Programación en C

Autor: **J. Carlos López Ardao**

Diciembre de 2001

Índice

1. Introducción **1**

2. Estructura y fases de creación de un programa C **1**

3. Generalidades **3**
3.1. Reglas de ámbito 4
3.2. Palabras clave 6
3.3. Constantes 6
3.4. Variables en C 7
3.5. Operadores en C 8
3.5.1. Operadores aritméticos 8
3.5.2. Operadores relacionales 8
3.5.3. Operadores lógicos 9
3.5.4. Operadores a nivel de bit 9

4. Directivas del preprocesador **9**
4.1. Definición de macros 10
4.2. Inclusión de ficheros 11
4.3. Compilación condicional 12

5. Tipos de datos en C **13**
5.1. Tipos básicos 13
5.2. El tipo “dirección” (punteros) 14
5.3. Matrices 15
5.4. Relación entre punteros y matrices 16
5.4.1. Punteros a punteros. Matrices de punteros 17
5.5. El modificador `const` 18
5.6. El tipo `void` 19
5.7. Conversión de tipos 19
5.7.1. Conversión de tipos explícita: El *cast* 20

6. Estructuras de datos **21**
6.1. Estructuras 21
6.2. Uniones 23
6.3. Enumeraciones 24
6.4. Creación de tipos mediante `typedef` 25

7. Funciones en C **26**
7.1. La función main() 27
7.2. Reglas de ámbito de las funciones: Ficheros de cabecera 28
7.3. Ficheros de tipos 30
7.4. Librerías 31
7.5. Punteros a funciones 32

8. Tipos de variables. Reglas de ámbito **33**
8.1. Variables Locales 33
8.2. Variables Globales 34
8.3. Parámetros Formales 35

9. Control de flujo en C 37
9.1. Sentencia condicional: `if` . . . 38
9.2. Sentencia `switch` . . . 38
9.3. bucle `while` . . . 40
9.4. Sentencia `do-while` . . . 41
9.5. bucles infinitos con `while` y `do-while` . . . 41
9.6. Bucle `for` . . . 41
9.7. Sentencia `break` . . . 42
9.8. Sentencia `continue` . . . 43

10. Entrada/Salida en C 43
10.1. E/S por dispositivo estándar . . . 43
10.1.1. Salida formateada: Función `printf()` . . . 44
10.1.2. Entrada formateada: Función `scanf()` . . . 46
10.2. E/S de fichero . . . 48

11. Asignación dinámica de memoria 50

12. Otras funciones de la librería estándar 51
12.1. Manejo de cadenas . . . 51
12.2. Funciones de caracteres . . . 54
12.3. Utilidades generales . . . 54
12.4. La librería matemática . . . 55
12.5. Señales . . . 56
12.5.1. Manejo de señales: Instalación de un *handler* . . . 57
12.5.2. Posibles problemas con señales . . . 58
12.5.3. Generación de señales . . . 59
12.5.4. Temporizadores . . . 59
12.6. Fecha y hora . . . 61
12.7. Funciones útiles en la red . . . 62

13. Herramientas útiles para la programación en C 63
13.1. El editor `emacs` . . . 63
13.2. Compilación con `make` . . . 65
13.3. El manual en línea: `man` . . . 67
13.4. El depurador: `ddd` . . . 68

14. Comandos Unix/Linux más importantes 68

1. Introducción

El lenguaje C se conoce como un lenguaje de medio nivel, pues podríamos situarlo entre los lenguajes de bajo nivel o de máquina (ensamblador) y los de alto nivel como el PASCAL, por ejemplo.

Un lenguaje de medio nivel nos ofrece un conjunto básico de sentencias de control y de manipulación de datos que nos permitirá construir sentencias y estructuras de nivel más alto.

En la actualidad existen numerosos compiladores de C, todos ellos con sus peculiaridades, sin embargo, prácticamente todos son compatibles con el C normalizado por ANSI, el **ANSI C**, en el que nos centraremos partir de ahora.

Entre las características de C podemos citar:

- 32 palabras clave (BASIC 128; TPASCAL 48).
- Eficiencia de ejecución del código generado (entre los lenguajes de bajo y alto nivel).
- Portabilidad.
- No impone tantas restricciones como los lenguajes de alto nivel, dando más libertad al programador.

C es un lenguaje estructurado como PASCAL, y como tal, un programa en C cumple todas las características de la programación estructurada. De hecho, en C todo el código se estructura en funciones totalmente independientes entre sí. Incluso el programa principal se incluye en una función especial denominada `main()`.

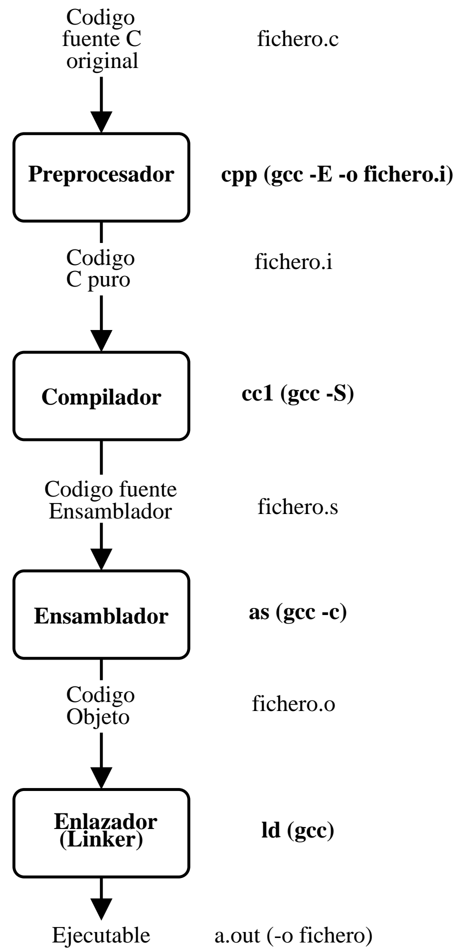
A pesar de ello, todas las ventajas enumeradas lo convierten en un lenguaje que requiere mayor experiencia del programador, puesto que los compiladores detectan muchos menos errores y dan mucha más libertad al programador que los lenguajes de alto nivel, fundamentalmente en lo que se refiere a la construcción y manipulación de estructuras de datos.

2. Estructura y fases de creación de un programa C

Un programa C está formado exclusivamente por **funciones** independientes. A diferencia de lenguajes como PASCAL, en C no se puede crear una función dentro de otra función.

El código de un programa C suele distribuirse entre varios ficheros fuente (con extensión `.c`), denominados **módulos**. Así, habitualmente un programa de cierto tamaño constará de un módulo que contiene el programa principal (ubicado en una función especial denominada `main()`), y de otros módulos con el código de diversas funciones agrupadas según algún criterio, o con definiciones de tipos utilizados. Además, todos los compiladores ponen a disposición del programador una enorme cantidad de funciones, accesibles por medio de librerías (ver sección 7.4). De hecho, si el programa es suficientemente pequeño puede constar de un sólo módulo, el principal, que haga uso de las librerías.

En la Figura se muestran las 4 fases en la creación de un programa ejecutable desarrollado en el lenguaje C.



- **Preprocesado**: El código C desarrollado es enviado al preprocesador que interpreta y ejecuta todas las directivas de preprocesado (ver sección 4). De esta tarea se encarga el programa **cpp**, que también puede ser invocado con la opción de parada **-E** del compilador (indicando adicionalmente `-o fichero.i`).

- **Compilación**: El código C generado por el preprocesador es analizado sintáctica, léxica y semánticamente por el compilador. Si el análisis es correcto, se genera el correspondiente código en lenguaje ensamblador. Adicionalmente, el compilador suele realizar un importante número de optimizaciones. Esta tarea es realizada por el programa **cc1**, que también puede ser invocado con la opción de parada **-S** del compilador (por defecto se genera `fichero.s`, no siendo necesaria la opción `-o`).

- **Ensamblado**: El código en ensamblador es transformado en código binario (objeto), tarea que realiza el programa **as**, que también puede ser invocado con la opción de parada **-c** del compilador (por defecto se genera `fichero.o`, no siendo necesaria la opción `-o`).

- **Enlazado**: Es la operación final consistente en agrupar los ficheros obje-

to de todos los módulos y librerías, y editar los enlaces para generar el ejecutable. Esta compleja tarea es realizada por el programa **ld**, que es invocado finalmente por el compilador, por lo que basta con no usar ninguna opción de parada. Habitualmente se usa la opción **-o** para indicar el nombre del fichero ejecutable, que por defecto es `a.out`.

Normalmente, dado el elevado número de parámetros que requieren los distintos programas en cada fase, nunca se utilizan individualmente, sino a través del compilador que se encarga de invocarlos. Así, las fases invocadas por el compilador dependerán de la extensión del fichero o ficheros pasados al compilador, y de la opción de parada. Por ejemplo, suponiendo que usemos el compilador **gcc** o **g++** de GNU:
`> gcc -E ejemplo.c -o ejemplo.i` implementa sólo el preprocesado, generando el fichero `ejemplo.i`
`> gcc -c ejemplo.c` implementa las tres primeras fases y genera el fichero objeto `ejemplo.o`
`> gcc -S ejemplo.i` implementa sólo la compilación y genera el fichero en ensamblador `ejemplo.s`
`> gcc ejemplo.c -o ejemplo` implementa el proceso completo y genera el ejecutable `ejemplo`
`> gcc ejemplo.s -o ejemplo` implementa las dos últimas fases y genera el ejecutable `ejemplo`

Otras opciones interesantes de estos compiladores son las siguientes:

-g: Inserta símbolos para poder usar el depurador (ver sección 13.4).

-Dnombre_macro: Define una macro, de forma equivalente a `#define nombre_macro` (ver sección 4.1. Resulta muy útil para compilación condicional cuando se combina con directivas `#ifdef (#ifndef) macro ... #endif` (ver sección 4.3).

-Dmacro=valor: Define una macro y le asigna un valor (ver sección 4.1).

-Idirectorio: Lista de directorios donde **cpp** debe buscar ficheros de cabecera (ver secciones 7.2 y 7.4).

-Ldirectorio: Lista de directorios donde **ld** debe buscar las librerías (ver sección 7.4).

-lnombre: Indica a **ld** que enlace la librería `libnombre` (extensiones `.a` para estáticas y `.so` para dinámicas). Ver sección 7.4.

-On: Indica el nivel de optimización desde `n` 0 (no optimización) hasta 6 (nivel máximo).

-Wall: Indica al compilador que muestre todos los mensajes de alerta (*warnings*).

3. Generalidades

- Toda sentencia simple debe terminar en punto y coma.

- Toda sentencia compuesta (bloque de sentencias simples) irá entre llaves, '`{`' al principio, y '`}`' al final (frente al BEGIN – END del Pascal).
- El símbolo de asignación es '`=`' (frente al '`:=`' del Pascal).
- Los comentarios empiezan por '`/*`' y terminan en '`*/`' (frente a '`(*`' y '`*)`' en Pascal). Algunos compiladores (como el `g++`) también ignorarán cualquier línea que comience con el símbolo '`//`'.
- Las sentencias del tipo :

  variable = variable **operador** expresión;

  se pueden escribir de la forma:

  variable **operador** = expresión;

  Ejemplo: `x = x − 100;` equivale a `x −= 100;`

3.1. Reglas de ámbito

Como veremos muy a menudo a lo largo de este documento, todo programa C se construye básicamente mediante tres tipos de objetos: funciones, variables y tipos de datos. Todos ellos deben poseer un identificador unívoco y ser definidos antes de su utilización.

Si bien el inicio del ámbito de uso de un objeto queda determinado por la declaración, la finalización de éste se determina atendiendo a las denominadas **reglas de ámbito**, que adicionalmente deben resolver toda ambigüedad que se pudiese plantear.

Si entendemos por bloque un conjunto de sentencias encerrado entre llaves, es decir, una función o una sentencia compuesta, según las reglas de ámbito del lenguaje C, el ámbito de uso de un objeto se restringe exclusivamente al bloque donde es definido. Pero si se halla definido en un módulo (fichero fuente), fuera de cualquier bloque, su ámbito será todo el módulo, es decir podrá ser usado en cualquier bloque del módulo posteriormente a su definición. En cambio, no podrá ser usado en otros módulos salvo que se importe en éstos: en el caso de las variables, mediante una declaración usando la palabra clave `extern` (ver sección 8), y en el caso de las funciones, mediante la declaración de la cabecera de la función (ver sección 7.2).

Desafortunadamente, en C no se ha previsto una forma de importar definiciones de tipos, por lo que su ámbito es, a lo sumo, un módulo. Ello podría plantear problemas de consistencia. En la sección 7.3 se plantea una solución sencilla a este problema.

Las reglas de ámbito deben resolver también la ambigüedad que se plantea cuando el ámbito de un objeto está incluido en un ámbito mayor de otro objeto que usa el mismo identificador, como ocurre, por ejemplo, cuando una variable local de una función y una global en el mismo módulo poseen el mismo nombre. Esta situación se denomina **solapamiento de ámbitos**.

Obviamente, no se produce solapamiento si el ámbito de ambos objetos es el mismo, como ocurre siempre entre las funciones, ya que siempre son globales al módulo. En estos casos, se produciría simplemente un error de compilación por hallarse duplicado el identificador.

En caso de solapamiento de ámbitos, las regla general determina que el objeto de ámbito mayor no es visible en el ámbito menor, es decir, son dos objetos distintos. Tal regla resulta de gran importancia de cara a la programación estructurada, porque permite el diseño de funciones con total y absoluta independencia, no importando los nombres usados para funciones, variables locales, parámetros formales o tipos de datos. Veamos unos ejemplos.

En el siguiente caso, la variable `a` se declara en cinco lugares dentro de un mismo módulo: como variable global al módulo, como parámetro formal de `funcion1()` (es una variable local más), como variable local en `funcion2()`, como variable local en una sentencia `if` de `funcion2()` y como variable local en una sentencia `if` del programa principal. En este ejemplo, la variable global al módulo, declarada fuera de todo bloque, no existe dentro de ambas funciones ni dentro de la sentencia `if` del programa principal, por haberse declarado variables locales con el mismo identificador. Igualmente, la variable local a `funcion2()` no existe dentro del `if`, donde se ha declarado otra variable con el mismo nombre.

En el mismo ejemplo, podemos un caso donde se produce solapamiento entre la función `funcion1()` y la variable local de igual nombre definida en `funcion2()`. De esta forma, la función `funcion1()` no puede ser usada en `funcion2()` como normalmente podría hacerse si no existiese dicha definición de la variable local.

```
int a; /* a puede ser usada en todo el módulo pero */
       /* no es visible en funcion1 y funcion2 */

int funcion1(int a) /* a es local a funcion1 */
{
  ...
  a = 5;
  ...
}
char funcion2(char b) /* a es local a funcion2 */
{
  float a = 7.5;
  float funcion1; /* Es una variable local que oscurece a la
                     función funcion1() */
  if (a)      /* Se refiere a la variable local a funcion2() */
  {
    char a = -2;  /* Variable local al if */
    printf("a=%d\n", a); /* Imprime -2 */
  }
  printf("a=%f\n", a); /* Imprime 7.5 */
  ...
}

main()
{
  ...
  a = 25;
  ...
  if (a)  /* Se refiere a la variable global al módulo */
```

```
  {
    float a = 0;   /* Variable local al if */
    printf("a=%f\n", a); /* Imprime 0 */
  }
  printf("a=%d\n", a); /* Imprime 25 */
}
```

3.2. Palabras clave

El lenguaje **ANSI C** está formado por 32 palabras clave que no pueden ser utilizadas como nombres de variable ni de función.

La lista de palabras clave es:

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

Las palabras clave deben escribirse siempre en minúsculas. En C las mayúsculas y las minúsculas son diferentes: `else` es una palabra clave; `ELSE` no. Las funciones `exit()` y `EXIT()` son distintas.

3.3. Constantes

Una constante entera puede ser decimal, octal (comienza con `0`, dígitos `0-7`) o hexadecimal (comienza con `0X` ó `0x`, dígitos `0-9` y `a-f` ó `A-F`). Si al final de la constante se añade la letra `l` ó `L`, se indica explícitamente que ésta es `long`, es decir, de 4 bytes (en UNIX/LINUX no sería necesario, pues lo enteros son de 4 bytes por defecto).

Ejemplos:

```
124   -2   /* Decimal */
-7L   5l   /* Decimal long */
7a         /* Error de compilación: a no es un dígito decimal */
0377  01   /* Octal */
01    05L  /* Octal long */
086        /* Error de compilación: 8 no es un dígito octal */
0xAf  0X1e /* Hexadecimal */
0xffffl    /* Hexadecimal long */
0X0g       /* Error de compilación: g no es un dígito
              hexadecimal */
```

Las constantes reales se pueden representar de dos formas:

- Mediante el punto decimal, como en

```
-2.3    12.0 ó 12.    0.5 ó .5
```

Nótese la necesidad de indicar la parte fraccionaria como 0 para distinguir de la constante entera `12`.

- Mediante la notación científica, donde la letra `E` ó `e` separa la mantisa del exponente (en base 10):

```
1E2 (=100.0)   12.3e-3 (=0.0123)  -2.0E3 (=-2000.0)
```

Las constantes reales son siempre `double` (8 bytes). Si se desea otra precisión, debe añadirse al final de la constante una `f` ó `F` para `float` (4 bytes), o bien una `l` ó `L` para `long double` (12 bytes).

Una constante carácter es un único carácter entre comillas simples, como en `'a'`, que se traducirá como el correspondiente código ASCII. También se pueden utilizar secuencias de escape, que comienzan con el símbolo \ y van seguidas de:

- uno, dos o tres dígitos octales: Ej. `‘\130’`
- la letra `x` y uno o dos dígitos hexadecimales: Ej. `‘\xf5’`
- otro símbolo con significado especial: Ejs. `‘\n’` (nueva línea), `‘\a’` (timbre), `‘\t’` (tabulador horizontal), `‘\v’` (tabulador vertical), etc.

La constante cadena (*string*) es una secuencia de caracteres entre comillas dobles:

```
"Hola"
"Hola\n\a" /* Incluye un cambio de línea y una alarma */
""         /* Cadena vacía */
```

Cabe destacar que las cadenas son almacenadas añadiendo al final de la cadena un `0`, denominado carácter nulo o de fin de cadena, y que se representa como `‘\0’`.

3.4. Variables en C

Los nombres de las variables en C pueden contener cualquier número de letras (se incluye el símbolo de subrayado, `‘_’`) o números, pero el primer carácter ha de ser, necesariamente, una letra (o el símbolo de subrayado).

Según las reglas de ámbito del lenguaje C (ver sección 3.1), todo objeto (variable, función, tipo, etc.) debe existir (ser declarado) antes de su uso. Las reglas de ámbito de las variables se tratan en la sección 8.

La declaración se hará indicando primero el tipo de dato, luego el nombre de la variable, y al final un punto y coma. También se pueden declarar varias variables del mismo tipo en una misma declaración mediante una lista de nombres separados por comas.

Como veremos más detalladamente en la sección 5.1, en C existen cinco tipos básicos de datos: dos para datos enteros (`char` (1 byte) e `int` (2 ó 4 bytes)), y tres para reales (`float` (4 bytes), `double` (8 bytes) y `long double` (12 bytes)).

