Capítulo

1

Programación en (ANSI) C: Primeros pasos

El primer tema se dedica a describir la estructura de un programa en lenguaje (ANSI) C, la declaración de datos (escalares), las instrucciones más sencillas y la construcción de expresiones. Y se termina con las sentencias para realizar la entrada/salida básica que permitirán hacer los primeros programas.

Introducción

El lenguaje C fue creado por Dennis Ritchie en los laboratorios Bell de AT&T en 1972 para tener un lenguaje de alto nivel (la mayoría de los autores le colocan entre medio y bajo nivel) en el que poder programar el sistema UNIX. Curiosamente se le llamó C por que existían versiones anteriores (A y B, la B creado por K. Thompson). Es uno de los lenguajes más empleados en la actualidad[1] (ver Ilustración 1), además sus sucesores se cuentan entre los cinco primeros: C++ y C#, y ha tenido influencia sobre JS, PHP, Java, Perl o Python y posee la ventaja de ser fácilmente extensible a nuevas arquitecturas y como hemos dicho, poder tener acceso a bajo nivel (registros de memoria). Dada su naturaleza, es empleado para programar sistemas POSIX, microcontroladores, procesadores empotrados o DSP multimedia.

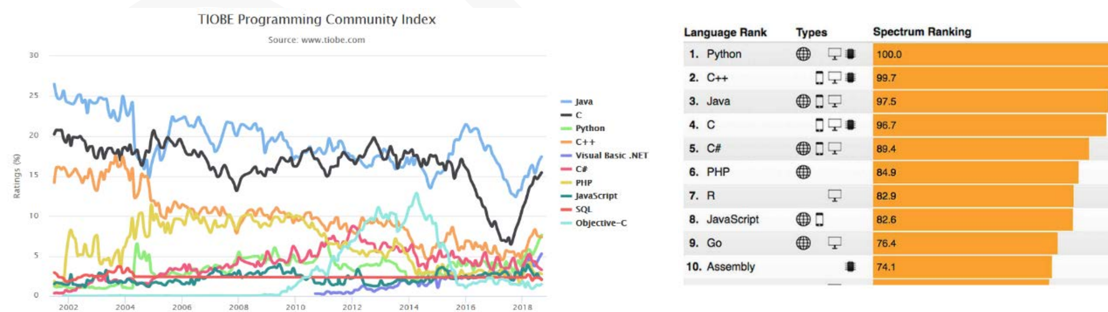

Ilustración 1. Porcentaje de uso del Lenguaje C.

La primera especificación del lenguaje fue escrita en 1978: “El lenguaje de programación C” de Kernighan y Ritchie, de hecho, al C especificado se le llamó K&R C. Una versión estandarizada de este lenguaje en el que se evitan algunos de los problemas de la primera versión (poco *tipado*) y la proliferación de librerías, fue creada en 1989 por el comité ANSI y se le llamó ANSI C o estándar C. Hubo una segunda edición del libro de K&R que cubría esta estandarización (ver **¡Error! No se encuentra el origen de la referencia.**). En ella se modificó el paso de parámetros a las funciones,

[1] Ranking lenguajes de programación 2018 según TIOBE e IEEE.

se crearon los tipos de datos `void` y enumerado, el calificador `const`, una librería estándar, los prototipos de funciones y que éstas puedan devolver estructuras.

En 1990 ANSI C fue adoptado por ISO y se convirtió en C90 y en 1999 se creó el nuevo estándar, el C99. Las aportaciones fundamentales de este estándar fueron la declaración de variables en cualquier sitio, las funciones inline, tipos de datos de 64 bits, arrays de longitud variable, comentarios de una línea (// estilo C++) y tipos de datos lógicos y complejos. El C11 es el más nuevo estándar (ISO/IEC 9899:2011) que como característica principal da soporte al *multithreading* para poder programar eficientemente sistemas *multicore*.

Ilustración 2. ANSI C.

El lenguaje C tiene la ventaja de que produce código muy compacto y eficiente, pero a costa, si no se tiene disciplina, de hacer los programas fuentes menos comprensibles de lo que serían en otros lenguajes como Pascal o Ada. Por eso, cuando se programa en este lenguaje, se debe hacer un esfuerzo adicional para que los programas resultantes sean claros y fáciles de leer. Además no soporta excepciones, objetos, polimorfismo o chequeo de rangos (como el C++).

Estructura de un programa

Un lenguaje de programación desde el punto de vista léxico (una de las fases de la compilación es el análisis léxico -ver apéndice B-) es un conjunto de *tokens* (la unidad más pequeña que tiene un único significado) dentro de los cuales nos encontramos con los identificadores (definidos por el programador con las reglas del lenguaje), los símbolos (operadores), las constantes (literales) numéricas o de carácter y las palabras clave o reservadas (usadas por el lenguaje). Con todos ellos construiremos un programa.

Un identificador será un nombre compuesto por letras, dígitos y el carácter "_", que deberá comenzar por una letra. No se admitirán letras como la ñ o las vocales acentuadas o caracteres que hayan sido creados en países de habla no inglesa. Los identificadores pueden ser de cualquier longitud pero normalmente sólo son significativos los 31 primeros caracteres (esto dependerá de la implementación, antes eran 8). Las mayúsculas y minúsculas son distintas.

