CAPÍTULO 1

El lenguaje de programación C

1.1 INTRODUCCIÓN

C es un lenguaje de programación de alto nivel desarrollado por Dennis Ritchie para codificar el sistema operativo UNIX. Las primeras versiones de UNIX se escribieron en ensamblador, pero a partir de 1973 pasaron a escribirse en C. Actualmente, sólo un pequeño porcentaje del núcleo de UNIX se sigue codificando en ensamblador; en concreto aquellas partes íntimamente relacionadas con el hardware. Todas las órdenes y aplicaciones estándar que acompañan al sistema UNIX también están escritas en C.

El lenguaje posee instrucciones que constan de términos que se parecen a expresiones algebraicas, además de ciertas palabras clave inglesas como `if`, `else`, `for`, `do` y `while`. En este sentido, C recuerda a otros lenguajes de programación estructurados como Pascal y Fortran.

El lenguaje C presenta las siguientes características:

- Se puede utilizar para programación a bajo nivel cubriendo así el vacío entre el lenguaje máquina y los lenguajes de alto nivel más convencionales.
- Permite la redacción de programas fuentes muy concisos, debido en parte al gran número de operadores que incluye el lenguaje.
- Tiene un repertorio de instrucciones básicas relativamente pequeño, aunque incluye numerosas funciones de biblioteca que mejoran las instrucciones básicas. Además los usuarios pueden escribir bibliotecas adicionales para su propio uso.
- Los programas escritos en C son muy portables. C deja en manos de las funciones de biblioteca la mayoría de las características dependientes de la computadora. De esta forma, la mayoría de los programas en C se puede compilar y ejecutar en muchas computadoras diferentes sin tener que realizar en la mayoría de los casos ninguna modificación en los programas.

- Los compiladores de C son frecuentemente compactos y generan programas objeto que son pequeños y muy eficientes.

1.2 CICLO DE CREACIÓN DE UN PROGRAMA

Un *compilador* es un programa que toma como entrada un texto escrito en un lenguaje de programación de alto nivel, denominado *fuente* y da como salida otro texto en un lenguaje de bajo nivel (ensamblador o código máquina), denominado *objeto*. Asimismo, un *ensamblador* es un compilador cuyo lenguaje fuente es el lenguaje ensamblador.

Un compilador no es un programa que funciona de manera aislada, sino que normalmente se apoya en otros programas para conseguir su objetivo: obtener un programa ejecutable a partir de un programa fuente en un lenguaje de alto nivel. Algunos de esos programas son:

- *El preprocesador*. Se ocupa (dependiendo del lenguaje) de incluir ficheros, expandir macros, eliminar comentarios y otras tareas similares.
- *El enlazador (linker)*. Se encarga de construir el fichero ejecutable añadiendo al fichero objeto generado por el compilador las cabeceras necesarias y las funciones de librería utilizadas por el programa fuente.
- *El depurador (debugger)*. Permite, si el compilador ha generado adecuadamente el programa objeto, seguir paso a paso la ejecución de un programa.
- *El ensamblador*. Muchos compiladores en vez de generar código objeto, generan un programa en lenguaje ensamblador que debe convertirse después en un ejecutable mediante un programa ensamblador.

A la hora de crear un programa en C, se ha de empezar por la edición de un fichero de texto estándar que va a contener el código fuente escrito en C. Este fichero se nombra, por convenio, añadiéndole la extensión `.c`. Se va suponer en lo que resta de sección que dicho fichero se llama `prog.c`. Si se utiliza el editor `vi` disponible en UNIX, la forma de editar el programa desde la línea de comandos del terminal (`$`) es:

```
$ vi prog.c
```

Los compiladores de C más utilizados son el `cc` y el `gcc` que se encargan de generar el fichero ejecutable a partir del fichero fuente escrito en C. Para invocarlo, desde la línea de comandos del terminal se teclea la orden:

```
$ gcc prog.c
```

Si no existen errores de compilación está orden se ejecutará correctamente generando como resultado el fichero ejecutable `a.out`.

Si se quiere personalizar el nombre del fichero de salida se debe escribir la orden

```
$ gcc –o nombre_ejecutable prog.c
```

De esta manera se creará un programa ejecutable con nombre `nombre_ejecutable`.

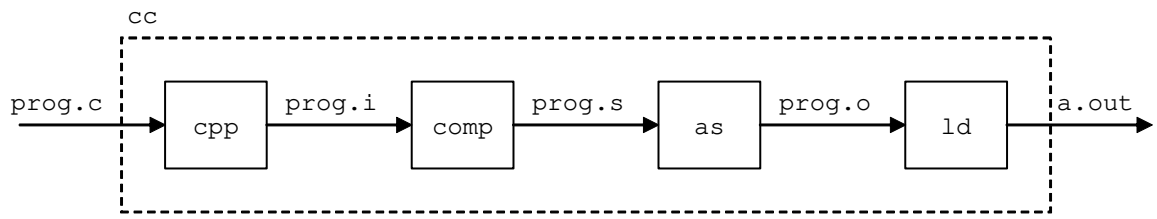

Figura 1.1: Fases del proceso de compilación

Con respecto a `cc` comentar que en realidad no es el compilador sino una interfaz entre el usuario y los programas que intervienen en el proceso de generación de un programa ejecutable (ver Figura 1.1). Dichos programas son:

- El *preprocesador* `cpp`, que genera un archivo con extensión `*.i`.
- El *compilador* `comp`, que genera un archivo con extensión `*.s` que contiene código fuente ensamblador
- El *ensamblador* `as`, que genera un archivo con extensión `*.o` que contiene código objeto.
- El *enlazador* `ld`, que genera el programa ejecutable con extensión `*.out` a partir de ficheros con código objeto (`.o`) y bibliotecas (`.a`).

1.3 ESTRUCTURA DE UN PROGRAMA EN C.

Todo programa C consta de uno o más módulos llamados *funciones*. Una de estas funciones es la función principal que se llama `main`. El programa siempre comenzará por la ejecución de la función `main`, la cual puede acceder a las demás funciones. Las definiciones de las funciones adicionales se deben realizar aparte, bien precediendo o bien siguiendo a `main`. De forma general, se puede afirmar que la estructura de un programa en C es la que se muestra en el Cuadro 1.1.

```
# Directivas del preprocesador.
Definición de variables globales.
Definición prototipo de la función 1
.
.
Definición prototipo de la función N

Función main()
{
      Definición de variables locales
      Código
}
Funcion1(parámetros formales)
{
      Definición de variables locales
      Código
}
.
.
FuncionN(parámetros formales)
{
      Definición de variables locales
      Código
}
```

Cuadro 1.1: Estructura general de un programa en C

En primer lugar, se escriben las *directivas del preprocesador*, que son órdenes que ejecuta el preprocesador para generar el fichero con el que va a trabajar el procesador. Dos son las directivas más utilizadas:

- `#include <xxxx.h>`. Que se emplea para indicar al compilador que recupere el código de un *fichero de cabecera* `xxxx.h` donde están identificadores, constantes, variables globales, macros, prototipos de funciones, etc. La utilización de los archivos de cabecera permite tener las declaraciones fuera del programa principal. Esto implica una mayor modularidad.

- `#define`. Que se emplea para declarar identificadores que van a ser sinónimos de otros identificadores o constantes. También se emplea para declarar *macros.*

En segundo lugar, se escriben las *declaraciones de las variables globales* del programa, en el caso de que existan, que pueden ser utilizadas por todas las funciones del mismo.

En tercer lugar se escriben la *declaración de los prototipos* de las N funciones que se vayan a utilizar en el programa (salvo `main`). En cuarto lugar se escribe el *cuerpo del programa* o función `main`. Y finalmente se procede a escribir las N funciones cuyos prototipos se han definido anteriormente.

♦ **Ejemplo 1.1:**

Considérese el siguiente programa escrito en lenguaje C

```
/* Mi primer programa de C*/
#include <stdio.h>
void main(void)
{
      printf(" HOLA A TODOS ");
}
```

La primera línea del programa es un comentario sobre el programa. En C los comentarios se escriben comenzando con '`/*`' y terminando con '`*/`'. La segunda línea es una directiva del preprocesador del tipo `#include` que hace referencia al fichero de cabecera o librería `stdio.h` que contiene funciones estándar de entrada/salida. La tercera línea es la declaración de función principal `main`. El indicador de tipo `void` al comienzo de la línea indica que `main` no genera ningún parámetro de salida. Mientras que el indicador de tipo `void` encerrado entre paréntesis al final de la línea indica que `main` no tiene parámetros de entrada. Algunos compiladores pueden dar mensajes de aviso o inclusos errores de compilación por usar el indicador de tipo `void` como tipo del parámetro de salida (y/o de entrada), por lo que alternativamente a la sentencia

```
void main(void)
```

se puede utilizar dependiendo del compilador la sentencia

```
main(void)
```

o la sentencia

```
main()
```

Las restantes líneas de este programa se corresponden al cuerpo de la función principal, que en este caso sólo consta de una sentencia simple del tipo `printf` que imprime en el dispositivo de salida estándar (típicamente el monitor) mensajes de texto. Obsérvese que de forma general todas las sentencias simples terminan en punto y coma.

Supóngase que el código fuente de este programa se encuentra en un fichero de texto llamado `programa1_1.c` y que al fichero ejecutable de este programa se le desea llamar `prog1`. La orden que hay que invocar desde la línea de órdenes (`$`) para generar este fichero ejecutable es:

```
$ gcc –o prog1 programa1_1.c
```

Para ejecutar `prog1` se teclea la orden

```
$ prog1
```

la ejecución de este programa mostrará por pantalla el mensaje

```
" HOLA A TODOS "
```

Debe tenerse en cuenta que si el directorio de trabajo actual donde está el fichero `prog1` no está añadido en la lista de directorios de la variable de entorno PATH (ver sección 3.6.9) entonces el programa no se ejecutará ya que el intérprete de comandos no lo encontrará y mostrará un mensaje en la pantalla avisando de esta circunstancia.

♦

1.4 CONCEPTOS BÁSICOS DE C

1.4.1 Identificadores, palabras reservadas, separadores y comentarios

Un identificador es una secuencia de letras o dígitos donde el primer elemento debe ser una letra o los caracteres _ y `$`. Las letras mayúsculas y minúsculas se consideran distintas. Los identificadores son los nombres que se utilizan para representar variables, constantes, tipos y funciones de un programa. El compilador sólo reconoce los 32 primeros caracteres del identificador, pero éste puede ser de cualquier tamaño.

El lenguaje C distingue entre letras mayúsculas y minúsculas. Por ejemplo, el identificador `valor` es distinto a los identificadores `VALOR` y `Valor`

Las *palabras reservadas* son identificadores predefinidos que tienen un significado especial para el compilador de C. Las palabras claves siempre van en minúsculas. En la Tabla 1.1 se recogen las palabras reservadas según ANSI C, que es la definición estandarizada del lenguaje C creada por el ANSI[1].

El lenguaje C también utiliza una serie de caracteres como elementos *separadores*: `{,},[,],(,),;,->,..`

En C es posible escribir *comentarios* como una secuencia de caracteres que se inicia con `/*` y termina con `*/`. Los comentarios son ignorados por el compilador.

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

Tabla 1.1: Palabras reservadas en C según el estándar ANSI

1.4.2 Constantes

Las constantes se refieren a valores fijos que no se pueden alterar por el programa. El lenguaje C tiene cuatro tipos básicos de constantes:

- *Constantes enteras*. Es un número con un valor entero, consistente en una secuencia de dígitos. Las constantes enteras se pueden escribir en tres sistemas numéricos distintos: decimal (base 10), octal (base 8) y hexadecimal (base16).

  - *Constante entera decimal*. Puede ser cualquier combinación de dígitos tomados del conjunto `0` a `9`. Si la constante tiene dos o más dígitos, el primero de ellos debe ser distinto de `0`.

  - *Constante entera octal*. Puede estar formada por cualquier combinación de dígitos tomados del conjunto `0` a `7`. El primer dígito debe ser obligatoriamente `0`, con el fin de identificar la constante como un número octal.

[1] ANSI es el acrónimo derivado del término inglés "*American National Standards Institute*" (Instituto Nacional Americano de Estándares).

  - *Constante entera hexadecimal*. Debe comenzar por `0x` o `0X`. Puede aparecer después cualquier combinación de dígitos tomados del conjunto `0` a `9` y de `a` a `f` (tanto minúsculas como mayúsculas).

- *Constantes en coma flotante*. Es un número decimal que contiene un punto decimal o un exponente (o ambos).

- *Constantes de carácter*. Es un sólo carácter encerrado con comillas simples. Cada carácter tiene un valor entero equivalente de acuerdo con el código ASCII[2]

- *Constantes de cadenas de caracteres*. Consta de cualquier número de caracteres consecutivos encerrados entre comillas dobles.

♦ **Ejemplo 1.2:**

A continuación se muestran algunos ejemplos de diferentes tipos de constantes:

`0  10  743  32767`	(Constantes enteras decimales)
`01  025  0547  07777`	(Constantes enteras octales)
`0x1  0X1a  0X7F  0xabcd`	(Constantes enteras hexadecimales)
`0.5  2.3  827.602  1.666E12`	(Constantes en coma flotante)
`'Z'  '?'  ' '  'a'`	(Constantes de carácter)
`"trabajo"  "Sistema operativo UNIX"`	(Constantes de cadena de caracteres)

♦

Las constantes enteras y en coma flotante representan números, por lo que se las denomina en general como *constantes de tipo numérico*. Las siguientes reglas se pueden aplicar a todas las constantes numéricas:

1) No se pueden incluir comas ni espacios en blanco en la constante.

2) Una constante puede ir precedida de un signo menos (`-`). Realmente, el signo menos es un operador que cambia el signo de una constante positiva.

[2] ASCII es el acrónimo derivado del término inglés "*American Standard Code for Information Interchange*" (Código Estadounidense Estándar para el Intercambio de Información).

3) El valor de una constante no puede superar un límite máximo y un límite mínimo especificados. Para cada tipo de constante, estos límites dependen del compilador de C utilizado.

Las constantes se declaran colocando el modificador `const` delante del tipo de datos. Otra forma de definir constantes es usando la directiva de compilación `#define`.

