Tema 2
Programación básica en C

1 SINTAXIS DEL LENGUAJE C

Existen seis clases de *componentes o elementos sintácticos* el lenguaje C: ***constantes***, ***identificadores***, ***palabras clave***, ***operadores, comentarios*** *(en realidad no son elementos ya que el compilador o el intérprete los ignoran)* y ***separadores***. Veamos más en detalle cada uno de estos elementos.

1.1 Constantes

Así como las variables pueden cambiar de valor, las constantes permanecen inalterables durante la ejecución del programa.

En C existen distintos tipos de constantes:

1. ***Constantes numéricas***. Son valores numéricos, enteros o de coma flotante.
Ejemplos: `278 57629.50 3.1416`
2. ***Constantes carácter***. Cualquier carácter individual encerrado entre comillas simples es considerado por C como un carácter.
Ejemplos: `‘a’ ‘Y’ ‘)’ ‘+’ ‘0’`
3. ***Cadenas de caracteres***. Una secuencia de caracteres alfanuméricos encerrados entre comillas dobles es una constante. También se le suele llamar literal alfanumérico. Una cadena de caracteres no es una constante carácter, aunque conste de un único carácter.
Ejemplos: `"Blas de Lezo", "Esto es una cadena de caracteres", “23”, “1”, “0”`
4. ***Constantes simbólicas***. Las constantes simbólicas tienen un nombre (identificador). Además, NO PUEDEN cambiar de valor a lo largo de la ejecución del programa. En C se pueden definir utilizando la cláusula ***#define*** (que en realidad es parte del preprocesador de C y no del lenguaje C) o por medio de la palabra clave ***const***.

Ejemplos:

#define PI = 3.141592

const float e = 2.718281;

Las constantes simbólicas se utilizan para mejorar la comprensión del programa. Por ejemplo, es mejor utilizar la constante pi que utilizar su valor en diferentes puntos del programa (pensar que el valor se podría utilizar con diferente número de cifras significativas)

Nota: Obsérvese que la línea correspondiente al preprocesador no finaliza con punto y coma; sin embargo, la instrucción de C que define una constante debe finalizar en punto y coma tal como lo hacen todas las instrucciones del lenguaje.

1.2 Identificadores

Un ***identificador*** es un nombre con el que se hace referencia a una función o a una variable. Cada lenguaje tiene sus propias reglas respecto a las posibilidades de elección de nombres para las funciones y variables. En C estas reglas son las siguientes:

1. Un ***identificador*** se forma con una secuencia de ***letras*** (minúsculas de la ***a*** a la ***z***; mayúsculas de la ***A*** a la ***Z***; y ***dígitos*** del ***0*** al ***9***). Tener en cuenta que los caracteres acentuados, actualmente, no pueden utilizarse para formar identificadores y lo mismo le ocurre a las letras ñ y Ñ.
2. El carácter ***subrayado*** o ***underscore*** (_) se considera como una letra más.
3. Un identificador no puede contener espacios en blanco, ni otros caracteres distintos de los citados, como por ejemplo (*,;.:-+, etc.).
4. El primer carácter de un identificador debe ser siempre una letra o un (_)
5. Se hace distinción entre letras mayúsculas y minúsculas. Así, **Valor** es considerado como un identificador distinto de **valor** y de **VALOR**.
6. Se pueden definir identificadores de hasta 31 caracteres de longitud.

Ejemplos de identificadores válidos son los siguientes:

tiempo, distancia1, caso_A, PI, velocidad_de_la_luz

Por el contrario, los siguientes nombres no son identificadores válidos

1_valor, tiempo-total, dolares$, %final

Una variable (definida mediante un identificador) es un espacio de memoria que se reserva en un programa para guardar los datos con los que va a operar. El contenido de una variable puede modificarse a lo largo del programa.

Mediante un identificador se pueden definir variables y funciones.

