[t] = toupper(cadena[t]);
    putchar(cadena[t]);
  }
}
```

Notas:

- Cuando se pasa una variable por referencia hay que pasar su dirección, por lo tanto ha de utilizarse el operador **&**, que podría tomarse como *la dirección de ....*
- Cuando en la función se quiera acceder al contenido de la variable, ha de utilizarse en todo momento el operador *****, que podría tomarse como: *el contenido de la dirección ....*

6.4. Los argumentos argc y argv

La forma de pasar argumentos en línea de comando a un programa en C es pasarle argumentos a la función **main()**. Esta función sólo puede tener dos argumentos: **argc** y **argv**.

- **argc** especifica el número de cadenas de caracteres que se pasan como argumento. Cada una de ellas irá separada por un espacio en blanco. **argc** siempre será mayor o igual a 1, puesto que al menos existe el nombre del programa.
- **argv** es un puntero a un array de cadenas: *char *argv[];*. Los corchetes vacíos indican que la longitud de las cadenas es indeterminada a priori. La forma de acceder a cada uno de los argumentos es indexando **argv**: así *argv[0]* apunta al primer argumento, que es el nombre del programa, *argv[1]* apunta al segundo argumento, etc.

Ejemplo de programa que imprime todos los argumentos utilizados para llamarlo:

```
main(argc, argv)
int argc;
char *argv[];
{
```

```
        int t, i;

        for (t=0; t < argc; ++t) {
           i = 0;
           while (argv[t][i]) {
              putchar (argv[t][i]);
              ++i;
           }
        }
    }
```

Aunque en teoría podrían ponerse hasta 32767 argumentos, generalmente los sistemas operativos no permitirán más que unos pocos. Normalmente se utilizan para pasar opciones o bien nombres de ficheros que van a utilizarse.

6.5. Funciones que devuelven valores no enteros

Las funciones en C devuelven valores de tipo entero por defecto. Si no se especifica lo contrario, C devolverá un valor entero o convertirá el tipo del dato a devolver a entero.

Pero es posible devolver otros tipos de datos. Simplemente ha de especificarse antes del nombre de la función. De la siguiente manera:

especificador_tipo nombre_funcion ([lista_de_argumentos])
[lista_argumentos, declaración]

{ Llave abierta que indica el comienzo de la función
⋮
cuerpo de la función;
⋮
} Llave cerrada que indica el fin de la función.

Por ejemplo:

```
float fsum(x,y)
float x, y;
```

```
{
  return (x+y);
}
```

Es fundamental realizar bien la declaración de tipos que devuelve una función, puesto que C lo interpretará e interpretará el resultado en función del tipo de dato esperado y no del devuelto. Por ejemplo, si se dice que devuelve float y se devuelve un int, tomará 6 bytes además de los dos de longitud del entero. Y viceversa, si espera int y le devuelven float, sólo tomará dos bytes de los seis u ocho posibles.

En C la comprobación de tipos sólo se hace cuando se está compilando, y sólo se detectan los errores cuando las definiciones de funciones y sus llamadas están en el mismo fichero.

6.6. Devolución de punteros

Cuando se devuelven punteros a variables hay que especificarlo, puesto que los punteros pueden incrementarse y decrementarse, y cada incremento o decremento variará en función del tamaño del dato. Por ejemplo, cuando un puntero se incrementa apuntará al siguiente dato de su tipo. Y el desplazamiento variará de unos tipos a otros.

```
char *coincidencia(c, s)
char c, *s;
{
    int cuenta;
    cuenta = 0;

    while (c != s[cuenta] && s[cuenta] != '\0') cuenta++;

    return (&s[cuenta]);
}

main()
{
   char s[80], *p, ca;

   gets(s);

   ca = getchar();
```

```
    p = coincidencia(ca, s);

    if (*p != '\0') /* Hay coincidencia*/  printf ("%s", p);
    else printf ("No hay coincidencia");
}
```

6.7. Recursión

En C una función puede llamarse a si misma. Y en cada llamada los valores que tenía la función antes de la llamada se almacenan y se recuperan cuando la función vuelve de la llamada recursiva. **Una llamada recursiva no hace una copia nueva de la función, simplemente crea nuevos argumentos.**

La utilización de llamadas recursivas no aumenta la velocidad del programa. Más bien si la profundidad de la recursión es alta disminuirá. La opción de utilizar llamadas recursivas es personal, y se suelen emplear cuando la solución del problema es más natural de forma recursiva que iterativa.

Ejemplo: Cálculo del factorial de un número, de forma recursiva e iterativa.