Importante: Debe tenerse en cuenta que habitualmente en C todas las variables contienen valores indeterminados tras su declaración, por lo que se recomienda su inicialización, que puede ser realizada en la misma declaración.

Por prudencia, en el caso de reales, aún cuando la constante no tenga parte real, debe indicarse.

Ejemplos:

```
int a=-2, i;
float b=2.0, c=-2.1;
```

3.5. Operadores en C

3.5.1. Operadores aritméticos

OPERADOR	ACCIÓN
-	resta, menos unario
+	suma
*	multiplicación
/	división
%	módulo
--	decremento
++	incremento

Cabe destacar que la división entre enteros da como resultado la división entera.

Ejemplos:

```
int x=5, y=2, z;
float f=2.0, g;
z=x-y;   /* z=3   */
z=x*y;   /* z=10  */
g=x/y;   /* g=2.0 */
g=x/f;   /* g=2.5 */
g=x/4.0; /* g=1.25 */
g=x/4;   /* g=1.0 */
z=x%y;   /* z=1   */
x++;     /* x=6   */
++y;     /* y=3   */
z--;     /* z=0   */

x=5; y=++x; /* y=6, x=6 */ /* Incremento y luego asignación */
x=5; y=x++; /* y=5, x=6 */ /* Asignación y luego incremento */
```

3.5.2. Operadores relacionales

OPERADOR	ACCIÓN
>	mayor
>=	mayor o igual
<	menor
<=	menor o igual
==	igual
!=	distinto

3.5.3. Operadores lógicos

OPERADOR	ACCIÓN
`&&`	and
`||`	or
`!`	not

En C, El valor FALSE está asociado al "0". Por tanto, una expresión o valor de cualquier tipo (real, entero o puntero) distinto de "0" se interpreta como TRUE en una expresión lógica. No existen como tal los identificadores TRUE y FALSE, ni tampoco el tipo `boolean`.

Utilizando operadores relacionales y lógicos se puede construir cualquier expresión lógica (condicional).

Ejemplos:

```
if ((a >= b) & !p) ...;
while ((a != b) & (a < c)) ...;
```

3.5.4. Operadores a nivel de bit

Los operadores siguientes pueden utilizarse en operaciones bit a bit con cualquier tipo entero de datos (`char, int, long int, ...`), pero no con tipos reales (`double` y `float`). En la siguiente sección se definen los tipos aquí citados.

OPERADOR	ACCIÓN
`&`	and
`|`	or
`^`	or exclusivo
`~`	complemento a dos
`>>`	desplaz. a derecha
`<<`	desplaz. a izquierda

Ejemplos:

```
char a=5, b;
char mask=0x01;   /* Máscara del bit menos significativo */

b = a & mask;     /* b = 1  */
b = a << 2;       /* b = 20 */
b = ~a;           /* b = -6 */
```

4. Directivas del preprocesador

La potencia y notación del lenguaje C pueden ser extendidas mediante el uso de un **preprocesador**. Este preprocesador trabaja asociado con el compilador y se usa para reconocer sentencias especiales, denominadas **directivas del preprocesador**, que se hallan incluidas en un programa C. Las facilidades ofrecidas por el preprocesador permiten desarrollar programas más fáciles de leer, de modificar y de transferir a otros sistemas.

Como su nombre indica, y tal como se vio en la sección 2, el preprocesador analiza las directivas y modifica el fichero fuente previamente a la fase de compilación.

Estas directivas ocupan una línea que debe comenzar obligatoriamente por el símbolo `'#'`. La sintaxis de estas directivas es totalmente independiente del lenguaje C. Se resumen a continuación las más utilizadas.

4.1. Definición de macros

Una macro es una cadena de caracteres identificada por un nombre, que será colocada en el fichero fuente allá donde el nombre aparezca. Para ello se utiliza la directiva

```
#define identificador macro
```

Ejemplos:

```
#define BUFFER_SIZE 1024
#define CAPACIDAD 4 * BUFFER_SIZE
```

En el ejemplo, cualquier ocurrencia del identificador `BUFFER_SIZE` será sustituida por la cadena `1024`. Así, podemos definir una constante habitualmente utilizada de forma que, si necesita ser cambiada, bastará con hacerlo en su definición. Además, nos permite asignarle un identificador acorde con su significado. La sustitución se hará incluso en la definición de otras macros. En el ejemplo anterior, el preprocesador sustituirá `CAPACIDAD` por `4 * 1024`.

El único lugar donde no se realizará sustitución alguna es dentro de constantes de tipo cadena (encerradas entre comillas dobles, "..."), como por ejemplo en:

```
printf("El valor de CAPACIDAD es %d bytes\n", cap);
char cadena[] = "La CAPACIDAD es muy baja";
```

Las macros también pueden recibir parámetros, que deben indicarse entre paréntesis junto al identificador. Por ejemplo

```
#define SWAP(x, y)       {int temp = x; x = y; y = temp;}
#define ES_BISIESTO(z)   (z%4 == 0) && (z%100 != 0) || \
                            (z%400 == 0)
#define LEE_ENTERO(a)    scanf("%d\n" &a);
...
int año, prox_bisiesto;
LEE_ENTERO(año);
if (ES_BISIESTO(año))
  SWAP(año, prox_bisiesto);
```

No obstante, debe ponerse cuidado al utilizar macros pues, debido a las reglas de precedencia, la expansión podría no dar el resultado esperado. Basta ver que la macro

```
#define CUADRADO(x)   x * x
```

en la sentencia

```
y = CUADRADO(a+1)
```

será sustituida por `y = a+1*a+1` ⟹ `y = a + (1*a) + 1`
debido a la precedencia de '*' frente a '+'. Así, por prudencia, una definición que pudiese dar problemas de este tipo debería usar paréntesis para cada parámetro y para la macro completa:

```
#define CUADRADO(x) ((x) * (x))
```

Con los parámetros de una macro pueden utilizarse dos operadores especiales:

- El operador cadena (#) se utiliza precediendo a un parámetro e indica al preprocesador que encierre entre comillas dobles el argumento correspondiente.
- El operador concatenación (##) se utiliza entre dos parámetros e indica al preprocesador que concatene sin espacios los dos argumentos correspondientes.

Ejemplos:

```
#define DEBUG(a)            printf(#a " = %f\n", a);
#define FUNCION(nombre,x)   func_##nombre(x);
```

`DEBUG(x+y);` es expandido a `printf("x+y" " = %f\n", x+y);`
`funcion(suma,z);` es expandido a `func_suma(z);`

Finalmente, debe tenerse en cuenta que el compilador recibe un fichero procesado previamente por el preprocesador y que, por tanto, no contiene directivas de compilación. Por ejemplo, si un programa contiene la directiva

```
#define a 5
```

el preprocesador sustituirá todo identificador `a` por `5`, y así una declaración

```
int a;
```

equivaldrá a

```
int 5;
```

dando lugar posteriormente a que el compilador genere un error de compilación. aunque puede no tener mucha importancia pues, en general, son errores fácilmente subsanables, para evitarlos suelen reservarse los identificadores que sólo usan mayúsculas para los nombres de macros.

4.2. Inclusión de ficheros

Para incluir el contenido de un fichero de texto en otro se usa la directiva

```
#include ''fichero''
#include <fichero>
```

El preprocesador incluirá el fichero indicado en el fichero actual, siendo procesado como si apareciera en el lugar de la directiva `include`. En la primera forma (con comillas dobles), debe darse el camino completo del fichero (por defecto, en el directorio actual). En la segunda forma (con <...>), el preprocesador buscará en el directorio por defecto, que en UNIX/LINUX es `/usr/include`, aunque

pueden indicarse más directorios por defecto con la opción `-idirectorio` del compilador, como vimos en la sección 2.

Un fichero incluido puede, a su vez, contener otras directivas `#include`. La profundidad de anidamiento es dependiente de la implementación.

Como veremos en las secciones 7.2 y 7.4, esta directiva es particularmente Útil para incluir ficheros que contienen declaraciones de funciones, tipos, macros, etc.

4.3. Compilación condicional

El preprocesador de C permite incorporar o eliminar sentencias de un programa fuente antes de su compilación, facilidad de gran utilidad denominada **compilación condicional**. Para ello se usan las directivas

```
#if expresión-constante
 ...
#else
...
#endif
```

El preprocesador evaluará la `expresión-constante` en tiempo de compilación. por tanto, Ésta sólo puede contener constantes y operadores C. Si su valor es `0` (falso lógico), el primer conjunto de líneas es descartada y el segundo es pasado al compilador. Si fuese distinto de cero, se pasaría sólo el primer conjunto de líneas. la parte `#else` de la directiva puede no existir. También pueden anidarse directivas `#if` mediante la directiva `#elif` (abreviación de `else if`).

El conjunto de líneas puede incluir cualquier número de líneas con declaraciones, sentencias, otras directivas, etc.

Entre las muchas posibilidades de aplicación, podría destacarse la fase de depuración de un programa, donde se podría incluir y eliminar sentencias de seguimiento, según se desee.

```
#define debug_1 1
#define debug_2 0
...
#if debug_1
fprintf(stderr, "x = %f\n", x);
...
#elif debug_2
fprintf(stderr, "x = %f\ty = %d\n", x, y);
...
#endif
```

Si se desea compilar las sentencias de `debug_2`, debe definirse éste con un valor distinto de cero, y poner `debug_1` a cero. Habitualmente, en vez de las dos primeras directivas, suele controlarse la compilación cuando ésta se realiza mediante la opción `-d` del compilador (ver sección 2):

```
> gcc -ddebug_1=1 ...}
```

o

```
> gcc -ddebug_2=1 ...
```

No obstante, prefiere utilizarse la directiva alternativa `#ifdef` (o `#ifndef`), que se evalúa a TRUE simplemente si el identificador que le sigue está (no está) definido, es decir, es conocido por el preprocesador, aunque no se le haya asignado valor alguno. Así, el ejemplo anterior sería:

```
#define debug_1
...
#ifdef debug_1
fprintf(stderr, "x = %f\n", x);
...
#endif
#ifdef debug_2
fprintf(stderr, "x = %f\ty = %d\n", x, y);
...
#endif
```

bastando con cambiar la primera directiva por `#define debug_2` si se desea compilar la segunda parte y no la primera. También puede controlarse la compilación cuando se realiza ésta mediante la opción `-d` del compilador:

```
> gcc -ddebug_1 ...
```

o

```
> gcc -ddebug_2 ...
```

Dado que el preprocesador no permite redefinir una macro a no ser que la definición sea idéntica, esta directiva es también útil cuando se desea asignar un valor distinto a una macro que pudiese estar ya definida en alguno de los ficheros de cabecera incluidos. para ello, debe eliminarse previamente la definición anterior con la directiva `#undef`.

```
#ifdef PI
#undef PI
#endif
#define PI 3.141592
```

o también, si se desea definir por si no lo estuviera aún:

```
#ifndef PI
#define PI 3.141592
#endif
```

5. Tipos de datos en C

5.1. Tipos básicos

En C existen cinco tipos básicos de datos: dos para datos enteros (`char` e `int`), y tres para reales (`float`, `double` y `long double`). Con los tipos enteros se pueden usar además los modificadores de tamaño (`short, long`), y/o los modificadores de signo (`signed, unsigned`).

El número de bits que ocupa cada tipo en memoria depende de la máquina y compilador con que se trabaje. Actualmente, sin embargo, hay un acuerdo

bastante extendido y las únicas diferencias se hallan en el tipo `int`, que mientras en DOS se toma `signed short` por defecto, en los compiladores para UNIX/LINUX se toma `signed long`. En todos los demás casos, el modificador por defecto para el signo es `signed`. También se puede usar sólo los modificadores de signo y/o tamaño, tomándose en este caso el tipo `int` por defecto.

Las posibles combinaciones de tipos básicos y modificadores son:

TIPO	BYTES	RANGO
char	1	-128 a 127
unsigned char	1	0 a 255
short int (int en dos)	2	-32768 a 32767
unsigned short int	2	0 a 65535
long int (int en unix/linux)	4	-2147483648 a 2147483647
unsigned long int	4	0 a 4294967295
float	4	3.4e-38 a 3.4e+38 (con signo)
double	8	1.7e-308 a 1.7e+308 (con signo)
long double	12	3.4e-4932 a 3.4e+4932 (con signo)

5.2. El tipo "dirección" (punteros)

En C hay un tipo básico de datos adicional, aunque un poco especial, que es el tipo "dirección de memoria" y que ocupa 4 bytes en memoria. Las constantes y variables de este tipo se utilizan como direccionamiento indirecto de otros datos, por lo que habitualmente se dice que son "punteros" a dichos datos.

En C hay dos operadores especiales para el manejo de punteros: `'&'` y `'*'`. Estos operadores coinciden con los ya definidos para **and** y **multiplicación**, pero se diferencian de éstos porque se anteponen **sin espacios** al operando, que debe ser el nombre de una variable o función.

Así, el operador `'&'` se interpreta como **dirección de memoria**, mientras que el operador `'*'` se interpreta como **contenido de la dirección**.

En la declaración de una variable puntero debe indicarse también el tipo de los datos apuntados (los direccionados indirectamente), de la siguiente forma:

tipo_datos_apuntados *nombre_variable_puntero

Por ejemplo, la siguiente declaración indica que la variable `p` es una dirección donde se almacenan datos de tipo `int` o, como se dice habitualmente, `p` es un puntero a un `int`.

```
int *p;
```

Supongamos, en el siguiente ejemplo, que la variable `nu` ocupa la posición de memoria 1000 y su valor es 20 (es decir, el contenido de la dirección 1000 es 20).

```
int nu ,q ,*m;

m = &nu; /* almacenamos en m la dirección de nu, i.e., 1000 */
q = *m;  /* contenido de la dirección m (1000) ==> q = 20   */
```

Con los punteros se pueden utilizar los operadores aritméticos `+` y `-`, y por tanto también `++` y `--`. Siendo `apuntador` un puntero, consideremos las sentencias siguientes:

```
apuntador++; /* incrementamos en una unidad la dirección */
apuntador--; /* decrementamos en una unidad la dirección */
```

Importante: sin embargo, el valor de “**una unidad**” depende del tipo al que apunte `apuntador`. Supongamos, por ejemplo, que `apuntador` tiene el valor 1000. Si la variable apuntada es de tipo `char`, la sentencia

```
apuntador = apuntador + 9;
```

hará que el nuevo valor de apuntador sea 1009, pues el tipo `char` ocupa un byte. Pero, en cambio, si la variable apuntada fuese de tipo `int`, dicha sentencia haría que el nuevo valor de apuntador fuese 1036, pues el tipo `int` ocupa 4 bytes.

Un valor especial de los punteros es el `‘0’` (o NULL, como se halla definido en `<stdio.h>`), que indica un puntero nulo, es decir, que no apunta a nada. El uso de este valor es de gran utilidad cuando se manejan listas enlazadas de estructuras para indicar un elemento inicial o final de la lista (ver sección 6.1).

Debe ponerse especial cuidado en no acceder al contenido de un puntero nulo, pues se traducirá en un error muy grave durante la ejecución del programa que hará que éste se termine, indicando un fallo de segmentación de memoria (*segmentation fault*).

Finalmente, debemos destacar una aplicación fundamental de los punteros, el paso de parámetros por referencia a una función, aunque este tema será tratado en la sección 8.3.

5.3. Matrices

Las matrices (o *arrays*) pueden ser de cualquier tipo de variable. su definición general es:

tipo nombre_matriz[n];

donde n es el número de elementos que contendrá la matriz.

Importante: en C, si bien los índices de la matriz van de **0 a n-1**, el compilador no chequea la violación de los límites de la matriz, hecho que tendría consecuencias desastrosas, en general, por lo que el programador debe poner un especial cuidado de no violar tal límite.

Las matrices también pueden ser inicializadas en su declaración con una simple enumeración entre llaves. en este caso no es necesario indicar el tamaño de la matriz.

```
float lista[3] = { 1.0, -3.4, 2.1 };
```

o también

```
float lista[] = { 1.0, -3.4, 2.1 };
```

En C, una cadena (*string*) es simplemente una matriz de caracteres.

Para inicializar una cadena puede hacerse de forma completa, o carácter a carácter.

```
char cadena[5] = "hola";
char cadena[] = {‘h‘,‘o‘,‘l‘,‘a‘,‘\0‘};
```

Recuérdese que `‘\0’` es el carácter nulo que determina el fin de la cadena. si la cadena se construye carácter a carácter, hemos de introducirlo explícitamente. Si la cadena se construye mediante comillas dobles (“ ”), entonces es el compilador quien lo introduce automáticamente. En ambos casos tampoco sería necesario indicar el tamaño.

Debido al carácter de fin de cadena, habrá que definir las cadenas de un elemento más que la longitud máxima prevista.

Ejemplo:

```
main() {
  char cadena[60];       /* cadena de caracteres       */
  gets(cadena);     /* función de librería que lee la cadena */
  printf("%s",cadena);   /* imprimimos la cadena en pantalla,
                            desde cadena[0] hasta el que se
                            encuentre el carácter nulo */
  printf("%c",cadena[1]); /* imprimimos el segundo carácter */
}
```

Una matriz bidimensional se define de la forma:

tipo nombre_matriz[num_filas][num_columnas];

pudiendo también inicializarse en la propia declaración:

```
int numeros[2][3] = {{1,2,3}, {-1,0,5}}
```

También se pueden definir matrices n-dimensionales, estando el número **n** limitado únicamente por el propio compilador:

tipo nombre$[a_1][a_2]...[a_n]$**;**

5.4. Relación entre punteros y matrices

En C existe una estrecha relación entre las matrices y los punteros.

Cuando definimos una matriz, el propio nombre de la matriz es la dirección de memoria a partir de la cual se almacenan los elementos de la matriz. Es decir, el nombre de la matriz es una constante puntero (que no puede ser modificada) al primer elemento de la matriz. Por ejemplo, si tenemos:

```
char cadena[100];
```

el nombre de la matriz, `cadena`, es la dirección del primer carácter, `cadena[0]`:

```
cadena
```

equivale a

```
&cadena[0]
```

Por tanto, para acceder a un elemento cualquiera de la cadena, además de mediante la forma tradicional, podríamos hacerlo mediante el operador `*`. Es decir,

```
cadena[5]
```

equivale a

```
*(cadena+5)
```

Cabe destacar también que la definición de la matriz implica la asignación, en tiempo de compilación, de espacio contiguo en memoria para almacenar la matriz, como si hubiésemos definido 100 variables de tipo `char`. En cambio, si hubiésemos hecho

```
char *cadena;
```

tan sólo estaríamos declarando una variable puntero a carácter, que puede ser modificada, y que no implica reserva alguna de memoria. a este respecto, conviene ver la subsección 11.

También, como veremos en la sección 8.3, el paso de una matriz como argumento a una función equivale exactamente al paso por valor de la dirección de su primer elemento. En ningún caso se pasa la matriz entera.