Habrá identificadores reservados que no podrán utilizarse porque ya tienen un significado propio, son las palabras reservadas (*keywords*). El conjunto de éstas depende del lenguaje que utilicemos y algunas veces del compilador en particular. También habrá otros identificadores que ya tienen un significado predefinido, y que si les usamos perderemos su función, este sería el caso de incluir una librería matemática con una función para calcular la arcotangente de un número y definir un identificador como *atan*.

Para el ANSI C estándar las palabras reservadas son:

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

También hay que tener en cuenta que la librería estándar de C usa otros identificadores que no deberían ser usados o redefinidos. Estos identificadores son los nombres de las funciones de esta librería.

En la construcción de un programa también se usan delimitadores que separen identificadores, un delimitador podrá ser un espacio en blanco, un tabulador, un retorno de carro, un comentario (algo no "compilable" que se utiliza para aclarar o explicar algo) o una combinación de los anteriores.

Desde el punto de vista de la estructura, un programa en C se compone de una serie de declaraciones de datos globales y una serie de funciones. La parte de declaración está compuesta de una lista de variables de alguno de los tipos de datos del lenguaje. La parte de definición de las funciones es la que indica lo que tiene que hacer el programa con los datos. Una de las funciones es la principal, ***main***, dónde se empieza a ejecutar el programa. Además, el programa puede contener instrucciones al preprocesador (ya se verá lo que es).

Cada función está compuesta de dos partes: por un lado la cabecera, con el nombre de la función seguida entre paréntesis por los posibles argumentos que puede tener; y por otro lado está el cuerpo de la misma (bloque), que estará encerrado entre llaves ({ }) y podrá contener declaraciones de datos locales a la función y las sentencias que definen su funcionamiento.

Esta estructura se puede ver con el siguiente ejemplo:

```
#include <stdio.h> /* fichero de cabecera para el preprocesador */
#define CONVERSION 166.386   /* constante del preprocesador */

int euros = 4;    /* variable entera inicializada */

main ()   /* Pasa 4 euros a pesetas */
{
   int pesetas;    /* variable local a main */

   pesetas = CONVERSION * euros;   /* conversiones de tipo */
   printf("Hay %d pesetas en %d euros \n", pesetas, euros); /*salida*/
   pinta();    /* llamada a función externa */
}

pinta()   /* cabecera de función */
{
   printf("Solo hago esto \n");
}
```

En azul estarían las instrucciones para el preprocesador (antecedidas del símbolo "#"). En verde la declaración de datos globales. En rojo la función principal (dentro sus propias declaraciones locales en naranja). Y en morado otra función. Destacados en marrón los comentarios que están limitados por los símbolos /* y */ .

En otros lenguajes las funciones de un programa pueden estar anidadas (Pascal o Ada), en el caso del C todas las funciones están al mismo nivel (ver Ilustración 3. Anidamiento de funciones.). En el programa de ejemplo también se observa que hay líneas que empiezan por "#", estas instrucciones no serán tratadas por el compilador si no por el preprocesador (ver apéndice A), que es el que entrega el código fuente al compilador.

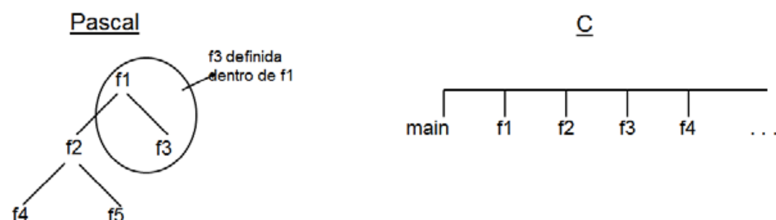

Ilustración 3. Anidamiento de funciones.

Un programa en C podría escribirse en una sola línea larguísima -si el editor nos dejara- pero sería difícil de entender y de escribir, por eso, al desarrollar un programa tenemos que adoptar cierta disciplina: como el sangrado –*indentation*– (ver Cuadro 1), es decir, dejar unos espacios (tabulación) antes de escribir una línea que pertenece a un mismo grupo de sentencias entre llaves; usar comentarios para aclarar los puntos oscuros del programa y definir las funciones (incluida la *main*); usar nombres de variables adecuados y escribir una sola sentencia por línea. Si no adoptamos una estrategia de escritura, el programa podría resultar ilegible:

```
int a = 4; main () {int b; b = 5 * a; printf("Hay %d pesetas en %d
euros\n",b, a); pinta(); } pinta();{printf("Solo hago esto\n"); }
```

Éste, es el mismo programa que el anterior, funcionaría de la misma manera, pero sería mucho más difícil de entender (pensar en un programa de varios cientos de líneas escritas así). En él, hemos quitado los comentarios y hemos sustituido los nombres de las variables por letras que no dicen nada. Además, no hemos respetado el sangrado.