```
fact(n) /* no recursiva */
int n;
{
  int t, respuesta;
  respuesta = 1;
  for (t=1; t < n; t++)
     respuesta *= t;
  return (respuesta);
}
```

```
factr(n) /* recursiva */
int n;
{
   if (n==1) return(1);
   return(n*factr(n-1));
}
```

Capítulo 7

Funciones de la biblioteca estándar

En C existe una librería estándar de funciones, que podemos incorporar a nuestros programas.

Aunque la versión más extendida de esta librería estándar es la que corresponde a la librería de estándar de funciones del UNIX, existen algunas funciones cuyo nombre variará de unas a otras, con lo cual es conveniente cerciorarse de cuál es la que corresponde a nuestro sistema antes de utilizarla.

Además si nosotros utilizamos con mucha frecuencia determinadas funciones propias, estas pueden incluirse en otra librería o biblioteca que nosotros definamos y pueden incluirse en los programas. Estas funciones pueden estar compiladas e incluídas en una librería o bien podemos incluirlas mediante una sentencia:

```
#include <mibiblioteca.h>
```

7.1. Entrada y Salida sobre consola

Existen funciones que realizan la Entrada y Salida, que se encuentran en las bibliotecas estándar. En C, la entrada y salida está **orientada a carácter**. Uno puede leer o escribir bytes.

7.1.1. Las funciones getchar() y putchar()

getchar(): Lee un carácter de la entrada estándar. Generalmente teclado.

putchar(): Escribe un carácter en la salida estándar. Generalmente en pantalla.

En el siguiente ejemplo aparecen otras funciones de la biblioteca estándar:

islower(ca): Devuelve 1 si ca es una minúscula. 0 en caso contrario.

isupper(ca): Devuelve 1 si ca es una mayúscula. 0 en caso contrario.

toupper(ca): Convierte el carácter ca a mayúscula.

tolower(ca): Convierte el carácter ca a minúscula.

Estas funciones no se ven afectadas ni afectan a los caracteres no alfabéticos.

```
main()  /* Cambia tipo de letra */
{
     char ch;
     do {
        ch = getchar();
        if (islower(ch)) putchar (toupper(ch));
        else putchar (tolower(ch));
     } while (ch != '.'); /* Usa un punto para terminar */
}
```

7.1.2. Las funciones gets() y puts()

gets(): Devuelve una cadena terminada con nulo en el array de caracteres que recibe como argumento. Al escribir una cadena y pulsar un retorno de carro, gets() lee la cadena y a continuación le cambia el retorno de carro por el terminador nulo.
puts(): Imprime en pantalla una cadena de caracteres, que pueden incluir los códigos que llevan asociados la barra invertida, como '\n'.

Ejemplo:

```
getnum()
{
   char num[80], n;

   do {
      gets(num);
```

```
      if (! numero(num)) {
         puts ("Debe ser un numero\n");
         n = 0;
      }
      else n = 1;
   } while (!n);
   return (atoi(num));
}
```

7.2. Entrada y Salida formateada por consola

7.2.1. Salida formateada: printf()

Su sintaxis se puede resumir:

printf (çadena de control", lista de argumentos);

La cadena de control contiene tanto caracteres como comandos de formato. De tal forma que los comandos de formato y los argumentos se aparean por orden.

Código de printf()	**Formato**
%c	Un único carácter
%d	Decimal
%e	Notación científica
%f	Coma flotante decimal
%g	Utiliza %e o %f, la que sea más corta
%o	Octal
%s	Cadena de caracteres
%u	Decimal sin signo
%x	Hexadecimal

Los códigos de control de formato pueden tener modificadores que especifiquen la anchura de campo, el número de decimales y un indicador de ajuste a la izquierda.

Ejemplo de salida formateada con printf()