Finalmente, debe ponerse especial cuidado cuando se comparan o copian cadenas, pues aunque intuitivamente parecería lógico hacerlo de la siguiente forma:

```
char cad1[] = "hola", cad2[10];

if (cad1 == cad2) /* Siempre falso pues las direcciones */
                  /* cad1 y cad2 son distintas          */

cad1 = cad2 /* Daría un error de compilación pues no se puede */
            /* modificar la dirección de inicio de la cadena  */
```

debemos tener en cuenta que en realidad son simples direcciones de memoria. Para realizar las tareas anteriores debe procederse como si se comparasen o copiasen bloques de memoria, haciendo uso de funciones de la librería estándar como `strcmp()`, `strcpy()`, `memcmp()`, `memcpy()` (en `<string.h>`), o `sprintf()` (en `<stdio.h>`). Estas funciones serán descritas brevemente en la sección 12.1.

5.4.1. Punteros a punteros. Matrices de punteros

Dado que una variable puntero puede contener la dirección de (apuntar a) cualquier variable, también puede contener la dirección de otra variable puntero, en cuyo caso tendríamos un puntero a puntero.

Por ejemplo, en

```
char **punt;
```

definimos una variable puntero a un puntero a `char`.

Dada la relación entre punteros y matrices, la forma más intuitiva de interpretar el ejemplo anterior es como una matriz de punteros, suponiendo que se haya reservado la memoria adecuadamente. así, en

```
char *punt[10];
```

se define igualmente una matriz de punteros a `char`, pero reservando adicionalmente memoria para almacenar 10 elementos, es decir, 10 punteros. Por ejemplo, `punt[0]` es un puntero a `char` pero, una vez más, podría interpretarse como una matriz, es decir, una cadena de caracteres, con lo cual, en realidad hemos definido una matriz de cadenas, pero reservando memoria sólo para los punteros, y no para las cadenas. Para ello debería definirse una matriz bidimensional:

```
char punt[10][12];
```

así, definimos y reservamos memoria para una matriz de 10 cadenas, donde cada una de ellas puede contener hasta 12 caracteres. De la misma forma, haciendo uso de la inicialización, podría hacerse:

```
char *punt[] = { "Lunes", "Martes", "Jueves" };
```

o también

```
char **punt = { "Lunes", "Martes", "Jueves" };
```

donde definimos e inicializamos una matriz de 3 cadenas.

Un uso muy habitual de las matrices de punteros se halla en el paso de parámetros al programa principal, como veremos en la sección 7.1, donde `char *argv[]` es una matriz de cadenas que contiene los parámetros de la línea de comandos.

Otro uso habitual de los punteros a punteros es el paso por referencia a una función de una variable puntero, tema que será tratado en la sección 8.3.

5.5. El modificador `const`

Todos los tipos de variables pueden usar la palabra reservada `const` en la definición, para indicar al compilador que su valor no puede ser modificado. Para ello, en la definición debe obligatoriamente inicializarse la variable. Cualquier intento de modificación provocará un error en la compilación. También provocará un error un intento de acceder a su dirección. Veamos unos ejemplos:

```
const int a = 6;
int *p;

p = &a; /* Error! */
a = 15; /* Error! */
```

Para declarar una constante puntero se usa `*const` en vez de `*` en la declaración:

```
int a = 6, b = 5;     /* normal */
int *const p = &a;    /* puntero constante */

*p = 7;               /* OK, a vale ahora 7 */
p = &b;               /* Error!, el puntero no puede cambiar */
```

Si deseásemos un puntero modificable, aunque no el objeto al que apunta, deberíamos declarar constante sólo el contenido:

```
int a = 6, b = 8;
const int *p = &a; /* puntero modificable,
                      pero no su contenido      */

p = &b;            /* OK, p puede ser cambiado  */
*p = 10;           /* Error!, el objeto apuntado
                      no puede cambiarse        */
```

Y por último, para tener tanto un objeto como un puntero constantes:

```
int a = 6, b = 5;
const int *const p = &a; /* puntero y objeto constantes */

*p = 7;   /* Error!, el objeto no puede cambiar  */
p = &b;  /* Error!, el puntero no puede cambiar */
```

Como veremos en la sección 8.3, el modificador `const` también puede aplicarse de la misma forma a los parámetros formales de una función para indicar que dichas variables no pueden ser modificadas dentro de la función.

5.6. El tipo `void`

C define otro tipo básico que es el tipo nulo, o ausencia de tipo, y que se representa por la palabra clave `void`. No obstante, carece de sentido definir variables de este tipo, a excepción de que se trate de punteros. En el caso de punteros, el tipo `void` se refiere a un puntero genérico a cualquier tipo de dato. Es decir, una dirección de memoria que contiene un tipo de datos sin especificar. Por tanto, aunque su valor puede ser modificado libremente, no se puede acceder a su contenido mediante el operador `*`. Por ejemplo:

```
char a = 5, *g;
void *p;

p = &a; /* OK ! */
*p = 1; /* Error de compilación */
g = p;  /* OK ! */
*g = 2; /* OK !. ahora a valdría 2 */
```

El uso del tipo de `void` se restringe básicamente para indicar que una función no devuelve nada (`void funcion()`), que devuelve un puntero genérico (`void *funcion()`), o que recibe como parámetro un puntero genérico (`void funcion(void *parametro)`). Ver también secciones 7 y 11.

5.7. Conversión de tipos

En C es posible operar con variables de tipos distintos. Toda la aritmética entera se realiza con una precisión mínima de un `int`. Por tanto, en el caso UNIX/LINUX, toda la aritmética entera se realiza con operandos de 32 bits con signo. Como consecuencia de ello, aunque una expresión contenga sólo `char` y `short int`, los operandos son extendidos a `int` antes de su evaluación, proceso denominado **promoción de enteros**. No obstante, si la promoción en un `int` no pudiese representar todos los valores del tipo original, la conversión se realizaría a un `unsigned int`. En el caso de UNIX/LINUX, ello afectaría sólo al caso de que la expresión involucrase un `unsigned int`.

Si en una expresión existen tipos reales, entonces el compilador extenderá todos los operandos al tipo real de mayor longitud.

La conversión de tipos implícita puede llevar consigo problemas al asignar a una variable una expresión de distinto tipo, ya que el resultado de la expresión podría exceder la capacidad de representación del tipo de la variable a la que se asigna, hecho que el compilador no detectaría. En cualquier caso, aunque no haya conversión implícita de tipos, el compilador nunca detecta un desbordamiento (*overflow*), aunque puede avisar de ello en algunas ocasiones (mediante un mensaje *Warning*).

También debe tenerse cuidado si se asigna a una variable de tipo entero una expresión de tipo real, ya que se realizará un truncamiento.

Así, en el siguiente ejemplo:

```
int a;
char b;
float c;
double d;

a = (c * b) + d;
```

la expresión del lado derecho de la sentencia será de tipo `double`, y estamos intentando asignársela a una variable de tipo `int`. Si el resultado cabe en una variable de dicho tipo no habrá problemas, ya que se realizará una conversión automáticamente, aunque se perderá la parte real. No obstante, si el resultado excede de un `int`, el resultado es impredecible.

Lo mismo ocurre cuando se pasan argumentos a una función, pues equivale a una asignación del argumento al parámetro formal de la función, o cuando se asigna el resultado de una función a una variable. Por ejemplo, como veremos en la sección 11, la función `malloc` devuelve un puntero de tipo `void` que será convertido al tipo de puntero de la variable a la que se asigna.

```
double *p;

p = malloc(12); /* El puntero genérico se convierte a
                   puntero a double */
```

5.7.1. Conversión de tipos explícita: El *cast*

Además de las conversiones de tipo implícitas, C facilita una forma de realizar conversiones de tipo de forma explícita denominada *cast*. Una expresión *cast* consta de un paréntesis `'('`, un tipo, un paréntesis `')'` y una expresión:

(type) expresion

Por ejemplo, si `a` es una variable de tipo `int`, entonces

```
(double) a
```

fuerza a que el valor de `a` sea extendido a la precisión de un `double`. Debe notarse que el *cast* sólo afecta a la primera expresión (constante, variable o expresión entre paréntesis) que se halla a la derecha. Así,

```
(int) a + b
```

forzaría sólo a que `a` fuese de tipo `int`, mientras que

```
(int) (a + b)
```

forzaría a que se extendiese el resultado de la suma la precisión de un `int`.

Veamos un ejemplo sencillo donde el *cast* puede resultar de utilidad.

```
int b; float c; double d;
d = c * b;
```

En este caso, la expresión `c * b` se calculará con la precisión de un `float`, por lo que el resultado podría exceder su rango de representación. Para evitar esto, y forzar a que se calcule con la precisión de un `double`, bastaría con forzar uno de los operandos a `double`, de la forma:

```
d = (double) c * b;
```

Importante: Si bien, la asignación entre tipos básicos no plantea problemas, pues se realiza una conversión implícita, algunos compiladores podrían no permitir la asignación entre punteros a distintos tipos, fundamentalmente si se hallan involucrados tipos compuestos como estructuras, debiendo para ello hacer uso de la conversión explícita (*cast*). Se volverá a este tema en la sección 8.3. Por ejemplo,

```
struct trama a;
char *c;
c = (char *)&a;
```

6. Estructuras de datos

La creación de tipos nuevos de datos en C se puede hacer de cuatro formas:

1. Combinando un conjunto de variables en una única variable, lo que denominaremos **estructura**.

2. Permitiendo que la misma zona de memoria sea compartida por varias variables, mediante una **unión**.

3. Mediante la enumeración de los distintos valores que puede tomar la variable se crea el **tipo enumerado**.

4. Mediante la palabra clave `typedef`.

6.1. Estructuras

Una estructura es un conjunto de variables referenciadas por un nombre común (similar al registro en PASCAL). El formato general de la definición de una estructura es:

```
struct nombre_de_estructura {
  tipo_1 elemento_1;
  tipo_2 elemento_2;
  ...
  tipo_n elemento_n;
} variable_struct_1, variable_struct_2,..;
```

Es muy común también definir una matriz cuyos elementos son estructuras. Este tipo de matrices deben declararse una vez declarada la estructura, como cualquier otra variable, o bien en la propia declaración de ésta última.

Ejemplo:

```
struct agenda {
  char nombre[10];
  char apelli[15];
  int  edad;
  struct agenda *prev; /* Útil para crear una lista enlazada */
  struct agenda *next; /* Útil para crear una lista enlazada */
} listado, *p_listado, libro[10];
```

La referencia a un elemento de la estructura se hace mediante el operador `'.'`, o bien mediante el operador `'->'` si se dispone de puntero a una estructura. Una estructura puede también ser inicializada de forma similar a las matrices, mediante una lista ordenada de los valores de cada elemento de la estructura, separados por comas y entre llaves. Veamos unos ejemplos:

```
struct fecha {
  int dia;
  char mes[12];
  int año;
};

struct fecha date = {12, "Diciembre", 2000}, *p_date, fechas[10];

printf("%s\n", date.mes); /* Imprime "Diciembre" */
p_date = &date;   /* p_date apunta a la estructura date */
p_date->dia = 21;
date.dia++;      /* Tras la asignación y este incremento
                    dia vale 22 */
printf("%s\n", fechas[2].mes);
if (fechas[3].mes[1] = 'e') ...;
p_date = &(fechas[4]); /* Ahora p_date apunta a la quinta
                          estructura de la matriz fechas */
```

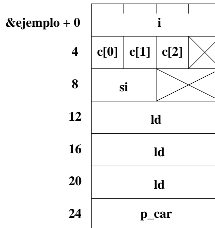
Las estructuras son especialmente útiles para crear listas enlazadas de nodos, donde cada nodo de la lista es una estructura. En este caso, resulta obligado incluir como elemento de la estructura un puntero, al menos, a otro nodo de la lista. Habitualmente, como en el caso de la `struct agenda` definida anteriormente, se tratará de un puntero al anterior o al siguiente nodo de la lista. En este caso, debería usarse el valor `NULL` para el puntero `prev` del primer nodo de la lista, y para el puntero `next` del último.

Todos los elementos de una estructura se almacenan en posiciones de memoria consecutivas, según el orden establecido en su definición. Sin embargo, dado que las estructuras se componen habitualmente de elementos de diferentes tipos, y los tipos de datos suelen poseer diferentes requerimientos de alineación en memoria (dependiendo de la máquina), pueden aparecer "huecos" en el espacio en memoria ocupado por la estructura. Actualmente, aunque no es norma general pero sí el caso de UNIX/LINUX, no se permite que variables de tamaño inferior a una palabra de memoria (4 bytes), caso de los tipos `char` y `short int`, se extienda en dos palabras de memoria. Por otro lado, las variables de tamaño 4 o múltiplo de 4 (caso de las direcciones (punteros) y los tipos `long int, float, double` y `long double`), deben ocupar palabras de memoria completas (1, 2 ó 3). Este esquema se mantiene incluso con matrices de estos tipos, pues una matriz de "n" elementos equivaldría a "n" elementos consecutivos del mismo tipo.

Así, en el siguiente ejemplo, tras los tres `char`, el siguiente `short int` se pasa a la siguiente palabra. Lo mismo ocurre con el siguiente `long double`, que pasa a ocupar las tres palabras siguientes.

```
struct alineacion {
```

```
  int i;
  char c[3];
  short int si;
  long double ld;
  char *p_car;
} ejemplo;
```



Puede observarse, pues, que la ocupación en memoria de una estructura es dependiente de la máquina, por lo que no se debiera hacer depender el código de ello para asegurar la portabilidad. En cualquier caso, si la estructura sólo contiene variables o matrices del mismo tipo, no se producirán huecos. Este hecho puede resultar de gran utilidad cuando se construyen paquetes de datos, por ejemplo, mediante estructuras que contengan una cadena de caracteres por campo del paquete, pues se dispone de una forma muy flexible y cómoda de acceder de forma individual a cada campo del paquete.

Una estructura, o un elemento de la misma, puede pasarse como argumento a una función, tanto por valor como por referencia (en este último caso, mediante un puntero a la estructura o a un elemento de la misma). Veamos algunos ejemplos usando el tipo `struct agenda` definido anteriormente.

```
struct agenda listado, *p_listado;

funcion(listado.edad);        /* Valor de edad */
funcion(listado.nombre[0]);   /* Primer carácter de nombre */
funcion(&(listado.edad));     /* Puntero al elto. edad */
funcion(listado.nombre);      /* Puntero al primer carácter de
                                 nombre */
funcion(&(listado.nombre[0])) /* Idem anterior sentencia */
funcion(listado.next)         /* Puntero a siguiente estructura
                                 en lista enlazada */
funcion(listado);             /* Paso estructura completa */
funcion(&listado);            /* Puntero a la estructura
                                 completa */
p_listado = &listado;
funcion(p_listado);           /* Otro puntero a la misma
                                 estructura */
```

6.2. Uniones

Una unión es un lugar de la memoria utilizado por un conjunto de variables que pueden ser de diversos tipos. El formato general de su definición es:

```
union nombre_union {
     tipo_1 elemento_1;
     tipo_2 elemento_2;
      ...
     tipo_n elemento_n;
     } variable_union_1, variable_union_2,..;
```

Por lo demás, la forma de declarar nuevas variables y de acceder a los elementos de la unión se realiza de la misma forma que con las estructuras. Veamos un ejemplo.

```
union mezcla {
   int i;
   char ca;
   }; /* Definición del tipo union mezcla */
union mezcla union1, *p_union1; /* Definición de dos variables
                                   del tipo anterior */
union1.i = 10;      /* Asignación al entero de la unión */
union1.ca = 0;      /* Asignación al carácter de la unión */
p_union1 = &union1;
p_union1->i = 20;   /* Nueva asignación al entero de
                       la unión */
p_union1->ca = 'a'; /* Nueva asignación al carácter de
                       la unión */
```

El compilador se encarga de reservar la cantidad de memoria suficiente para contener el tipo de mayor tamaño de los que forman la unión. Así, en el ejemplo anterior, la variable `union1` ocupará 4 bytes de memoria. Aunque para `ca` se necesita tan sólo un byte, son necesarios 4 para `i`. Es importante destacar que se comparte la zona de memoria, por lo que, en el ejemplo, tras asignar el valor 0 a `ca` estamos modificando también el valor de `i`.

6.3. Enumeraciones

Al igual que en **PASCAL**, una enumeración es un conjunto de constantes enteras con nombre, que especifica todos los valores válidos que una variable de ese tipo puede tener. La sintaxis es la siguiente:

enum etiqueta {lista_de_enumeraciones} lista_de_variables;

Tanto la etiqueta como la lista de variables son opcionales, pero debe existir alguna de las dos. Ejemplos:

```
/* Defino el tipo enumerado marcas */
enum marcas {opel, seat, volkswagen, citroen, ford};
/* Declaro la variable coche del tipo enumerado marcas */
enum marcas coche;
```

También podría hacerse:

```
enum marcas {opel, seat, volkswagen, citroen, ford} coche;
/* Definimos el tipo y a la vez declaramos la variable coche */
```

o también

```
enum {opel, seat, volkswagen, citroen, ford} coche;
/* Declaramos la variable coche de tipo enumerado pero no */
/* definimos ningún tipo enumerado en concreto */
```

Cada uno de los símbolos de la enumeración corresponde a un valor entero, de forma que pueden usarse en cualquier expresión entera, aunque su utilidad debiera restringirse al uso cómodo y flexible de nemónicos que, además, evita la asignación de valores erróneos. A menos que se inicialice de otro modo, el valor del primer símbolo es 0, el del segundo 1, y así sucesivamente. Sin embargo, en

```
enum {opel=2, seat, volkswagen=10, citroen, ford} coche;
```

los valores serían, respectivamente, 2, 3, 10, 11, y 12.

6.4. Creación de tipos mediante `typedef`

La palabra clave `typedef` permite dar un nuevo nombre a un tipo ya existente, de la forma

typedef tipo_existente nombre_tipo;

Ejemplos:

```
typedef int Edad;
Edad a;
typedef struct {
    char a;
    int b;
    int c;
} Numeros;
Numeros a, *b, c[4];
typedef union {
    int i;
    char ca;
} Mezcla;
Mezcla a;
typedef enum {opel, seat, volkswagen, citroen, ford} Coches;
Coches coche;
```

En muchas ocasiones, el uso de `typedef` ayuda a hacer más legible las definiciones, sobre todo si éstas incluyen punteros a funciones. Véase el ejemplo del tipo `sighandler_t` en la sección 7.5.

Mientras que los nombres usados para las estructuras, uniones y enumerados nunca interferirán con los dados a variables y funciones, debido a que los identificadores siempre se componen de las palabras clave `struct`, `union`, `enum` y el nombre, deshaciendo la posible ambigüedad, no ocurre así lo mismo con los nombres dados a los tipos creados mediante `typedef`, que son identificadores normales expuestos a las reglas de ámbito.

Por ejemplo, la definición

```
struct agenda agenda;
```

no plantea ambigüedad alguna, ya que se está declarando la variable `agenda` del tipo `struct agenda`. En cambio, en el siguiente ejemplo puede observarse como el nombre del tipo `Numeros` entra en conflicto con otras variables y se resuelve según las reglas de ámbito.

```
typedef struct {
  char a;
  int b;
  int c;
} Numeros;
int funcion1()
{
Numeros Numeros; /* No es ambiguo: Un tipo global y una variable
                    local */
Numeros n;       /* Error de compilación: el tipo global Numeros
...               * ha sido oscurecido por la definición de la
...               * variable local Numeros en la sentencia
...               * anterior          */
}
int funcion2()
{
Numeros a;   /* Definición de variable de tipo global Numeros */
int Numeros; /* Definición de variable local Numeros, que en */
...          /* adelante oscurece el tipo global Numeros     */
}
```

Para evitar conflictos entre identificadores, y al igual que ocurría con las macros, suelen reservarse los identificadores que comienzan con mayúscula para las definiciones de tipos mediante `typedef`.