Algunos estilos de sangrado:

a) La llave abierta se pone en la misma línea con la estructura de control y la llave de cierre va en una línea a la altura del inicio de la estructura:

```
if (a==1) {
  b = 1;
  c = 2;
}
```

b) Ídem, pero la llave de cierre se dispone un poco a la derecha:

```
if (a==1) {
  b = 1;
  c = 2;
  }
```

c) La llave abierta va en una línea sola, al igual que la llave cerrada. Ambas se disponen a la altura de la estructura que gobierna el bloque **–esta parece la más razonable y es la que sigo siempre–**:

```
if (a==1)
{
  b = 1;
  c = 2;
}
```

d) Ídem, pero las dos llaves se disponen más a la derecha y el contenido del bloque más a la derecha:

```
if (a==1)
   {
     b = 1;
     c = 2;
   }
```

e) Y aún otro, con las llaves a la misma altura que el contenido del bloque:

```
if (a==1)
    {
    b = 1;
    c = 2;
    }
```

No hay un estilo mejor que otro, aunque provoca muchas discusiones entre los desarrolladores de C.

Cuadro 1. Estilos de sangrado.

El uso de comentarios en cada parte que creamos conveniente explicar es muy importante para entender un programa, estos comentarios deberán estar entre los símbolos /* y */ (no se pueden anidar). Una buena técnica de programación es explicar cada bloque de código explicando que es lo que hace, su autor/es, versión, partes, entradas y salidas, etc. y usar los comentarios de línea (admitidos en C99) para explicar líneas de código o para usar detrás de la última "}" del bloque para indicar el final explícitamente. En C todas las sentencias acaban con ";" que hace de terminador. De hecho, varios ";" seguidos constituirían varias sentencias vacías.

Declaración de datos y sus tipos

En C existen dos clases de datos: constantes y variables. Una constante es un símbolo, que se asocia a un dato de valor constante y que no se podrá cambiar. Una variable es una posición de la memoria, referenciada por un nombre –identificador– donde se almacena el valor de un dato que se puede cambiar a lo largo del programa. Así, en el programa anterior, el factor de conversión era una constante y el número de euros y pesetas eran variables.

Las variables -o las constantes- pueden almacenar o referenciar distintos **tipos de datos**, estos pueden ser numéricos, alfabéticos o agrupaciones de los anteriores (tipos estructurados). A diferencia de otros lenguajes, el K&R C sólo tenía esos dos tipos básicos de datos, por ejemplo, no existían datos de tipo lógico (se asocian al dato de tipo entero, 0 para FALSE y distinto de cero para TRUE), tampoco existían el tipo subrango o el enumerado. Es una de las cosas que solucionó el ANSI C. El uso de punteros en C (direcciones de memoria) es habitual, ya que como hemos dicho, es un lenguaje de medio nivel, de hecho, el nombre de una variable de tipo *array* es un puntero (o asociando el operador "&" a una variable obtenemos su dirección en memoria).

Otra de las características del C es que nos permite renombrar nuestros propios tipos de datos, para ello se usa la palabra reservada **typedef**.

En la siguiente figura pueden verse los tipos básicos de datos:

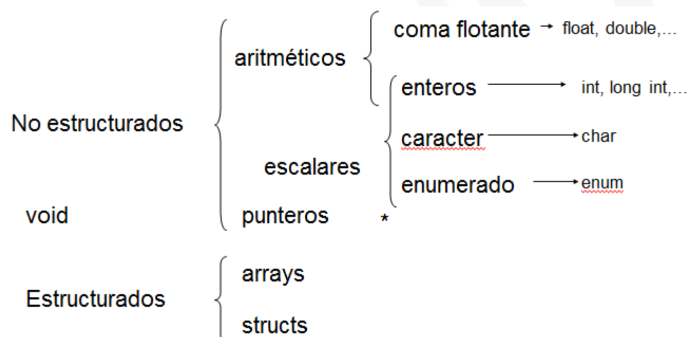

Ilustración 4. Tipos de datos en C.

Un tipo entero es un número positivo o negativo que no tiene parte fraccionaria. Dentro del tipo entero podemos destacar varios subtipos: el ***short*** normalmente de un byte (-128..127), el ***int*** (-32768..32767) de dos bytes, y el ***long*** (-2147483648..2147483647) de cuatro bytes. La longitud en bytes de cada tipo de dato depende de la implementación del compilador sobre la máquina que usemos. Siempre podemos averiguar el número de bytes de un tipo de dato con el operador ***sizeof()*** (devuelve un dato de tipo `unsigned long integer`). Es habitual que el entero tenga una longitud de palabra, si la palabra en la máquina es de 32 bits, el *int* ocupará 4 bytes. En una máquina Linux de 64 bits los valores son 2, 4 y 8.

También hay enteros sin signo poniendo a cada subtipo el calificador ***unsigned****: unsigned short* (0..255) de un byte, *unsigned int* (0..65535) de dos bytes y *unsigned long* (4 mil millones) de cuatro bytes. Como hemos dicho el número de bytes depende de la máquina y el compilador. Los enteros son por defecto con signo (***signed)*** pero no es necesario ponerlo. Cuando se programa, se tiene que tener en cuenta al implementar el algoritmo, el rango de los números que estamos usando para no producir un "*overflow*", es decir, no sobrepasar la aritmética del tipo de número escogido (lo mismo para el "*underflow*").