```
Sentencia printf()                    Salida
```

```
("%-5.2f", 123.234)                |123.23       |
("%5.2f", 3.234)                   |         3.23|
("%10s", "hola")                   |         hola|
("%-10s", "hola")                  |hola         |
("%5.7s", "123456789")             |      1234567|
```

7.2.2. Entrada formateada: scanf()

Rutina de lectura de propósito general, que permite leer datos formateados y que puede convertir automáticamente la información numérica a enteros o a flotantes. El formato general es:

scanf (çadena de control", lista de argumentos);

La cadena de control está formada por códigos de formatos de entrada, que están precedidas por un signo

Código de scanf()	**Formato**
%c	Lee un único carácter
%d	Lee un Decimal
%e	Lee un número en coma flotante
%f	Lee un número en coma flotante decimal
%h	Lee un entero corto
%o	Lee un número en octal
%s	Lee una cadena de caracteres
%x	Lee un número en hexadecimal

Los comandos de formato pueden utilizar modificadores de longitud de campo y son números enteros colocados entre el % y el código de comando de formato.

Los datos de entrada tienen que estar separados por espacios en blanco, tabuladores o cambios de línea. Los signos de puntuación como los puntos y comas, los puntos y las comas no sirven como separadores.

Todas las variables utilizadas para recibir valores a través de scanf() se pasan por sus direcciones. Esta es la forma en la que C hace una llamada por referencia a un procedimiento.

Por ejemplo para leer un entero **cuenta**, ha de utilizarse:

```
scanf("%d",&cuenta);
```

Para leer una cadena de caracteres se puede pasar el nombre de la cadena, que sin ningún índice identifica la dirección de la cadena.

```
   scanf("%s", cadena);
   scanf("%20s", direccion);   /* Con modificador de longitud maxima */
```

Ejemplo:

```
entrar()
{
  char s[20];
  int slot;

  for (slot = 0; slot < 50; slot++) {
      if (!nombre[slot*20]) break;   /* Buscar uno libre */
  }
  if (slot == 50) {
     printf ("Lista clientes completa\n");
     return (0);
  }
  printf ("Introducir nombre y balance: \n");
  scanf ("%19s %d", &nombre[slot*20], &balance[slot]);
}
```

Capítulo 8

Punteros

La correcta utilización de punteros en C agilizará los programas. Además dado el uso extensivo que de los punteros se hace en C, es conveniente tener muy clara la sintaxis y semántica asociada a los punteros.

8.1. Punteros como direcciones

Para más referencia ver el apartado de punteros comentado con anterioridad.

Sólo recordar que un **puntero** es una variable en la que se almacena la dirección de memoria donde se encuentra otra variable, y que la forma de acceder a la dirección o al contenido de dicha variable es mediante los operadores **&** y *. Como ejemplo:

```
/* Asignacion de y = x utilizando como variable intermedia un puntero */

    float x, y, *p;

    p = &x;
    y = *p;
```

8.2. Asignaciones con punteros

Los punteros pueden formar parte de cualquier asignación, tanto en el lado izquierdo como derecho. El valor del puntero es como cualquier otro, y es una dirección en memoria. En el siguiente ejemplo, se imprime el valor de **y**, que será la dirección de **x**, no su valor.