♦ **Ejemplo 1.3:**

La siguiente sentencia declara la constante `MAXIMO` de tipo entero que es inicializada al valor `9`.

```
const int MAXIMO=9;
```

Otra forma equivalente de declararla es con la sentencia:

```
#define MAXIMO 9
```

Esta sentencia se ejecuta de la siguiente forma: en la fase de compilación al ejecutar `#define` el compilador sustituye cada aparición de `MAXIMO` por el valor `9`. Además no se permite asignar ningún valor a esa constante. Es importante darse cuenta que esta declaración no acaba en punto y coma '`;`'

♦

Se denomina *secuencia de escape* a una combinación de caracteres que comienza siempre con una barra inclina hacia atrás `\` y es seguida por uno o más caracteres especiales. Una secuencia de escape siempre representa un solo carácter, aun cuando se escriba con dos o más caracteres. En la Tabla 1.2 se listan las secuencias de escape utilizadas más frecuentemente.

Carácter	Secuencia de escape
Sonido (alerta)	`\a`
Tabulador horizontal	`\t`
Tabulador vertical	`\v`
Nueva línea	`\n`
Comillas	`\"`
Comilla simple	`\'`
Barra inclinada hacia atrás	`\\`
Signo de interrogación	`\?`
Nulo	`\0`

Tabla 1.2: Secuencias de escape utilizadas más frecuentemente

El compilador inserta automáticamente un carácter nulo (`\0`) al final de toda constante de cadena de caracteres, aunque este carácter no aparece cuando se visualiza

la cadena. El saber que el último carácter de una cadena de caracteres es siempre el carácter nulo es de gran ayuda si la cadena es examinada carácter a carácter, como sucede en muchas aplicaciones.

Es importante darse cuenta que una constante de carácter (por ejemplo `'J'`) y su correspondiente cadena de caracteres (por ejemplo `"J"`) no son equivalentes ya que la cadena no consta de un único carácter sino de dos (`'J'` y `'\0'`).

1.4.3 Variables

Una *variable* es un identificador que se utiliza para representar cierto tipo de información dentro de una determinada parte del programa. En su forma más sencilla, una variable es un identificador que se utiliza para representar un dato individual; es decir, una cantidad numérica o una constante de carácter. En alguna parte del programa se asigna el dato a la variable. Este valor se puede recuperar después en el programa simplemente haciendo referencia al nombre de la variable.

A una variable se le pueden asignar diferentes valores en distintas partes del programa. De esta forma la información representada puede cambiar durante la ejecución del programa. Sin embargo, el tipo de datos asociado a la variable no puede cambiar. Las variables se declaran de la siguiente forma:

```
tipo nombre_variable;
```

Donde `tipo` será un tipo de datos válido en C con los modificadores necesarios y `nombre_variable` será el *identificador* de la misma.

Las variables se pueden declarar en diferentes puntos de un programa:

- Dentro de funciones. Las variables declaradas de esta forma se denominan *variables locales*.
- En la definición de funciones. Las variables declaradas de esta forma se denominan *parámetros formales*.
- Fuera de todas las funciones. Las variables declaradas de esta forma se denominan *variables globales*.

La inicialización de una variable es de la forma:

```
nombre_variable = constante;
```

Donde `constante` debe ser del tipo de la variable.

♦ **Ejemplo 1.4:**

A continuación se muestran algunos ejemplos de declaración e inicialización de variables:

```
/*Declaración */
      char letra;
      int entero;
      float real;
/*Inicialización*/
      letra='a';
      entero=234;
      real=123.333;
```

♦

1.4.4 Tipos fundamentales de datos

En el lenguaje C se consideran dos grandes bloques de datos:

- *Tipos fundamentales*. Son aquellos suministrados por el lenguaje.
- *Tipos derivados*. Son aquellos definidos por el programador.

Los *tipos fundamentales* se clasifican en:

- *Tipos enteros*. Se utilizan para representar subconjuntos de los números naturales y enteros.
- *Tipos reales*. Se emplean para representar un subconjunto de los números racionales.
- Tipo `void`. Sirve para declarar explícitamente funciones que no devuelven ningún valor. También sirve para declarar punteros genéricos.

1.4.4.1 Tipos enteros

Se distinguen los siguientes tipos enteros:

- `char`. Define un número entero de 8 bits. Su rango es [-128, 127]. También se emplea para representar el conjunto de caracteres ASCII.
- `int`. Define un número entero de 16 o 32 bits (dependiendo del procesador).
- `long`. Define un número entero de 32 o 64 bits (dependiendo del procesador).
- `short`. Define un número entero de tamaño menor o igual que `int`. En general se cumple que: tamaño(`short`) ≤ tamaño(`int`) ≤ tamaño(`long`).

Estos tipos pueden ir precedidos del modificador `unsigned` que indica que el tipo sólo representa números positivos o el cero. Es usual utilizar la abreviatura `u` para designar a `unsigned`, en dicho caso el modificador precede al tipo de datos pero sin un espacio entre medias.

♦ **Ejemplo 1.5:**

Con las sentencias

```
int numero=3;
char letra='c';
unsigned short contador=0;
ulong barra=123456789;
```

se están declarando: la variable `numero` de tipo `int` inicializada a 3, la variable `letra` de tipo `char` inicializada a 'c', la variable `contador` de tipo `unsigned short` inicializada a 0 y la variable `barra` de tipo `ulong`, abreviatura de `unsigned long`, inicializada a `123456789`.

1.4.4.2 Tipos reales

Se distinguen los siguientes tipos reales:

- `float`. Define un número en coma flotante de precisión simple. El tamaño de este tipo suele ser de 4 bytes (32 bits).
- `double`. Define un número en coma flotante de precisión doble. El tamaño de este tipo suele ser de 8 bytes (64 bits). Este tipo puede ir precedido del modificador `long`, que indica que su tamaño pasa a ser de 10 bytes.

♦ **Ejemplo 1.6:**

Con las sentencias

```
float ganancia=125.23;
double diametro=12.3E53;
```

se están declarando: la variable `ganancia` de tipo `float` inicializada a `125.23` y la variable `diametro` de tipo `double` inicializada a `'12.3E53'`.

♦

1.4.5 Tipos derivados de datos

Los tipos derivados se construyen a partir de los tipos fundamentales o de otros tipos derivados. Se van a describir las características de los siguientes: arrays, punteros, estructuras y uniones.

1.4.5.1 Arrays

Un *array* es una colección de variables del mismo tipo que se referencian por un nombre común. El compilador reserva espacio para un array y le asigna posiciones de memoria contiguas. La dirección más baja corresponde al primer elemento y la más alta al último. Se puede acceder a cada elemento de un array con un índice. Los índices, para acceder al array, deben ser variables o constantes de tipo entero. Se define la dimensión de un array como el total de índices que son necesarios para acceder a un elemento particular del array.

La declaración formal de una array multidimensional de tamaño N es la siguiente:

$$\texttt{tipo\_array nombre\_array[rango}_1\texttt{][rango}_2\texttt{]...[rango}_N\texttt{];}$$

donde $\texttt{rango}_1$, $\texttt{rango}_2$,... y $\texttt{rango}_N$ son expresiones enteras positivas que indican el número de elementos del array asociados con cada índice.

A los arrays unidimensionales se les denomina *vectores* y a los bidimensionales se les denomina *matrices*. A los arrays unidimensionales de tipo `char` se les denomina *cadena de caracteres* y a los arrays de cadenas de caracteres (matrices de caracteres) se les denomina *tablas*. Para indexar los elementos de un array, se debe tener en cuenta que los índices deben variar entre 0 y M-1, donde M es el tamaño de la dimensión a la que se refiere el índice.

♦ **Ejemplo 1.7:**

La declaración de una variable denominada `matriz_A` de números de tipo `float` de dos filas y dos columnas sería de la siguiente forma:

```
float matriz_A[2][2];
```

La declaración de una variable denominada `X` que es un vector de 4 números de tipo `int` sería de la siguiente forma:

```
int X[4];
```

Para el vector `X` anteriormente declarado, el índice variará entre 0 y 3: `X[0]`, `X[1]`, `X[2]`, `X[3]`;

♦

Los array pueden ser inicializados en el momento de su declaración. Para ello los valores iniciales deben aparecer en el orden en que serán asignados a los elementos del array, encerrados entre llaves y separados por comas. Todos los elementos del array que no sean inicializados de forma explícita serán inicializados por defecto al valor cero.

El tamaño de un array unidimensional no necesita ser especificado explícitamente cuando se incluyen los valores iniciales de los elementos. Con un array numérico el tamaño del array será fijado automáticamente igual al número de valores iniciales incluidos dentro de la definición del array.

En el caso de las cadenas de caracteres la especificación del tamaño del array normalmente se omite. El tamaño adecuado de la cadena es asignado automáticamente como el número de caracteres que aparecen en la cadena más el carácter nulo `'\0'` que se añade automáticamente al final de cada cadena de caracteres.

♦ **Ejemplo 1.8:**

La siguiente definición

```
int datos[5]={1, 6, -5, 4, 0};
```

o equivalentemente la definición

```
int datos[]={1, 6, -5, 4, 0};
```

declara el array `datos` de cinco elementos de tipo entero que son inicializados con los siguientes valores: `datos[0]=1`, `datos[1]=6`, `datos[2]=-5`, `datos[3]=4` y `datos[4]=0`.

La definición

```
float X[3]={12.26, 25.31};
```

declara el array `X` de tres elementos de tipo coma flotante que son inicializados con los siguientes valores: `X[0]=12.26`, `X[1]=25.31` y `X[2]=0`. Nótese que la definición

```
float X[]={12.26, 25.31};
```

no es equivalente a la anterior ya que ahora al no especificarse explícitamente el tamaño del array este se calcula a partir del número de valores iniciales especificados. Luego ahora la dimensión del array `X` es de 2 elementos y no de 3 como en el caso anterior.

La definición

```
char nombre[]="Rocio";
```

declara una cadena de caracteres de seis elementos: `nombre[0]='R'`, `nombre[1]='o'`, `nombre[2]='c'`, `nombre[3]='i'`, `nombre[4]='o'` y `nombre[5]='\0'`.

La definición anterior se podría haber escrito equivalentemente como

```
char nombre[6]= "Rocio";
```

Nótese como al especificar la dimensión del array se debe incluir un elemento extra para el carácter nulo `'\0'`.

♦

En el caso de la inicialización de los array multidimensionales se debe tener cuidado en el orden en que se asignan los elementos del array. La regla que se utiliza es que el último índice (situado más a la derecha) es el que se incrementa más rápido y el primer índice (situado más a la izquierda) es el que se incrementa más lentamente. De esta forma, los elementos de un array bidimensional se deben asignar por filas comenzando por la primera.

El orden natural en el que los valores iniciales son asignados se puede alterar formando grupos de valores iniciales encerrados entre llaves. Los valores dentro de cada par interno de llaves serán asignados a los elementos del array cuyo último índice varíe más rápidamente. Por ejemplo, en un array bidimensional, los valores almacenados dentro del par interno de llaves serán asignados a los elementos de la primera fila, ya que el segundo índice (columna) se incrementa más rápidamente. Si hay pocos elementos dentro de cada par de llaves, al resto de los elementos de cada fila se les asignará el valor cero. Por otra parte, el número de valores dentro de cada par de llaves no puede exceder del tamaño de fila definido.

♦ **Ejemplo 1.9:**

La siguiente definición

```
int base[2][3]={1, 6, -5, 4, 0, 8};
```

o equivalentemente la definición

```
int base[2][3]={
      {1, 6, -5},
      {4, 0, 8}}
};
```

declara el array bidimensional `base` que puede ser considerado como una tabla de dos filas y tres columnas (tres elementos por fila) que es inicializada con los siguientes valores:

```
base[0][0]=1      base[0][1]=6      base[0][2]=-5
base[1][0]=4      base[1][1]=0      base[1][2]=8
```

Nótese que los valores iniciales se asignan por filas, el índice situado más a la derecha se incrementa más rápidamente.

La definición

```
int base[2][3]={
      {1, 6},
      {4, 0}}
};
```

asigna valores únicamente a los dos primeros elementos de cada fila. Luego los elementos del array `base` tendrán los siguientes valores iniciales:

```
base[0][0]=1      base[0][1]=6      base[0][2]=0
base[1][0]=4      base[1][1]=0      base[1][2]=0
```

La definición

```
int base[2][3]={
      {1, 6, -5, 6},
      {4, 0, 8, -9}}
};
```

produciría un error de compilación, ya que el número de valores dentro de cada par de llaves (cuatro valores en cada par) excede el tamaño definido del array (tres elementos en cada fila).

♦

1.4.5.2 Punteros

Un *puntero* es una variable que es capaz de guardar una dirección de memoria, dicha dirección es la localización de otra variable en memoria. Así, si una variable A contiene la dirección de otra variable B decimos que A apunta a B, o que A es un puntero a B.

Los punteros son usados frecuentemente en C ya que tienen una gran cantidad de aplicaciones. Por ejemplo, pueden ser usados para pasar información entre una función y sus puntos de llamada. En particular proporcionan una forma de devolver varios datos desde una función a través de los argumentos de una función. Los punteros también permiten que referencias a otras funciones puedan ser especificadas como argumentos de una función. Esto tiene el efecto de pasar funciones como argumentos de una función dada.

Los punteros se definen en base a un tipo fundamental o a un tipo derivado. La declaración de un puntero se realiza de la siguiente forma:

```
tipo_base *puntero;
```

Los punteros una vez declarados contienen un valor desconocido. Si se intenta usar sin inicializar podemos incluso dañar el sistema operativo. La forma de inicializarlos consiste en asignarles una dirección conocida.

Existen dos operadores que se utilizan para trabajar con punteros:

- El operador `&`, da la dirección de memoria asociada a una variable y se utiliza para inicializar un puntero.
- El operador `*`, se utiliza para referirse al contenido de una dirección de memoria.

El acceso a una variable a través de un puntero se conoce también como *indirección*. Obviamente antes de manipular el contenido de un puntero hay que inicializar el puntero para que referencie a la dirección de dicha variable.

♦ **Ejemplo 1.10:**

Supóngase la siguiente sentencia

```
int *z;
```

Se trata de la declaración de la variable puntero `z`, que contiene la dirección de memoria de un número de tipo `int`.

Supónganse ahora las siguientes tres sentencias:

```
int z, *pz;

pz=&z;

*pz=25;
```

La primera sentencia está declarando una variable `z` de tipo entero y una variable puntero `pz` que contiene la dirección de una variable de tipo `int`.

La segunda sentencia, inicializa la variable puntero `pz` a la dirección de la variable `z`.

Por último la tercera sentencia es equivalente a la sentencia '`z=25;`'. A través de `pz` se puede modificar de una forma indirecta el valor de la variable `z`.