Un tipo real (flotante) es un número decimal. Desde el punto de vista matemático, entre un rango de enteros hay un número finito de elementos (por eso los datos de tipo entero son escalares), pero entre dos reales el número es infinito, para representar esto con el ordenador se utilizan aproximaciones usando la notación científica (exponencial). Así, una parte de la memoria dedicada a almacenar el real se encarga del exponente y otra parte a la mantisa. En C hay tres subtipos de

reales: el ***float***, el ***double*** y el ***long double***. Su tamaño (que implicará rango de números y precisión) dependerá de la implementación. También se deberá tener en cuenta que se puedan producir desbordamientos (*overflow* o *underflow*). En una máquina Linux de 64 bits los valores son 4, 8 y 16.

Un tipo carácter describe un carácter alfanumérico (todos los caracteres de la tabla ASCII). Como es de suponer este tipo de datos también es escalar. Podrían existir dudas cuando nos referimos a un carácter que indica un número, para diferenciarlo del dato numérico, todos los caracteres se escriben entre apóstrofes, así, no es lo mismo el número 8 que carácter '8'. Los datos de tipo carácter ocupan 1 byte.

Las constantes –literales– enteras serán los números que se escriben sin punto decimal, si queremos que una constante sea de tipo *long* se pondrá una **L** detrás del número y **u** para *unsigned*. También se pueden escribir en distintas bases numéricas: para octal tendrán que ser precedidas por un **0** y para hexadecimal por un **0X**.

Larga: 31L Unsigned: 31u Octal: 037 Hexadecimal: 0X1F

Los literales reales se pueden escribir de dos maneras: en notación decimal o en notación científica (exponencial potencia de 10). En la primera se utiliza el punto para separar la parte entera de la fraccionaria y en la segunda se da una cifra y una potencia de 10 separadas por una e (E) sin espacios, también se usan letras para decir el tipo de flotante, **f** (F) para *float*, y **L** (l) para *long double*:

Decimal 3.1416 100. float .2f científica 4e16 .8E-5 double -4.3e-5L

Las constantes de tipo ***char*** se especifican con los apóstrofes y serán sólo un carácter (byte). También se pueden escribir refiriéndose a la tabla ASCII, colocando el código en octal o en hexadecimal, después de la barra atrás (secuencias de escape).

'3' 't' secuencias '\007' '\x7'

Hay definidas una serie de secuencias de escape para ciertos caracteres de control:

\n nueva línea	\f salto de página	\0 nulo
\t tabulador	\\ barra atrás	\v tab vertical
\b retroceso	\' apóstrofe	\a alerta
\r retorno de carro	\" comillas	

Las \\ \' \" están indicadas cuando hay que incluirlas en una tira de caracteres o en la definición de *char* y no confundirlo con la delimitación de las mismas.

Una característica especial del C es que las variables se pueden inicializar en su definición con las constantes apropiadas. Esto nos servirá para ver ejemplos de declaraciones de datos:

```
int    coeficiente          = 1024;
float  pi                   = 3.1416;
char   letra1, letra2       = 'A';
char  letra3 = ‘a’, letra4 = ‘b’;
char   car                  = '\n';
```

Como se observa, también se pueden hacer varias declaraciones de variables del mismo tipo, separándolas por comas. Si se hace una inicialización en este caso, sólo tendrá efecto en la última variable. Así, `letra2` valdrá 'A', pero el valor de `letra1` estará indefinido. Detrás hay un ejemplo de cómo se evitaría esta indefinición.

Existe un tipo estructurado, que es la tira (*string* o *array* de caracteres terminados en el carácter '\0') de la cual también existen constantes (literales). Para su definición y delimitación se usan las comillas (se verán posteriormente).

```
"Esto es una constante/literal de tipo string y se delimita por \".\n"
```

donde se han utilizado los caracteres especiales '\"' y '\n' para poner unas comillas y una nueva línea en el mensaje.

Por último, en lenguaje C existen dos maneras de declarar constantes (de cualquier tipo). La primera -y más clásica- es la de utilizar constantes simbólicas con el preprocesador (del que se ha hablado en el apéndice A). Para ello se utiliza la instrucción `define`, que como todas las del preprocesador (no va a ser leída por el compilador) debe ir precedida del símbolo “#”. La idea es que el preprocesador actuará sobre el código fuente buscando el nombre de la constante y lo sustituirá por su valor. Por costumbre, para este tipo de constantes, se utilizan mayúsculas, aunque no es obligatorio. La segunda es utilizando la palabra reservada ***`const`*** antes de la declación. Obviamente si declaramos algo constante no se puede modificar su valor en el programa. A continuación, vemos unos ejemplos:

```
const int    coeficiente = 1024;
const float  pi          = 3.1416;
#define      LETRA        'A'
#define      CANTIDAD     345
#define      FRASE        "Hola a todos"
```

Expresiones y sentencias

Los cuerpos de las funciones estarán compuestos de declaraciones similares a las anteriores y de sentencias que trabajan con los datos definidos. Habrá varios tipos de sentencias, una de las más utilizadas es la asignación, que sirve para cambiar a una variable de valor.

Esta sentencia estará compuesta de dos partes separadas por el operador de asignación **"="**. La parte de la izquierda será la variable a la que queremos cambiar de valor definida por un identificador (lexema en términos de compiladores), la parte de la derecha será una expresión compuesta por operandos y operadores que formarán una expresión cuyo resultado debe ser del mismo tipo que la variable, sino se producirá una conversión, no un error. Es conveniente indicar explícitamente (como veremos posteriormente con *typecast*) si se producen conversiones por claridad de código.