7. Funciones en C

Un programa C está formado exclusivamente por **funciones** independientes, que se pueden comportar como una función o como un procedimiento en PASCAL.

Podríamos definir una función como un conjunto de sentencias C que, a partir de unos argumentos de entrada, realizan una determinada tarea, y devuelven un valor de salida.

La función está caracterizada por su tipo (puede ser cualquiera, incluso un puntero), su nombre y la lista de parámetros que recibe. Como todo bloque de sentencias, las correspondientes a una función van encerradas entre llaves. Si no se indica el tipo de la función, se toma por defecto el tipo `int`, aunque se recomienda indicar el tipo siempre por prudencia y portabilidad.

El nombre puede ser cualquiera excepto el de "`main`", que se reserva para la función principal, la que inicia la ejecución del programa.

La función debe definirse del tipo de dato que se quiere que devuelva como resultado. Si la función no devuelve ningún valor (como si fuese un procedimiento de PASCAL), debería usarse el tipo `void` (ausencia de tipo). Aunque no es estrictamente necesario usarlo, se recomienda hacerlo por prudencia y portabilidad, de la misma forma que se recomienda definir una función `int` como tal aunque sea el tipo por defecto.

Los parámetros irán precedidos por su tipo, y separados unos de otros por comas. Por ejemplo, una función que devuelve un puntero a un carácter (o una cadena) sería:

```
char *ordena(int x, char *s, float a, float b) {
  if (s[x] == 0) s++;
  ....
  return(s);
}
```

Se permite también que una función no reciba ningún argumento (a veces, en este caso se usa también el tipo `void`, aunque es poco habitual dada la imposibilidad de ambigüedad). Veamos un ejemplo:

```
void lee_datos() { ... }

main()
  {
    ...
     lee_datos();
    ...
   }
```

Se puede utilizar la palabra clave `return` para que una función devuelva un valor y el programa vuelva inmediatamente a la función llamante. En este caso, debe asegurarse que el tipo de la expresión devuelta coincide con el de la función.

Se puede utilizar también `return` sólo para volver a la función llamante, sin devolver valor alguno (caso de las funciones de tipo `void`). Si la función no contiene ningún `return`, el programa simplemente vuelve al finalizar la función (cuando llegamos a la '`}`' final).

Como ya vimos, el código de un programa C suele distribuirse entre varios ficheros fuente (con extensión `.c`), denominados **módulos**. Así, habitualmente un programa de cierto tamaño constará de un módulo principal que contiene, al menos, la función `main()`, otros módulos con el código de diversas funciones agrupadas según algún criterio, o con definiciones de tipos utilizados. No obstante, si el programa es suficientemente pequeño puede constar de un único módulo, el principal.

A diferencia de lenguajes como PASCAL, en C no se puede definir una función dentro de otra función, incluida la función `main()`.

7.1. La función main()

En C, el código del programa principal se halla en una función especial denominada `main()`, que es la primera que se llama al ejecutar el programa. Debe existir siempre una y sólo una función `main()`, y puede hallarse en cualquier parte de cualquier módulo (fichero fuente).

Esta función puede no llevar argumentos, y de llevarlos sólo puede recibir dos:

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

que permiten pasar parámetros al programa principal cuando es invocado desde la línea de comandos.

- `argc` será el número de parámetros (cadenas de caracteres) que van en la línea de comandos. Cada parámetro debe separarse por espacios en blanco.
- `argv` es una matriz de cadenas que contiene los parámetros (cadenas de caracteres) de la línea de comandos, tal como vimos en la sección 5.4.1. El primer parámetro siempre será el nombre del programa, y estará en `argv[0]`.

Por ejemplo, si llamamos a un programa `prueba` con la siguiente orden desde la línea de comandos:

```
> prueba valores.dat 25 a 2.7
```

dentro de la función `main()` tendremos que:

`argc`	vale	`4`
`argv[0]`	es	`‘prueba’`
`argv[1]`	es	`‘valores.dat’`
`argv[2]`	es	`‘25’`
`argv[3]`	es	`‘a’`
`argv[4]`	es	`‘2.7’`

Debe notarse que cada parámetro es una cadena de caracteres, lo que debe ser muy tenido en cuenta sobre todo si leen de la línea de comandos valores que van a ser usados en expresiones matemáticas. Así, antes de ser usados, las cadenas deben ser convertidas en los números que representan. Para ello deben usarse las funciones de la librería estándar (prototipos en `<stdlib.h>`), `int atoi(const char *)` y `double atof(const char *)`, que convierten una cadena de caracteres ASCII en los correspondientes `int` y `double` respectivamente (ver sección 12.3). También puede resultar especialmente útil la función `sscanf()`, que veremos en la sección 12.1.

7.2. Reglas de ámbito de las funciones: Ficheros de cabecera

Siguiendo las reglas de ámbito del lenguaje C (ver sección 3.1), para poder llamar a una función desde otra, su código debe hallarse en el mismo módulo antes de ser llamada. De no ser así, puede ocurrir:

- que el código se halle en otro módulo,
- que el código se halle posteriormente en el mismo módulo, o
- que se trate de una función de librería cuyo código será enlazado posteriormente.

En estos casos, la función debe ser declarada como una variable más antes de ser llamada. La declaración consta exclusivamente de la cabecera de la función, también llamada prototipo. Aunque sólo es necesario indicar los tipos de los parámetros, opcionalmente y por claridad, se pueden añadir también los nombres:

tipo nombre_función (lista de tipos de todos los parámetros);

En el caso de que la función se halle en otro módulo, también podría añadirse el modificador `extern` como en el caso de las variables, pero aquí resulta superfluo e innecesario, pues no hay ambigüedad posible entre lo que es la definición

de la función (que incluye su código) y lo que es una simple declaración (sólo la cabecera, terminando la declaración con punto y coma).

Para facilitar la tarea de programación y evitar tener que declarar funciones a menudo, lo que puede llevar consigo errores, habitualmente por cada fichero fuente conteniendo código de funciones (con extensión `.c`) existe un fichero con el mismo nombre (y extensión `.h`) que contiene exclusivamente las cabeceras o prototipos (declaraciones) de las funciones.

De esta forma, tal y como vimos en la sección 4.2, basta con incluir el fichero mediante una directiva `#include` para declarar todas las funciones contenidas en `fichero.c`, pudiendo ser utilizadas sin más.

De la misma forma, si se desea usar una función de librería, debe incluirse el fichero de cabecera que contiene la declaración de dicha función. En el caso de la librería estándar, al hallarse estos ficheros en el directorio por defecto (pueden indicarse más con la opción `-I` del compilador, como vimos en la sección 2), debe usarse la forma

`#include <fichero.h>`

Veamos un ejemplo ilustrativo.

```
#include <stdio.h> /* Declara todas las funciones de esta librería,
                      como printf() */

#include "mis_funciones.h" /* Declara todas las funciones propias
                              contenidas en el fichero
                              mis_funciones.c, como formatea()  */

int lee(char *); /* Función residente en otro módulo que puede
                    usarse en cualquier función del módulo tras la
                    declaración de su prototipo */

char modifica(int a)
{
....
....
return(lee((char *) &a)); /* Función residente en otro módulo que
                             se declara antes en este módulo */
}

main()
  {
    int *x, *entero(char); /* Función residente en este módulo
                              posteriormente, que debe declararse
                              para poder ser usada */

    char car, escribe(char); /* Función residente en otro módulo
                                que puede usarse en esta función
                                tras esta declaración */

    lee(&car);   /* Función residente en otro módulo que se
                    hallla declarada antes en este módulo */
```

```
    escribe(car);   /* Función residente en otro módulo que se */
                    /* halla declarada antes en esta función   */

    x = entero(car);   /* Función residente en este módulo
                          posteriormente, que se halla declarada
                          en esta función */

    car = modifica(*x); /* Al estar definida antes en este */
                        /* módulo, puede usarse sin más    */

    formatea(car); /* Declaración incluida en el fichero de
                      cabecera mis_funciones.h */
    printf("%c",car); /* Declaración incluida en el fichero de
                         cabecera stdio.h */
}

int *entero(char c)
  {
    int a;

    a = (int)c;
    return(&a);
  }
```

7.3. Ficheros de tipos

La directiva `#include` puede utilizarse también para incluir cualquier otro fichero que desee el programador, como habitualmente suele ocurrir con ficheros que contienen definiciones de tipos, constantes, macros, etc.

El caso de las definiciones de tipos es, además, bastante habitual pues en C no se pueden importar, por lo que el ámbito de una definición de tipo es, a lo sumo, un módulo. Ello podría plantear problemas de consistencia si dos objetos en módulos distintos se refieren al mismo tipo como, por ejemplo, una variable local a la que se asigna el valor devuelto por una función del mismo tipo definida en otro módulo:

```
  fichero1.c
  ----------
  typedef struct {
    char a;
    int c;
    int b;
  } Str;

  main()
  {
    Str f(), var;
    var=f();
  ...
 }
```

```
fichero2.c
----------
typedef struct {
  char a;
  int b;
} Str;

Str f()
{
  Str numeros;

  numeros.a = 5;
  numeros.b = 5;
  return(numeros);
}
```

En este ejemplo, aún cuando ni compilador ni enlazador detectan error alguno, pues la función está correctamente declarada, el valor de la variable `var` tras la llamada a `f()` es indeterminado y del todo incorrecto, dado que los tipos `Str` aunque tienen el mismo nombre, son distintos. No obstante, si fuesen definidos exactamente de la misma forma, el resultado sería el esperado.

Para evitar estos problemas, la solución más sencilla y habitual es ubicar en un fichero con extensión `.h` aquellas definiciones de tipos que serán usadas en más de un módulo. Posteriormente basta con incluir este fichero con una directiva `#include` en aquellos módulos que deseen usarlas.

7.4. Librerías

Las librerías constituyen una forma simple de reunir varios ficheros objeto conjuntamente. Las librerías pueden ser de dos tipos:

- **Estáticas**: El código de la función es integrado con el ejecutable en la fase de enlazado.

- **Dinámicas**: El código de la función es cargado cuando se ejecuta el programa. Las librerías dinámicas permiten economizar espacio en disco pero, sobre todo, memoria porque son cargadas sólo una vez en memoria y el código puede ser compartido entre todos los programas que la necesiten.

Las cabeceras o prototipos (declaraciones) de las funciones de cada librería se hallan en ficheros de cabecera con extensión `.h`. Tantos estos ficheros como las librerías se hallan en determinados directorios conocidos por el compilador. Por ejemplo, en UNIX/LINUX, los ficheros de cabecera se buscan por defecto en el directorio `/usr/include`, y las librerías en el directorio `/usr/lib`. Como ya vimos en la sección 2, se pueden indicar ubicaciones adicionales de ficheros cabecera (mediante la opción `-I` del compilador) o de librerías (mediante la opción `-L` del compilador).

Los compiladores de C poseen una librería de funciones más o menos extensa, pero todos soportan la librería estándar definida por **ANSI**, para garantizar la portabilidad. Esta librería dinámica denominada **librería C** (`libc`) incluye la

mayor parte de las funciones que se necesitan habitualmente en un programa C, a excepción de la librería de funciones matemáticas (`libm`, cuyo fichero de cabecera es `math.h`). Otras librerías especiales, como por ejemplo las de gráficos, tampoco se hallan contenidas en la librería estándar.

Dado que el compilador, por defecto, enlaza los ejecutables dinámicamente con la **librería C**, no es necesario indicárselo. Sin embargo, si se usan, por ejemplo, las funciones matemáticas declaradas en `math.h`, debe indicarse al compilador que enlace el ejecutable con la librería matemática, tal y como se indica en el siguiente ejemplo. De lo contrario, se produciría un error de enlazado:

```
> gcc ejemplo.c -o ejemplo -lm
```

Dado que la creación de una librería dinámica es una tarea de cierta complejidad, y que se sale del ámbito de este manual, veremos una forma sencilla de crear una librería estática. Supongamos que disponemos de un fichero `nuestras.c` que contiene una serie de funciones creadas por nosotros. Los comandos básicos para crear una librería estática de nombre `libpropia.a` serían los siguientes:

`>gcc -c nuestras.c` $\Longrightarrow$ `nuestras.o`

`>ar cru libpropia.a nuestras.o`

`>ranlib libpropia.a` $\Longrightarrow$ Ya hemos creado nuestra librería estática.

Para comprobar que todo ha ido correctamente puede ejecutarse el comando

```
> nm -s libpropia.a
```

que nos debería listar todas las funciones incluidas en la librería. Ahora, para poder usar las funciones de nuestra librería, bastaría con crear el fichero con todas las cabeceras de las funciones de la librería, por ejemplo `nuestras.h`, e incluirlo mediante la directiva `#include` en el fichero fuente donde se van a usar. Una vez hecho esto, cuando compilemos nuestro programa, basta añadir la opción `-lpropia` y `-Ldirectorio` para indicar su ubicación si ésta no es la estándar.

7.5. Punteros a funciones

Al igual que ocurría con las matrices, el nombre de una función es la dirección de memoria a partir de la cual se ubica su código. Es decir, el nombre de la función es un puntero constante (que no puede ser modificado) al inicio de la función.

En consecuencia, pueden definirse también variables de tipo puntero a función, indicando tanto el tipo devuelto por la función como los tipos de sus parámetros.

Pero debe ponerse especial cuidado en su declaración. Por ejemplo, en

```
int i, *pi, *fpi(int), (*fi)(int);
```

estamos declarando un entero `i`, un puntero a un entero `pi`, una función `fpi` que devuelve un entero, y un puntero `fi` a una función que devuelve un entero y recibe un entero como parámetro. Nótese la necesidad de los paréntesis que, de no existir, implicaría que `fi` fuese una función al igual que `fpi`.

De esta forma, podría llamarse a la función apuntada por `fi` mediante `fi(x)`.

Veremos también en la sección 8.3 que, al igual que cualquier otro tipo de puntero, se puede pasar como parámetro a una función un puntero a otra función, lo que permite disponer de una función que llama a distintas funciones en cada ejecución.

Igualmente, una función puede devolver un puntero a una función. Así, para definir una función `p_func()` que devuelve un puntero a otra función de tipo `type` y que recibe dos parámetros de los tipos `type_p1,type_p2`, debe hacerse de la siguiente forma:

```
type (*p_func(...))(type_p1, type_p2)
```

Ello, como ya se adelantó en la sección 6.4, hace que tales definiciones puedan llegar a ser difíciles de entender y, más aún, si algún parámetro es también del tipo puntero a función. Para tener definiciones más claras y legibles, el uso de `typedef` resulta muy útil.

Por ejemplo, la función de librería `signal`, que veremos en la sección 12.5.1, devuelve un puntero a una función de tipo `void` que recibe como parámetro un `int`. Adicionalmente, uno de los parámetros de la función `signal`, es un puntero a una función del mismo tipo. Es decir, su cabecera es:

```
void (*signal(int signum, void (*handler)(int)))(int)
```

Para hacer más legible esta definición se puede hacer uso de la definición del tipo `sighandler_t`, como un puntero a una función de tipo `void` que recibe como parámetro un `int`:

```
typedef void (*sighandler_t)(int); /* Definido en signal.h */
```

Así, el prototipo de la función `signal` quedaría mucho más claro:

```
sighandler_t signal(int signum, sighandler_t handler)
```

8. Tipos de variables. Reglas de ámbito

Como ya se adelantó en la sección 3.1, toda variable debe ser declarada antes de su utilización. No obstante, si bien el inicio de su ámbito de aplicación queda determinado por la declaración, la finalización de éste se determina atendiendo a las denominadas **reglas de ámbito**.

Según las reglas de ámbito, C distingue tres tipos de variables: locales, globales y parámetros formales.

8.1. Variables Locales

Las variables definidas dentro de un bloque (delimitado por `'{'` y `'}'`) serán locales, es decir, su ámbito se restringe exclusivamente a dicho bloque, que puede ser una función o una sentencia compuesta. A su vez, las variables locales pueden ser de tres tipos:

- **Automáticas**: Se crean al entrar en el bloque y se destruyen al salir de él. Se definen anteponiendo la palabra clave `auto` al tipo de la variable en la declaración. Por defecto, todas las variables locales a un bloque son automáticas.

- **Estáticas**: No son destruidas al salir del bloque, y su valor permanece inalterable cuando se vuelve a él. Se definen anteponiendo la palabra clave `static` al tipo de la variable en la declaración.

- **Registro**: Son variables estáticas, exclusivamente de tipo `char` o `int`, que se almacenan en un registro de la CPU en vez de en la memoria. Si se utiliza con variables referenciadas con mucha frecuencia se acelera la ejecución. Se definen anteponiendo la palabra clave `register` en la declaración.

8.2. Variables Globales

Las variables globales son aquellas que se definen en un módulo (fichero fuente) fuera de cualquier función (incluida también la función `main`). Las variables globales existen siempre, y no pueden usarse registros.

Si bien el ámbito de una variable global se restringe al módulo en que es definida, este ámbito puede ampliarse a cualquier módulo que la declare, anteponiendo la palabra clave `extern`, que indica al compilador que su definición se halla en otro módulo. Será el enlazador quien se encargue de editar los enlaces adecuadamente, es decir, hacer corresponder ambas variables con la misma dirección de memoria.

No obstante, puede evitarse que una variable global sea exportada fuera del módulo donde es definida (es decir, sea visible desde otros módulos). Para ello debe anteponerse la palabra `static` en su definición, creando así una variable global estática. De esta forma, si en un módulo existe la siguiente definición

```
static int a;
```

la siguiente declaración en otro módulo

```
extern int a;
```

provocará un error de enlazado, de la misma forma que si no se hubiese definido la variable original a importar.

En cambio, la definición

```
int a;
```

se referirá a otra variable global distinta a la del primer módulo.

Aunque no es realmente necesario anteponer la palabra `extern` para importar una variable global no estática de otro módulo (el enlazador las considerará la misma variable), se recomienda hacerlo encarecidamente, pues ello ayudará a detectar errores de programación.

Por razones obvias, una declaración mediante `extern` no admite inicializaciones. Éstas sólo pueden ser realizadas en la definición original de la variable.

Importante: Dado que uno de los conceptos claves de la programación estructurada es que un programa sea desarrollado como una serie de módulos independientes, que reduce enormemente la posibilidad de que cometer errores no detectados, se recomienda encarecidamente no usar más variables globales que las estrictamente necesarias, que habitualmente son muy pocas o ninguna. Y de ser necesarias, es preferible siempre hacer uso de variables globales estáticas, no visibles desde otros módulos.

8.3. Parámetros Formales

Los parámetros formales de una función son, a todos los efectos, variables locales automáticas, y su valor al entrar en la función es el resultado de asignarle el valor actual del argumento pasado en la llamada a ésta.