```
int x;
unsigned y;
int *p1, p2;

p1 = &x;
p2 = p1;      /* La direccion de x a p2 */
y = p2;
printf ("%u", y);   /* imprime el valor decimal de la direccion de x,
                       ¡ no del contenido de x */
```

8.3. Expresiones con punteros

8.3.1. Aritmética de punteros

Sólo hay dos operadores aritméticos que se pueden utilizar con los punteros: son el + u el −.

Por ejemplo, si tengo un puntero **p** que es un puntero a entero, el resultado de utilizar **p**++ o **p** − − implica apuntar al entero siguiente o anterior. Por tanto, si el contenido de **p** era la dirección 2000, el resultado de hacer $p++$ no es poner en **p** 2001, sino 2002, porque si existe un entero en la posición 2000, lo normal es que ocupe dos posiciones, dos bytes, y por tanto el siguiente no puede estar en la posición 2001, sino que tiene que estar en la posición 2002. Lo mismo ocurre con la operación $p-\,-$, que pondrá en **p** 1998.

Todos los punteros se incrementarán o decrementarán según la longitud del tipo de dato al que apuntan.

También se pueden hacer sumas y restas de un valor entero con el puntero, pero nada más. El sumar o restar un valor a un puntero le indica que apunte al n-ésimo elemento del mismo tipo anterior o posterior al actual.

```
int *p;

p = p + 9; /* p apunta ahora al noveno entero que habia tras p */
```

8.3.2. Comparaciones entre punteros

Sólo tiene sentido comparar punteros cuando ambos son del mismo tipo. Podemos comparar los punteros para saber si dos elementos ocupan la misma posición de memoria, o si estamos recorriendo posiciones de memoria y llegamos a una determinada.

El siguiente ejemplo utiliza el concepto de **pila**, que será definido en clase. Simplemente comentar que las pilas son estructuras dinámicas de tipo LIFO "Last Input First Output.º sea "Último en Entrar, Primero en Salir".

El proceso de introducir un elemento en una pila es simple. Simplemente se comprueba que hay sitio en la pila, esto es no hemos llegado al tope, y si es así se introduce el elemento en la siguiente posición.

```
/* Asignacion del tope de la pila: ocupa 25 posiciones de entero, o sea,
50 posiciones caracter. Para eso se utiliza la funcion malloc */

int *p1, *cpi;
char *malloc();  /* La funcion malloc reserva x posiciones de memoria para
                    una variable de tipo caracter */
p1 = malloc(50);
cpi = p1;        /* En cpi guardamos la direccion del tope de la pila */

/* Funciones push y pop para poner y quitar un elemento de una pila */

push(i)
int i;
{
   p1++;     /* Hace sitio para colocar el nuevo entero */
   if (p1 == (cpi+25)) {
      printf ("Desbordamiento de la pila");
      exit();
   }
   *p1=i;     /* Coloca el entero i en la posicion p1 */
}


pop(i)
int i;
{
   if ((p1-1) == cpi) {
         printf ("Desbordamiento por abajo");
         exit();
   }
   p1--;   /* Coloca la cabeza de la pila en el anterior */
   return (*(p1+1));
}
```

Capítulo 9

Entrada y Salida sobre Ficheros en disco

9.1. Una forma primitiva de comunicarse con ficheros

Los operadores de reenvío del Sistema Operativo: $>$ y $<$ permiten leer datos de un fichero y escribir datos en un fichero sustituyendo a la Entrada/Salida estándar. Esto se hace en la línea de comando al ejecutar el programa, de la siguiente manera:

libros >listalib

Este programa podría pedir una lista completa de libros y almacenarlos en un fichero, que tendría por nombre *listalib*.

Existe en C una opción más potente de manejo de ficheros.

9.2. Ficheros en C

Para el programador C un fichero no es más que una porción de almacenamiento de memoria. El C representa un fichero como una estructura, de hecho el fichero **stdio.h** contiene la definción de un fichero como si fuese una estructura:

```
struct _iobuf {
   char *_ptr;       /* Puntero al buffer actual  */
   int _cnt;           /* Contador del byte actual */
```

```
    char *_base;    /* Dirección base del buffer de E/S */
    char _flag;       /* Flags de control */
     char _file;        /* Número de fichero */
};

#define FILE  struct _iobuf;     /* Notació abreviada */
```

Ideas con las que tenemos que quedarnos:

- El fichero se maneja como una estructura.
- **FILE** corresponde al patrón del fichero. Cuando se manejen ficheros, se utilizará este tipo de dato estructurado.

9.2.1. Programa sencillo de lectura de fichero

El siguiente programa lee el contenido de un fichero llamado **TEST** y lo imprime por pantalla.