La asignación no es como una ecuación matemática, simplemente, la parte de la derecha se evalúa y se asigna este valor a la parte de la izquierda, matemáticamente, "x = x + 1" no tiene sentido, pero esa sentencia en los lenguajes informáticos, incluido el C, significa que el valor actual de x se ha incrementado en uno respecto al antiguo.

Además, en C existe la asignación múltiple, de tal manera que se puede asignar valor a varias variables:

```
a = b = c = 780;
```

El C no es un lenguaje fuertemente tipado, de tal manera que se pueden hacer asignaciones de datos de distinto tipo, produciéndose la consiguiente conversión (mejor indicarlo):

```
caracter = caracter + 1;
```

Para construir la expresión se usan los operadores, los aritméticos son:

Operador	Símbolo
Adición	+
Substracción	-
Negación	-
Multiplicación	*
División	/
Resto	%

Los operadores "+", "-", "*", y "/" son la suma, la resta, la multiplicación y la división y pueden ser de tipo real o entero (se dice que los operadores están sobrecargados porque tienen el mismo

símbolo –lexema– para diferentes operadores), dependiendo del tipo de operandos. Hay que tener en cuenta que la división entera trunca el resultado, es decir, 5/3 es 1). El resto es el "%" (5 % 3 = 2) que es entero (obviamente no se usa con flotantes).

Como el C intenta producir código compacto hay cinco combinaciones de operadores aritméticos de asignación:

Operador Asignación	Símbolo
Adictivo	+=
Substractivo	-=
Multiplicativo	*=
Divisivo	/=
De resto	%=

Así, es equivalente:

```
x = x + 3;   ⇔   x += 3;
```

Las expresiones pueden estar formadas por variables y constantes (operandos) y operadores. Cuando la expresión es compleja e incluye a varios operadores y operandos, hay que introducir unas normas de evaluación de la expresión, que son las normas de **precedencia** y **asociatividad.** Por ejemplo si nosotros queremos sumar un operando con otro y el resultado multiplicarlo por un tercero, no podremos hacer:

```
valor = operando1 + operando2 * operando3;
```

ya que la asociatividad del "*" es binaria y tiene mayor precedencia que el "+", por lo tanto, afectará al operando segundo y tercero, y al resultado del producto le sumaremos el primero (el **orden** de evaluación no está definido, es decir que término de la suma se evalúa primero en el C clásico K&R, en el ANSI C sería de izquierda a derecha para operadores binarios).

Para evitar esto tendríamos que haber utilizado:

```
valor = (operando1 + operando2) * operando3;
```

ya que el **operador paréntesis** es el de máxima precedencia.

En la siguiente tabla aparecen las características[1] (precedencia y asociatividad) de los operadores aritméticos, teniendo en cuenta que el operador de asignación es el que tiene la menor precedencia. La otra característica de los operadores es su asociatividad, es decir, a que número de operandos afecta: primario (que encierra), unario sólo a uno, binario a dos, ternario a tres.

Operador	Símbolo	Asociatividad	Precedencia
Adición	+	Binario	4º
Substracción	-	Binario	4º
Incremento	++	Unario	2º (pre)
Decremento	--	Unario	2º (pre)
Signo	-	Unario	2º
Multiplicación	*	Binario	3º
División	/	Binario	3º
Resto	%	Binario	3º

[1] En el capítulo séptimo hay una tabla completa de precedencia de todos los operadores.

Paréntesis	( ) [ ]	Primario	1º
Asignación	= += -= *= /= %=	Binario	5º

En la tabla aparecen dos nuevos operadores que son el **incremento** "++" y el **decremento** "--", que son unarios y tienen mayor precedencia que los binarios. Su misión es incrementar o decrementar en una unidad el operando. Estos operadores se pueden usar de dos maneras, por un lado, anteponiéndolos al operando (en prefijo) o posponiéndolos (en sufijo), lo cual hará variar su precedencia. Cuando se usan en una expresión con otros términos hay que tener en cuenta su precedencia y su **orden** de evaluación:

Prefijo	Sufijo

izda = 3 * ++dcha;
⇓
dcha = dcha + 1;
izda = 3 * dcha;
	izda = 3 * dcha++;
	⇓
	izda = 3 * dcha;
	dcha = dcha + 1;

La precedencia nos dice donde se colocarían los paréntesis por defecto, no el orden de evaluación, en el último ejemplo a "dcha", no a "3 * dcha", el uso (prefijo o sufijo) nos dice el orden de evaluación. Además en ambos casos la variable “dcha” estará incrementada.

Hay que tener mucho cuidado al utilizar estos operadores ya que cuando se utiliza el mismo operando en una expresión, no se garantiza que se hace primero (en K&R C):

```
respuesta = num / 2 + 3 * (1 + num++);
```

Es decir, al evaluar la expresión anterior no está garantizado que se realice primero "num / 2" que el otro término "3 * (1 + num++)", con lo cual podemos tener "num" incrementado o no y el resultado puede ser diferente (para num = 5 podría ser 20 ó 21, lo normal es que sea 20 si se evalúa de izquierda a derecha como ocurre en ANSI C).