En general es muy aconsejable ***elegir los nombres*** de las funciones y de las variables de forma que permitan conocer a simple vista que representan utilizando para ello tantos caracteres como sean necesarios (sin exagerar). Esto simplifica enormemente la tarea de programación y, sobre todo, la corrección y el mantenimiento de los programas. Es cierto que los nombres largos son más laboriosos de teclear, pero en general resulta rentable tomarse esa pequeña molestia (sobre todo cuando el programa que se está desarrollando es complejo).

1.3 Palabras clave del lenguaje C

En C, como en cualquier otro lenguaje de programación, existen una serie de palabras clave (*keywords*) que **no pueden utilizarse como identificadores** (nombres de variables y/o de funciones). Estas palabras sirven para indicar al ordenador que realice una tarea muy determinada y tienen un especial significado para el compilador.

El C es un lenguaje muy conciso, con muchas menos palabras clave que otros lenguajes. A continuación se muestran en la primera tabla las palabras clave que utilizaremos en el curso y en la segunda tabla se muestra el resto de palabras del lenguaje.

const	int	float	double	long	void
char	if	else	do	while	for
return	break	continue	unsigned		

auto	case	union	signed	static	enum
register	default	switch	typedef	short	goto
struct	extern	sizeof	volatile		

1.4 Operadores

Los ***operadores*** son signos especiales (a veces, conjuntos de dos caracteres) que indican determinadas operaciones a realizar con las variables y/o constantes sobre las que actúan en el programa.

El lenguaje C es particularmente rico en distintos tipos de operadores, los que nosotros utilizaremos son los siguientes:

Aritméticos	+	-	*	/	%
Asignación	=				
Auto incremento	++				
Auto decremento	--				
Igualdad	==				
Desigualdad	!=				
Relacionales	<	>	<=	>=	
Lógicos	&&	||	!		
Dirección	&				
Indirección	*				

1.5 Separadores

Los *separadores* están constituidos por uno o varios espacios en blanco, tabuladores, y caracteres de nueva línea. Su papel es ayudar al compilador a descomponer el programa fuente en cada uno de sus ***elementos básicos***. Es conveniente introducir espacios en blanco incluso cuando no son estrictamente necesarios, con objeto de mejorar la legibilidad de los programas.

También son separadores los paréntesis, los corchetes y las llaves.

El resto de separadores son la coma, el punto y coma y los dos puntos. Explicaremos cómo se utiliza el punto y coma. La coma y los dos puntos sólo los utilizaremos en ocasiones y entonces explicaremos su significado.

1.6 Comentarios

El lenguaje C permite que el programador introduzca ***comentarios*** en los ficheros fuente que contienen el código de su programa. La misión de los comentarios es servir de explicación o aclaración sobre cómo está hecho el programa, de forma que pueda ser entendido por cualquier persona distinta de la que lo hizo o por el propio programador algún tiempo después. El compilador ignora por completo los comentarios.

Los caracteres (**/***) se emplean para iniciar un comentario introducido en el código del programa; el comentario termina con los caracteres (***/**). No se puede introducir un comentario dentro de otro. Todo texto introducido entre los símbolos de comienzo (**/***) y final (***/**) de comentario son ignorados por el compilador. Por ejemplo:

```
variable_1 = variable_2;

/* En la línea anterior se asigna a variable_1
   el valor contenido en variable_2 */
```

Un error frecuente consiste en olvidarse de cerrar un comentario.