En la Figura 1.2 se representa la relación entre el puntero `pz` y la variable `z`.

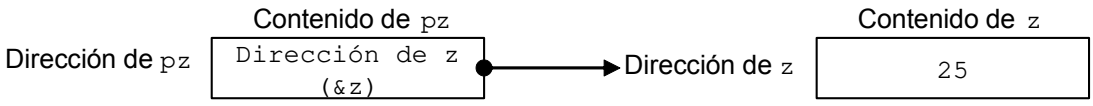

Figura 1.2: Relación entre el puntero pz y la variable z

♦

Por otra parte, es posible realizar ciertas operaciones con una variable puntero, a continuación se resumen las operaciones permitidas:

- A una variable puntero se le puede asignar la dirección de una variable ordinaria (por ejemplo, `pz=&z`).
- A una variable puntero se le puede asignar el valor de otra variable puntero (`pz=py`), siempre que ambos punteros apunten al mismo tipo de datos.
- A una variable puntero se le puede asignar un valor nulo (cero). En general no tiene sentido asignar un valor entero a una variable puntero. Pero una excepción es la asignación de 0, que a veces se utiliza para indicar

condiciones especiales. En tales situaciones, la práctica recomendada es definir una constante simbólica `NULL` que representa el valor 0 y usar `NULL` en la inicialización del puntero. Un ejemplo de esta forma de inicialización es

```
#define NULL 0
float *pv=NULL;
```

- A una variable puntero se le puede sumar o restar una cantidad entera (por ejemplo, `px +3`, `pz++`, etc).
- Una variable puntero puede ser restada de otra con tal que ambas apunten a elementos del mismo array.
- Dos variables puntero pueden ser comparadas siempre que ambas apunten a datos del mismo tipo.

Los punteros están muy relacionados con los arrays y proporcionan una vía alternativa de acceso a los elementos individuales del array. Se puede acceder a cualquier posición del array usando el nombre y el índice (indexación) o bien con un puntero y la aritmética de punteros.

En general, el nombre de un array es realmente un puntero al primer elemento de ese array. Por tanto, si `x` es un array unidimensional, entonces la dirección al elemento `0` (primer elemento) del array se puede expresar tanto como `&x[0]` o simplemente como `x`. Además el contenido del elemento `0` del array se puede expresar como `x[0]` o como `*x`. La dirección del elemento `1` (segundo elemento) del array se puede escribir como `&x[1]` o como `(x+1)`. Mientras que el contenido del elemento `1` del array se puede expresar como `x[1]` o como `*(x+1)`. En general, la dirección del elemento `i` del array se puede expresar bien como `&x[i]` o como `(x+i)`. Mientras que el contenido del elemento `i` del array se puede expresar como `x[i]` o como `*(x+i)`.

♦ **Ejemplo 1.11:**

El siguiente programa en C ilustra las dos formas equivalentes de acceso a los elementos de un array unidimensional:

```
#include <stdio.h>
main()
{
      int vector[3]={1, 2, 3};
      printf("\n %d %d %d", vector[0],vector[1],vector[2]);
```

```
        *vector=4;     /*Esta sentencia equivale a vector[0]=4*/
        *(vector+1)=5; /*Esta sentencia equivale a vector[1]=5*/
        *(vector+2)=6; /*Esta sentencia equivale a vector[2]=6*/

        printf("\n %d %d %d", *vector,*(vector+1),*(vector+2));
}
```

♦

Puesto que el nombre de un array es realmente un puntero al primer elemento dentro del array, es posible definir el array como una variable puntero en vez de como un array convencional. Sintácticamente las dos definiciones son equivalentes. Sin embargo, la definición convencional de un array produce la reserva de un bloque fijo de memoria al principio de la ejecución del programa, mientras que esto no ocurre si el array se representa en términos de una variable puntero. En este último caso la reserva de memoria la debe realizar el programador de forma explícita en el código del programa mediante el uso de la función `malloc` (ver sección 1.9).

Un array multidimensional se puede expresar como un array de punteros. En este caso el nuevo array será de una dimensión menor que el array multidimensional. Cada puntero del array indicará el principio de un array de dimensión *N*-1. Así la declaración de un array multidimensional de orden N

$$\texttt{tipo\_array nombre\_array[rango}_1\texttt{][rango}_2\texttt{]...[rango}_N\texttt{];}$$

puede sustituirse equivalentemente por la declaración de un array de punteros de dimensión N-1:

$$\texttt{tipo\_array *nombre\_array[rango}_1\texttt{]...[rango}_{N-1}\texttt{];}$$

Obsérvese que cuando un array multidimensional de dimensión N se expresa mediante un array de punteros de dimensión N -1 no se especifica $\texttt{[rango}_N\texttt{]}$.

El acceso a un elemento individual dentro del array se realiza simplemente usando el operador `*`.

Al igual que en el caso unidimensional, el usar un array de punteros para representar a un array multidimensional requiere la reserva explícita de memoria mediante el uso de la función `malloc`.

♦ **Ejemplo 1.12:**

Considérese que `z` es un array bidimensional de números en coma flotante con 3 filas y 4 columnas. Se puede definir `z` como

```
float z[3][4];
```

o como un array unidimensional de punteros escribiendo

```
float *z[3];
```

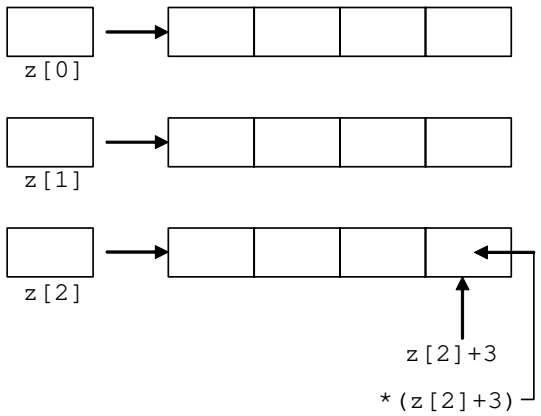

Figura 1.3: Uso de un array de punteros para referirse a un array bidimensional de 3 filas y 4 columnas.

En este segundo caso (ver Figura 1.3), `z[0]` apunta al elemento 0 (primer elemento) de la fila 0 (primera fila), `z[1]` apunta al elemento 0 de la fila 1 (segunda fila) y `z[2]` apunta al elemento 0 de la fila 1 (tercera fila). Obsérvese que el número de elementos dentro de cada fila no está especificado.

Para acceder al elemento de la fila 2 situado en la columna 3 (`z[2][3]`) se puede escribir

```
*(z[2]+3)
```

En esta expresión `z[2]` es un puntero al elemento 0 (primer elemento) de la fila 2, de modo que `(z[2]+3)` apunta al elemento 3 (cuarto elemento) de la fila 2. Luego `*(z[2]+3)` se refiere al elemento en la columna 3 de la fila 2, que es `z[2][3]`.

♦

El uso de arrays de punteros es muy útil para almacenar cadenas de caracteres. En esta situación, cada elemento del array es un puntero de tipo carácter que indica donde comienza la cadena. De esta forma un array de punteros de *n* elementos de tipo carácter puede apuntar a *n* cadenas diferentes. Cada cadena puede ser accedida haciendo uso de su puntero correspondiente. También es posible asignar un conjunto de valores iniciales como parte de la declaración del array de punteros. Estos valores iniciales serán cadenas de caracteres, donde cada una representa a un elemento distinto del array.

♦ **Ejemplo 1.13:**

Supóngase que se desean almacenar durante la ejecución de un programa las siguientes cadenas de caracteres en un array de tipo carácter:

```
Sol
Tierra
Luna
Vía Láctea
```

Estas cadenas se pueden almacenar, usando las sentencias apropiadas, en un array bidimensional de tipo carácter, por ejemplo

```
char astronomía[4][11];
```

Nótese que `astronomía` tiene 4 filas para 4 cadenas. Cada fila debe ser suficientemente grande para almacenar por lo menos 11 caracteres, ya que `Vía Láctea` tiene 10 caracteres más el carácter nulo (`\0`) al final. Otra forma de almacenar este array bidimensional es definir un array de 4 punteros:

```
char *astronomía[4];
```

Nótese que no es necesario incluir el número de columnas dentro de la declaración del array. No obstante, posteriormente en el código del programa se tendrá que reservar una cantidad especifica de memoria para cada cadena de caracteres haciendo uso de la función `malloc` (ver sección 1.9). También se podía haber inicializado el array con las cadenas de caracteres al realizar la declaración del array.

```
char *astronomía[4]={
                        "Sol"
                        "Tierra"
                        "Luna"
                        "Vía Láctea"
                    };
```

Así el elemento 0 (primer elemento) del array de punteros `astronomía[0]` apuntará a `Sol`, el elemento 1 (segundo elemento) `astronomía[1]` apuntará a `Tierra` y así sucesivamente. Como la declaración del array incluye valores iniciales, no es necesario especificar en la declaración de forma explícita el tamaño del array. Por lo tanto, la declaración anterior se puede escribir equivalentemente de la siguiente forma:

```
char *astronomía[]={
                        "Sol"
                        "Tierra"
                        "Luna"
                        "Vía Láctea"
                    };
```

♦

1.4.5.3 Estructuras

Una estructura es un agregado de tipos fundamentales o derivados que se compone de varios campos. A diferencia de los arrays, cada elemento de la estructura puede ser de un tipo diferente. La forma de definir una estructura es la siguiente:

```
struct nombre_estructura
{
      tipo1 campo1;
      tipo2 campo2;
      ...
      tipoN campoN;
}
```

Para acceder a los campos de una estructura se utiliza el operador '.'. Asimismo es posible declarar:

- Arrays de estructuras.
- Punteros a estructuras. En este caso, el acceso a los campos de la variable se hace por medio del operador '`->`'.

Además, puesto que el tipo de cada campo de una estructura puede ser un tipo fundamental o derivado, también puede ser otra estructura. Con lo que es posible declarar estructuras dentro de estructuras.

♦ **Ejemplo 1.14:**

Las siguientes sentencias

```
struct cliente{
      int cuenta;
      char nombre[100];
      unsigned short dia;
      unsigned short mes;
      unsigned int año;
      float saldo;
}
```

están declarando una estructura del tipo `cliente`. Por otra parte la sentencia:

```
struct cliente uno;
```

está declarando la variable de estructura `uno` del tipo `cliente`. Con la sentencia

```
uno.cuenta=12530;
```

Se está asignando al campo `cuenta` de la estructura `uno` el valor `12530`.

♦

♦ **Ejemplo 1.15:**

Sea la siguiente declaración de una estructura

```
struct
{
      float real, imaginaria;
} vectorC[5];
```

La variable `vectorC` es un array unidimensional de números complejos. Para modificar la parte real del elemento 2 (recuérdese que los elementos de un array se comienzan a numerar por el 0), se escribe la sentencia:

```
vectorC[1].real=0.23;
```

♦

♦ **Ejemplo 1.16:**

Considérense las siguientes sentencias:

```
struct altura
{
      char nombre[100];
      float h;
};
struct altura persona, *p;
p=&persona;
p->h=1.65;
```

El puntero `p` apunta a la estructura `persona` del tipo `altura`. Con la última sentencia se está asignando al campo `h` de la estructura `persona` el valor 1.65. Esa sentencia es equivalente a `persona.h=1.65;`

♦

♦ **Ejemplo 1.17:**

Las siguientes sentencias son un ejemplo de como es posible declarar una estructura dentro de otra estructura:

```
struct fecha {
      int día, mes, año;
}
struct alumno {
      char nombre[100];
      struct fecha fecha_nacimiento;
      float nota;
};
struct alumno ficha;
```

♦

1.4.5.4 Uniones

Las uniones se definen de forma parecida a las estructuras, es decir, contienen miembros cuyos tipos de datos pueden ser diferentes. Sin embargo, todos los miembros que componen una unión comparten la misma área de almacenamiento dentro de la memoria, mientras que cada miembro dentro de una estructura tiene asignada su propia área de almacenamiento. Por lo tanto, las uniones se utilizan para ahorrar memoria. Resultan bastante útiles para aplicaciones que requieren múltiples miembros donde únicamente se requiere asignar simultáneamente valor a un único miembro. El tamaño de la memoria reservada a una unión va a ser fijado por el compilador tomando como referencia el tamaño del mayor de los miembros de dicha unión. La declaración de una unión es la siguiente:

```
union nombre_unión
{
      tipo1 campo1;
      tipo2 campo2;
      ...
      tipoN campoN;
};
```

La asignación de valores a los miembros de una unión se realiza de forma parecida a como se realiza para los miembros de una estructura.

♦ **Ejemplo 1.18:**

Considérese la siguiente declaración de una unión.

```
union multiuso
{
      int numeroZ;
      char campo[6];
} uno;
```

Las sentencias anteriores son la definición de la variable de unión `uno` del tipo `multiuso`. Esta variable puede representar en un momento dado o un número entero (`numeroZ`) o una cadena de 6 caracteres (`campo`). Puesto que la cadena necesita más memoria que el entero, el compilador reserva para esta variable de unión un bloque de memoria suficientemente grande para poder almacenar la cadena de 6 caracteres.

♦

1.4.5.5 Alias para los nombres de tipo

Es posible hacer que un identificador sea considerado el nombre de un nuevo tipo, para ello hay que emplear la palabra clave `typedef`, con ella es posible actuar sobre cualquier tipo fundamental o derivado.

♦ **Ejemplo 1.19:**

Con las sentencias

```
typedef int entero;
entero y;
```

se está definiendo el identificador `entero` como sinónimo de `int` y se está declarando la variable `y` de tipo `entero`.

♦

1.4.6 Tipos de almacenamiento

Las variables se pueden clasificar por su *tipo de almacenamiento* al igual que por su tipo de datos. El tipo de almacenamiento especifica la parte del programa dentro del cual se reconoce a la variable. Hay cuatro especificaciones diferentes de tipo de almacenamiento en C: automática, externa, estática y registro. Están identificadas por las siguientes palabras reservadas: `auto`, `extern`, `static` y `register`.

Si se desea especificar el tipo de almacenamiento de una variable éste debe colocarse antes del tipo de datos de la variable:

```
tipo_almacenamiento tipo_dato nombre_variable;
```

A veces se puede establecer el tipo de almacenamiento asociado a una variable por la posición de su declaración en el programa por lo que no es necesario utilizar la palabra reservada asociada a dicho tipo de almacenamiento. En otras situaciones, la palabra reservada que especifica un tipo particular de almacenamiento se tiene que colocar al comienzo de la declaración de la variable.