Por lo tanto, para usarlos con seguridad hay que tener las siguientes precauciones:

1. No emplearlo en expresiones con el mismo operando.
2. No se emplean en argumentos de funciones con el mismo operando.

Hay otro operador que da el tamaño en bytes de una variable o un tipo y tiene precedencia 2º, siendo también unario, su nombre es *sizeof*. El valor que devuelve es de tipo *unsigned long int*. Se puede usar con un tipo o una variable (con variable con o sin paréntesis):

```
sizeof(int)          sizeof dias        sizeof (dias)
```

Existe otro operador que es el de **moldeado** (*type cast*) para hacer conversiones explícitas de tipo (poniendo el tipo resultante entre paréntesis). Como hemos dicho, esto siempre es conveniente hacerlo explícitamente como buena metodología de programación. Es de la misma precedencia que los unarios. En el ejemplo siguiente el resultado sería 1:

```
(int) 1.6
```

Cuando se hacen conversiones implícitas o se combinan operandos de diferentes tipos hay que tener cuidado ya que se pueden realizar promociones o pérdidas de rango según el siguiente esquema de mayor a menor: **long double -> double -> float -> long -> int -> short -> char .**

En promociones no hay problemas adicionales, en pérdidas se pueden dar resultados insospechados, hay que recordar que el compilador ¡**no dará error**! al hacer cosas como ésta, salvo que usemos la opción –Wall (warning all):

```
ch = flotante;
ch = 123.3e10;
```

Cuando se combinan en una expresión operandos de distintos tipos, antes de la evaluación se produce una promoción de rango del menor para que el resultado sea del tipo del mayor y la operación se produzca entre operandos del mismo tipo.

Hay que tener cuidado si las operaciones se realizan entre datos del mismo tipo para luego ser promocionados:

```
int total = 500;
int numero = 12;
float media = 0.0;
media = total / numero;
```

Sabiendo esto podrías decir ¿cuánto valdría "media"?

Entrada y salida básica

Los lenguajes de programación siempre dan un método por el cual el ordenador recibe y/o devuelve información a través de una unidad de entrada/salida de una forma legible por una persona. En C se utiliza una librería que proporciona funciones para realizar esta Entrada/Salida (E/S o I/O). Se considera que es de uso habitual, por lo que no hay que incluirla al compilar (está incluida en `libc` que se incluye por defecto), pero lo que tenemos que hacer es declarar la forma de estas funciones (se verá en el tema cuarto que son los prototipos).

Para ello se debe incluir un fichero de código fuente (ya realizado y llamado `stdio.h`), donde está la forma de esas funciones (prototipos), así como las constantes simbólicas que se pueden usar en nuestros programas en relación a la E/S. Para incluir este fichero (los ficheros de cabecera son .h) se utiliza otra instrucción del preprocesador que es *include* (los ángulos "<>" le dicen al preprocesador que busque el fichero en el directorio por defecto). Así, casi todos nuestros programas empezarán con una línea como esta:

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

Una de las funciones que realiza la salida es ***printf*** (imprime formateado). Tiene la siguiente estructura:

printf (control, lista de variables);

Está compuesta de dos partes, por un lado está una tira (mensaje) de control donde están encerrados entre comillas los caracteres a imprimir y los **conversores** y por otro lado está la lista de variables que se deben pintar, que es opcional pero recomendable.

Los **conversores** indican el lugar y el formato, en la tira de control, de las variables a imprimir, por lo que su número deberá ser igual al número de variables en la lista. La lista de las cosas que queremos escribir, tendrá que ir separada por comas y podrá ser cualquier expresión.

La salida podrá ser formateada de diferente manera de acuerdo al conversor:

Conversor	Variable	Salida	Conversor	Variable	Salida
%d %i	int short	int	%f	Float double	double (punto)
%u	int	Unsigned int	%e %E	Float double	double (exponencial)
%o, %x %X	int	Octal, hexadecimal	%g %G	Float double	double (conveniencia)
%c	char	char	%s %p	String puntero	string puntero

Ejemplos son (sentencia ⇢ resultado):

printf("el valor de entero es %d .\n", entero); ⇢ el valor de entero es 23 .

printf("%c%d\n", '$', entero); ⇢ $23

Los tipos de variables no tienen porque coincidir con los conversores (de ahí su nombre), es decir, que por ejemplo se puede pintar un carácter en forma de entero (número ASCII).

Si dentro de la tira de control se quiere imprimir el símbolo "%", para no confundirlo con un conversor se deberá duplicar:

```
printf("el 10%% de %d es %d\n",100,10);
```

Existen unos modificadores de salida que se sitúan entre el conversor y el "%". Hay varios tipos:

Significado	Modificador	Ejemplo
Justificación izquierda	-	%-d
Anchura de campo	número	%4f
Precisión	.número	%.2f
tipo short	h	%hd
tipo long / double	ℓ	%ld / %lf
tipo long double	L	%Lf

Ejemplo:

printf("el valor de pi es %.4f\n",pi); ⇢ el valor de pi es 3.1416

Hay que tener en cuenta que la justificación por defecto es la derecha y que si ponemos una anchura insuficiente de campo se ignorará.