En C todos los argumentos pasados a una función son por valor. Es decir, cuando el argumento es una variable, tan sólo sirve para inicializar el parámetro formal con el valor actual de esta variable. Así, aunque el parámetro formal puede ser alterado posteriormente dentro de la función, como cualquier otra variable local, la variable pasada como argumento no puede ser modificada. Veamos un ejemplo:

```
int incrementa(int x) {
  return(++x); /* Se modifica "x", pero no "a",
                  el argumento pasado */
}
main() {
  int a=5, b;
  b = incrementa(a); /* b = 6, pero a = 5 */
}
```

Importante: Tal como se comentó en la sección 5.7, el paso de un argumento a una función equivale a realizar una asignación del argumento al parámetro formal de la función. Si bien, la asignación entre tipos básicos no plantea problemas, pues se realiza una conversión implícita, algunos compiladores podrían no permitir la asignación entre punteros a distintos tipos, fundamentalmente si se hallan involucrados tipos compuestos como estructuras, debiendo para ello hacer uso de la conversión explícita (*cast*). Por ejemplo,

```
struct trama a;
char *c;
int f(char *datos);
c = (char *)&a;
f((char *)&a);
```

El paso de punteros a una función resulta imprescindible si se desea que la función modifique una variable (lo que en **PASCAL** se denomina paso por referencia). Para ello, basta con pasar (por valor) un puntero a dicha variable, es decir, su dirección. De esta forma, aunque no se puede modificar el puntero pasado, sí se puede modificar su contenido. Veamos un ejemplo:

```
void intercambio(int *x, int *y) /* x e y contendrán las dirs. */
  {                              /* de a y b */
    int temp;

    temp = *x;  /* En este caso temp = 1 (el valor de a) */
    *x = *y;    /* Aqui a = b = 3                        */
    *y = temp;  /* Aquí b = temp = 1                     */
  }

main()
  {
```

```
    int a, b;

    a = 1;
    b = 3;
    intercambio(&a, &b); /* a y b cambiarán de valor: */
  }                      /* a = 3 y b = 1 */
```

El paso de una matriz como argumento a una función equivale exactamente a pasar por valor la dirección de su primer elemento. En ningún caso se pasa la matriz entera. Por tanto, si una función va a recibir una matriz como argumento resulta indiferente que el parámetro formal sea un puntero o una matriz. Es decir,

`int f(char *cadena)` equivale a `int f(char cadena[])`

No obstante, en este caso, el parámetro formal `cadena` es una variable puntero que se puede modificar, cuyo valor inicial es la dirección de inicio de la matriz pasada. Por ejemplo, podría hacerse, de forma equivalente:

```
int f(char *cadena)            int f(char cadena[])
{                              {
   ...                            ...
   c = cadena[5];                 c = *(cadena+5);
   cadena++;                      cadena[1] = 2;
   *cadena = 2;                   cadena++;
   ...                            ...
}                              }
```

En el caso de que la matriz sea de dimensión superior a uno, deben indicarse en la definición las magnitudes de todas las dimensiones excepto la primera. Es decir:

`int f(float valores [][10][5])`

Como ya se comentó en la sección 5.4.1, si queremos pasar por referencia un puntero, realmente lo que debemos hacer es pasar (por valor) un puntero a dicho puntero. Para ilustrar su uso con mayor claridad, modificaremos la función anterior `intercambio()` para que intercambie el valor de dos punteros en lugar del valor de dos variables:

```
void p_intercambio(int **x, int **y) /* x e y contendrán las */
  {                                  /* dirs. de pa y pb     */
    int *temp;
    temp = *x;  /* En este caso temp = pa  */
    *x = *y;    /* Aquí pa = pb            */
    *y = temp;  /* Aquí pb = temp = pa     */
  }

main()
  {
    int *pa, *pb, a, b;
    pa = &a;
    pb = &b;
    p_intercambio(&pa, &pb); /* "pa" y "pb" intercambiarán
                                su valor; "pa" apuntará a "b",
```

```
                                  y "pb" apuntará a "a" */
   *pa = 5;                    /* b = 5 */
   *pb = 2;                    /* a = 2 */
}
```

También puede pasarse como parámetro a una función un puntero a otra función, lo que permite disponer de una función que llama a distintas funciones en cada ejecución. Por ejemplo, supongamos que las funciones `media()` y `varianza()` se hallan definidas en un módulo cualquiera:

```
main() {
 double media(float *, int), varianza(float *, int), a, b;
 double statistics(double (*)(float *, int), float *, int);
 float *numeros;
 int N;

 /* Llamada con la función media    */
 a = statistics(media, numeros, N);

 /* Llamada con la función varianza */
 b = statistics(varianza, numeros, N);
 ...
}

double
statistics(double (*func)(float *, int), float *valores,
           int numero)
{
 ...
 return(func(valores, numero));
}
```

El modificador `const`, estudiado en la sección 5.5, también puede ser aplicado a los parámetros formales de una función para indicar que dichos parámetros no pueden ser modificados dentro de la función. Por ejemplo, en la siguiente función, `tiempo` es una constante, `horas` y `minutos` son dos punteros constantes, aunque el objeto apuntado puede ser modificado, `segundos` es un puntero a una constante, pero el puntero puede ser modificado, y `sesenta` es un puntero constante a una constante, donde ni puntero ni objeto apuntado pueden ser modificados.

```
void proceso(const int tiempo, int *const horas,
             int *const minutos, const int *segundos,
             const int *const sesenta)
```

9. Control de flujo en C

En adelante, cuando nos refiramos a sentencias, debe entenderse que éstas pueden ser simples (terminadas en '`;`' o compuestas (un bloque de sentencias simples encerradas entre llaves).

9.1. Sentencia condicional: `if`

Estructura general:

```
if (expresión)
  sentencia_1;
else /* Opcional */
  sentencia_2;
```

Si la `expresión` posee un valor distinto de cero (valor lógico TRUE), se ejecuta la `sentencia_1`. En cambio, si es cero (valor lógico FALSE), no se ejecutará, a menos que exista un bloque `else`, en cuyo caso se ejecutará la la `sentencia_2`.

Para evitar ambigüedades en el caso de sentencias `if-else` anidadas, el compilador asociará el `else` con el `if` más próximo sin resolver.

Ejemplo:

```
int a, *p;

if (a <= 6)
  if (p)   /* Si el puntero no es NULL */
   {
     *p = 5;
     p = &a;
   }
  else   /* Se asociaría con if(p) */
    p = &a;
```

En C existe un operador especial, denominado **operador condicional**, que se halla estrechamente ligado a la sentencia `if-else`. Su formato es:

condición ? expresión_true : expresión_false

El valor devuelto por este operador depende de la `condición`: si ésta es cierta devolverá `expresión_true`, y si es falsa `expresión_false`.

Por ejemplo, en la sentencia

```
maximo = a > b ? a : b
```

se asignará a `maximo` el valor mayor entre `a` y `b`, de igual forma que se haría con una sentencia `if-else`:

```
if (a > b) maximo = a;
else maximo = b;
```

9.2. Sentencia `switch`

La sentencia `switch` da una solución mucho más elegante al uso de múltiples sentencias `if` anidadas.

Estructura general:

```
switch (expresión) {
    case cte_1: sentencia_11;
                sentencia_12;
                ...
    case cte_2: sentencia_21;
```

```
                sentencia_22;
                ...
         ...
         ...
         ...
      case cte_n: sentencia_n1;
                  sentencia_n2;
                  ...
        default: sentencia_default1;
                 sentencia_default2;
                 ...
  }
```

Se comprueba la coincidencia del valor de `expresión` (que debe ser de un tipo entero) con las constantes de tipo entero que siguen a `case`, denominadas etiquetas, y que no pueden hallarse duplicadas. En caso de coincidencia con alguna etiqueta, se ejecuta el bloque de sentencias correspondiente, y se continua con todas las sentencias hasta el final del `switch`. Si se desea ejecutar sólo el bloque correspondiente a la etiqueta coincidente, debe añadirse al final del bloque de sentencias la sentencia `break`, que finaliza la ejecución del `switch`. De esta forma, el `switch` se comporta como el **CASE** del PASCAL. Véase la sentencia `break` más adelante.

La palabra clave `default` identifica a un bloque de sentencias opcional que se ejecutará si no se encuentra ninguna coincidencia.

También se permite que no existan sentencias asociadas a una etiqueta, en cuyo caso, siguiendo la regla, se ejecutarían las correspondientes a la siguiente etiqueta. Ello permite ejecutar un bloque de sentencias si la expresión coincide con cualquier etiqueta de un conjunto. Veamos algunos ejemplos:

```
int n=2;                              SALIDA
switch(n) {                           ------
  case 1  : printf("Uno\n");          Dos
  case 2  : printf("Dos\n");          Tres
  case 3  : printf("Tres\n");         Default
  default : printf("Default\n");
 }

int n=2;                              SALIDA
switch(n) {                           ------
  case 1  : printf("Uno\n");          Dos
            break;
  case 2  : printf("Dos\n");
            break;
  case 3  : printf("Tres\n");
            break;
  default : printf("Default\n");
 }

int n=2;                              SALIDA
switch(n) {                           ------
  case 1  : printf("Uno\n");          Dos o Tres
```

```
          break;
  case 2  :
  case 3  : printf("Dos o Tres\n");
          break;
  default : printf("Default\n");
 }

int n=7;                              SALIDA
switch(n) {                           ------
  case 1  : printf("Uno\n");          Default
  case 2  : printf("Dos\n");
  case 3  : printf("Tres\n");
  default : printf("Default\n");
 }

int n=7;                              SALIDA
switch(n) {                           ------
  case 1  : printf("Uno\n");          Fin
  case 2  : printf("Dos\n");
  case 3  : printf("Tres\n");
}
printf("Fin\n");
```

9.3. bucle `while`

Estructura general:

```
while (condición_de_continuación)
    sentencia;
```

La `condición_de_continuación` se evalúa antes de entrar al bucle en cada iteración, incluso antes de la primera vez, continuando el bucle, es decir, ejecutándose `sentencia`, mientras la condición sea verdadera. De esta forma, si la condición es falsa la primera vez, el bucle no se ejecutaría.

En el siguiente ejemplo podemos ver cómo se puede recorrer una lista enlazada de principio a fin imprimiendo el nombre de aquellas personas que tengan 20 años.

```
  struct personas {
    int edad;
    char nombre[30];
    struct personas *next;
  } *cabeza, *actual;
  ...
  actual = cabeza;
  while(actual)  /* Mientras el puntero no sea nulo */
  {
    if (actual->edad == 20)
      printf("%s\n", actual->nombre);
    actual = actual->next;  /* Actualizamos el puntero */
                            /* que recorre la lista */
```

```
}
```

9.4. Sentencia `do-while`

Estructura general:

```
do
   sentencia;
while (condición_de_continuación);
```

Es muy similar al bucle `while` salvo que, al comprobarse la `condición_de_continuación` al final, siempre tendrá una iteración al menos, siendo el único bucle en C con esta característica.

9.5. bucles infinitos con `while` y `do-while`

Es posible crear **bucles infinitos**, que no terminan nunca, con las sentencias `while` y `do-while`, sin más que usar una `condición_de_continuación` que sea siempre cierta, es decir, cualquier valor distinto de `'0'`. Por ejemplo:

```
#define TRUE 1;

while (TRUE)          do {
  {                        ...
    ...                    ...
  }                  } while(2)
```

9.6. Bucle `for`

Estructura general:

```
for (inicialización; condición_de_continuación; actualización)
  sentencia;
```

- **Inicialización**: Se ejecuta justo antes de entrar en el bucle. Se pueden inicializar una o más variables. En caso de que sean más de una, se separan por comas.
- **Condición de continuación**: La condición se evalúa antes de cada iteración del bucle, incluso antes de la primera vez, continuando el bucle mientras la condición sea verdadera.
- **Actualización**: Se realiza como si fuera la última sentencia del bloque de sentencias del bucle, es decir, sólo se ejecutará si la condición es verdadera. Si se realizase más de una actualización, también se separarían por comas.

Ejemplos:

```
for(x = 0;  x < 10; x++)
for(x = 0, y = 1; x+y < 10; ++x, ++y)
```

El bucle `for` es semánticamente equivalente a un bucle `while` de la forma:

```
inicialización;
while(condición_de_continuación) {
  sentencia;
  actualización;
}
```

Así, al igual que en el ejemplo del `while`, también podemos recorrer la lista con un bucle `for`:

```
for (actual = cabeza; actual != NULL; actual = actual->next)
  if (actual->edad == 20)
    printf("%s\n", actual->nombre);
```

Ninguna de las tres partes de la sentencia **FOR** es necesaria, aunque sí los "`;`".

Veamos un ejemplo donde la actualización y la inicialización se hacen fuera de la sentencia `for`, mientras que la condición va en la propia sentencia `for`:

```
main() {
  ...
  x = 0;
  ...
  for( ; x < 10; ) {
    ...
    x++;
    ...
  }
}
```

Además de con `while` y `do-while`, como ya vimos, es posible crear **bucles infinitos** con la sentencia `for`, sin más que prescindir de la condición que, al no existir, se cumplirá siempre. Ejemplos:

```
for(;;)       for(x=0;;)    for(;;x++)    for(x=0;;x++)
  {             {             {             {
    ...           ...           ...           ...
  }             }             }             }
```

9.7. Sentencia `break`

La sentencia `break` se usa para provocar la finalización inmediata de un bucle `do`, `for`, o `while`, o de una sentencia `switch`.

Se utiliza cuando interesa que un bucle deje de repetirse al cumplirse una condición especial distinta de la condición normal de finalización del bucle. Para su uso en una sentencia `switch` ver subsección 9.2.

Ejemplo:

```
main() {
  int t;

  do {
    t = getnum();
```

```
    if (t == 0) break;
    printf("%d\n", t);
  } while (t < 100);
  sentencia; /* cuando t = 0 salta a esta sentencia */
}
```

Un `break` dentro de varios bucles anidados causa la salida del bucle más interno.

9.8. Sentencia `continue`

Esta sentencia actúa de forma contraria a `break`, en lugar de finalizar la ejecución del bucle, fuerza a que se evalúe la condición y se ejecute inmediatamente la iteración siguiente o se salga del bucle, dependiendo de dicha evaluación.

Ejemplo:

```
main() {
  int t;

  for(t = 0; t < 100; t++) {
    x = getnum();
    if (x < 0) continue; /* Si x es negativo no se hace el */
      printf("%d",x);    /* printf() y se pasa a la        */
  }                      /* iteración siguiente            */
}
```

10. Entrada/Salida en C

Todas las funciones de E/S en C, ya sean por el dispositivo estándar (pantalla y teclado), o por fichero, se encuentran en la librería estándar `<stdio.h>`.

10.1. E/S por dispositivo estándar

Los dispositivos estándar en C son, por defecto, el teclado y la pantalla. No obstante, éstos son tratados como si fuesen ficheros con los nombres `stdin` y `stdout`, respectivamente.

- `int getchar(void)`

  Devuelve un carácter de la entrada estándar. Si hay algún error, o se alcanza el final del fichero, devuelve `EOF`[1]

- `int putchar(int ch)`

  Imprime el carácter `ch` por la salida estándar. Si hay error devuelve `EOF`, sino devuelve el carácter.

Las dos funciones anteriores se pueden generalizar para que operen con cadenas de caracteres, dando lugar, así, a las funciones `gets()` y `puts()`.

[1] `EOF` (*End Of File*) es una macro definida en `stdio.h` que significa **fin de fichero**.

- `char *gets(char *string)`

  Lee la cadena de caracteres `string` desde la entrada estándar, devolviendo un puntero a la cadena. La orden de final de lectura es el retorno de carro, pero la cadena devuelta tendrá el carácter nulo (`‘\0’`) al final, sin incluir el fin de línea (`‘\n’`)

- `int puts(const char *string)`

  Imprime la cadena de caracteres `string` por la salida estándar, devolviendo el último carácter escrito, y `EOF` si hubiese error. Su realización es muy sencilla:

```
int puts(char *s) {
    int i;
    for(i = 0; s[i]; ++i)
      putchar(s[i]);
}
```

Continuando el proceso de generalización de la funciones de E/S estándar, llegamos a las funciones `scanf()` y `printf()`.

10.1.1. Salida formateada: Función `printf()`

Es una función de salida de propósito general, que permite escribir datos de distintos tipos con formato. El formato general es:

int printf(çadena_de_control", lista_de_argumentos);

`cadena_de_control` es una lista de caracteres que puede contener:

- Una especificación de conversión, a la que precede el símbolo `‘%’`, y que indica adicionalmente el formato de visualización de un argumento.

- Caracteres normales que no forman parte de una especificación de conversión, y que se copian tal cuál a la salida. Para provocar un cambio de línea se usa el carácter `‘\n’`.

En `cadena_de_control` existirán tantas especificaciones como argumentos, correspondiendo cada una a un argumento, por lo que deberán colocarse en el mismo orden. Los argumentos pueden ser variables, constantes o cualquier expresión. La función devuelve el número de caracteres escritos, y `EOF` en caso de error.

Las especificaciones de conversión constan obligatoriamente del símbolo `‘\n’`, por el que comienzan, y de la operación de conversión, que son las mostradas en la siguiente tabla:

OPERACIÓN	ACCIÓN
c	Imprime el carácter ASCII correspondiente
d, i	Conversión decimal con signo de un entero
e/E	Conversión a coma flotante con signo en notación científica
f	Conversión a coma flotante con signo, usando punto decimal
g/G	Conversión a coma flotante, usando la notación que requiera menor espacio
o	Conversión octal sin signo de un entero
u	Conversión decimal sin signo de un entero
s	Cadena de caracteres (terminada en `'\0'`)
x/X	Conversión hexadecimal sin signo
p	Dirección (puntero) en la forma segmento:desplazamiento
`%%`	Imprime el simbolo `%`

Notas: La elección `e/E` simplemente determina si se usa una `e` minúscula o mayúscula para separar mantisa y exponente, al igual que `g/G` cuando resulta elegida la notación científica. De la misma forma, `x/X` determina si se usan los caracteres `a-f` o `A-F`. Las operaciones `d` e `i` son equivalentes.

Ejemplos:

`printf("%s %c %d","esto es una cadena", 'C', 100);` ⇒ esto es una cadena C 100

`printf("esto es una cadena %d", 100);` ⇒ esto es una cadena 100

`printf("el %d es decimal, %f es real", 10, 5.23);` ⇒ el 10 es decimal, 5.23 es real

Una especificación de conversión puede incluir opcionalmente otros modificadores relativos al campo donde se imprimirán los valores. Estos modificadores irán entre el símbolo "`%`" y la operación de conversión, en el siguiente orden:

- *Flag*: Puede ser:
  - "`-`": El valor se justifica a la izquierda en su campo.
  - "`+`": El valor incluirá siempre el signo (sólo para `d/i, e/E, f, g/G`).
  - Un blanco: Los números no negativos irán precedidos de un blanco (sólo para `d/i, e/E, f, g/G`).
  - "`#`": Los valores en octal irán precedidos por un `0`. Los valores en hexadecimal irán precedidos por `0x` (o `0X`). Las conversiones a coma flotante siempre incluirán el punto decimal. Para `g/G`, se eliminan los ceros del final.
  - "`0`": Se usa el carácter `0`, en lugar de blancos, como relleno a la izquierda del valor en el campo.
- Ancho del campo: Es un entero positivo que indica el ancho mínimo del campo de visualización.
- Precisión: Es un entero positivo, precedido por un punto, usado para indicar el número máximo de dígitos a la derecha del punto decimal en una conversión de coma flotante (por defecto, se toma 6). Cuando se usa con

`d/i, o, u, x/X`, especifica el número mínimo de dígitos. Cuando se usa con `s`, determina el número máximo de caracteres.