A partir del año 2000 se permite utilizar como comentario la secuencia de dos símbolos de dividir seguidos; la cual obliga a que el compilador ignore el resto de caracteres hasta el final de la línea. Por ejemplo:

```
Superficie = base * altura / 2;        // Calcula el área de un triángulo (comentario)
```

Dentro de una cadena de caracteres los comentarios no lo son ya que forman parte de la propia cadena. Por ejemplo:

```
char linea[]= "esta linea /* no contiene comentarios */ aunque lo parezca";
```

2 TIPOS DE DATOS FUNDAMENTALES EN C

El C, como cualquier otro lenguaje de programación, tiene posibilidad de trabajar con datos de distinta naturaleza: texto formado por caracteres alfanuméricos, números enteros, números reales con parte entera y parte fraccionaria, etc. Además, algunos de estos tipos de datos admiten distintos números de cifras (rango y/o precisión), posibilidad de ser sólo positivos o de ser positivos y negativos, etc. En este tema se verán los *tipos fundamentales* de datos admitidos por el C, así como otros tipos de datos, *derivados* de los fundamentales. La siguiente tabla recoge los tipos de datos fundamentales en C.

TIPO	**P.CLAVE**	**EJEMPLOS**
Caracter	**char**	'a' '3' '#'
Entero	**int**	3 -8 572 -247
Real (precisión 6 cifras)	**float**	12.57 -27.9 63.0 3.141591
Real (precision 15 cifras)	**double**	3.141592653589793

La palabra ***int*** indica que se trata de un número entero, mientras que ***float*** se refiere a un número real. Para números de mayor precisión se utiliza la palabra reservada **double** Los números pueden ser positivos o negativos, salvo para el caso de los números enteros que pueden ser exclusivamente positivos utilizando **unsigned int**

La palabra ***char*** hace referencia a un carácter (una letra mayúscula o minúscula, un dígito, un carácter especial,...). Por ejemplo: **'3'** (con comillas simples) es un carácter, mientras que 3 (sin comillas) es un entero y "3" es una cadena de caracteres que contiene un carácter. Esta cadena de caracteres es almacenada por el compilador en alguna parte del programa y consta de dos caracteres, el primero es el carácter 3 y el segundo el carácter cero que se utiliza para finalizar la cadena de caracteres.

En C, cada carácter es equivalente a un número entero que depende de la implementación.

En C no está establecido el juego de caracteres, sólo se exige un conjunto mínimo de caracteres que puede ampliarse según las preferencias del desarrollador.

En C, los caracteres tienen un tratamiento muy similar a los enteros, existiendo una relación de orden entre ellos y siendo posible aplicarles operaciones propias de los números enteros.

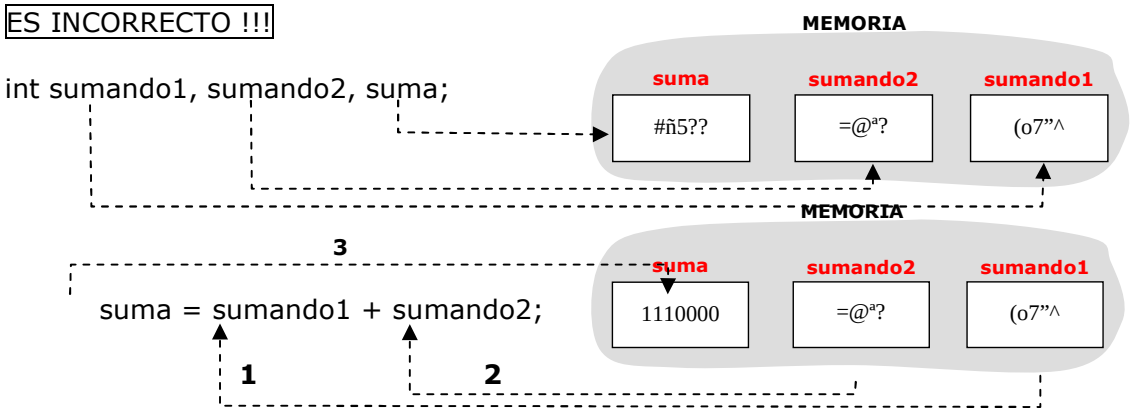
'a' < 'b' < … < 'z'
La operación 'f' + 'j' es posible

E*n C es necesario declarar todas las variables que se vayan a utilizar*, indicando el tipo de datos que pueden contener. Una variable no declarada produce un mensaje de error en la fase de compilación.