1.4.6.1 Variables automáticas

Las *variables automáticas* se declaran siempre dentro de una función y son locales a dicha función. Luego una variable automática no mantiene su valor cuando se transfiere el control fuera de la función en que está definida. Asimismo las variables automáticas definidas en funciones diferentes serán independientes unas de otras, incluso aunque tengan el mismo nombre.

Cualquier variable declarada dentro de una función se interpreta por defecto como una variable automática a menos que se incluya dentro de la declaración un tipo de almacenamiento distinto. Por lo tanto no es obligatorio incluir al principio de la declaración de una variable automática la palabra reservada `auto`.

1.4.6.2 Variables externas (globales)

Las *variables externas o globales*, a diferencia de las variables automáticas, no están confinadas a una determinada función. Su ámbito se extiende desde el punto de definición hasta el resto del programa. Por lo tanto, a una variable externa se le puede asignar un valor dentro de una función y este valor puede usarse (al acceder a la variable externa) dentro de otra función.

No es necesario escribir el especificador de tipo de almacenamiento `extern` en una definición de una variable externa, porque las variables externas se identifican por la localización de su definición en el programa. De hecho algunos compiladores producen errores cuando se usa esta palabra clave en la declaración de una variable global.

1.4.6.3 Variables estáticas

Las *variables estáticas* se definen dentro de funciones individuales y tienen, por tanto, el mismo ámbito que las variables automáticas, es decir, son locales a la función en que están definidas. Sin embargo, a diferencia de las variables automáticas, las variables estáticas retienen sus valores durante todo el tiempo de vida del programa. Por lo tanto, si se sale de la función y posteriormente se vuelve a entrar, las variables estáticas definidas dentro de esa función mantendrán sus valores previos. Esta característica permite a las funciones mantener información permanente a lo largo de toda la ejecución del programa.

Las variables estáticas se definen dentro de una función de la misma forma que las variables automáticas, excepto que la declaración de variables tiene que comenzar con la palabra clave `static`.

En ocasiones dentro de un mismo programa se definen variables automáticas o estáticas que tienen el mismo nombre que variables externas. En tales casos las variables locales tienen precedencia sobre las variables externas, aunque los valores de las variables externas no se verán afectados por la manipulación de las variables locales.

♦ **Ejemplo 1.20:**
Supóngase el siguiente programa escrito en lenguaje C:

```
int r=1, s=2, t=3;
void func1(void);
main()
{
      static int r=4;
      printf("\nA: %d %d %d\n", r, s, t);
      r=r+1;
      func1();
      printf("\nB: %d %d %d\n", r, s, t);
      func1();
      printf("\nC: %d %d %d\n", r, s, t);
}
void func1(void)
{
      static int r=0;
      int s=0;
      if (r==0)
            r=r+s+t+2;
      else
            r=r+10;
      printf("\nD: %d %d %d\n", r, s, t);
}
```

En este programa las variables de tipo entero `r`, `s` y `t` son variables externas. Sin embargo, `r` ha sido redefinida dentro de `main` como variable de tipo entero estática. Las modificaciones que se realicen a la variable `r` dentro de la función `main` serán locales a esta función y no afectarán al valor de la variable global `r`. Luego dentro de la función `main` sólo se reconocen como variables externas a `s` y `t`

De forma análoga, en la función `func1` se define la variable estática `r` y la variable automática `s`, ambas de tipo coma flotante. Luego `r` mantendrá su valor previo si se vuelve a entrar dentro de la función `func1`, mientras que `s` perderá su valor siempre que se transfiera el control del programa fuera de `func1`. Por otra parte dentro de func1 sólo se reconoce como variable externa a `t`.

La ejecución del fichero ejecutable que se genera al compilar este programa mostraría la siguiente traza de ejecución en pantalla:

```
A: 4 2 3
D: 5 0 3
B: 5 2 3
D: 15 0 3
C: 5 2 3
```

♦

1.4.6.4 Variables registro

Los registros son áreas especiales de almacenamiento dentro de la unidad central de procesamiento (CPU) de una computadora cuyo tiempo de acceso es mucho más pequeño que la memoria principal. El tiempo de ejecución de algunos programas se puede reducir considerablemente si ciertos valores pueden almacenarse dentro de los registros en vez de en la memoria principal.

En C, los valores de las *variables registro* se almacenan dentro de los registros de la CPU. A una variable se le puede asignar este tipo de almacenamiento simplemente precediendo la declaración de tipo con la palabra reservada `register`. Pero sólo puede haber unas pocas variables registro dentro de cualquier función. Típicamente dos o tres, aunque depende de la computadora y del compilador.

Los tipos de almacenamiento `register` y `automatic` están muy relacionados, ya que su ámbito definición es el mismo, es decir son locales a la función en la que han sido declaradas.

El declarar varias variables como `register` no garantiza que sean tratadas realmente como variables de tipo `register`. La declaración será válida solamente si el espacio requerido de registro está disponible. En caso contrario, la variable declarada se tratará como si fuese una variable automática.

Finalmente comentar que el operador dirección (`&`) no se puede aplicar a las variables registro.

1.5 EXPRESIONES Y OPERADORES EN C

En una expresión van a tomar parte variables, constantes y operadores. Los operadores establecen la relación entre las variables y las constantes a la hora de evaluar la expresión. Los paréntesis también pueden formar parte de una expresión y se emplean para modificar la precedencia de los operadores.

1.5.1 Operadores aritméticos

Los posibles operadores aritméticos son los que se muestran en la Tabla 1.3. Las expresiones aritméticas se evalúan de izquierda a derecha. Si en una expresión aritmética intervienen variables o constantes de diferentes tipos, el tipo del resultado va a coincidir con el tipo mayor que aparezca en la expresión. Por ejemplo, si se multiplica una variable `float` por una variable `int`, el resultado será `float`.

Operador	Acción
-	Resta
+	Suma
*	Multiplicación
/	División[3]
%	Resto de la división
--	Decremento
++	Incremento

Tabla 1.3: Operadores aritméticos

La suma y la diferencia sobre una misma variable tienen una representación simplificada mediante los operadores `++` y `--`.

♦ **Ejemplo 1.21:**

Las siguientes sentencias son un ejemplo del uso de la representación simplificada del operador suma y del operador resta:

```
int x;
++x; /* Es equivalente a x=x+1. Preincremento*/
x++; /* Es equivalente a x=x+1. Postincremento*/
--x; /* Es equivalente a x=x-1. Predecremento*/
x--; /* Es equivalente a x=x-1. Postdecremento*/
```

La diferencia entre la posición prefija y la posición sufija de los operadores anteriores queda puesta de manifiesto en las siguientes sentencias:

```
x=10;
printf("%d\n",++x); /*Incrementa "x" en 1, por lo que imprime 11*/;
x=10;
printf("%d\n",x++); /*Imprime 10 e incrementa x en 1*/;
```

♦

1.5.2 Operadores de relación y lógicos

Tanto los operadores de relación como los operadores lógicos se emplean para formar expresiones booleanas. Recuérdese que una expresión booleana únicamente puede tomar dos valores: `Verdadero (TRUE)` o `Falso (FALSE)`. En el lenguaje C, por convenio se considera que si una expresión booleana da como resultado 0 entonces su valor lógico es `Falso`. Por el contrario si al evaluarla su valor es distinto de 0,

[3] La división de números enteros produce un truncamiento del cociente (por ejemplo, 3/2=1).

entonces su valor lógico es `Verdadero`. En la Tabla 1.4 y 1.5 se muestran los operadores lógicos y los operadores de relación, respectivamente.

Operador	Significado
&&	AND lógica
||	OR lógica
!	Negación lógica

Tabla 1.4: Operadores lógicos

Operador	Relación
>	Mayor
>=	Mayor o igual
<	Menor
<=	Menor o igual
==	Igual
!=	Distinto

Tabla 1.5: Operadores de relación

1.5.3 Operadores para el manejo de bits

El lenguaje C dispone de operadores para la manipulación de bits o constantes enteras. Se debe tener mucho cuidado de no confundir las operaciones a nivel de bit con las operaciones lógicas. En la Tabla 1.6 se muestran los operadores para el manejo de bits.

Operador	Significado	Ejemplo
&	AND	`1001&0011=>0001`
|	OR	`1001|0011=>1011`
^	XOR	`1001^0011=>1010`
~	Complemento a 1	`~1001=>0110`
<<	Desplazamiento a la izquierda	`0110<<1=>1100`
>>	Desplazamiento a la derecha	`0110>>1=>0011` `1011>>1=>1101`

Tabla 1.6: Operadores para el manejo de bits

1.5.4 Expresiones abreviadas

El lenguaje C permite utilizar algunas expresiones abreviadas para indicar ciertas operaciones. En la Tabla 1.7 se muestran las expresiones abreviadas más comunes.

Expresión abreviada	Expresión equivalente
x+=y	x=x+y
x-=y	x=x-y
x*=y	x=x*y
x/=y	x=x/y
x&=y	x=x&y
x|=y	x=x|y
x^=y	x=x^y
x<<=y	x=x<<y
x>>=y	x=x>>y

Tabla 1.7: Expresiones abreviadas

1.5.5 Conversión de tipos

Los operandos de una expresión que difieren en el tipo pueden sufrir una conversión de tipo antes de que la expresión alcance su valor final. En general, el resultado final se expresará con la mayor precisión posible, de forma consistente con los tipos de datos de los operandos. Asimismo aparte de esta conversión implícita, si se desea, es posible convertir explícitamente el valor resultante de una expresión a un tipo de datos diferente. Para ello la expresión debe ir precedida por el nombre del tipo de datos deseado, encerrado con paréntesis:

```
(tipo de datos) expresión;
```

A este tipo de construcción se le denomina *conversión de tipos* (*cast*).

♦ **Ejemplo 1.22:**

Supóngase que `h` es una variable en coma flotante cuyo valor es `3.5`. La expresión

```
f%3
```

no es válida ya que `h` está en coma flotante en vez de ser un entero. Sin embargo, la expresión

```
((int)h)%3
```

hace que el primer operando se transforme en un entero y por lo tanto es válida, dando como resultado el resto entero `0`. Sin embargo, obsérvese que `h` sigue siendo una variable en coma flotante con un valor de 3.5, aunque el valor de `h` se convirtiese en un entero (`3`) al efectuar la operación del resto.

Supóngase ahora que `a` es una variable entera de valor 20. La expresión

```
((int)(a+h))%3
```

hace que el primer operando `(a+h)` se transforme en un entero y da como resultado el resto entero 2.

♦

1.6 ENTRADA Y SALIDA DE DATOS EN C

El lenguaje C va acompañado de una colección de funciones de biblioteca que incluye un cierto número de funciones de entrada/salida. Como norma general, el archivo de cabecera requerido para la entrada/salida estándar se llama `stdio.h`, entre todas las funciones que contiene algunas de las más usadas son: `getchar`, `putchar`, `scanf`, `printf`, `gets` y `puts`. Estas seis funciones permiten la transferencia de información entre la computadora y los dispositivos de entrada/salida estándar tales como un teclado y un monitor.

1.6.1 Entrada de un carácter: función `getchar`

Mediante la función de biblioteca `getchar` se puede conseguir la entrada de un carácter a través del dispositivo de entrada estándar, por defecto el teclado. Su sintaxis es

```
variable = getchar();
```

donde `variable` es alguna variable de tipo carácter declarada previamente.

♦ **Ejemplo 1.23:**

Considérense el siguiente par de sentencias:

```
char c;
c=getchar();
```

En la primera sentencia se declara la variable `c` de tipo carácter. La segunda instrucción hace que se lea del dispositivo de entrada estándar un carácter y se le asigne a `c`.

♦

1.6.2 Salida de un carácter: función `putchar`

Mediante la función de biblioteca `putchar` se puede conseguir la salida de un carácter a través del dispositivo de salida estándar, por defecto el monitor. Su sintaxis es

```
putchar(variable);
```

donde `variable` es alguna variable de tipo carácter declarada previamente.

♦ **Ejemplo 1.24:**

Considérense el siguiente par de sentencias:

```
char c='a';
putchar(c);
```

En la primera instrucción se declara la variable `c` de tipo carácter y se inicializa con el carácter '`a`'. La segunda instrucción hace que se visualice el valor de `c` en el dispositivo de salida estándar.

♦

1.6.3 Introducción de datos: función `scanf`

Mediante la función de biblioteca `scanf` se pueden introducir datos en la computadora a través del dispositivo de entrada estándar. Esta función permite introducir cualquier combinación de valores numéricos, caracteres sueltos y cadenas de caracteres. La función devuelve el número de datos que se han conseguido introducir correctamente. Su sintaxis es

```
scanf(cadena de control,arg1,arg2,...,argN);
```

donde `cadena de control` hace referencia a una cadena de caracteres que contiene cierta información sobre el formato de los datos y `arg1,arg2,...,argN` son argumentos (punteros) que indican las direcciones de memoria donde se encuentran los datos.

Carácter de conversión	Significado del dato
`c`	Carácter
`d`	Entero decimal
`e`	Coma flotante
`f`	Coma flotante
`g`	Coma flotante
`h`	Entero corto
`o`	Entero octal
`s`	Cadena de caracteres seguida de un carácter de espaciado
`u`	Entero decimal sin signo
`x`	Entero hexadecimal
`[...]`	Cadena de caracteres que puede incluir caracteres de espaciado

Tabla 1.8: Caracteres de conversión de los datos de entrada de uso común

En la cadena de control se incluyen grupos individuales de caracteres, con un grupo de caracteres por cada dato de entrada. Cada grupo de caracteres debe comenzar con el signo de porcentaje `%`. En su forma más sencilla, un grupo de caracteres estará formado por el signo de porcentaje, seguido de un *carácter de conversión* que indica el tipo de dato correspondiente. En la Tabla 1.8 se muestran los caracteres de conversión de los datos de entrada de uso común.

Los argumentos pueden ser variables o arrays y sus tipos deben coincidir con los indicados por los grupos de caracteres correspondientes en la cadena de control. Cada

nombre de variable debe estar precedido por un carácter ampersand (`&`), salvo en el caso de los arrays.