- Modificador de tamaño: Indica el tamaño del tipo involucrado en la operación de conversión. En el caso de `d/i, o, u, x/X`, puede ser `h` (`short`) o `l` (`long`, por defecto en UNIX/LINUX). En el caso de `e/E, f, g/G`, puede ser `L` para indicar un `long double`. De no existir, se usan por defecto `int` y `float/double`.

Veamos unos ejemplos ilustrativos, en los que la salida irá entre corchetes para mostrar claramente el campo de visualización. Se usará el carácter "`_`" para representar un espacio en blanco.

VARIABLE	LLAMADA	SALIDA
`char c = 'A';`	`printf("[%c]", c)`	[A]
	`printf("[%3c]", c)`	[__A]
	`printf("[%-4c]", c)`	[A___]
`int j = 45;`	`printf("[%4d]", j)`	[__45]
	`printf("[%-5d]", j)`	[45___]
	`printf("[%+05d]", j)`	[+0045]
`short int k = -123;`	`printf("[%2hd]", k)`	[-123]
`float x = 12.345;`	`printf("[%e]", x)`	[1.234500e+01]
	`printf("[%+12.2E]", x)`	[___+1.24E+01]
	`printf("[%10.0e]", x)`	[_____1e+01]
	`printf("[%#10.0e]", x)`	[____1.e+01]
`double y = -678.9;`	`printf("[%-12.1E]", y)`	[-6.8E+02____]
`long double z = 757.6;`	`printf("[%.2Le]", z)`	[7.58e+02]
`float x = 12.345;`	`printf("[%-10.2f]", x)`	[12.35_____]
	`printf("[%+8.1f]", x)`	[___+12.4]
	`printf("[%08.2f]", x)`	[00012.35]
	`printf("[%+06.1f]", x)`	[+012.4]
	`printf("[%#4.0f]", x)`	[_12.]
`double y = -678.9;`	`printf("[%f]", y)`	[-678.900000]
	`printf("[%.2f]", y)`	[-678.90]
`int j = 45;`	`printf("[%5o]", j)`	[___55]
	`printf("[%-4o]", j)`	[55__]
`unsigned int l = 0127;`	`printf("[%#6o]", l)`	[__0127]
`unsigned int m = 123;`	`printf("[%6u]", m)`	[___123]
	`printf("[%-4u]", m)`	[123_]
`int i = 0xf4;`	`printf("[%6x]", i)`	[____f4]
	`printf("[%#6X]", i)`	[__0XF4]
`char *cadena = "Hola";`	`printf("[%s]", cadena)`	[Hola]
	`printf("[%8s]", cadena)`	[____Hola]
	`printf("[%-8s]", cadena)`	[Hola____]
	`printf("[%6.2s]", cadena)`	[____Ho]
	`printf("[%-10.6s]", cadena)`	[Hola______]
`int *ptr = &j;`	`printf("[%p]", ptr)`	[115A:F56B]

10.1.2. Entrada formateada: Función `scanf()`

Es una función de entrada de propósito general con formato, que permite leer datos de distintos tipos. El formato general es:

int scanf(çadena_de_control", lista_de_punteros_a_variables);

La función lee de la entrada estándar elementos que son convertidos, según las especificaciones de conversión contenidas en la `cadena_de_control`, y asignados a las variables cuyas direcciones se pasan en la lista. La `cadena_de_control` es un mapa de la entrada que se espera, y puede contener:

- Espacios en blanco: Uno o más espacios en blanco en la `cadena_de_control` encajan con uno o más espacios en blanco de la entrada.

- Otros caracteres no blancos, no contenidos tampoco en las especificaciones de conversión (por ejemplo, el de cambio de línea (`\n`), deben encajar exactamente con la entrada, o de lo contrario la operación fallará.

- Las especificaciones de conversión, que constan obligatoriamente del símbolo `'%'`, por el que comienzan, y de la operación de conversión, pero pueden incluir adicionalmente otros campos opcionales. El orden y función es el siguiente:

  - Símbolo `'%` (obligatorio).
  - Símbolo de supresión, `'*'` (opcional).
  - Campo de longitud (opcional): En el caso de ir precedido por `'*'`, indica cuántos caracteres de la entrada se deben saltar. En caso contrario indica cuántos se deben leer como máximo. De no existir, se leerán todos hasta el próximo blanco o hasta el último carácter válido (por ejemplo, si se trata de una operación `d` no se leerá un carácter `'a'`).
  - Modificador de tamaño (opcional): Indica el tamaño del tipo involucrado en la operación de conversión. Puede ser `h` (`short`) o `l` (`long`) para conversión decimal, y `l` (`double`) o `L` (`long double`) para conversión en coma flotante. De no existir, se usan por defecto `int` y `float`.
  - Operación de conversión (obligatorio): Son los mostrados en la siguiente tabla

OPERACIÓN	ACCIÓN
c	Lee y asigna un `char`
d/i	Conversión decimal con signo y asignación a `short, int` o `long`, según modificador
e/E), f o g/G	Conversión a coma flotante y asignación a `float, double,` o `long double`, según modificador
o	Conversión octal sin signo y asignación a `short, int` o `long`, según modificador
u	Conversión decimal sin signo y asignación a `unsigned short, int` o `long`, según modificador
s	Lee y asigna una cadena (entre blancos)
x/X	Conversión hexadecimal sin signo y asignación a `short, int` o `long`, según modificador

La función devuelve el número de campos de `cadena_de_control` procesados con éxito. Veamos unos ejemplos.

VARIABLE	LLAMADA	ENTRADA	EFECTO
`char ch;`	`scanf("%c", &ch)`	ABC	ch = 'A'
	`scanf("%*2c%c", &ch)`	ABC	ch = 'C'
`char *cad;`	`scanf("%2c", cad)`	ABC	cad[0] = 'A'; cad[1] = 'B'
`short s;`	`scanf("%hd", &s)`	__–12B	s = –12
`int i;`	`scanf("%*2dHola%d", &i)`	12Hola34__	i = 34
`float x;`	`scanf("%f", &x)`	_2c_	x = 2.0
`double y;`	`scanf("%2f%le", &x, &y)`	1.34.8R	x = 1.0; y = 34.8
`long double z;`	`scanf("%*f%Lg", &z)`	_–6.7_5.2	z = 5.2
`short s;`	`scanf("%*3o%ho", &s)`	76517W	s = 15
`int i;`	`scanf("%X", &i)`	1F	i = 31
`unsigned int k;`	`scanf("%*2u%u", &k)`	__123_	k = 3
`char cad1[10];`	`scanf("%4s", cad1)`	__A12cXd_	cad1 = .A12c"
`char cad2[15];`	`scanf("%s%s", cad1, cad2)`	_Hola_76_	cad1 = "Hola"; cad2 = "76"
`char cad1[10];`	`scanf("%*4s%s", cad2)`	__ABC12CD_	cad2 = "2CD"

10.2. E/S de fichero

- `FILE *fopen(char *nombre_fichero, char *modo)`

  Sirve para abrir el fichero que vayamos a utilizar, devolviendo un puntero al fichero (`FILE` es un tipo de dato específico definido en `<stdio.h>`), o `NULL`[2] si hubo algún error en la apertura, tal como no encontrar el fichero. `modo` es una cadena que puede tener los valores:

  - `r`: Abrir un fichero existente para lectura.
  - `w`: Borrar el contenido de un fichero existente o crear uno nuevo para escritura.
  - `a`: Añadir al contenido de un fichero existente o crear uno nuevo para escritura.
  - `r+`: Abrir un fichero existente para lectura y escritura, posicionándose al inicio.
  - `w+`: Borrar el contenido de un fichero existente o crear uno nuevo para lectura y escritura, posicionándose al inicio.
  - `a+`: Añadir al contenido de un fichero existente o crear uno nuevo para lectura y escritura, posicionándose al final.

  Ejemplo:

  ```
  FILE *fp;
  fp = fopen("fichero.txt", "w+");
  ```

- `int putc(int ch, FILE *fp)`

  Permite escribir un carácter en un fichero abierto con `fopen()`, y en modo "`w`". `fp` es el puntero al fichero y `ch` el carácter a escribir en el fichero. En caso de error devuelve `EOF`. Es idéntica a la función `fputc()`. `putc(ch, stdout)` equivale a `putchar(ch)`.

[2] `NULL` es un identificador definido en `stdio.h` y `stdlib.h` que posee el significado de puntero nulo. Cualquier intento de acceder a su contenido provoca un grave error.

- `int getc(FILE *fp)`

  Función complementaria de `putc()`, que permite leer un carácter de un fichero abierto con `fopen()` en modo "`r`", y cuyo puntero es `fp`. Devuelve el carácter leído y `EOF` si hubo error de lectura. Es idéntica a la función `fgetc()`. `getchar()` es equivalente a `getc(stdin)`.

- `char *fgets(char *buffer, int n, FILE *fp)`

  Lee una cadena de caracteres del fichero apuntado por `fp`, almacenándola a partir de la dirección `buffer` y añadiendo el carácter nulo de fin de cadena. La lectura terminará si se alcanza el fin del fichero, el fin de línea (el carácter '\n' también es leído), o hasta que se hayan leído `n-1` caracteres.

  La función devolverá un puntero NULL ante cualquier error o si se alcanza el fin del fichero sin haber leído ningún carácter.

- `char *fputs(char *s, FILE *fp)`

  Escribe la cadena de caracteres `s` (terminada en nulo) al fichero identificado por el puntero `fp`. La función devuelve EOF en caso de error, y un valor no negativo en otro caso.

- `int fclose(FILE *fp)`

  Cierra el fichero cuyo puntero es `fp`, y que fue previamente abierto con `fopen()`.

- Funciones `fscanf()` y `fprintf()`: Su comportamiento es exactamente igual a las anteriores `scanf` y `printf()`, salvo que la entrada y salida es desde y hacia un fichero, identificado por un puntero `fp`. Su sintaxis es:

  int fscanf(FILE *fp, çadena_de_control", lista_de_argumentos);

  int fprintf(FILE *fp, çadena_de_control", lista_de_argumentos);

Cabe recordar que, si bien los dispositivos estándar en **C** son, por defecto, el teclado y la pantalla, éstos son tratados como si fuesen ficheros con los nombres `stdin` y `stdout`, respectivamente. Es decir, `printf("...")` y `scanf("...")` equivalen a `fprintf(stdout, "...")` y `fscanf(stdin, "...")`, respectivamente. También existe una salida estándar de error, denominada `stderr`, que puede utilizarse para generar mensajes de error ajenos al funcionamiento normal de un programa mediante `fprintf(stderr, "...")`.

Estos ficheros pueden ser redireccionados en la línea de comandos mediante los operadores `<` (para `stdin`), `>` (para `stdout`) y `>&` (para `stderr`). Por ejemplo, si disponemos de un programa `prueba`, el comando

```
> prueba < in.txt > out.txt >& err.txt
```

hace que la entrada estándar sea desde el fichero `in.txt` en lugar del teclado, la salida estándar sea hacia el fichero `out.txt` en lugar de la pantalla, y la salida estándar de error sea el fichero `err.txt` también en vez de la pantalla.

- `void perror(const char *s)`

  La mayor parte de las llamadas al sistema devuelven -1 cuando algo falla y dejan en la variable global entera `errno` (ya definida en `<stdlib.h>`) un código indicativo del error. Esta función sirve para traducir este código en un mensaje de error. Su funcionamiento es equivalente a

  ```
  fprintf(stderr, "%s : %s\n", s,
          "mensaje de error relativo a errno")}.
  ```

- Otras funciones de E/S, cuyos prototipos se hallan en `<stdio.h>`, son: `clearerr`, `feof`, `ferror`, `fflush`, `fgetpos`, `fread`, `fseek`, `fsetpos`, `ftell`, `fwrite`, `remove`, `rename`, `rewind`.

11. Asignación dinámica de memoria

Cuando un programa necesita una cantidad variable, generalmente grande, de memoria, resulta prácticamente imprescindible hacer uso de la asignación dinámica de memoria, que permite reservar, usar y liberar zonas de memoria, de forma dinámica durante la ejecución del programa. Obviamente, para acceder a tales zonas deberán utilizarse punteros.

Hay una serie de funciones de la librería estándar `<stdlib.h>` dedicadas a la gestión dinámica de la memoria.

- `void *malloc(unsigned tamaño)`

  Sirve para reservar una zona de memoria contigua de `tamaño` bytes, devolviendo un puntero al byte de comienzo de dicha zona. Si no hubiese memoria disponible (o `tamaño` fuese cero), se devolvería un puntero nulo (`NULL`). Es muy importante verificar que el valor devuelto no es un puntero nulo, antes de tratar de usarlo, pues de hacerlo así provocaríamos un error grave del sistema, que muchas veces acaba suponiendo la caída de éste.

  Cabe destacar que el puntero devuelto es del tipo genérico `void`, y que puede asignarse a cualquier variable puntero, sean del tipo que sean los datos a los que apunte. No obstante, por portabilidad, es costumbre usar el operador **cast**, para convertirlo al tipo deseado. Ejemplo:

  `char *cadena;`

  `cadena = malloc(6);` o también `cadena = (char *)malloc(6);`

  Cabe resaltar la utilidad de la palabra clave `sizeof`, que aplicada sobre un tipo ya definido `type` nos devuelve un `unsigned`, indicando el tamaño (en bytes) ocupado en memoria por dicho tipo de datos: `sizeof(type)`

  `type` puede ser un tipo básico (int, float, etc.) o un tipo definido por el usuario o en alguna librería. Se usa habitualmente con la función `malloc()` (y las que veremos a continuación) para pasarles el tamaño de los bloques. Por ejemplo, si deseamos reservar un bloque de memoria para una variable del tipo `struct agenda` y asignárselo al puntero `p_agenda`, deberíamos hacer:

```
struct agenda *p_agenda; /* Sólo se define el puntero   */
                         /* No se reserva  memoria para */
                         /* la estructura               */
p_agenda = (struct agenda *)malloc(sizeof(struct agenda));
```

- `void *calloc(unsigned num, unsigned tamaño)`

  Su uso es similar al de `malloc()`, excepto que permite reservar `num` bloques contiguos de `tamaño` bytes (en total `num`*`tamaño` bytes), devolviendo un puntero al primer byte del primer bloque. Al igual que ocurría con `malloc()`, devuelve un puntero nulo si no hay suficiente memoria disponible.

  ```
  p_agenda = (struct agenda *)calloc(10, sizeof(struct agenda));
  ```

- `void *realloc(void *ptr, void *tamaño_nuevo)`

  Sirve para cambiar el tamaño de la memoria a la que apunta `ptr`, y que fue asignada anteriormente (con `malloc()` o `calloc()`), al nuevo tamaño `tamaño_nuevo`, que puede ser mayor o menor que el original.

  La función devuelve un puntero, que por razones obvias, puede apuntar a una zona distinta, aunque en ese caso se copiaría el contenido del bloque viejo en el nuevo. Si el bloque no pudiese reasignarse (o `tamaño_nuevo` fuese cero), devolvería un puntero `NULL`.

- `void free(void *ptr)`

  Sirve para liberar una zona de memoria asignada anteriormente, y cuya dirección de comienzo es `ptr`, para que ésta pueda ser asignada posteriormente.

  Es obligatorio que sólo se llame a `free()` con un puntero que fue asignado previamente con alguna de las funciones de asignación dinámica, descritas anteriormente. Usando un puntero inválido se podría destruir el mecanismo de gestión de memoria y provocar la caída del sistema.

12. Otras funciones de la librería estándar

12.1. Manejo de cadenas

La librería `<string.h>` contiene toda una serie de funciones relacionadas con el manejo de cadenas de caracteres. A continuación describimos brevemente las más importantes, según su utilidad. Para más información, puede recurrirse al manual en línea, `man`, tal y como se comenta en la sección 13.3.

Debe tenerse en cuenta que todas las funciones de manejo de cadenas suponen que el final de una cadena es el carácter nulo. El tipo `size_t` usado se halla definido en el propio fichero y equivale a un `unsigned long`.

- Concatenación de la cadena `src` tras la cadena `dest`:

  ```
  char *strcat(char *dest, const char *src)

  /* Sólo n caracteres de src */
  char *strncat(char *dest, const char *src, size_t n)
  ```

- Búsqueda de `c` en la cadena `strchr`:

```
char *strchr(const char *s, int c)  /* Primera ocurrencia */
char *strrchr(const char *s, int c) /* Última ocurrencia  */
```

- Comparación de dos cadenas:
  - Devolviendo un valor negativo si `s1 <s2`, o si `s1 = s2`, y un valor positivo si `s1 >s2`:

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

    /* Sólo hasta n caracteres */
    int strncmp(const char *s1, const char *s2, size_t n)
    ```

  - Devolviendo el número de caracteres iniciales hallados en la cadena `s` que coinciden con los de la cadena `accept`, o que no coinciden con los de la cadena `reject`:

    ```
    size_t strspn(const char *s, const char *accept)
    size_t strcspn(const char *s, const char *reject)
    ```

  - Devolviendo un puntero a la primera ocurrencia en la cadena `s` de cualquier carácter de la cadena `accept`:

    ```
    char *strpbrk(const char *s, const char *accept)
    ```

- Copia de los caracteres de la cadena `src` en la cadena `dest`, que no pueden solaparse:

```
char *strcpy(char *dest, const char *src)

/* Sólo hasta n caracteres */
char *strncpy(char *dest, const char *src, size_t n)
```

- Longitud de una cadena `s`, excluyendo el carácter nulo.

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

Para realizar operaciones similares a las vistas con las cadenas, pero con dos zonas de memoria, byte a byte, se han previsto otras funciones, declaradas también en el fichero `<string.h>`. A diferencia de las anteriores, éstas no suponen que una zona de memoria termina con el carácter nulo, por lo que siempre se debe indicar el tamaño de las zonas manejadas. Por lo demás, el funcionamiento es muy similar. Sus prototipos son los siguientes:

```
void *memchr(const void *s, int c, size_t n)         /* Búsqueda */
int memcmp(const void *s1, const void *s2, size_t n) /* Comparación */
void *memcpy(void *dest, const void *src, size_t n)  /* Copia */
```

Dos funciones adicionales de memoria son:

```
void *memmove(void *dest, const void *src, size_t n)
```

para mover `n` bytes desde la zona apuntada por `src` hasta la apuntada por `dest`, que en este caso, a diferencia de `strcpy()` y `memcpy()`, sí pueden solaparse.

```
void *memset(void *s, int c, size_t n)
```

para rellenar `n` bytes a partir de `s` con el valor `c`.