Cuando se declara una variable, se le reserva memoria de acuerdo con el tipo indicado en su declaración. Tras la declaración de una variable, no es posible conocer el valor que contiene, ya que éste será el contenido de la porción de memoria que el compilador le ha asignado. Puede ocurrir que este contenido sea la denominada *basura informática*, valores almacenados en usos anteriores de esa misma zona de memoria para otros fines. Por este motivo, y con el objetivo de garantizar el adecuado contenido de las variables declaradas en nuestros programas, es necesario iniciarlas (o establecerlas) adecuadamente antes de utilizarlas en las operaciones.

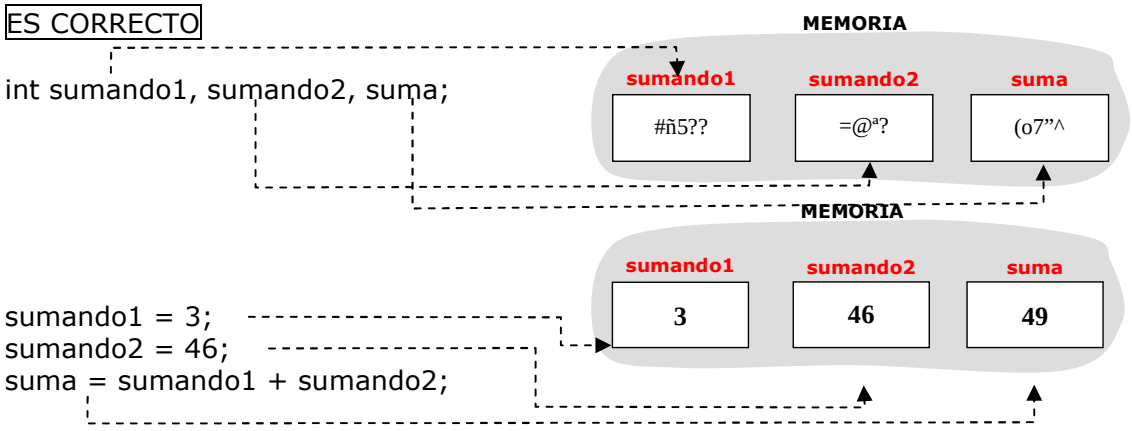
Ejemplos:

ES INCORRECTO !!!



RESULTADO INCOHERENTE ➡ **ERROR**

ES CORRECTO



A continuación se muestra un programa en el que se intenta reproducir la explicación gráfica dada sobre la forma de iniciar las variables.

```
int main(void)
{
     int sumando1, sumando2, suma;

     suma = sumando1 + sumando2;        // esta suma puede dar cualquier valor
```

```
    printf("sumando1 = %d\n", sumando1);
    printf("sumando2 = %d\n", sumando2);
    printf("suma = %d\n", suma);

    sumando1 = 3;
    sumando2 = 46;

    suma = sumando1 + sumando2;

    printf("sumando1 = %d\n", sumando1);
    printf("sumando2 = %d\n", sumando2);
    printf("suma = %d\n", suma);                    // esta suma debe dar 49

    system("pause");
    return 0;
}
```

A continuación se va a explicar cómo se almacena en C un dato de cada tipo fundamental.

2.1 Caracteres (tipo char)

Las variables del tipo carácter (*tipo **char***) contienen un único carácter y se almacenan en un ***byte*** de memoria (8 bits). En un bit se pueden almacenar (en diferentes instantes de tiempo) dos valores diferentes (0 y 1); con dos bits se pueden almacenar $2^2 = 4$ valores (00, 01, 10, 11 en binario; 0, 1 2, 3 en decimal). Con 8 bits se podrán almacenar $2^8 = 256$ valores diferentes (normalmente entre 0 y 255; ciertos compiladores pueden almacenar valores entre -128 y 127).