♦ **Ejemplo 1.25:**

Considérese el siguiente programa en C:

```
#include <stdio.h>
main()
{
      char concepto[20];
      int no_partida;
      float coste;
      scanf("%s %d %f", concepto, &no_partida, &coste);
}
```

Dentro de la función `scanf` de este programa, la cadena de control es `"%s %d %f"`. Contiene tres grupos de caracteres. El primer grupo, `%s`, indica que el primer argumento (`concepto`) representa a una cadena de caracteres. El segundo grupo, `%d`, indica que el segundo argumento (`&no_partida`) representa un valor entero decimal y el tercer grupo, `%f`, indica que el tercer argumento (`&coste`) representa un valor en coma flotante.

Obsérvese que las variables numéricas `no_partida` y `coste` van precedidas por ampersands (`&`) dentro de la función `scanf`. Sin embargo, delante de `concepto` no hay ampersand, ya que `concepto` es el nombre del array.

♦

1.6.4 Escritura de datos: función `printf`

Mediante la función de biblioteca `printf` se puede escribir datos en el dispositivo de salida estándar. Esta función permite escribir cualquier combinación de valores numéricos, caracteres sueltos y cadenas de caracteres. Su sintaxis es

```
printf(cadena de control,arg1,arg2,...,argN);
```

donde `cadena de control` hace referencia a una cadena de caracteres que contiene información sobre el formato de salida y `arg1,arg2,...,argN` son argumentos que representan los datos de salida.

La cadena de control está compuesta por grupos de caracteres, con un grupo de caracteres por cada dato de salida. Cada grupo de caracteres debe empezar por un signo de porcentaje (`%`). En su forma sencilla, un grupo de caracteres consistirá en el signo de porcentaje seguido de un carácter de conversión que indica el tipo de dato correspondiente. En la Tabla 1.9 se muestran los caracteres de conversión de los datos de salida de uso común.

Carácter de conversión	Significado del dato visualizado
c	Carácter
d	Entero decimal con signo
e	Coma flotante con exponente
f	Coma flotante sin exponente
g	Coma flotante con o sin exponente según el caso. No se visualizan ni lo ceros finales ni el punto decimal cuando no es necesario
i	Entero con signo
o	Entero octal sin el cero inicial
s	Cadena de caracteres
u	Entero decimal sin signo
x	Entero hexadecimal sin el prefijo 0x

Tabla 1.9: Caracteres de conversión de los datos de salida de uso común

♦ **Ejemplo 1.26:**

Considérese el siguiente programa en C:

```
#include <stdio.h>
main()
{
      char concepto[20]="cremallera";
      int no_partida=12345;
      float coste=0.05;
      printf("%s %d %f", concepto, no_partida, coste);
}
```

Dentro de la función `printf` de este programa, la cadena de control es `"%s %d %f"`. Contiene tres grupos de caracteres. El primer grupo, `%s`, indica que el primer argumento (`concepto`) representa a una cadena de caracteres. El segundo grupo, `%d`, indica que el segundo argumento (`no_partida`) representa un valor entero decimal y el tercer grupo, `%f`, indica que el tercer argumento (`coste`) representa un valor en coma flotante.

El resultado de la ejecución de estas instrucciones del programa es visualizar en el monitor la siguiente salida:

```
cremallera 12345 0.050000
```

♦

1.6.5 Las funciones `gets` y `puts`

La función de biblioteca `gets` permite leer una cadena de caracteres terminada con el carácter de nueva línea `'\n'` desde el dispositivo de entrada estándar. La declaración de esta función es:

```
char *gets(char *s);
```

El significado de esta declaración es el siguiente: `gets` es una función que acepta un argumento que es un puntero a carácter `char *s` y devuelve un puntero a carácter `char *gets(. . .)`.

La sintaxis típica de esta función es

```
gets(s);
```

donde `s` es el nombre de la cadena de caracteres (o el nombre del puntero a carácter) donde se va almacenar la información leída. Esta función reemplaza el carácter de nueva línea `'\n'` que aparece al final de la cadena y lo sustituye por el carácter nulo `'\0'`.

Nótese que la función `gets` desconoce la longitud de `s` ya que solamente se le está pasando como argumento la dirección de inicio del array de caracteres. En consecuencia si se introduce una línea con más caracteres de los que se pueden almacenar en `s` se estaría sobrescribiendo en la zona de memoria que hay a continuación del espacio reservado a `s` y que estará asociado a otras variables, con lo que se estaría modificando el valor de las mismas. No será posible darse cuenta de este error hasta que no se usen estas variables. Este es el motivo por el cual si se compila un programa que contiene la función `gets` el compilador muestra un aviso (warning) que indica que el uso de esta función puede ser peligroso. Una posible solución es usar la función `fgets` definida en la biblioteca `stdio.h` (ver Complemento 2.A).

La función de biblioteca `puts` permite visualizar una cadena de caracteres en el dispositivo de salida estándar. Su sintaxis es

```
puts(s);
```

donde `s` es la cadena de caracteres que se desea visualizar. Esta función añade al final de la cadena `s` el carácter de nueva línea `'\n'`.

♦ **Ejemplo 1.27:**

El siguiente programa en C permite leer y escribir una línea de texto:

```
#include <stdio.h>
main()
```

```
{
      char linea[80];
      gets(linea);
      puts(linea);
}
```

♦

1.7 INSTRUCCIONES DE CONTROL EN C

1.7.1 Proposiciones y bloques

Se denomina *proposición* a una expresión seguida de punto y coma '*;*'. Asimismo se denomina *bloque o proposición compuesta* a un conjunto de declaraciones y proposiciones agrupadas entre llaves `'{', '}'`.

♦ **Ejemplo 1.28:**

Las siguientes sentencias son un ejemplo de bloque de proposiciones:

```
{  /* Comienzo bloque*/
      float z; /*Declaración*/
      /*Proposiciones*/
      z=0.256;
      z=z+1;
      printf("%f\n",z);
}  /* Final bloque*/
```

♦

1.7.2 Ejecución condicional.

1.7.2.1 La instrucción if

La instrucción `if` presenta la siguiente sintaxis

```
if(expresión) proposición;
```

La `proposición` se ejecutará sólo si `expresión` tiene un valor no nulo, es decir es `Verdadera`. En caso contrario no se ejecutará. La `proposición` puede ser simple o compuesta.

♦ **Ejemplo 1.29**

Las siguientes sentencias ilustran el uso de la instrucción `if`:

```
if (debito>0) credito=0; /*If con proposición simple*/
if(x<=3.0) {  /*If con proposición compuesta*/
      y=0.25*x;
      z=1.26*y;
```

```
        printf("%f\n",y);
}
```

♦

1.7.2.2 La instrucción if - else

La sintaxis de una instrucción `if` que incluye la sentencia `else` es:

```
if(expresión)
        proposición1;
else
        proposición2;
```

Si la `expresión` tiene un valor no nulo, es decir es `Verdadera` se ejecuta la `proposición1`, en caso contrario se ejecuta la `proposición2`. Tanto `proposición1` como `proposición2` pueden ser simples o compuestas.

♦ **Ejemplo 1.30:**

Las siguientes sentencias ilustran el uso de la instrucción `if - else`:

```
if (estado=='S')
        tasa=0.20*pago;
else
        tasa=0.14*pago;
if (debito>0) {
        prestamo=0;
        x=y+z;
}
else {
        x=y-z;
        d=0;
}
```

♦

1.7.2.3 La instrucción else if

La sintaxis de una instrucción `else if` es:

```
if(expresión1)
        proposición1;
else if(expresión2)
        proposición2;
...
else if(expresión n-1)
        proposición n-1;
```

```
else
      proposición_n;
```

1.7.3 Bucles

1.7.3.1 La instrucción for

La forma general de la instrucción `for` es:

```
for(inicialización;expresión;progresión)
      proposición;
```

donde `inicialización` se utiliza para inicializar algún parámetro (denominado índice) que controla la repetición del bucle, `expresión` representa una condición que debe ser satisfecha para que se continúe la ejecución del bucle y `progresión` se utiliza para modificar el valor del parámetro inicialmente asignado por `inicialización`. `Proposición` se ejecutará mientras `expresión` sea `Verdadera`. La proposición puede ser simple o compuesta.

♦ **Ejemplo 1.31:**

Las siguientes sentencias ilustran el uso de la instrucción `for`:

```
int dígito;
for (dígito=0; dígito<=3; dígito++)
      printf("%d\n",dígito);
```

En este bucle el índice de control `dígito` se inicializa al valor `0`. La condición que debe ser satisfecha para que se continúe la ejecución del bucle es que `dígito` sea menor o igual a `3`. El índice se incrementa en una unidad (`dígito++`) al finalizar una ejecución del bucle. En cada ejecución del bucle se muestra en el dispositivo en la pantalla el valor de `dígito`. Luego la ejecución de estas líneas generaría la siguiente traza en la pantalla:

```
0
1
2
3
```

Al finalizar la ejecución de este bucle el valor de `dígito` es `4`.

♦

1.7.3.2 La instrucción while

La forma general de la instrucción `while` es:

```
while(expresión)
      proposición;
```

`Proposición` se ejecutará mientras `expresión` sea `Verdadera`. La proposición puede ser simple o compuesta.

♦ **Ejemplo 1.32:**

Las siguientes sentencias ilustran el uso de la instrucción `while`:

```
int digito=0;
while (dígito<=9){
      printf("%d\n",dígito);
      ++dígito;
}
```

♦

1.7.3.3 La instrucción do - while

La forma general de la instrucción `do - while` es:

```
do
{
      proposición;
} while(expresión);
```

`Proposición` se ejecutará mientras `expresión` sea `Verdadera`. La primera vez siempre se ejecuta.

♦ **Ejemplo 1.33:**

Las siguientes sentencias ilustran el uso de la instrucción `do - while`:

```
int dígito=0;
do{
      printf("%d\n",dígito++);
}
while (dígito<=9)
```

♦

1.7.4 Las instrucciones `break` y `continue`

La instrucción `break` se utiliza para terminar la ejecución de bucles o salir de una instrucción `switch`. Se puede utilizar dentro de una instrucción `while`, `do - while`, `for` o `switch`.

La instrucción `break` se puede escribir sencillamente sin contener ninguna otra expresión o instrucción de la siguiente forma:

```
break;
```

La instrucción `continue` se utiliza para saltarse el resto de la pasada actual a través de un bucle. El bucle no termina cuando se encuentra una instrucción `continue`. Sencillamente no se ejecutan las instrucciones que se encuentran a continuación de `continue` y se salta directamente a la siguiente pasada a través del bucle.

La instrucción `continue` se puede incluir dentro de una instrucción `while`, `do – while` o `for`. Simplemente se escribe sin contener ninguna otra expresión o instrucción de la siguiente forma:

```
continue;
```

♦ **Ejemplo 1.34:**

Las siguientes sentencias ilustran el uso de las instrucciones `break` y `continue`:

```
int x=100;
while (x<=100){
      x=x-1
      if (x<0){
            printf("Valor negativo de x\n");
            break;
      }
      if (x==50){
            printf("Reducción de x a la mitad\n");
            continue;
      }
      printf("Decrementar\n");
}
printf("Final\n");
```

Se tiene un bucle de tipo `while` cuya condición para ejecutarse es que el valor de `x` sea menor o igual a `100`. El valor inicial de esta variable es `100`, de acuerdo a su declaración. En cada pasada del bucle en primer lugar se decrementa en una unidad el valor de `x`. En segundo lugar se comprueba si `x` es menor que cero, en caso afirmativo se imprime en la pantalla el mensaje

```
Valor negativo de x
```

y se ejecuta la instrucción `break` que hace que finalice el bucle `while`. Con lo que la próxima instrucción que se ejecuta es `printf("Final\n");`

En tercer lugar, si `x` no es menor que cero, se comprueba si es igual a `50`, en caso afirmativo se imprime en la pantalla el mensaje

```
Reducción de x a la mitad
```

y se ejecuta la instrucción `continue` que hace que se salte directamente a la siguiente pasada del bucle `while`, por lo que no se ejecuta en la pasada actual del bucle la instrucción `printf("Decrementar\n");`

En cuarto y último lugar, si las dos comprobaciones anteriores han dado resultado negativo se muestra por pantalla el mensaje

```
Decrementar
```

Y se procede a realizar la siguiente pasada a través del bucle.

♦

1.7.5 La instrucción `switch`

La instrucción `switch` hace que se seleccione un grupo de instrucciones entre varios grupos disponibles. La selección se basa en el valor de una expresión que se incluye en la instrucción `switch`. La primera instrucción de cada grupo debe ir precedida por una o varias etiquetas `case`. Estas etiquetas permiten identificar el grupo de instrucciones asociado a un determinado valor de la expresión. Uno de los grupos de instrucciones se puede etiquetar con `default`. Este grupo se seleccionará si el valor de la expresión no coincide con ninguno de los valores especificados en las etiquetas `case`. Si no se especifica un grupo de instrucciones con la etiqueta `default` y el valor de la expresión no coincide con ninguno de los valores especificados en las etiquetas `case` entonces la instrucción `switch` no hace nada.

De forma general la sintaxis de una instrucción `switch` es:

```
switch(expresión)
{
      case valor_expresión_1:
            instrucción 1;
            instrucción 2;
            ...
            break;

      case valor_expresión_2:
            instrucción 1;
            instrucción 2;
            ...
            break;
      ...
      default:
```

```
            instrucción 1;
            instrucción 2;
            ...
            break;
    }
```

También es posible asignar varias etiquetas `case` a un mismo grupo de instrucciones:

```
        case valor_expresión_1:
        case valor_expresión_2:
        case valor_expresión_3:
        ...
        case valor_expresión_m:
            instrucción 1;
            instrucción 2;
            ...
            break;
```

♦ **Ejemplo 1.35:**

Las siguientes sentencias ilustran el uso de la instrucción `switch`:

```
    char eleccion;
    switch (eleccion=getchar()){
    case 'f':
    case 'F':
        printf("Aviso 1\n");
        break;
    case 'g':
    case 'G':
        printf("Aviso 2\n");
        break;
    case 't':
    case 'T':
        printf("Aviso 3\n");
        break;
    default:
        printf("\nEntrada errónea");
}
```

Si el carácter introducido por el teclado es `'f'` o `'F'` se mostrará por pantalla el mensaje `Aviso 1`. Si el carácter introducido es `'g'` o `'G'` se mostrará por pantalla el mensaje `Aviso 2`. Finalmente si el carácter introducido es `'t'` o `'T'` se mostrará por pantalla el mensaje `Aviso 3`.