Finalmente, también relacionadas con las cadenas, se hallan las funciones `sscanf()` y `sprintf()`, contenidas en el fichero `<stdio.h>`, y cuyo comportamiento es exactamente igual a las ya vistas `scanf()` y `printf()` de las secciones 10.1.1 y 10.1.2, salvo que la entrada y salida con formato es desde y hacia una cadena de caracteres a la que apunta `buffer`. Su sintaxis es:

```
int sprintf(char *buffer, "cadena_de_control", lista_de_argumentos); }
int sscanf(const char *buffer, "cadena_de_control", lista_de_argumentos);}
```

Por ejemplo, podría usarse `sprintf()` para copiar una cadena en otra, de forma similar a `strcpy()`:

```
char *origen = "prueba", destino[30];
sprintf(destino, "%s", origen); /* Equivale a strcpy(destino, origen) */
```

Pero, `sprintf()` da mayor flexibilidad en otro tipo de casos que no sea la simple copia de cadenas. Por ejemplo, si queremos copiar el contenido de `origen`, y añadirle una extensión `.dat`, podría hacerse:

```
sprintf(destino, "\%s\%s", origen, ".dat");
```

Veamos un ejemplo de uso de `sscanf()` para asignar campos de una cadena con formato.

Supongamos que el programa principal recibe un único parámetro consistente en una dirección IP en notación decimal (Ej: `129.34.98.231`), que queremos almacenar en una matriz de 4 caracteres. Para ello deberemos usar la operación `d`, que convierte la cadena de dígitos en el entero que representan, incluyendo caracteres `'.'` en la cadena de control para que se los salte.

Como debemos pasar a `sscanf()` punteros a un entero, usaremos una matriz intermedia de enteros, `ent[]`. No debemos usar la matriz de caracteres, `ip[]`, con un cast `(int *)`, tal y como se halla comentado en el programa, pues cada entero leído se almacenará en 4 bytes a partir de la dirección pasada, con lo que, al escribir el cuarto entero, sobreescribiremos las posiciones `ip[4], ip[5]` e `ip[6]`, que se hallan fuera de la matriz!!!. Finalmente, simplemente copiaremos la matriz intermedia de enteros en la matriz de caracteres.

```
#include <stdio.h>
main(int argc, char *argv[]) {
  int ent[4];
  unsigned char ip[4];

  if (argc != 2) {
    printf("Uso: %s dir_ip\n", argv[0]);
    exit(0);
  }
```

```
  /* NO!!! if (sscanf(argv[1], "%d.%d.%d.%d", (int *)ip, (int *)(ip+1),
                    (int *)(ip+2), (int *)(ip+3)) != 4) */

  if (sscanf(argv[1], "%d.%d.%d.%d", ent, ent+1, ent+2, ent+3) != 4) {
    printf("Dirección IP incorrecta\n");
    exit(0);
  }
  for (int i=0; i<4; i++)
    ip[i] = ent[i];
  printf("%d.%d.%d.%d\n", ip[0], ip[1], ip[2], ip[3]);
}
```

Este ejemplo sirve tan sólo para ilustrar el uso de `scanf()` y posibles problemas derivados, dado que esta misma funcionalidad puede obtenerse mediante el uso de la función `inet_addr()`.

12.2. Funciones de caracteres

El fichero de cabecera `<ctype.h>` contiene un conjunto de funciones para procesar caracteres individualmente.

A excepción de las funciones

```
int tolower(c)
int toupper(c)
```

que convierten un carácter a minúscula y mayúscula, respectivamente, el resto de funciones citadas aquí, que comienzan por `is...`, devuelven un entero que es distinto de cero (TRUE) o cero (FALSE), dependiendo de si el carácter pasado como parámetro cumple o no cierta propiedad.

Dada la claridad del nombre de la función, nos limitaremos a citarlas. Para mayor información puede recurrirse al `man` (ver sección 13.3).

Las funciones booleanas son: `isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit`.

12.3. Utilidades generales

El fichero de cabecera `<stdlib.h>` contiene, además de las funciones para gestión dinámica de memoria vistas en la sección 11, otras funciones de propósito general, de entre las que destacamos las siguientes:

- Terminación de un programa:
  - `void exit(int status)`: Origina una terminación inmediata del programa desde cualquier parte del mismo. Según el argumento `status`, se pueden comunicar las causas de terminación del programa, correctas o no. Implica la limpieza de todos los búferes de E/S y el cierre de todos los ficheros.
  - `int atexit(void (*func)(void))`: Registra la función `func` para que sea ejecutada cuando el programa termina normalmente, tras llegar al final o mediante `exit()`.
- Conversión de cadenas ASCII:

  - `double atof(const char *s)`: Convierte la cadena ASCII `s`, conteniendo sólo dígitos, al número real representado por ésta.
  - `int atoi(const char *s)`: Convierte la cadena ASCII `s`, conteniendo sólo dígitos, al número entero representado por ésta.

- Generación de números pseudoaleatorios:
  - `int rand(void)`: Devuelve un número pseudoaleatorio entre `0` y `RAND_MAX`.
  - `void srand(unsigned int semilla)`: Establece la semilla a partir de la cual se genera la secuencia de números pseudoaleatorios que se extraerán con `rand()`. Esta semilla es, por defecto, 1.

- Búsqueda y ordenación en una matriz `base` de `n` elementos, cada uno de ellos de tamaño `size`, basándose en el valor devuelto por una función `cmp()`, que debe devolver un entero negativo (cero o positivo) si el primer elemento apuntado es menor (igual o mayor) que el segundo:
  - `void qsort(void *base, size_t n, size_t size,`
    `int (*cmp)(const void *, const void *))`:
    Ordena en sentido ascendente la matriz, según el valor devuelto por `cmp()`, usada para comparar dos elementos de la matriz.
  - `void *bsearch(const void *clave, const void *base, size_t n, size_t size,`
    `int (*cmp)(const void *, const void *))`:
    Busca el elemento `clave` en la matriz `base`, ordenada en sentido ascendente. La función `cmp()` se usa para comparar `clave` con cada elemento de la matriz.

- Otras:
  - `int abs(int j)`: Devuelve el valor absoluto de un entero.
  - `int system (const char *string)`: Ejecuta un comando del procesador de comandos (*shell*), contenido en la cadena `string`.

12.4. La librería matemática

Las funciones matemáticas se hallan declaradas en el fichero de cabecera `<math.h>`. Tal y como se comentó en la sección 7.4, las funciones matemáticas no se hallan en la librería estándar C, `libc`, enlazada automáticamente, por lo que si se usan las funciones matemáticas declaradas en `math.h`, debe indicarse al compilador que enlace el ejecutable con la librería matemática, `libm`, añadiendo la opción `-lm` en el comando de compilación. Por ejemplo:

```
> gcc prueba.c -o prueba -lm
```

Todas las funciones devuelven un `double`, los parámetros `x` e `y` son también de tipo `double`, y `n` es de tipo `int`. Las funciones trigonométricas operan con radianes.

- `acos(x)`: Función arco coseno. Resultado entre 0 y π.
- `asin(x)`: Función arco seno. Resultado entre $-\pi/2$ y $\pi/2$.

- `atan(x)`: Función arco tangente. Resultado entre $-\pi/2$ y $\pi/2$.
- `atan2(x, y)`: Arco tangente de `x/y`. Resultado entre $-\pi$ y π.
- `ceil(x)`: Número real equivalente al menor entero mayor o igual que `x`.
- `cos(x)`: Función coseno.
- `cosh(x)`: Función coseno hiperbólico.
- `exp(x)`: Función exponencial (e^x).
- `fabs(x)`: Valor absoluto.
- `floor(x)`: Número real equivalente al mayor entero menor o igual que `x`.
- `fmod(x, y)`: Resto de la división entera `x/y` con igual signo que `x`.
- `frexp(x, int *exp)`: Divide el real `x` en mantisa (devuelta por la función) y exponente (devuelto en *exp).
- `ldexp(x, n)`: Calcula $x \cdot 2^n$.
- `log(x)`: Logaritmo natural.
- `log10(x)`: Logaritmo base 10.
- `modf(x, double *ent)`: Divide el real `x` en su parte entera (devuelta en *ent) y su parte fraccionaria (devuelta por la función).
- `pow(x)`: Calcula x^y.
- `sin(x)`: Función seno.
- `sinh(x)`: Función seno hiperbólico.
- `sqrt(x)`: Raíz cuadrada.
- `tan(x)`: Función tangente.
- `tanh(x)`: Función tangente hiperbólica.

12.5. Señales

El fichero `<signal.h>` facilita una serie de funciones para manejar eventos excepcionales (interrupciones) que aparecen durante la ejecución de un programa. Estas señales pueden provenir de errores del programa, como una referencia inválida a memoria, o de señales de interrupción de fuentes externas, como un temporizador, el teclado o un comando `kill` (en la sección 14 se describe este comando).

12.5.1. Manejo de señales: Instalación de un *handler*

Todas estas señales, identificadas por un número, son atendidas por una función denominada *handler* (rutina de atención), de tipo `void`, y que posee un parámetro de tipo `int`, a través del cual se pasa el número que identifica a la señal atendida. En `<signal.h>` se halla la definición de este tipo de funciones, con nombre `sighandler_t`:

```
typedef void (*sighandler_t)(int);
```

Para cambiar el *handler* de una señal se usa la función `signal()`:

```
sighandler_t signal(int signum, sighandler_t handler)
```

donde `signum` es el número de la señal (que será pasada a la función de atención), y `handler` es un puntero a la nueva función que atenderá la señal. Si hay algún error, `signal` devuelve `SIG_ERR`, en caso contrario devuelve un puntero al *handler* anterior, por si se desea restaurar posteriormente.

Debe notarse que tras la ejecución de nuestro *handler*, el sistema restaurará de nuevo el *handler* por defecto. Por ello, si se desea que esto no ocurra debe llamarse de nuevo a la función `signal()` dentro de nuestro *handler*.

Si `handler` es `SIG_IGN` se ignorará la señal, y si es `SIG_DFL` se restaura la señal a su comportamiento por defecto. Se muestra a continuación la lista de las señales más importantes (macros definidas en `<signal.h>`), excepto las usadas por los temporizadores, que se verán en la siguiente sección:

SEÑAL	ACCIÓN	COMENTARIOS
`SIGINT`	A	Interrupción desde teclado [CTRL-C]
`SIGILL`	A	Instrucción ilegal
`SIGFPE`	C	Excepción de coma flotante (error aritmético)
`SIGSEGV`	C	Referencia a memoria inválida
`SIGBUS`	A	Error de bus
`SIGABRT`	C	Terminación anormal
`SIGTRAP`	CD	*Breakpoint* mediante un depurador
`SIGTERM`	A	Terminación enviada por el comando `kill`
`SIGUSR1`	A	Señal definida por el usuario num. 1
`SIGUSR2`	A	Señal definida por el usuario num. 2
`SIGSTP`	D	Suspensión desde teclado [CTRL-Z]
`SIGCONT`		Continua si parado
`SIGSTOP`	DEF	Suspensión irrevocable
`SIGKILL`	AEF	Terminación irrevocable

Las letras de acción indican que:

- A: La acción por defecto es terminar el programa.
- B: La acción por defecto es ignorar la señal.
- C: La acción por defecto es generar un `core dump`.
- D: La acción por defecto es parar el programa.
- E: La señal no será capturada.

- F: La señal no será ignorada.

La función

```
char *strsignal(int sig)
```

en `<string.h>` puede usarse para obtener una descripción de la señal `sig`.
También resulta útil la función

```
void psignal(int sig, const char *s)
```

en `<signal.h>}` para mostrar mensajes de error relativos a una señal. Su funcionamiento equivale a:

```
fprintf(stderr, "%s : %s\n", s, "descripción de la señal sig")}.
```

Veamos un ejemplo sencillo del uso de señales mediante una señal de usuario.

```
main () {
 void manejador(int sig);

 if (signal(SIGUSR1, manejador) == SIG_ERR)
   {
     perror("Error en signal()");
     exit(1);
   }
 for (;;) ;
}

void manejador(int sig) {
 printf("Se ha recibido la señal: %s\n", strsignal(sig));
}
```

Suponiendo que el programa se llama `ejemplo`, la ejecución sería:

```
> ejemplo &
  [1] 2056  /* Es el identificador del proceso "ejemplo" */
> kill -USR1 2056
> Se ha recibido la señal: User defined signal 1
```

Si se ejecuta de nuevo el comando `kill -USR1 2056`, el programa terminará, dado que el sistema habrá restaurado el *handler* por defecto. Para que ello no ocurra, nuestro *handler* debería ser:

```
void manejador(int sig) {
 signal(SIGUSR1, manejador);   /* Restauramos este handler de nuevo */
 printf("Se ha recibido la señal: %s\n", strsignal(sig));
}
```

12.5.2. Posibles problemas con señales

A diferencia de lo que ocurre con los *handler* por defecto, cuando se instala un *handler* mediante una llamada a `signal()`, si la ejecución de éste interrumpe una llamada al sistema (típicamente, de entrada/salida) ésta no será reiniciada tras finalizar la atención de la señal. Para cambiar este comportamiento debe utilizarse la función

```
int siginterrupt(int sig, int flag)
```

Si el argumento `flag` es `0`, las llamadas al sistema interrumpidas por la señal `sig` serán reiniciadas. Si es `1`, y todavía no se han transferido datos, la llamada al sistema interrumpida devolverá `-1` y la `errno` se pone a `EINTR`. La función devuelve `0` normalmente y `-1` si `sig` es un número de señal inválido.

Resulta fundamental que las funciones *handler* no hagan uso de funciones no reentrantes, como es el caso de las funciones de librería que manipulan datos estáticos (`malloc, realloc, calloc`.

12.5.3. Generación de señales

La siguientes funciones, cuyos prototipos se hallan en `<signal.h>`, pueden ser utilizadas para generar señales:

- `int kill(pid_t pid, int sig)`: Envía la señal de número `sig` al proceso con identificador `pid`. Posee una funcionalidad idéntica a la del comando `kill` del sistema, utilizado en el ejemplo anterior (ver sección 14).
- `int raise (int sig)`: Envía la señal de número `sig` al proceso actual. Equivale a `kill(getpid(),sig)`, donde la función `pid_t getpid(void)` devuelve el identificador del proceso llamante.

12.5.4. Temporizadores

El sistema permite a cada proceso utilizar tres temporizadores periódicos, cada uno en un dominio temporal distinto. Cuando cualquier temporizador expira, envía cierta señal al proceso y el temporizador, potencialmente, reinicia. Los tres temporizadores son:

- `ITIMER_REAL`: Decrementa en tiempo real y genera la señal `SIGALRM` tras su vencimiento.
- `ITIMER_VIRTUAL`: Decrementa sólo cuando el proceso se está ejecutando y genera la señal `SIGVTALRM` tras su vencimiento.
- `ITIMER_PROF`: Decrementa tanto cuando el proceso se está ejecutando como cuando el sistema está realizando alguna tarea para el proceso. Genera la señal `SIGPROF` tras su vencimiento.

Los valores del temporizador se definen mediante las siguientes estructuras:

```
struct itimerval {
  struct timeval it_interval; /* siguiente valor */
  struct timeval it_value;    /* valor actual */
};

struct timeval {
  long tv_sec;                /* segundos      */
  long tv_usec;               /* microsegundos */
};
```

La siguientes funciones pueden ser utilizadas para manejar los temporizadores del sistema:

- `int setitimer(int timer, const struct itimerval *value, struct itimerval *ovalue)`

  A través de la estructura `value` se fija el valor del temporizador mediante `it_value`, y el de reinicio tras el vencimiento mediante `it_interval`. El temporizador especificado en `timer` decrementará desde `it_value` hasta cero, generará la señal y reiniciará con el valor `it_interval`. Si `it_value` es cero el temporizador parará. De la misma forma, si `it_interval` es cero, el temporizador parará tras su vencimiento. Si `ovalue` no es cero (`NULL`), la función devuelve en esta estructura los valores antiguos del temporizador.

  Los temporizadores nunca expiran antes del plazo señalado, pero pueden, en cambio, hacerlo algún tiempo pequeño después, dependiendo de la resolución del reloj del sistema (habitualmente, 10 msgs.). Si el temporizador expira mientras se está ejecutando el proceso (siempre cierto para `ITIMER_VIRTUAL`), la señal será entregada inmediatamente. En otro caso, será entregada unos instantes después, dependiendo de la carga del sistema.

  Así, bajo Linux, donde sólo se permite un evento pendiente por señal, ante una carga anormalmente pesada del sistema, podría ocurrir que el temporizador `ITIMER_REAL` venciese de nuevo antes de que se hubiera entregado la señal correspondiente al vencimiento anterior. En este caso, la segunda señal se perderá. Veamos un ejemplo donde se instala un temporizador que ejecuta la función `handler_tempo()` cada 5 segundos.

```
 main() {
   struct itimerval tempo;
   void handler_tempo(int sig);
   tempo.it_interval.tv_sec = 5;         /* 5 segundos */
   tempo.it_interval.tv_usec = 0;
   signal(SIGALRM, handler_tempo);      /* Instalamos handler */
   setitimer(ITIMER_REAL, &tempo, NULL); /* Lanzamos temporizador */
   ...
   tempo.it_value.tv_sec = 0;
   tempo.it_value.tv_usec = 0;
   setitimer(ITIMER_REAL, &tempo, NULL); /* Paramos temporizador */
}

void handler_tempo(int sig) {
  signal(SIGALARM, handler_tempo); /* Restauramos este handler de nuevo */
  ...
}
```

- `unsigned int alarm(unsigned int seconds)` (en `<unistd.h>`):

  Esta función es una simplificación de la anterior `setitimer()`, cuando no se desea mucha resolución en la temporización, es decir, cuando sólo se especifican segundos. La señal generada es igualmente `SIGALRM`, y usa también el temporizador `ITIMER_REAL`, por lo que interferirá con llamadas a `setitimer()` para dicho temporizador. De hecho, si `seconds` vale cero éste se parará. Así, el ejemplo anterior es equivalente al siguiente:

```
main() {
 void handler_tempo(int sig);

 signal(SIGALRM, handler_tempo); /* Instalamos handler */
 alarm(5);                       /* Lanzamos temporizador */
 ...
 alarm(0);                       /* Paramos temporizador */
}

void handler_tempo(int sig) {
 signal(SIGALARM, handler_tempo); /* Restauramos este handler de nuevo */
 ...
}
```

- `unsigned int sleep(unsigned int seconds)` (en `<unistd.h>`):

  Esta función hace que el proceso actual se duerma hasta que hayan transcurrido los segundos indicados o llegue una señal no ignorada. La señal generada es `SIGALRM`, usando también el temporizador `ITIMER_REAL`, por lo que interferirá con llamadas a `setitimer()` y a `alarm()`.

- `int pause(void)` (en `<unistd.h>`):

  Esta función hace que el proceso llamante se duerma hasta que se reciba una señal no ignorada.