La declaración de variables tipo carácter puede tener la forma:

char c;
char c1, c2, car;

Se puede declarar más de una variable de un tipo determinado en una sola instrucción. Se puede también establecer el valor de la variable en la declaración. Por ejemplo, para definir la variable carácter **letra** y asignarle el valor '**a'**, se puede escribir:

char letra = 'a';

A partir de ese momento queda definida la variable **letra** con el valor correspondiente a la letra **a**. Recuérdese que el valor **'a'** utilizado para inicializar la variable **letra** es una constante carácter.

Internamente, el compilador reserva en memoria un sólo byte para la variable **letra**, almacenando su contenido como un número entero, el correspondiente a la letra **a** en el código ASCII. Este código es una codificación que establece la representación numérica correspondiente a cada carácter estándar. Existe también un código ASCII extendido que contiene caracteres especiales y caracteres específicos de los alfabetos de diversos países, como por ejemplo las *vocales acentuadas* y la *letra ñ* para el castellano. El código ASCII asocia números consecutivos a las letras mayúsculas y minúsculas ordenadas alfabéticamente. Esto simplifica notablemente ciertas operaciones de ordenación alfabética de nombres.

Volviendo al ejemplo de la variable **letra**, su contenido puede ser modificado cuando se desee por medio de una sentencia que le asigne otro valor, por ejemplo:

letra = 'z';

También puede utilizarse una variable ***char*** para dar valor a otra variable de tipo ***char***:

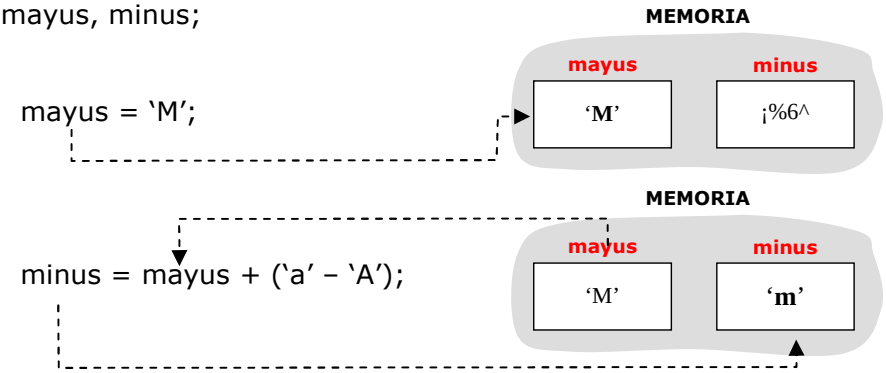
char caracter;
caracter = letra; // Ahora caracter es igual a 'z'

Como una variable tipo ***char*** es un número entero (entre 0 y 255), se puede utilizar el contenido de una variable ***char*** de la misma forma que se utiliza un número entero, por lo que están permitidas operaciones como:

letra = letra + 1;

En este ejemplo, si el contenido de **letra** era una '**a**', al incrementarse en una unidad pasa a contener una '**b**'.

char mayus, minus;



Este ejemplo es muy interesante: puesto que la diferencia numérica entre las letras minúsculas y mayúsculas es siempre la misma (en el sistema de codificación ASCII y en la mayoría de los sistemas de codificación actualmente utilizados), esta sentencia pasa una letra mayúscula a la correspondiente letra minúscula sumándole dicha diferencia numérica.
Recuérdese para concluir que las variables tipo ***char*** son y se almacenan como números enteros. Ya se verá más adelante que se pueden escribir como caracteres o como números según qué formato de conversión se utilice en la llamada a la función de escritura.

2.2 Números enteros (tipo int)

Las normas del C exigen que una variable tipo ***int*** se almacena al menos 2 bytes (16 bits), aunque la mayoría de compiladores actuales utilizan 4 bytes (32 bits).

Con 16 bits se pueden almacenar 2^{16} = 65536 números enteros diferentes: de 0 al 65535 para variables sin signo, y de -32768 al 32767 para variables con signo (que pueden ser positivas y negativas), que es la opción por defecto.