Al igual que existe una instrucción de salida, también existe una de entrada, ***scanf,*** que normalmente se realizará desde teclado. La estructura es la misma que la anterior y el formato lo proporciona el usuario al escribir la entrada, que debe coincidir con lo especificado. En la lectura no tiene sentido poner mensajes en la tira de control.

Las variables de la lista tienen que ser de tipo **puntero** (se verán posteriormente), por ahora esto significa que delante de las variables que no sean tiras o *arrays* (también se verán posteriormente) habrá que poner el símbolo "**&**", que como veremos después es el operador de dirección de tipo unario.

```
scanf ("%d %f", &edad, &sueldo);
scanf ("%2d%f", &edad, &sueldo);
scanf ("%s", tira);
scanf ("%10s", tira);
```

Cuando hay varios elementos en la lista de variables, deben ir separados por comas. Cuando tecleemos los valores para las variables tenemos que separarlos con una línea, un espacio o un tabulador, salvo cuando se lea un carácter, en cuyo caso se leerá lo que haya de forma literal, lo que **suele plantear problemas**, ya que si hay un espacio entre la lectura de dos números y leemos un carácter, el carácter leído será el espacio que hemos dejado como separador entre los números. Así para la entrada anterior se puede responder:

```
23 12.3
21312.21
una_tira_de_caracteres
```

o también:

```
23 12.3 21312.21 una_tira_de_caracteres
```

o cualquiera de las posibles combinaciones. Es mejor hacerlas coincidir con lo indicado en la tira, así no es lo mismo dejar un espacio entre `%d %f` que no hacerlo. Y si ponemos un valor numérico en el conversor se parará la lectura en el número de caracteres indicado, en el ejemplo 10.

A pesar de que exista el conversor "%c", para la entrada y salida de caracteres, estas funciones combinadas suelen dar problemas ya que filtran los caracteres espacios, retornos y tabuladores. Por ello es mejor limitar la entrada con **scanf** a números.

En C existen otras funciones que están en la librería *stdio.h* para este fin. Estas funciones son ***getchar***() y ***putchar***(). *getchar*() no tiene argumentos y devuelve un carácter, *putchar()* tiene como argumento un carácter.

Así podríamos hacer un programa:

```
#include <stdio.h>
main()
{
  char ch;
  ch = getchar();
  putchar(ch);
}
```

que escriba el primer carácter que se introduzca (doble eco). Se pueden abreviar las dos sentencias haciéndolas "más C" (es mala programación ya que el código se hace menos legible):

```
putchar(getchar());
```

El comportamiento de las funciones será distinto si el sistema posee o no buffer de entrada (donde cabe una línea). Si tiene buffer los caracteres no se leerán hasta que no se pulse la tecla ENTER, leyéndose todos de golpe. Si no se tiene buffer (con eco) los caracteres se leerán inmediatamente sin esperar al ENTER.

Un ejemplo con los dos tipos de entrada/salida sería:

Con sistema de buffer:	Esto es una prueba Esto es una prueba
Sin sistema de buffer:	EEssttoo eess uunnaa pprruueebbaa

Existen otras dos formas de entrada que no están en el estándar ANSI de C, pero que son utilizadas cuando no se quiere esperar al retorno de carro (cuando no se quiere eco), estas son *getch*() y *getche*(), cuya salida sería:

Sin sistema de buffer:	Esto es una prueba

Cuadro 2. Entrada / Salida no estándar.

La E/S en Linux / UNIX se puede producir en tres canales lógicos: el de entrada, el de salida y el de errores. Estos canales se definen en *stdio.h* con tres variables ya definidas: `stdin`, `stdout` y `stderr`. Cuando usamos *printf* usamos la salida `stdout`, con *scanf* la entrada `stdin` y para la salida de errores el fprintf (ya puedes imaginar que es lo que hace una llamada a *printf*):

```
fprintf (stderr, "el valor de entero es %d .\n", entero);
```

La diferencia entre el canal de salida normal y el de errores es que el de salida es *bufereado*. Esto significa que una línea sólo sale en pantalla si es terminada con \n, si no se quedará en el buffer de salida, al contrario que en el canal de errores donde cada carácter sale inmediatamente. Esto es muy importante a la hora de depurar el programa, ya que para saber el valor de las variables en un momento dado siempre tendremos que terminar la línea con la secuencia de escape "/n".

En la librería *stdio*, también se define una forma de parada de lectura que es la macro EOF (End Of File), normalmente definido como -1, con lo cual habría que hacer la entrada entera y conocer las sentencias de control y de iteración del siguiente tema.