Si el carácter introducido no es ninguno de los anteriores se ejecutará el grupo de instrucciones asociado a la etiqueta `default`, que en este caso consta de una única instrucción que muestra por pantalla el mensaje

```
Entrada errónea
```

♦

1.8 FUNCIONES

Una *función* es un segmento de programa que realiza determinadas tareas bien definidas. Todo programa en C consta de una o más funciones. Una de estas funciones debe ser la función principal `main`. La ejecución del programa siempre comenzará por las instrucciones contenidas en `main`.

Si un programa contiene varias funciones, sus definiciones pueden aparecer en cualquier orden, pero deben ser independientes unas de otras. Es decir, una definición de una función no puede estar incluida en otra.

Generalmente, una función procesará la información que le es pasada desde el punto del programa en que se accede a ella y devolverá un solo valor. La información se le pasa a la función mediante unos identificadores especiales llamados *argumentos* (también denominados *parámetros*) y es devuelta por medio de la instrucción `return`. Sin embargo, algunas funciones aceptan información pero no devuelven nada (por ejemplo, la función de biblioteca `printf`), mientras que otras funciones (por ejemplo, la función `scanf`) devuelven varios valores. Una función no puede devolver otra función, ni tampoco un array. Una función puede devolver un puntero a cualquier tipo de datos. La organización de un programa grande en funciones sencillas permite que el programa sea estructurado y más fácil de depurar y mantener.

1.8.1 Definición, prototipo y acceso a una función

De forma general la definición de una función tiene la siguiente forma:

```
tipo nombre_función (tipo1 arg1, tipo2 arg2,...,tipoN argN)
{
      variables_locales;
      proposiciones;
      return(expresión);
}
```

En esta definición se observan dos componentes principales: la primera línea y el cuerpo de la función.

La primera línea de la definición de una función contiene la especificación del tipo de valor devuelto por la función (`tipo`), seguido del nombre de la función (`nombre_función`) y un conjunto de argumentos (`tipo1 arg1, tipo2 arg2,...,tipoN argN)`, separados por comas y encerrados entre paréntesis. Cada argumento viene precedido por su declaración de tipo. Es posible definir una función que no requiera argumentos en dicho caso al nombre de la función le seguirán un par de paréntesis vacíos. Los tipos de datos se suponen enteros sino se indican explícitamente. Sin embargo, la omisión de los tipos de datos se considera una práctica de programación poco elegante.

Los argumentos de la definición de una función se denominan *argumentos o parámetros formales*, ya que representan los nombres de los elementos que se transfieren a la función desde la parte del programa que hace la llamada. Los identificadores que son usados como argumentos formales son locales, es decir, no son reconocidos fuera de la función.

Al resto de líneas que componen la definición de la función se le denomina el *cuerpo de la función* y contiene los siguientes elementos: la definición de diferentes variables locales, diversas proposiciones y una instrucción `return` para devolver el valor de `expresión` al punto del programa desde donde se llamó a la función. La aparición de `(expresión)` en `return` es opcional, si se omite simplemente se devuelve el control al punto de llamada, sin transferir ninguna información. Sólo se puede incluir una expresión en la instrucción por lo tanto, una función sólo puede devolver un valor al punto de llamada mediante la instrucción `return`. No es obligatorio que la instrucción `return` aparezca en la definición de una función, sin embargo su omisión se considera una práctica de programación poco elegante.

Se denomina *prototipo de una función* a la primera línea de una definición de función añadiendo al final un punto y coma. La forma general del prototipo de una función es por lo tanto:

```
tipo nombre_función (tipo1 arg1, tipo2 arg2,...,tipoN argN);
```

Los prototipos de funciones normalmente se escriben al comienzo del programa, delante de todas las funciones definidas por el programador (incluida `main`). Los prototipos de funciones no son obligatorios en C. Sin embargo, son aconsejables ya que facilitan la comprobación de errores.

Al escribir el prototipo de una función es posible omitir los nombres de sus argumentos, sin embargo los tipos de datos de dichos argumentos no pueden ser omitidos ya que son esenciales. Además los nombres de los argumentos del prototipo de una función pueden ser distintos a los nombres de los argumentos de la definición de esa misma función. Es por este motivo por el que a los argumentos del prototipo de una función se les denomina en ocasiones *argumentos ficticios*.

Por otra parte, se puede *llamar o acceder a una función* especificando su nombre, seguido de una lista de argumentos encerrados entre paréntesis y separados por comas.

Los argumentos o parámetros que aparecen en la llamada a la función se denominan *argumentos reales*, en contraste con los argumentos formales que aparecen en la primera línea de la definición de la función. Los argumentos reales pueden ser constantes, variables simples, o expresiones más complejas. El nombre de los argumentos reales puede ser distinto del nombre de los argumentos formales. El número de argumentos reales debe coincidir con el número de argumentos formales. Además cada argumento real debe ser del mismo tipo de datos que el correspondiente argumento formal. El valor de cada argumento real es transferido a la función y asignado al correspondiente argumento formal.

♦ **Ejemplo 1.36:**

Considérese el siguiente programa escrito en lenguaje C que calcula el factorial de un número entero positivo menor de 26 que debe ser introducido por el usuario a través del dispositivo estándar de entrada (usualmente el teclado).

```
#include <stdio.h>
double factorial(int x);
main()
{
      int n, ok=1;

      while (ok==1)
      {
            printf("\n n= ");
            scanf("%d", &n);
            if (n<=25)
            {
                  if (n<0) n=-n;
                  printf("\n n!= %.0f", factorial(n));
                  ok=0;
            }
```

```
            else
                  printf("\n Error n>25\n");
      }
}

double factorial(int x)
{
      int i;
      double prod=1;
      if (x>1)
            for(i=2;i<=x;++i)
                  prod *=i;
      return(prod);
}
```

En este programa se definen dos funciones: `main` y `factorial`. El acceso que se realiza a la función `factorial` en este programa tiene como argumento real al entero `n` cuyo valor es pasado al argumento formal `x`. Obsérvese por lo tanto que no es necesario que el nombre del argumento real coincida con el del argumento formal pero si debe ser el mismo tipo de dato.

El prototipo de `factorial` aparece antes de la definición de la función `main`, esto indica al compilador que más adelante en el programa se definirá una función `factorial`, que acepta una cantidad entera y devuelve un número en coma flotante de doble precisión. Nótese que aunque el factorial de un número entero *n* es otro número entero para evitar errores numéricos se ha usado el tipo `double` (64 bits) en vez del tipo `int` (8 bits) o `long int` (32 bits).

Otras formas posibles de escribir el prototipo de la función `factorial` serían:

```
double factorial(int);
double factorial(int var);
```

En el primer caso se ha omitido el nombre del argumento, mientras que en el segundo se ha utilizado otro nombre distinto para dicho argumento.

♦

1.8.2 Paso de argumentos a una función

1.8.2.1 Formas de pasar argumentos a una función

Existen dos formas de pasar los argumentos a una función: *paso por valor* y *paso por referencia*.

Cuando se le pasa un valor simple a una función mediante un argumento real, se copia el valor del argumento real a la función. Por tanto, se puede modificar el valor del argumento formal dentro de la función, pero el valor del argumento real en la rutina que

efectúa la llamada no cambiará. Este procedimiento para pasar el valor de un argumento a una función se denomina *paso por valor*.

A menudo los punteros son pasados a la funciones como argumentos. Esto permite que datos de la parte del programa en la que se llama a la función sean accedidos por la función, modificados dentro de ella y luego devueltos modificados al programa. Este procedimiento para pasar el valor de un argumento a una función se denomina *paso por referencia*.

Cuando se pasa un argumento por referencia la dirección del dato es pasada a la función. El contenido de esta función puede ser accedido libremente, tanto dentro de la función como dentro de la rutina de la llamada. Además cualquier cambio que se realiza al dato (al contenido de la dirección) será reconocido en ambas, la función y la rutina de llamada. Así, el uso de punteros como argumentos de funciones permite que el dato sea alterado globalmente desde dentro de la función

Cuando los punteros se utilizan como argumentos formales de una función deben ir precedidos por un asterisco `*`. Esta regla se aplica también a los prototipos de las funciones.

♦ **Ejemplo 1.37:**

Considérese el siguiente programa escrito en lenguaje C

```
#include <stdio.h>
void f1(int u, int v);     /*Prototipo de la función f1*/
void f2(int *pu, int *pv); /*Prototipo de la función f2*/
main()
{
      int x=2;
      int y=4;
      printf("\nAntes de la llamada a f1:  x=%d  y=%d", x, y);
      f1(x,y);
      printf("\nDespués de la llamada a f1:  x=%d  y=%d", x, y);
      printf("\nAntes de la llamada a f2:  x=%d  y=%d", x, y);
      f2(&x,&y);
      printf("\nDespués de la llamada a f2:  x=%d  y=%d", x, y);
}
void f1(int u, int v)
{
      u=0;
      v=0;
```

```
        printf("\nDentro de f1:  u=%d  v=%d", u, v);
        return;
}
void f2(int *pu, int *pv)
{
        *pu=0;
        *pv=0;
        printf("\nDentro de f2:  *pu=%d  *pv=%d", *pu, *pv);
        return;
}
```

Este programa contiene dos funciones `f1` y `f2`. La función `f1` tiene como argumentos dos variables enteras. Estas variables tienen asignado inicialmente los valores `2` y `4`, respectivamente. Los valores son modificados a `0` y `0` dentro de `f1`. Sin embargo, los nuevos valores no son reconocidos en `main`, ya que los argumentos fueron pasados por valor y cualquier cambio sobre los argumentos únicamente es local a la función en la cual se han producido los cambios.

La función `f2` tiene como argumentos dos punteros a variables enteras. Dentro de esta función los contenidos de las direcciones apuntadas son modificados a `0` y `0`. Como estas direcciones son reconocidas tanto en `f2` como en `main`, los nuevos valores serán reconocidos dentro de `main` tras la llamada a `f2`. Por lo tanto, las variables enteras `x` e `y` habrán cambiado sus valores de `2` y `4` a `0` y `0`.

De acuerdo con el análisis realizado la ejecución de este programa genera la siguiente salida por el dispositivo de salida estándar:

```
Antes de la llamada a f1: x=2  y=4
Dentro de f1:  x=0  y=0
Después de la llamada a f1:  x=2  y=4
Antes de la llamada a f2:  x=2  y=4
Dentro de f2:  x=0  y=0
Después de la llamada a f2:  x=0  y=0
```

Finalmente, comentar que otra forma equivalente de escribir los prototipos de las funciones `f1` y `f2` es omitiendo los nombres de los argumentos:

```
void f1(int, int);
void f2(int *, int *);
```

♦

1.8.2.2 Paso de arrays a una función

Para pasar un array (unidimensional o multidimesional) a una función como argumento real únicamente se escribe el nombre del array sin corchetes ni índices. Recuérdese que el nombre de un array representa la dirección del primer elemento del array y por lo tanto se trata como un puntero cuando se pasa a una función. En

consecuencia, el paso de arrays como parámetros reales de funciones siempre se realiza por referencia.

Por otra parte, cuando se declara un array multidimensional como un argumento formal se debe especificar el tamaño en todos los índices excepto en el primero (el situado más a la izquierda) cuyo par de corchetes se deja vacío. El prototipo correspondiente debe escribirse de la misma manera.

♦ **Ejemplo 1.38:**

Considérese el siguiente boceto de programa escrito en lenguaje C

```
float fun1(int x, float a[], int b[][3]);
main()
{
      int n;
      float vector[4];
      int matriz[2][3];
      ...
      r=fun1(n,vector,matriz);
      ...
}
float fun1(int x, float a[], int b[][3])
{
...
}
```

Dentro de la función `main` existe una llamada a la función `fun1`, que tiene tres argumentos reales: la variable entera `n`, el array unidimensional de cuatro números en coma flotante `vector` y el array bidimensional de 2 filas y 3 columnas de números enteros `matriz`. Obsérvese que ni `vector` ni `matriz` incluyen sus corchetes e índices.

Por otra parte, la primera línea de la definición de la función incluye tres argumentos formales: la variable entera `x`, el array unidimensional de números en coma flotante `a` y el array bidimensional de números enteros `b`. Obsérvese que dentro de la declaración formal del argumento `a` no se especifica el tamaño del array, aparece únicamente el par de corchetes vacío. Asimismo dentro de la declaración formal del argumento `b` no se especifica el primer índice apareciendo su par de corchetes asociados vacíos.

Finalmente comentar que el prototipo de `fun1` se podría haber escrito equivalentemente omitiendo los nombres de los argumentos:

```
float fun1(int, float[], int[][3]);
```

♦

1.8.2.3 Paso de estructuras o uniones a una función

Una estructura o una unión se pueden transferir por referencia a una función pasando un puntero a la estructura como argumento. En este caso si cualquiera de los miembros de una estructura es alterado dentro de la función, las alteraciones serán reconocidas fuera de la función. Asimismo, una estructura se puede pasar por valor a una función. De este modo si cualquiera de los miembros de la estructura es alterado dentro de la función, las alteraciones no serán reconocidas fuera de la función. Los compiladores más recientes de C permiten que una estructura completa sea transferida directamente a una función como argumento y devuelta directamente mediante la instrucción `return`.

También es posible pasar los miembros de una estructura como argumentos de la llamada a una función. Asimismo, un miembro de una estructura puede ser devuelto por una función mediante una instrucción `return`.