Con 32 bits se pueden almacenar 2^{32} = 4294967296 números enteros diferentes: de 0 al 4294967295 para variables sin signo, y de -2147483648 al 2147483647 para variables con signo (que pueden ser positivas y negativas), que es la opción por defecto.

Una variable entera (tipo ***int***) se declara, o se declara y se inicia en la forma:

int numero; // declaración
int nota = 10; // declaración e asignación de valor inicial de la variable nota.

Las variables ***numero*** y ***nota*** podrán estar entre -2147483648 y 2147483647. Cuando a una variable ***int*** se le asigna en tiempo de ejecución un valor que quede fuera del rango permitido (situación de ***overflow*** o valor excesivo), se produce un error que no es fácil de

detectar. *El programa no avisa al usuario de esta circunstancia* pero el error se manifiesta ocasionando como consecuencia resultados imprevisibles.

Por ejemplo, si sumamos una unidad al máximo valor de un entero puede observarse que el valor que se obtiene no es 2147483648 sino que se obtiene un valor negativo (-2147483648) debido a que se excede la capacidad de almacenamiento del tipo int.

Podéis probarlo directamente en el intérprete, introduciendo las siguientes instrucciones:

int x = 2147483647;

x = x + 1;

De este modo el valor de x será -2147483648

2.3 Números enteros (tipo long)

Existe la posibilidad de utilizar enteros con un rango mayor si en su declaración se especifica como tipo ***long***:

long numero_grande;

El rango de un entero ***long*** puede variar según el ordenador o el compilador que se emplee, pero de ordinario se utilizan 4 bytes (32 bits) para almacenarlos, por lo que se pueden representar 2^{32} = 4.294.967.296 números enteros diferentes. Si se utilizan números con signo, podrán representarse números entre -2.147.483.648 y 2.147.483.647.

En los compiladores e intérpretes que utilizaremos nosotros el tipo long tiene la misma capacidad que el tipo int.

2.4 Números reales (tipo float)

En muchas aplicaciones hacen falta variables reales, capaces de representar magnitudes que contengan *una parte entera y una parte fraccionaria* o *decimal*. Estas variables se llaman también de ***coma flotante***. De ordinario, en base 10 y con notación científica, estas variables se representan por medio de la ***mantisa***, que es un número mayor o igual que 0.1 y menor que 1.0, y un ***exponente*** que representa la potencia de 10 por la que hay que multiplicar la mantisa para obtener el número considerado. Por ejemplo, el número π se representa como $0.3141592654 * 10^1$ y en lenguaje C se expresa como 0.3141592654e1
Tanto la ***mantisa*** como el ***exponente*** pueden ser positivos y negativos.

Los ordenadores trabajan en base 2. Por eso un número de tipo ***float*** se almacena en 4 bytes (32 bits), utilizando *24 bits para la mantisa* (1 para el signo y 23 para el valor) y *8 bits para el exponente* (1 para el signo y 7 para el valor). Es interesante ver qué clase de números de coma flotante pueden representarse de esta forma. En este caso hay que distinguir el ***rango*** de la ***precisión***. La *precisión* hace referencia al número de cifras con las que se representa la *mantisa*: con 23 bits el número más grande que se puede representar es, 2^{23} = 8.388.608 lo cual quiere decir que se pueden representar todos los números decimales de 6 cifras y la mayor parte –aunque no todos– de los de 7 cifras (por ejemplo, el número 9.213.456 no se puede representar con 23 bits). Por eso se dice que las variables tipo ***float*** tienen entre 6 y 7 cifras decimales de precisión.