Como avance podemos poner:

```
#include <stdio.h>
main()
{
        int ch;
        while ((ch = getchar()) != EOF)
              putchar(ch);
}
```

Ejemplos

```
1  /* Ejemplo 1.c
2  Compilalo con el comando $ gcc 1.c.
3  Comprueba que tienes un ejecutable y su nombre.
4  Ahora prueba con $ gcc -ansi 1.c
5  Ahora prueba con $ gcc -Wall 1.c y observa los warnings (no son errores)  */
6
7
8  #include <stdio.h>                          /* fichero de cabecera para el preprocesador */
9  #define CONVERSION 166.386                 /* constante del preprocesador */
10
11 int euros = 4;    /* variable entera inicializada */
12
13 main () /* Pasa 4 euros a pesetas */
14 {
15     int pesetas;    /* variable local a main */
16
17     pesetas = CONVERSION * euros;  /* conversiones de tipo */
18     printf("Hay %d pesetas en %d euros \n", pesetas, euros); /*salida*/
19     pinta();    /* llamada a funcion externa */
20         return(0);
21 }
22
23 pinta()   /* cabecera de funcion */
24 {
25     printf("Solo hago esto \n");
26 }
27
```

```
/* Ejemplo 2.c
Errores corregidos de 1.c  */

#include <stdio.h>                              /* fichero de cabecera para el preprocesador */
#define CONVERSION 166.386                     /* constante del preprocesador */

int euros = 4;    /* variable entera inicializada */
void pinta();

int main () /* Pasa 4 euros a pesetas */
{
    int pesetas;    /* variable local a main */

    pesetas = CONVERSION * euros;  /* conversiones de tipo */
    printf("Hay %d pesetas en %d euros \n", pesetas, euros); /*salida*/
    pinta();    /* llamada a funciÃ³n externa */
        return(0);
}

void pinta()   /* cabecera de funciÃ³n */
{
    printf("Solo hago esto \n");
}

```

```
/* Ejemplo 3.c
comprende los tipos de inicializaciones que hay
¿Cuanto vale b ?    */


#include <stdio.h>                              /* fichero de cabecera para el preprocesador */
#define CONVERSION 166.386                     /* constante del preprocesador */

int main ()
{
        char caracter;
        int pesetas;
        long datos;
        float variable=CONVERSION;
        double doble;

        char b, a = 'a';     /* cuanto vale b ? */
        char cerod = '0';
        char ceroo = '\060';
        char cerox = '\x30';
        char retorno = '\n';

        #define     FRASE       â€œHola a todosâ€

        int i=0;
        unsigned short dato=23u;
        #define MAX 8L       // otra constante
        long maximo=MAX;

        const float eps=1.0e-5f; // constante
        double real=doble;

        return(0);
}

```

```c
/* Ejemplo 4.c
Comprende las operaciones que hay.
Escribe cuanto valen las variables izda y dcha en cada paso.
Ejecuta el programa y mira si has acertado.    */


#include <stdio.h>                          /* fichero de cabecera para el preprocesador */
#define CONVERSION 166.386                 /* constante del preprocesador */

int main ()
{

    int izda;
    int dcha= 2;

    printf("%d %d\n", izda,dcha);
    izda = 3 * ++dcha;
    printf("%d %d\n", izda,dcha);
    izda = 3 * dcha++;
    printf("%d %d\n", izda,dcha);

    dcha = dcha + 1;
    izda = 3 * dcha;
    printf("%d %d\n", izda,dcha);
    izda = 3 * dcha;
    dcha = dcha + 1;
    printf("%d %d\n", izda,dcha);

    return(0);
}

```

```c
/* Ejemplo 5.c
Comprende las operaciones que hay.
Escribe cuanto valen las variables f , izda y c en cada paso.
Ejecuta el programa y mira si has acertado.      */


#include <stdio.h>                          /* fichero de cabecera para el preprocesador */
#define CONVERSION 166.386                 /* constante del preprocesador */

int main ()
{

        char a,b,c;
        a=b=c='a';
        b=a+1;

        float f=3.0+1/2;
        printf("%f \n", f);

        f=3.0+1/2.0;
        printf("%f \n", f);

        f=3.0+(float)1/2;
        printf("%f \n", f);

        int izda = f;
        printf("%d \n", izda);

        c=f*100.0;
        printf("%c \n", c);
        printf("%d \n", c);

        return(0);
}

```

```
1   /* Ejemplo 6.c
2   Comprende las operaciones que hay.
3   Piensa como aparecerán en pantalla lo indicado en los tres primeros printf.
4   Introduce valores desde teclado para los tres siguientes scanf y piensa que valores están dando a las variables.
5   Ensaya dando los valores en una línea o en varias y dejando espacios o no.
6   ¿Qué ocurre con el último scanf?
7   ¿Qué valor toma el ultimo carácter con getchar?     */
8
9
10  #include <stdio.h>                              /* fichero de cabecera para el preprocesador */
11  #define CONVERSION 166.386                     /* constante del preprocesador */
12
13  int main ()
14  {
15
16          char c='a';
17          float f=3.5;
18          printf("Los valores son %c %f \n", c,f);
19
20          printf("%e \n", f);
21
22          int d = 23;
23          printf("Valores %d %u %o %X \n", d,d,d,d);
24
25          printf("%20e \n", f);
26          printf("%.2f \n", f);
27
28          scanf("%d %f", &d, &f);
29          printf("Los valores son %d %f \n", d,f);
30
31          scanf("%d %c", &d, &c);
32          printf("Los valores son %d %c \n", d,c);
33
34          scanf("%d%c", &d, &c);
35          printf("Los valores son %d %c \n", d,c);
36
37          c=getchar();
38          putchar(c);
39
40          return(0);
41  }
42
43
```

© RIA