♦ **Ejemplo 1.39:**

Considérese el siguiente programa escrito en lenguaje C:

```
#include <stdio.h>
typedef struct{
                  long int dni;
                  float nota;
            } datos;
void cambiar(datos *a);
main()
{
      datos alumno={70534213, 7.5};
      printf("%ld %.1f\n", alumno.dni, alumno.nota);
      cambiar(&alumno);
      printf("%ld %.1f\n", alumno.dni, alumno.nota);
}
void cambiar(datos *a)
{
      a->dni=23789345;
      a->nota=8.0;
      return;
}
```

Este programa ilustra la transferencia de una estructura a una función por referencia. Se tiene una estructura `alumno` del tipo `datos` cuyos miembros tienen asignado unos valores iniciales. Cuando el programa se ejecuta en primer lugar se visualizan en la pantalla los valores iniciales de los

miembros de la estructura. A continuación se llama a la función `cambiar` pasándole como argumento la dirección de la estructura, es decir, un puntero. Dentro de esta función se le asignan nuevos valores a los miembros de la estructura. Finalmente estos nuevos valores son visualizados en la pantalla. De acuerdo con el funcionamiento comentado la ejecución de este programa genera la siguiente salida por pantalla:

```
70534213 7.5
23789345 8.0
```

♦

♦ **Ejemplo 1.40:**

Considérese el siguiente programa escrito en C:

```
#include <stdio.h>
typedef struct {
                  long int dni;
                  float nota;
            } datos;
void cambiar(datos a);
main()
{
      datos alumno={70534213, 7.5};
      printf("%ld %.1f\n", alumno.dni, alumno.nota);
      cambiar(alumno);
      printf("%ld %.1f\n", alumno.dni, alumno.nota);
}
void cambiar(datos a)
{
      a.dni=23789345;
      a.nota=8.0;
      return;
}
```

Este programa es similar al del ejemplo anterior pero ha sido convenientemente modificado para ilustrar la transferencia de una estructura a una función por valor. Nótese como ahora la función `cambiar` acepta una estructura del tipo `datos` en vez de un puntero a este tipo de estructura. Cuando se ejecuta este programa, se obtiene la siguiente salida:

```
70534213 7.5
70534213 7.5
```

Se comprueba que al haber pasado la estructura por valor a la función, los cambios realizados a los miembros de la estructura dentro de la función no son reconocidos fuera de la misma. Si se dispone de un compilador de C reciente toda la estructura puede ser devuelta al punto de llamada de la función. Simplemente habría que hacer algunos pequeños cambios en el código del programa:

```
#include <stdio.h>
typedef struct {
                  long int dni;
                  float nota;
            } datos;
datos cambiar(datos a);
main()
{
      datos alumno={70534213, 7.5};
      printf("%ld %.1f\n", alumno.dni, alumno.nota);
      alumno=cambiar(alumno);
      printf("%ld %.1f\n", alumno.dni, alumno.nota);
}
datos cambiar(datos a)
{
      a.dni=23789345;
      a.nota=8.0;
      return(a);
}
```

El prototipo y la primera línea de la definición de `cambiar` deben especificar el tipo `datos` como tipo de salida. La llamada a la función `cambiar` debe escribirse en la parte derecha de una expresión de asignación. La instrucción `return` dentro de la definición de `cambiar` debe devolver la estructura `a`.

La salida de este programa sería:

```
70534213 7.5
23789345 8.0
```

♦

La mayoría de los compiladores de C permiten transferir estructuras de datos complejas libremente entre funciones. Sin embargo, algunos compiladores pueden tener dificultades al ejecutar programas que involucran transferencias de estructuras de datos complejas, debido a ciertas restricciones de memoria.

1.8.3 Devolución de un puntero por una función

Una función puede devolver un puntero de cualquier tipo de dato a la parte del programa que hizo la llamada a una función. Esta característica puede ser útil, por ejemplo, cuando se pasan varias estructuras a una función y sólo una de ellas es devuelta.

♦ **Ejemplo 1.41:**

El prototipo

```
char *func(char *s);
```

declara una función llamada `func` que acepta un puntero `s` a carácter y devuelve un puntero a carácter.

El prototipo

```
int *p(int b, float *C);
```

declara una función llamada `p` que acepta un entero `b` y un puntero `C` a un número en coma flotante. La función devuelve un puntero a un número entero.

♦

♦ **Ejemplo 1.42:**

Considérese el siguiente programa escrito en C:

```
#include <stdio.h>
#define T 3
#define NULL 0
typedef struct {
                  long int dni;
                  float nota;
            } datos;
datos *buscar(datos a[], long int b);
main()
{
      datos alumnos[T]={
                        {70534213, 7.5},
                        {33356897, 8.5},
                        {85963472, 7.0}
                        };
      long int id;
      datos *pr;
      printf("\nIntroduzca el DNI del alumno: ");
      scanf("%ld", &id);
      pr=buscar(alumnos, id);
      if (pr!=NULL)
            printf("Nota[%ld]= %.1f\n", pr->dni, pr->nota);
      else
            printf("\nDNI no encontrado");
}
datos *buscar(datos a[], long int b)
```

```
{
        int h;
        for(h=0;h<T;++h)
        if(a[h].dni==b)
                return(&a[h]);
        else
                return(NULL);
}
```

En este programa dentro de la función `main` se define el array `alumnos` de tres estructuras del tipo `datos` cuyos miembros `dni` y `nota` tienen asignados unos valores iniciales. También se define un entero largo `id` y un puntero a una estructura del tipo `datos`. Cuando el programa se ejecuta en primer lugar se visualiza en la pantalla el mensaje

```
Introduzca el DNI del alumno:
```

En segundo lugar el programa se queda a la espera de que el usuario introduzca un entero largo y pulse la tecla de salto de línea. El valor introducido se asigna a la variable `id`. En tercer lugar, se llama a la función `buscar` pasándole como argumentos reales la dirección del array de estructuras `alumnos` y el valor de la variable `id`. Comienza por lo tanto a ejecutarse la función `buscar` que lo que hace es recorrer todo el array de estructuras hasta encontrar alguna cuyo campo `dni` coincida con `id`. Si se produce alguna coincidencia, la función devuelve a `main` un puntero a dicha estructura del array, se muestra en la pantalla el mensaje

```
Nota[dni]= nota
```

y el programa finaliza.

Si no se produce ninguna coincidencia después de buscar en todo el array, entonces la función `buscar` devuelve el valor `NULL` (cero) a `main`, se muestra por pantalla el mensaje

```
DNI no encontrado
```

y el programa finaliza.

♦

1.8.4 Punteros a funciones

Las funciones aunque no son variables tienen de igual modo una posición física en memoria, a la cual se le puede asignar un puntero. La dirección de memoria de una función es la entrada a la función, por tanto se puede usar un puntero para ejecutar una función y este puntero nos permitirá también pasar funciones como argumentos a otras funciones. Una aplicación típica del paso de punteros a funciones es el tratamiento de las interrupciones. Usando punteros a funciones se puede capturar una interrupción y tratarla usando una función determinada.

Un puntero a una función puede ser pasado como argumento a otra función. Esto permite que una función sea transferida a otra, como si la primera función fuera una variable. A la primera función se la denomina *función huésped* y a la segunda función se la denomina *función anfitriona*. De este modo la función huésped es pasada a la anfitriona, donde puede ser accedida.

La declaración de la función anfitriona se puede escribir de forma general de la siguiente forma:

```
tipo_a nombre_fun_a (puntero_fun_h, otros)
```

donde `tipo_a` es el tipo de dato que devuelve la función anfitriona cuyo nombre es `nombre_fun_a`. Esta función recibe como argumentos formales un puntero a la función huésped (`puntero_fun_h`). También puede recibir, como cualquier otra función, varios argumentos formales de diferentes tipos (`otros`).

La declaración de un puntero a una función huésped como argumento formal (`puntero_fun_h`) se puede escribir cómo:

```
tipo_h(*nombre_fun_h)(tipo1 arg1, tipo2 arg2,..., tipoN argN)
```

donde `tipo_h` es el tipo de dato de la cantidad devuelta por la función huésped, `nombre_fun_h` es el nombre de la función huésped, `tipo1, tipo2,..,tipoN` se refieren a los tipos de datos de los argumentos asociados con la función huésped y `arg1, arg2,..,argN` son los nombres de los argumentos asociados con la función huésped.

La declaración del argumento formal `puntero_fun_h` también se puede escribir omitiendo el nombre de los argumentos formales:

```
tipo_h(*nombre_fun_h)(tipo1, tipo2,..., tipoN);
```

Para acceder a la función huésped dentro de la función anfitriona el operador `*` debe preceder al nombre de la función huésped y ambos deben estar encerrados entre paréntesis, es decir:

```
(*nombre_fun_h)(argumento1, argumento2,..., argumentoN);
```

donde `argumento1, argumento2,..., argumentoN` son los argumentos reales de la llamada a la función.

♦ **Ejemplo 1.43:**

Considérese el siguiente programa escrito en C:

```
#include <stdio.h>
```

```
float a1(float (*b)(float a));
float a2(float a);
main(){
      int u[3]={501,501,0};
      float z=0;
      z=a1(a2);
      printf("\n[%d, %d, %d, %.2f]\n",u[0],u[1],u[2],z);
}
float a1(float (*b)(float a))
{
      float u=30.4;
      u=(*b)(u);
      return(u);
}
float a2(float a){
      int h;
      for(h=-1;h<3;h++)
            a=0.5*a;
      return(a);
}
```

Cuando se ejecuta este programa en primer lugar se invoca a la función anfitriona `a1` pasándole como argumento real la dirección de memoria de la función huésped `a2`.

La primera acción que se realiza al ejecutar la función `a1` es invocar a la función huésped `a2` pasándole como argumento real el valor inicial de la variable `u`, que es 30.4. Se comienza a ejecutar la función `a2`. La primera acción asociada a `a2` es ejecutar un bucle 4 veces dentro del cual se multiplica el valor de la variable `a` por 0.5. El resultado de la multiplicación se asigna a la variable `a`. En la primera ejecución del bucle (`h=-1`) `a`=0.5*30.4=15.2, en la segunda ejecución (`h=0`) `a`=0.5*15.2=7.6, en la tercera ejecución (`h=1`) `a`=0.5*7.6=3.8 y en la cuarta ejecución (`h=2`) `a`=0.5*3.8=1.9.

Finalizado el bucle se ejecuta la instrucción `return(a)` con lo que la función `a2` finaliza devolviendo 1.9 como valor de salida que es asignado a la variable `u` de la función `a1`. A continuación se ejecuta la instrucción `return(u)` de la función `a1` con lo que finaliza devolviendo el valor 1.9 como valor de salida que es asignado a la variable `z` del código principal.

Por último se imprime en pantalla el mensaje

```
[501, 501, 0, 1.90]
```

y el programa finaliza su ejecución.

Comentar que el prototipo de la función anfitriona `a1` se podría haber escrito omitiendo el nombre de la función huesped y el nombre de su argumento:

```
float a1(float (*)(float));
```

Asimismo la primera línea de la declaración de la función anfitriona se podría haber escrito omitiendo el nombre del argumento de la función huésped:

```
float a1(float (*b)(float))
```

♦

1.8.5 Argumentos de la función main()

Es posible pasar argumentos a la función `main()`, de tal modo que los use como opciones iniciales en la ejecución del programa desde la línea de comandos. En ese caso la primera línea de la definición de `main` es:

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

El significado de los argumentos formales de `main` es el siguiente:

- `int argc`. Es un número entero que contiene el número de argumentos de la línea de comandos. Como mínimo este argumento puede valer uno, puesto que el nombre del programa cuenta como primer argumento.

- `char *argv[]`. Es un array de punteros a caracteres. Cada puntero del array apunta a un argumento de la línea de ordenes. El contenido de `argv[0]` es el nombre del programa.

Los nombres `argc` y `argv` pueden ser sustituidos por otros nombres. Todos los argumentos de la línea de comandos son cadenas y deben ir separados por espacios en blanco. Si alguno de los argumentos de la línea de comandos se va a usar numéricamente en el programa, es obligación del programador pasar el argumento, que se considera una cadena, al tipo de datos numérico adecuado para la aplicación. Para realizar estas operaciones C tiene una extensa librería de funciones que permiten pasar datos de un formato a otro. Por ejemplo, la función `atoi` incluida en el fichero de cabecera `stdlib.h` convierte una cadena de caracteres numéricos en un número entero.

Cuando un programa usa argumentos la ausencia de uno de ellos en su invocación desde la línea de comandos puede provocar la ejecución incorrecta del programa. Luego es obligación del programador comprobar las condiciones iniciales de ejecución del programa.

♦ **Ejemplo 1.44:**

Considérese el siguiente programa en C:

```
#include <stdio.h>
#include <stdlib.h>
void main (int argc,char *argv[])
{
      if (argc<2)
      {
             printf ("Error, falta clave de acceso\n");
             exit(0);
      }
      else
      {
            if (!strcmp(argv[1],"Azul") )
                  printf("Acceso al programa...\n);
            else
            {
                  printf ("Acceso denegado\n");
                  exit(0);
            }
      }
```

Supuesto que el ejecutable de este programa lleva por nombre `clave`, en la línea de ordenes de la consola habría que llamarlo de la siguiente forma:

```
$ clave azul
```

Entonces aparecería el siguiente mensaje

```
Acceso al programa...
```

Por el contrario, si se llamase por ejemplo de la siguiente forma:

```
$ clave rojo
```

Entonces aparecería el siguiente mensaje

```
Acceso denegado
```

Finalmente, si se llamase por ejemplo de la siguiente forma:

```
$ clave
```

Entonces aparecería el siguiente mensaje

```
Error, falta clave de acceso
```

♦

1.9 ASIGNACIÓN DINÁMICA DE MEMORIA

Se denomina *asignación dinámica de memoria* a la acción de guardar un espacio de memoria de tamaño variable durante la ejecución de un programa. Un ejemplo típico donde es necesario realizar la asignación dinámica de memoria es cuando se usa un array de punteros para implementar un array multidimensional.

Las funciones `malloc` y `free` definidas en el fichero de biblioteca `stdlib.h` permiten reservar y liberar memoria, respectivamente, de una forma dinámica. Estrechamente relacionado con estas funciones se encuentra el operador `sizeof` que devuelve el tamaño en bytes de su operando. Su sintaxis es:

```
sizeof(operando)
```

♦ **Ejemplo 1.45:**

Considérese el siguiente programa:

```
#include <stdio.h>
main()
{
   int i;
   float x;
   double d;
   char c;

   printf("Entero: %d\n", sizeof(i));
   printf("Coma flotante: %d\n", sizeof(x));
   printf("Doble precisión: %d\n", sizeof(d));
   printf("Carácter: %d\n", sizeof(c));
}
```

Este programa muestra por pantalla el número de bytes asignados por el compilador a los tipos de datos `int`, `float`, `double` y `char`:

```
Entero: 2
Coma flotante: 4
Doble precisión: 8
Carácter: 1
```

Es importante observar que el número de bytes asignado a cada tipo de datos puede variar dependiendo del compilador utilizado.

♦

Una vez analizado el operador `sizeof`, es posible describir una de las sintaxis más habituales para la función `malloc`:

```
ptr= (tipo*) malloc(N*sizeof(tipo));
```

donde `tipo` hace referencia a un tipo de datos (`int`, `float`, `char`, ...) y `N` indica el número de elementos del tipo `tipo` para los que se va reservar espacio (en bytes) en memoria. Si la función se ejecuta con éxito entonces en `ptr` se almacena un puntero a la zona de memoria reservada. En caso contrario (cuando no pueda reservar memoria), devolverá `NULL`. Es importante resaltar que el espacio reservado por `malloc` está sin inicializar.