Respecto al *exponente de dos* por el que hay que multiplicar la *mantisa* en base 2, con 7 bits el número más grande que se puede representar es 127. El ***rango*** vendrá definido por la potencia, $2^{127} = 1.7014 * 10^{38}$, lo cual indica el número más grande representable de

esta forma. El número más pequeño en valor absoluto será del orden de 2^{-128} = 2.9385 * 10^{-39}. Las variables tipo ***float*** se declaran de la forma:
float numero_real;

Las variables tipo ***float*** pueden ser establecidas en el momento de la declaración, de forma análoga a las variables tipo ***int***.
float numero_real =3.14;

2.5 Números reales (tipo double)

Las variables tipo ***float*** tienen un *rango* y, sobre todo, una *precisión* muy limitada, insuficiente para la mayor parte de los cálculos técnicos y científicos. Este problema se soluciona con el tipo ***double***, que utiliza 8 bytes (64 bits) para almacenar una variable. Se utilizan *53 bits para la mantisa* (1 para el signo y 52 para el valor) y *11 para el exponente* (1 para el signo y 10 para el valor). La ***precisión*** es en este caso, 2^{52} = 4.503.599.627.370.496 lo cual representa entre 15 y 16 cifras decimales equivalentes. Con respecto al ***rango***, con un exponente de 10 bits el número más grande que se puede representar será del orden de 2 elevado a 2 elevado a 10 (que es 1024): 2^{1024} = 1.7977 * 10^{308}.

Las variables tipo ***double*** se declaran de forma análoga a las anteriores:

double real_grande;

Por último, existe la posibilidad de declarar una variable como ***long double***, aunque las normas establecidas no obligan a un *rango* y a una *precisión* mayor ***double***. Estas variables se declaran de la siguiente forma:

long double real_muy_grande;

El rango y la precisión de las variables long double no está normalizado y cada compilador lo implementa como considera oportuno.

2.6 Conversiones de tipo implícitas y explícitas (casting)

Cuando en una operación se mezclan variables de distintos tipos, C realiza conversiones de tipos implícitas, consiguiendo que la operación sea factible y tenga un tipo de resultado preestablecido por sus operandos. Así, por ejemplo, para poder sumar dos variables hace falta que ambas sean del mismo tipo. Si una es *int* y otra *float*, la primera se convierte a *float* (es decir, la variable del tipo de menor rango se convierte al tipo de mayor rango), antes de realizar la operación. En esta conversión automática e implícita de tipo, el programador no necesita intervenir, aunque sí conocer sus reglas, pues la variable de menor rango se transforma internamente para soportar al rango de la otra.

Así pues, cuando tipos diferentes de constantes y/o variables aparecen en una misma expresión relacionadas por un operador, el compilador convierte los operandos al mismo tipo de acuerdo con los rangos, que de mayor a menor se ordenan del siguiente modo:

long double > double > float > long > int > char

Otra clase de conversión implícita tiene lugar cuando el resultado de una expresión es asignado a una variable, pues dicho resultado se convierte al tipo de la variable (en este caso, ésta puede ser de menor rango que la expresión, por lo que esta conversión puede perder información y ser peligrosa).

Por ejemplo, si realizamos la siguiente operación con números enteros, obtendremos un entero:

```
1 / 3          // proporcionará el valor 0

1 / 3.0        // proporcionará el valor 0.33333

(float) 1/3    // proporcionará el valor 0.33333
```

Tal como hemos mostrado en el último ejemplo, en C existe la posibilidad de realizar *conversiones explícitas de tipo* (llamadas ***casting***, en la literatura inglesa). Para ello basta preceder la constante, variable o expresión que se desea convertir por el tipo al que se desea convertir, encerrado entre paréntesis. A continuación mostramos un ejemplo completo en el que se aprecia la utilización de un casting a float.

```
// Ejemplo de conversión explícita necesaria
//
// Transformar un número aleatorio del intervalo [0..32767]
// en un número aleatorio del intervalo [0..1)

void ej_casting_necesario()
{
      int n;
      float r;

      n = rand ();
      r = (float) n / 32767;

      printf ("valor real aleatorio del intervalo [0, 1) es el %f\n", r);
}
```

Puede observarse que si se elimina el casting a float, casi siempre se obtiene el valor 0 ya que el valor de rand pertenece al intervalo [0, 32767] y su valor sólo será 1 cuando la función rand() proporcione el valor 32767.