```
/* Nos dice qué hay en el fichero test */
#include <stdio.h>

main()
{
   FILE *in;     /* Declara un puntero a un fichero */
   int ch;

   if ((in = fopen("test", "r")) != NULL)
   /* Abre el fichero test para lectura, comprueba si existe */
   /* El puntero FILE apunta ahora a test */
   {
       while ((ch=getc(in)) != EOF )    /* Toma caracter de in */
             putc (ch, stdout);
        fclose(in);
   }
   else
   printf ("No puedo abrir el fichero \"test\".\n");
}
```

Vamos a comentar ahora los detalles más importantes:

9.3. fopen()

Tiene tres parámetros:

- Nombre de fichero: cadena de caracteres.
- Uso del fichero: Cadena de caracteres: r", "w.º .ª".
- Puntero a fichero que devuelve la función. En el ejemplo es **in**. Este descriptor es el que se utilizará a posteriori para referirse al fichero.

No hace falta definir la funció **fopen** en el programa como una función que devuelve un puntero a fichero:

```
FILE * fopen();
```

porque ya está declarada dentro de **stdio.h**.

Una cuestió importante es que si **fopen** no consigue abrir el fichero, devuelve **NULL** que es un 0 definido en **stdio.h**. Es importante realizar la comprobación de que se ha abierto correctamente el fichero antes de proceder a su utilización.

9.4. fclose()

Cierra el fichero.

Observar que el argumento es el puntero a fichero **in** y no el descriptor del fichero.

Cuando ha ocurrido algún problema al cerrar un fichero se devuelve el valor -1, mientras que si el cierre del fichero ha sido satisfactorio devuelve 0.

9.4.1. Ficheros de texto con Buffer

Las funciones **fopen()** y **fclose()** trabajan con ficheros de texto con buffer. Con ello queremos indicar que la entrada y salida se almacenan temporalmente en un área de memoria llamada el *buffer*. Cuando el buffer se llena, el contenido se traspasa a un bloque de memoria, y se recomienza el proceso. Una de las tareas principales de **fclose()** es el "vaciado"del buffer, que podría haber quedado parcialmente lleno al cerrar el fichero.

Recordar que un fichero de texto es aquel en el que la información se almacena como caracteres, utilizando su código ASCII o similar. Mientras que

un fichero binario, es aquel en el que se almacena información binaria como puede ser un código en lenguaje máquina.

Las funciones de Entrada/Salida que vamos a describir a continuació están diseñadas únicamente para trabajar con ficheros de texto.

9.5. getc() y putc()

Funcionan de forma similar a **getchar()** y **putchar()**, simplemente que aquí se ha de especificar el fichero sobre el que han de leer o escribir un carácter.

```
ch = getchar();
ch = getc(in);

putchar(ch);
putc(ch, out);
```

En ambos casos, tanto **in** como **out** han de ser punteros a **FILE**.

En el ejemplo anterior aparecía **stdout**, que no es más que la salida estándar predefinida. De la misma manera **stdin** está definida como la entrada estándar predefinida.

9.5.1. Un programa sencillo que reduce el texto leído de un fichero

```
/* Reduce un fichero a su tercera parte */
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
    FILE *in, *out;   /* Declara dos punteros a fichero */
    int ch;
    static char nombe[20];  /* Para guardar el fichero de salida */

    int con = 0;

    if (argc < 2)  /* Comprueba que existe fichero de entrada */
        printf ("Necesito nombre de fichero como argumento\n");
    else
```