Por su parte la sintaxis de la función `free` es:

```
free(ptr);
```

donde `ptr` es un puntero previamente inicializado con `malloc`. Si la función se ejecuta correctamente libera la zona de memoria apuntada por `ptr`.

♦ **Ejemplo 1.46:**

Considérese el siguiente programa escrito en C:

```
#include <stdio.h>
#include <stdlib.h>
void fun1(float a[],int b[][3]);
void fun2(float *a, int *b[2]);
main()
{
        float vector[]={1.5,2.5,3.5};
        int matriz[2][3]={
                          {2, 1, 3},
                          {4, 5, 6}
                    };
        int *d[2];
        d[0]=(int *) malloc(3*sizeof(int));
        d[1]=(int *) malloc(3*sizeof(int));
        *(d[0])=2;
        *(d[0]+1)=1;
        *(d[0]+2)=3;
        *(d[1])=4;
        *(d[1]+1)=5;
        *(d[1]+2)=6;
        fun1(vector, matriz);
        fun2(vector, d);
```

```
        free(d[0]);
        free(d[1]);
}
void fun1(float a[],int b[][3])
{
  printf("\n----Fun1----\nVector: %g  %g  %g\n",a[0],a[1],a[2]);
  printf("Fila 1: %d  %d  %d\n",b[0][0],b[0][1],b[0][2]);
}
void fun2(float *a, int *b[2])
{
  printf("\n----Fun2----\nVector: %g  %g  %g\n",*a,*(a+1),*(a+2));
  printf("Fila 1: %d  %d  %d\n",*b[0],*(b[0]+1),*(b[0]+2));
  printf("Fila 1: %d  %d  %d\n",*(*b),*(*(b)+1),*(*(b)+2));
  printf("Fila 2: %d  %d  %d\n",*b[1],*(b[1]+1),*(b[1]+2));
  printf("Fila 2: %d  %d  %d\n",*(*(b+1)),*(*(b+1)+1),*(*(b+1)+2));
}
```

En este programa ilustra como es posible usar un array de punteros para implementar un array multidimensional y como en dicho caso es necesario realizar una asignación dinámica de memoria. Se tiene un array `d` de dos punteros a números enteros con el que se desea implementar un array bidimensional de dos filas y tres columnas. En consecuencia es necesario reservar memoria para los tres enteros de la primera fila cuyo elemento 0 (primer elemento) es apuntado por `d[0]`y los tres enteros de la segunda fila cuyo primer elemento es apuntado por `d[1]`. Este programa muestra la siguiente traza de ejecución en pantalla:

```
----Fun1----
Vector: 1.5  2.5  3.5
Fila 1: 2  1  3
----Fun2----
Vector: 1.5  2.5  3.5
Fila 1: 2  1  3
Fila 1: 2  1  3
Fila 2: 4  5  6
Fila 2: 4  5  6
```

♦

COMPLEMENTO 1.A
Forma alternativa del uso de punteros para referirse a un array multidimensional

Los arrays multidimensionales pueden implementarse alternativamente como un puntero a un grupo de arrays unidimesionales contiguos en vez de como un array de punteros. Así la declaración de un array multidimensional de orden N

```
tipo_array nombre_array[rango1][rango2]...[rangoN];
```

puede sustituirse equivalentemente por:

```
tipo_array (*nombre_puntero)[rango2]...[rangoN];
```

Obsérvese que el nombre del puntero al array y el asterisco que le precede van entre paréntesis. Esto no es algo arbitrario sino que realmente la escritura de estos paréntesis es necesaria, ya que en caso contrario se estará definiendo un array de punteros en vez de un puntero a un grupo de arrays. Los corchetes y el asterisco se evalúan normalmente de derecha a izquierda. Además el índice `[rango1]` ya no se escribe.

Al igual que ocurría cuando se utilizaba un array de punteros para implementar un array multidimensional, si se utiliza un puntero a un grupo de arrays unidimesionales contiguos la reserva de memoria la debe realizar el programador de forma explícita en el código del programa mediante el uso de la función `malloc`.

Se puede acceder a un elemento individual de un array multidimensional mediante la utilización repetida del operador `*`.

♦ **Ejemplo 1A.1:**

Considérese que `z` es un array bidimensional de números en coma flotante con 3 filas y 4 columnas. Se puede declarar `z` como

```
float z[3][4];
```

o equivalentemente como

```
float (*z)[4];
```

En el segundo caso, `z` se define como un puntero a un grupo de arrays unidimensionales consecutivos de 4 elementos en coma flotante. Así `z` apunta al primero de los arrays de 4 elementos, que es en realidad la primera fila (fila 0) del array bidimensional original. Análogamente (`z+1`) apunta al segundo array de 4 elementos, que es la segunda fila (fila 1) del array bidimensional original, y así sucesivamente.

Para acceder al elemento de la fila 2 situado en la columna 3 (`z[2][3]`) se puede escribir

```
*(*(z+2)+3)
```

El significado de esta expresión (ver Figura 1A.1) es el siguiente: `(z+2)` es un puntero a la fila 2. Como la fila 2 es un array unidimensional `*(z+2)` es realmente un puntero al primer elemento de la fila 2. Se le suma 3 a ese puntero, Por lo tanto, `(*(z+2)+3)` es un puntero al elemento 3 (el cuarto elemento) de la fila 2. Luego `*(*(z+2)+3)` se refiere al elemento en la columna 3 de la fila 2, que es `z[2][3]`.

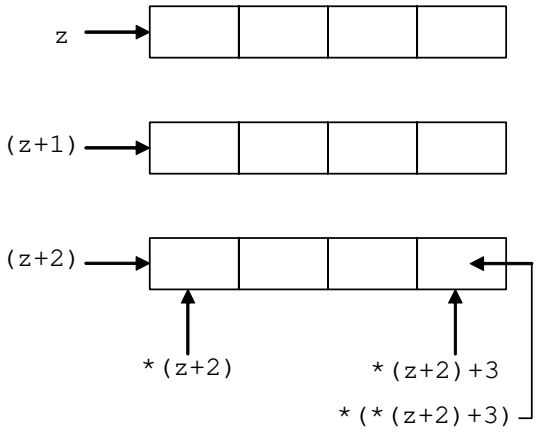

Figura 1A.1: Forma alternativa del uso de punteros para referirse a un array bidimensional de 3 filas y 4 columnas.

♦

COMPLEMENTO 1.B
Macros

La directiva del preprocesador `#define` aparte de para definir constantes también se emplea para definir *macros*, es decir, identificadores simples que son equivalentes a expresiones, a instrucciones completas o a grupos de instrucciones. En este sentido las macros se parecen a las funciones. No obstante, son definidas y tratadas de forma diferente que las funciones durante el proceso de compilación.

Las definiciones de macros están normalmente colocadas al principio de un archivo, antes de la definición de la primera función. El ámbito de definición de una macro va desde el punto de definición hasta el final del archivo donde ha sido definida. Sin embargo una macro definida en un archivo no es reconocida dentro de otro archivo.

Pueden ser definidas macros con varias líneas colocando una barra invertida (`\`) al final de cada línea excepto en la última. Esta característica permite que una sola macro (un identificador simple) represente una instrucción compuesta.

Una definición de *macro* puede incluir argumentos que están encerrados entre paréntesis. El paréntesis izquierdo debe aparecer inmediatamente detrás del nombre de la macro, es decir, no pueden existir espacios entre el nombre de la macro y el paréntesis izquierdo.

♦ **Ejemplo 1B.1:**

Supóngase el siguiente programa en C:

```
#include <stdio.h>
#define bucle(n) for (lineas=1; lineas<=n; lineas++){           \
                        for(cont=1; cont<=n-lineas; cont++)     \
                              putchar(' ');                     \
                        for(cont=1; cont<=2*lineas-1;cont++)    \
                              putchar(' ');                     \
                        printf("\n");                           \
                 }
main()
{
      int cont, lineas, n;
      printf("número de líneas= ");
      scanf("%d", &n);
      printf("\n");
      bucle(n);
}
```

Este programa contiene una macro `bucle(n)` de varias líneas, que representa a una instrucción compuesta. La instrucción compuesta consta de varios bucles `for` anidados. Notar la barra invertida (\) al final de la línea, excepto en la última.

Cuando el programa es compilado, la referencia a la macro es reemplazada por las instrucciones contenidas dentro de la definición de la macro. Así, el programa mostrado anteriormente se convierte en

```
main()
{
      int cont, lineas, n;
      printf("número de líneas= ");
      scanf("%d", &n);
      printf("\n");
      for (lineas=1; lineas<=n; lineas++){
            for(cont=1; cont<=n-lineas; cont++)
                  putchar(' ');
            for(cont=1; cont<=2*lineas-1;cont++)
                  putchar(' ');
```

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

♦

A veces las macros son usadas en lugar de funciones dentro de un programa. El uso de una macro en lugar de una función elimina el retraso asociado con la llamada a la función. Si el programa contiene muchas llamadas a funciones repetidas, el tiempo ahorrado por el uso de macros puede ser significativo.

Por otra parte, la sustitución de la macro se realizará en todas las referencias a la macro que aparezcan dentro de un programa. Así un programa que contenga varias referencias a la misma macro puede volverse excesivamente largo. Por tanto, se debe llegar a un compromiso entre la velocidad de ejecución y el tamaño del programa objeto compilado. El uso de la macro es más ventajoso en aplicaciones donde hay relativamente pocas llamadas a funciones pero la función es llamada repetidamente (por ejemplo, una función llamada dentro de un bucle).

COMPLEMENTO 1.C
Principales archivos de cabecera

El lenguaje C dispone de un gran número de *funciones de biblioteca* que realizan varias operaciones y cálculos de uso frecuente. Las funciones de biblioteca de propósitos relacionados se suelen encontrar agrupadas en programas objeto en *archivos de biblioteca o librerías* separados. Estos archivos de biblioteca se proporcionan como parte de cada compilador de C. Las prototipos de todas las funciones que forman parte de una misma librería se encuentran agrupados en un archivo denominado *archivo de cabecera* que se denota con la extensión `.h`. En este archivo también se pueden incluir: declaraciones de constantes, declaraciones de tipos de datos y macros. Las principales archivos de cabecera incluidos en la mayoría de los compiladores de C son:

- `<alloc.h>`. Contiene los prototipos de funciones para obtener y liberar memoria.
- `<ctype.h>`. Contiene los prototipos de funciones que indican características de los caracteres, por ejemplo, si está en mayúscula o en minúscula.
- `<errno.h>`. Contiene la definición de varias constantes y variables, entre ellas la variable global `errno`. Esta variable contiene el identificador numérico del error que se ha producido durante la ejecución de una llamada al sistema.

- `<fcntl.h>`. Define los indicadores o flags para los modos de apertura de un fichero.
- `<float.h>`. Establece algunas propiedades de las representaciones del tipo coma flotante.
- `<limits.h>`. Contiene macros que determinan varias propiedades de las representaciones de tipos enteros.
- `<math.h>`. Contiene los prototipos de funciones matemáticas elementales, entre ellas, las funciones trigonométricas, exponenciales y logarítmicas.
- `<stdarg.h>`. Contiene los prototipos de funciones que permiten acceder a los argumentos adicionales sin nombre en una función que acepta un número variable de argumentos.
- `<stdio.h>`. Incluye macros y los prototipos de funciones para realizar operaciones de entrada y salida sobre ficheros y flujos de datos.
- `<stdlib.h>`. Contiene los prototipos de funciones estándar, por ejemplo para convertir números a cadenas de caracteres o para realizar la asignación dinámica de memoria. (algunas de éstas también están declaradas en `alloc.h`.
- `<string.h>`. Contiene los prototipos de funciones para manejar cadenas de caracteres.
- `<time.h>`. Contiene los prototipos de funciones para manejar fechas.

En el Apéndice B se incluye un listado con las funciones de bibliotecas de uso más frecuente.

COMPLEMENTO 1.D
Compilación con gcc de un programa que consta de varios ficheros

Es una práctica frecuente en programación el descomponer la escritura de un programa en varios ficheros con objeto de tener una visión más clara del mismo. En el caso del compilador `gcc` de C bajo UNIX para compilar un programa que consta de varios ficheros se debe teclear la siguiente orden desde la línea de comandos:

```
$ gcc fichero1.c fichero2.c   ficheroN.c –o nombre_ejecutable
```

♦ **Ejemplo 1D.1:**

Supóngase que un programa se ha escrito en tres ficheros: `ejemplo.h`, `parte1.c` y `parte2.c`. El código del fichero de cabecera `ejemplo.h` es:

```
#define T 3
#define NULL 0
typedef struct {
                  long int dni;
                  float nota;
            } datos;
datos *buscar(datos a[], long int b);
```

El código del fichero `parte1.c` es:

```
#include <stdio.h>
#include "ejemplo.h"
main()
{
      datos alumnos[T]={
                        {70534213, 7.5},
                        {33356897, 8.5},
                        {85963472, 7.0}
                        };
      long int id;
      datos *pr;
      printf("\nIntroduzca DNI del alumno: ");
      scanf("%ld", &id);
      pr=buscar(alumnos, id);
      if (pr!=NULL)
            printf("Nota[%ld]= %.1f\n", pr->dni, pr->nota);
      else
            printf("\nDNI no encontrado\n");
}
```

El código del fichero `parte2.c` es:

```
#include "ejemplo.h"
datos *buscar(datos a[], long int b)
{
      int h;
      for(h=0;h<T;++h)
            if(a[h].dni==b)
                  return(&a[h]);
            else
                  return(NULL);
}
```

Si se desea compilar con `gcc` este programa, al que se le va a llamar `busca_notas`, se debe escribir la siguiente orden desde la línea de comandos:

```
$ gcc parte1.c parte2.c -o busca_notas
```

Obsérvese que no hace falta incluir en la orden el nombre fichero de cabecera ya que está incluido dentro de cada fichero `.c`. Asimismo nótese que el fichero de cabecera está escrito entre comillas (""). Originalmente esto se hacía para indicarle al compilador que se trata de un archivo de cabecera de usuario y diferenciarlo así de los archivos de cabecera del sistema que están escritos entre los signos menor y mayor (<>). La mayoría de los compiladores de C y los entornos de desarrollo actuales permiten especificar donde se encuentran los distintos archivos de cabecera. Sin embargo se sigue recomendando usar la misma nomenclatura por cuestiones de claridad en el código.

♦