12.6. Fecha y hora

El fichero de cabecera `<time.h>` contiene toda una serie de funciones relacionadas con la fecha y hora. En dicho fichero se definen también tres tipos importantes:

- `struct tm` es una estructura con los componentes de una fecha.

```
struct tm {
 int     tm_sec;         /* seconds */
 int     tm_min;         /* minutes */
 int     tm_hour;        /* hours */
 int     tm_mday;        /* day of the month */
 int     tm_mon;         /* month */
 int     tm_year;        /* year */
 int     tm_wday;        /* day of the week */
 int     tm_yday;        /* day in the year */
 int     tm_isdst;       /* daylight saving time */
};
```

- `clock_t` es un tipo entero definido según la implementación para representar el tiempo en el sistema. Para obtener el número de segundos debe dividirse por `CLOCKS_PER_SEC`.

- `time_t` es un tipo entero definido según la implementación para representar fechas.

Las funciones más importantes son:

- `clock_t clock(void)`: Devuelve una aproximación del tiempo de CPU usado por un proceso. Para obtener el número de segundos debe dividirse por `CLOCKS_PER_SEC`.
- `time_t time(time_t *t)`: Devuelve el tiempo transcurrido en segundos desde el 1 de Enero de 1970.
- `double difftime(time_t time1, time_t time0)`: Devuelve la diferencia, en segundos, entre los instantes `time1` y `time0`.

12.7. Funciones útiles en la red

Dado que las distintas máquinas en una red pueden usar convenios de representación distintos para los enteros (extremista mayor o menor), debe definirse un convenio común para la transmisión a través de la red.

En la red, como por ejemplo en Internet, se adopta el convenio extremista mayor, es decir, el byte más significativo se almacena en la posición más baja de memoria, siendo el primero en ser transmitido. De esta forma, una trama o paquete de datos será transmitida de izquierda a derecha, es decir, desde la posición de memoria más baja hacia la más alta.

En cambio, las máquinas basadas en procesadores de Intel, como es el caso de los laboratorios, usan el convenio extremista menor. Por tanto, si se almacena un entero en una trama o se lee un entero de una trama, debemos realizar una conversión de convenio para que se transmita adecuadamente, en el primer caso, o se almacene adecuadamente en memoria en el segundo caso. Para ello, pueden usarse las siguientes funciones cuyos prototipos se hallan en el fichero `<netinet/in.h>`.

```
unsigned long int htonl(unsigned long int hostlong)
unsigned short int htons(unsigned short int hostshort)
unsigned long int ntohl(unsigned long int netlong)
unsigned short int ntohs(unsigned short int netshort)
```

Estas funciones convierten un entero `short` de 2 bytes (la función termina en `s`) o `long` de 4 bytes (la función termina en `l`) desde el convenio de la red (la función comienza por `n`) al de la máquina (`toh`), o viceversa (`ton`).

Otras funciones útiles son las de conversión de direcciones IP entre la notación decimal estándar (con puntos) y su representación binaria (4 bytes), contenidas en `<arpa/inet.h>`. Aunque hay varias funciones con propósitos similares, las más útiles son:

- `unsigned long int inet_addr(const char *cp)`: Convierte la dirección IP (en notación decimal estándar) contenida en la cadena de caracteres `cp`, en un entero de 4 bytes representado en el convenio de la red (extremista mayor), de forma que puede ser copiado directamente en la trama a transmitir. Si se desea la representación de la máquina (extremista menor) debe usarse la función

  ```
  unsigned long int inet_network(const char *cp)}
  ```

Por ejemplo, supongamos que la dirección IP, en notación decimal, es el primer parámetro de un programa:

```
struct paquete {
...
int IP;
...
} trama;

trama.IP = inet_addr(argv[1]);
```

Si el campo correspondiente no pudiese ser un entero, sino una cadena, debido a la existencia de huecos (ver sección 6.1), podría hacerse:

```
struct paquete {
...
char IP[4];
...
} trama;
int dir;

dir = inet_addr(argv[1]);
memcpy(trama.IP, &dir, 4);
```

- `char *inet_ntoa(struct in_addr in)`: Toma una dirección IP de 32 bits en convenio de red (extremista mayor), y devuelve una cadena que contiene la correspondiente dirección IP en notación decimal. La estructura `in_addr`, utilizada, entre otras, en esta función se halla definida en `<netinet/in.h>`:

```
struct in_addr {
  unsigned long int s_addr;
}
```

Para consultar funciones relacionadas, puede hacerse uso del manual en línea, por ejemplo, mediante el comando:

```
> man inet_addr
```

13. Herramientas útiles para la programación en C

13.1. El editor emacs

El editor más ampliamente utilizado en entornos UNIX/LINUX es, sin duda, el `emacs` de GNU, dada su enorme potencia y flexibilidad para adaptarse a las necesidades del usuario y las de los distintos lenguajes de programación.

Su fichero de configuración es `.emacs`, escrito en lenguaje LISP y, entre otras cosas, puede contener definiciones de las teclas para comandos rápidos. Por ejemplo, las sentencias

```
(global-set-key [f1] 'save-buffer)
(global-set-key [C-f9] 'goto-line)
```

define las teclas F1 y CTRL-F9 con los comandos `save-buffer` y `goto-line`. Todos los comandos pueden ser también invocados desde el mini-búfer (bajo la línea negra en la parte inferior de la pantalla) mediante la combinación de teclas ESC-x. Un fichero `.emacs` para C, como el facilitado para el laboratorio de RST, podría contener las siguientes asociaciones entre teclas rápidas y comandos útiles:

```
[C-f1] isearch-forward -> Búsqueda hacia adelante
[C-f2] isearch-backward -> Búsqueda hacia atrás
[C-f3] 'query-replace   -> Buscar y reemplazar (ESPACIO acepta, BACKSPACE
ignora)
[C-f4] 'c-mark-function -> Marca el bloque correspondiente a la función
actual
[C-f5] 'kill-region -> Corta la región marcada y la deja en el portapapeles
[C-f6] 'hilit-yank -> Pega el contenido del portapapeles (resultado de lo
                   anterior o de haber marcado con el ratón
[C-f7] 'c-indent-defun -> Indenta una región o fichero escrito en C
[C-f8] 'comment-region -> Comenta una región
[C-f12] 'save-buffers-kill-emacs -> Salva ficheros (búferes) y sale
[f1] 'keyboard-quit -> Aborta comando anterior
[f2] 'save-buffer -> Salva fichero (búfer)
[f3] 'find-file -> Carga fichero
[f4] 'write-file -> Guarda fichero como ...
[f5] 'insert-file -> Inserta el contenido de un fichero
[f8] 'undo -> Deshace la operación o operaciones anteriores
[f9] 'goto-line -> Va al número de línea indicado
[f10] 'auto-fill-mode -> Alterna entre cambio de línea automático y manual
[f11] 'beginning-of-buffer -> Inicio de fichero
[f12] 'end-of-buffer -> Fin de fichero
[delete] 'delete-char -> Borra carácter bajo cursor
[home] 'beginning-of-line -> Principio de línea
[end] 'end-of-line -> Fin de línea
[prior] 'scroll-down -> Página abajo
[next] 'scroll-up) -> Página arriba
```

No obstante, por defecto, la mayor parte de comandos emacs se hallan asociados a alguna combinación de teclas. Por ejemplo, el comando `save-buffer` corresponde a CTRL-x CTRL-s. También, desde el menú superior se puede acceder a una parte de estos comandos. Para obtener un listado de estas asociaciones, hacer desde el menú superior `Help ->Describe ->List Key Bindings` (o CTRL-h b).

Si se dispone de varias ventanas sobre el editor, debido a la ejecución de un comando como el anterior, o por que se ha dividido ésta mediante CTRL-x 2, puede eliminarse la actual (en la que se halla el cursor) mediante CTRL-x 0, o mantener sólo la actual mediante CTRL-x 1.

13.2. Compilación con `make`

Como ya se dijo en la sección 2, habitualmente los grandes programas suelen dividirse en varios ficheros fuente (módulos) y otros ficheros de cabecera.

Puede ocurrir que distintos módulos deban ser compilados con opciones especiales y con ciertas definiciones y declaraciones. Además, el código puede requerir ciertas librerías bajo el control de opciones particulares. Desafortunadamente, es muy fácil que el programador olvide las dependencias entre los distintos módulos, o entre los módulos y los ficheros de cabecera, o qué ficheros fueron modificados recientemente, o la secuencia exacta de operaciones necesarias para obtener una nueva versión del programa. Por ejemplo, olvidarse de compilar una función que ha sido cambiada o que usa declaraciones modificadas usualmente dará lugar a un programa que no funcionará correctamente, y a un error muy difícil de hallar. Por otro lado, recompilar siempre todo, aunque seguro, es una pérdida de tiempo.

Los sistemas UNIX/LINUX proporcionan una herramienta, denominada `make`, que automatiza prácticamente todas las tareas relacionadas con el desarrollo y mantenimiento de un programa. `make` facilita un mecanismo simple de mantener actualizado (*up-to-date*) un programa.

Para ello, debe indicarse a `make` la secuencia de comandos necesaria para obtener ciertos ficheros, y la lista de dependencias entre ficheros. Siempre que se produce un cambio en cualquier parte del programa, `make` obtendrá los ficheros de forma simple, correcta y con el mínimo esfuerzo.

La operación básica de `make` es actualizar un programa objetivo (*target*), asegurando que todos los ficheros de los que depende existan y se hallen actualizados. Para ello, el programa `make` hace uso de las reglas y órdenes contenidas en el fichero `Makefile` o `makefile`.

Este fichero puede contener:

- **Comentarios**: Cualquier línea que comience con `#` será ignorada por `make`.
- **Variables**: Cualquier identificador, generalmente en mayúsculas, seguido de un signo `=` y una cadena es una variable. Para acceder a su valor se usa el símbolo `$` y el nombre de la variable entre paréntesis. Por ejemplo,

```
FUENTES = prueba.c
...
 cc -c $(FUENTES)
```

- **Reglas**: Las reglas pueden ser de dos tipos:
  - De dependencia: Establecen las dependencias entre ficheros, y se especifican designando los módulos objetivo a la izquierda, seguidos de dos puntos (`:`) y de los módulos de los cuales depende el objetivo. A este tipo de reglas puede seguir una órden.
  - De inferencia: Se diferencian de las anteriores en que tan sólo se especifica el objetivo, seguido de dos puntos (`:`), y debe ir seguido obligatoriamente de una órden en la siguiente línea. Sirven para especificar órdenes por medio de las cuales se obtiene el objetivo especificado.

    `make` posee cierto conocimiento interno referente al desarrollo de programas y sabe que los archivos con extensión `.c` o `.cc` son archivos

fuente C, con extensión `.o` módulos objeto, etc. Por ello, se permite especificar reglas de inferencia explícita (`.c.o:`) de cómo convertir ficheros fuente C a objeto.

En cualquier tipo de regla, los objetivos también pueden ser cualquier identificador que posteriormente pueda ser usado en la línea de comandos como parámetro del `make`. El objetivo por defecto cuando se ejecuta `make` sin parámetros es `all`.

- **Órdenes**: Una regla de dependencia puede venir seguida por las órdenes que deben ejecutarse si uno o más módulos dependientes (los de la derecha) se ha modificado. En el caso de una regla de inferencia, la órden es obligatoria y se ejecuta siempre que se necesite el objetivo.

  Las líneas de órdenes deben hallarse separadas desde el margen izquierdo por una tabulación (no valen espacios). En órdenes correspondientes a reglas de inferencia explícita (por ejemplo, `.c.o:`) suelen utilizarse las siguientes macros predefinidas:

  - `$@`: Es el nombre del objetivo (un `.o`, en este caso)
  - `$<`: Representa el fichero modificado (un `.c` en este caso) del cual depende el objetivo.

Para ilustrar su uso veamos el ejemplo de fichero `Makefile` entregado para la práctica del laboratorio de RST.

```
##### Variables que definen el compilador, flags de compilación,
##### directorios include y de librerías, y librerías a enlazar
CXX = g++
CXXFLAGS = -g
PATHLIBS =  -L /home/clave/labs/tlm/qt/lib   -L/usr/X11R6/lib
LIBS = -lqt -lpthread -lXext
PATHINC = -I/home/clave/labs/tlm/qt/include
##########################################
#### Variables que definen módulos y ficheros
PROGRAMA = arp
CABECERAS = comunicacion.h clase.hh
FUENTES = arp.cc
OBJETOS = comunicacion.o clase.o moc_clase.o
OBJETOSNUEVOS = arp.o
################ REGLAS INFERENCIA #################
### Como generar un objeto .o a partir de un fuente .c
### $@ representa al .o y $< al .c
.c.o:
	$(CXX) -c $(CXXFLAGS) $(PATHINC) -o $@ $<

### Como generar un objeto .o a partir de un fuente .cc
.cc.o:
	$(CXX) -c $(CXXFLAGS) $(PATHINC) -o $@ $<

### Objetivo "all" (por defecto)
all: $(PROGRAMA)
```

```
### Como obtener "arp", definido anteriormente
### como objetivo "all"
$(PROGRAMA):
   $(CXX) -o $(PROGRAMA) $(OBJETOS) \
   $(OBJETOSNUEVOS) $(PATHLIBS) \
   $(PATHINC) $(LIBS)

### Objetivo clean. Se ejecutará con > make clean
clean:
   -rm -f $(OBJETOSNUEVOS) *~ core

############## REGLAS DE DEPENDENCIA#################
### Si se modifica arp.o o cualquier otro en OBJETOS
### se buscará (generará) el objetivo "arp"
arp: arp.o $(OBJETOS)

### Si se modifica arp.cc o los ficheros de cabecera
### se buscará (generará) el objetivo "arp.o"
arp.o: arp.cc $(CABECERAS)
```

Una forma de determinar dependencias entre un fichero `ejemplo.c` y otros módulos o ficheros de cabecera, puede utilizarse el comando

```
> gcc -MM ejemplo.c
```

que nos mostrará las reglas de dependencia.

Si el programa `make` detecta que el objetivo está actualizado simplemente generará un mensaje informando de ello: `It's up-to-date`.

13.3. El manual en línea: `man`

El manual en línea, `man`, es una herramienta de documentación de gran utilidad pues permite obtener información, no sólo de todos los comandos y programas existentes en un sistema, sino de todas las funciones C contenidas en las distintas librerías existentes.

Para obtener información acerca de un comando, programa, o función de nombre `name`, basta ejecutar:

```
> man [seccion] name
```

Cuando distintos comandos o funciones poseen el mismo nombre, cada uno de ellos se halla en una sección. Por ello, cuando es así debe indicarse la sección a que nos referimos. No obstante, por defecto, se presenta la primera.

Para buscar todas las secciones existentes para un nombre `name`, debe hacerse:

```
man -k name
```

13.4. El depurador: ddd

El depurador más ampliamente utilizado en entornos UNIX/LINUX es el gdb de GNU. No obstante, dado que su uso es a través de una línea de comandos, habitualmente se utilizan otras programas que facilitan una interfaz gráfica mucho más amigable y fácil de usar. Este es el caso de los programas xxgdb, xwpe y ddd, fundamentalmente de éste último.

Para poder usar el depurador, debe haberse compilado el programa con la opción -g (ver sección 2) para que se introduzca información simbólica. La llamada a estos programas se realiza usando como argumento el nombre del programa ejecutable:

```
> ddd ejecutable &
```

Queda fuera del ámbito de este documento el describir detalladamente el uso de estos depuradores. Por otro lado, su uso es muy similar al de otros depuradores ya utilizados y, para más información, puede acudirse a la ayuda del propio programa o al man.

14. Comandos Unix/Linux más importantes

Se citan a continuación los principales comandos del sistema, acompañados de una breve descripción del mismo. Para una información más detallada sobre cada uno de ellos, se remite al lector al manual en línea, man.

cp: Copiar ficheros

mv: Mover o renombrar ficheros

ls: Listar contenido de directorio. La opción -l incluye detalles, y -a muestra ficheros ocultos (que empiezan con .)

rm: Borrar ficheros

more: Mostrar el contenido de un fichero página a página (ESPACIO mueve a la página siguiente, "b"mueve a la página anterior, ENTER mueve una línea, "q"termina el comando, y /"permite buscar un patrón).

mkdir: Crear directorio

rmdir: Eliminar directorio

cd: Cambiar de directorio de trabajo

ps: Lista procesos, e identificadores, en ejecución del usuario actual

pwd: Muestra directorio de trabajo actual

xterm: Abre un terminal X

grep: Imprime lineas de un fichero con un patrón determinado

telnet: Para conectarse a una máquina IP desde otra, que sirve como terminal virtual.

`logout`: Abandonar sesión

`startx`: Lanzar entorno de ventanas X-Window

Debe notarse que un comando o programa puede ser lanzado en *foregorund* (interactivo) o *backgorund* (no interactivo). En el modo *foregorund*, que es el modo por defecto, el sistema no devuelve el control hasta que el programa termina. Un proceso en *foregorund* puede finalizarse mediante CTRL-C, o suspenderse mediante CTRL-Z. Tras ser suspendido, puede volver a lanzarse al *foregorund* mediante el comando `fg` o al *backgorund* mediante el comando `bg`.

Para lanzar el programa o comando directamente en *backgorund* debe añadirse al final del comando el símbolo "`&`". Para terminar un proceso en *backgorund* debe usarse el comando de generación de señales `kill` (ver sección 12.5.1):

```
> kill [-señal] pid
```

donde `señal` es cualquiera de las vistas en la sección 12.5.1 (omitiendo `SIG`), y `pid` es el identificador del proceso, mostrado entre corchetes si se lanza al *backgorund*, o obtenido mediante el comando `ps`. La señal por defecto es `TERM`, que finalizará un proceso que no la capture. Para terminar un proceso de forma irrevocable debe usarse `KILL` (también puede usarse el número, en este caso `kill -9`). Mediante `kill -l` se obtiene un listado de las señales.

Para simplificar el acceso a unidades de disquete, se recomienda el uso de las herramientas `mtools`, que permiten que un sistema Linux acceda a disquetes formateados en DOS. Para referirse a la unidad de disquete se usa `a:`

Los principales comandos son:

`mcd`: Cambiar de directorio de trabajo en a:

`mcopy`: Copiar a/desde la unidad de disquete

`mdel`: Borrar fichero en a:

`mdeltree`: Eliminar un directorio y su contenido en a:

`mdir`: Listar contenidos de a:

`mformat a:` Formateo rápido DOS

`mmd`: Crear directorio en a:

`mmove`: Mover un fichero o directorio en a:

`mrd`: Eliminar directorio en a:

`mren`: Renombrar un fichero o directorio en a:

`mtype`: Mostrar el contenido de un fichero

Debe notarse que los nombres de los ficheros copiados a la unidad de disquete pueden diferir de los usados en el sistema Linux (por ejemplo, usar mayúsculas), dada la diferencia a este respecto entre ambos sistemas.

Cabe destacar también una facilidad incluida en Linux, como es la edición de comandos. Con las flechas arriba y abajo podemos movernos por la lista de comandos anteriores. CTRL-a nos lleva al inicio del comando y CTRL-e al

final. Las flechas izqda. y dcha. nos permiten también movernos por el comando. Finalmente, el tabulador tiene la función Autocompletar, tanto para comandos o programas (la primera palabra de la línea) como para ficheros. Si hay más de una opción se mostrarán éstas. Esta misma facilidad de edición de comandos y uso del tabulador puede usarse en el minibúfer de una ventana `emacs`.