```
    {
        if ((in = fopen(argv[1], "r" )) != NULL)
        {
            strcpy (nombre, argv[1]); /* Copia el nombre del fichero en un array*/
            strcat (nombre, ".red");   /* Crea fichero de extensión reducida */

            out = fopen (nombre, "w"); /* Abre  fichero como salida de escritura */

            while ((ch = getc(in)) != EOF)
                if (cont++ % 3 == 0)
                    putc (ch, out) ;  /* Envía un carácter de cada tres */

             fclose(in);
             fclose(out);
        }
        else
           printf ("No puedo abrir el fichero %s\n", argv[1]);
    }
}
```

El resto de funciones de entrada y salida de datos tienen su equivalente para el manejo de ficheros, con la única diferencia que tienen que llevar un descriptor de fichero como parámetro adicional.

9.6. fprintf() y fscanf()

Requieren un argumento adicional para indicar el descriptor de fichero, que se convierte en el primer argumento de la lista.

```
/* Formato para usar fprintf () y fscanf() */
#include <stdio.h>

main()
{
    FILE *fi;
    int edad;

    fi = fopen("pedro", "r");    /* Apertura en modo lectura */
fscanf (fi, "%d", &edad);    /* fi apunta a pedro */
fclose (fi);

fi = fopen ("datos", "a");   /* Modo aqadir */
```

```
fprintf (fi,"pedro tiene %d\n", edad);  /* fi apunta a datos */
fclose(fi);
}
```

A diferencia de **getc()** y **putc()**, aquí el puntero es el primer argumento, mientras que allí es el último.

9.7. fgets()

Utiliza tres argumentos:

- Puntero al lugar de destino de la línea que se va a leer.
- Longitud de la tira que se está leyendo. La función se detiene cuando se lee el carácter de nueva línea o cuando se han leido MAXLIN -1 caracteres.
- Puntero al fichero en el que se está leyendo.

Una diferencia entre **gets()** y **fgets()**, es que la primera convierte el carácter retorno de carro en $'\backslash 0'$, mientras que fgets lo mantiene.

Además devuelve el símbolo NULL cuando ha leído un fin de fichero (EOF).

Ejemplo:

```
/* Lee de un fichero una linea cada vez */
#include <stdio.h>

#define MAXLIN 80

main() {
FILE *fi;

char *tira[MAXLIN];

fi = fopen("cuanto","r");
while (fgets(tira, MAXLIN, fi) != NULL)
    puts(tira);
}
```

9.8. fputs()

Simplemente añade un argumento adicional al final de fputs() que es un puntero a fichero.

```
fputs("Esto es un ejemplo", fi);
```

donde **fi** debe ser un puntero a fichero previamente abierto mediante una sentencia fopen().

La forma general de uso es:

control = fputs(puntero-a-tira, puntero-a-fichero)

donde *control* es un entero que toma el valor EOF si se encuentra una marca de fin de fichero o un error.

Al igual que **puts()** esta función no copia el $'\backslash 0'$ del final de la tira. Pero tampoco añade un caracter de nueva línea a la salida.

9.9. Acceso aleatorio con fseek()

La funcion **fseek()** permite tratar los ficheros como vectores, moviendose directamente a un byte determinado del fichero abierto por **fopen()**.

Los argumentos de **fseek()** son los siguientes:

- Puntero a fichero.
- OFFSET: indica la distancia a que debemos movernos desde el punto de comienzo. Este parametro debe declararse de tipo **long** y puede ser positivo o negativo (movimiento hacia delante o hacia atras).
- Indicador del punto de referencia para el offset.

Modo	**El offset se mide desde**
0	el comienzo del fichero
1	posicion actual
2	fin del fichero

La funcion **fseek()** devuelve el valor 0 si todo ha ido bien, y devuelve el valor -1 si ha habido algun error.

Ejemplo:

```
/* Usa fseek() para imprimir el contenido de un fichero */
#include <stdio.h>
main(numero, nombres)      /* No es obligatorio usar argc y argv */
int numero;
char *nombres[];
{
FILE *fp;
long offset = 0L;     /* 0 de tipo long */

if (numero < 2)
    puts ("Necesito un fichero como argumento\n");
else
{
    if ((fp =fopen(nombres[1], "r")) == 0)
       printf ("No puedo abrir %s", nombres[1]);
    else {
        while (fssek(fp, offset++, 0) == 0)
           putchar(getc(fp));
        fclose(fp);
    }
}
}
```

Capítulo 10

Repaso de términos

De cara a usar el manual del compilador, vamos a dar algunos términos:

Código fuente: Texto de un programa escrito de forma que un usuario pueda leerlo; es lo que normalmente denominamos *programa*.

Código objeto: Traducción a código máquina del código fuente de una programa, de forma que el computador puede leerlo y ejecutarlo.

Encadenador o linker o Enlazador: Un programa que une en un sólo programa trozos de código separados en distintos programas. Además incluye las funciones de la librería estándar de C con nuestro programa.

Biblioteca: Fichero que contiene las funciones estándar que pueden ser utilizadas por su programa. Estas funciones incluyen tanto las de E/S como otras funciones de utilidad general.

Tiempo de compilación: Los eventos que ocurren mientras su programa está siendo compilado. Un hecho típico en tiempo de compilación es el de detectar algún error sintáctico.

Tiempo de ejecución: Los eventos que ocurren mientras el progrma es ejecutado.

Capítulo 11

Cómo crear un programa en C

Pasos a seguir para crear un programa C, que será siempre compilado:

- Crear el programa fuente (con extensión .c).
- Compilarlo.
- Encadenarlo con todas aquellas funciones de librerías que sean necesarias.
- Ejecutarlo.

La creación del programa fuente ha de realizarse con un editor de texto que sólo contenga caracteres ASCII. Puede usarse cualquiera de los editores del MS-DOS como el EDIT o un editor asociado a un paquete que incluya un compilador de C, como el MicroSoft-C o el Turbo-C, o bien un editor asociado a otros paquetes, como por ejemplo el que tiene el FORTRAN, o un editor de UNIX como el VI.

Instrucciones para compilar y ejecutar un programa en C

Para el sistema operativo DOS, las instrucciones son las siguientes:

```
cc programa.c
link programa, lib.c
programa
```

En MS-DOS se puede crear un fichero de comandos por lotes o fichero batch, que podremos utilizar para realizar siempre este tipo de operaciones.

El contenido del programa que se encarga de compilar y enlazar sería:

```
cc %1.c
link %1, lib.c
%1
```

Comentar simplemente que en el caso de utilizar otro compilador de C, por ejemplo Turbo C, se tendrán que sustituir las instrucciones **cc** y **link** por las respectivas de ese compilador, por ejemplo **tcc** y **tlink** en Turbo C.

De tal manera que cuando no haya errores ni de compilación ni de ejecución, simplemente con teclear:

mi-compilador *nombre-programa*

se obtiene la ejecución del programa.

En los primeros pasos, hasta que se depure el programa, habrá que realizar varias veces dicha operación.

11.1. Un programa de ejemplo

```
main() {
  int a, b, c;

  printf("Introduzca dos numeros\n");
  a = getnum();
  b = getnum();

  c = mul(a,b);
  printf ("a * b = %d", c);
}

mul(x,y) int x, y; {
  return (x*y);
}

getnum() {
  char s[80];

  gets(s);
  return(atoi(s));
}
```

Bibliografía

[1] C. Delannoy. *El libro de C. Primer lenguaje.* Eyrolles, Barcelona, 2000.

[2] F. García, J. Carretero, J. Fernández, and A. Calderón. *El lenguaje de programación C. Diseño e Implementación de programas.* Pearson Educación, Madrid, 2002.

[3] B. Gotfried. *Programación en C.* McGraw-Hill/Interamericana de España S.A.U., Madrid, segunda edition, 1997.

[4] B.W. Kernighan and D.M. Ritchie. *The C programming language.* Prentice Hall, nueva jersey, eeuu edition, 1988.

[5] D.R. Llanos Ferraris. *Curso de C bajo UNIX.* Paraninfo, Madrid, 2001.

[6] S. Senent and R. Domínguez. *Gestión de E/S en C.* Anaya Multimedia, Madrid, 1992.