3 OPERACIONES

El lenguaje C permite realizar distintas clases de operaciones sobre los datos que maneja un programa.

Un **operador** es un carácter o grupo de caracteres que actúa sobre uno, dos o más valores (ya sean variables o constantes) *para realizar una determinada **operación** y obtener un determinado **resultado***. Ejemplos típicos de operadores son el de la *suma* (+), la *diferencia* (-), el *producto* (*), etc. Los operadores pueden ser ***unarios***, ***binarios*** y ***n-arios***, según actúen sobre uno, dos o un n operandos, respectivamente.

En C existen muchos operadores de diversos tipos que se verán a continuación.

3.1 Operadores aritméticos

Los ***operadores aritméticos*** son los más sencillos de entender y de utilizar. Todos ellos son operadores binarios.

En C se utilizan los cinco operadores siguientes:

Suma: **+**
Resta: **-**
Multiplicación: *****
División: **/**
Resto: **%**

Todos estos operadores se pueden aplicar a constantes, variables y expresiones. El resultado es el que se obtiene de aplicar la operación correspondiente entre los dos operandos.
El único operador que requiere una explicación adicional es el operador ***resto*** %. En realidad su nombre completo es ***resto de la división entera***. Este operador se aplica solamente a constantes, variables o expresiones de tipo ***int***. Aclarado esto, su significado es evidente:

23%4 es 3, puesto que el resto de dividir 23 por 4 es 3.

Si ***a%b*** es cero, ***a*** es múltiplo de ***b***

Una ***expresión*** es un conjunto de variables y constantes –y también de otras expresiones más sencillas– relacionadas mediante distintos operadores. A continuación se muestran algunas expresiones válidas:

```
6*x*x – 4*x + 1

3*x*x – sin(x)*cos(x)
```

Las expresiones pueden contener ***paréntesis*** que agrupan a algunos de sus términos. Puede haber paréntesis contenidos dentro de otros paréntesis. El significado de los paréntesis coincide con el habitual en las expresiones matemáticas, con algunas características importantes que se verán más adelante. En ocasiones, la introducción de espacios en blanco mejora la legibilidad de las expresiones.

Una instrucción es una expresión finalizada mediante un punto y coma. Por ejemplo, las expresiones anteriores pueden expresarse como instrucciones del siguiente modo:

```
6*x*x – 4*x + 1;

3*x*x – sin(x)*cos(x);
```

3.2 Operador de asignación

El ***operador de asignación*** atribuye a una variable, es decir, deposita en la zona de memoria correspondiente a dicha variable, el resultado de una expresión o el valor de otra variable.

El operador de asignación se denota con el signo de la ***igualdad*** (=), que no debe ser confundido con la igualdad matemática (que en C se representa mediante: ==)

Su forma general es:

nombre_de_variable **=** expresion;

cuyo funcionamiento es: se evalúa **expresion** y el resultado se deposita en **nombre_de_variable**, sustituyendo cualquier otro valor que hubiera en esa posición de memoria. Una posible utilización de este operador es la siguiente:

variable **=** variable + 1;

Desde el punto de vista matemático este ejemplo no tiene sentido (**¡¡¡**equivale a 0 = 1**!!!**), pero sí lo tiene considerando que en realidad ***el operador de asignación (=) representa una sustitución***; en efecto, se toma el valor de **variable** contenido en la memoria, se le suma una unidad y el valor resultante vuelve a depositarse en memoria en la zona correspondiente al identificador **variable**, sustituyendo al valor que había anteriormente. El resultado ha sido incrementar el valor de **variable** en una unidad.

Así pues, una variable puede aparecer a la izquierda y a la derecha del operador (=).

A la izquierda del operador de asignación (=) no puede haber nunca una expresión, tiene que ser necesariamente el nombre de una variable. Es incorrecto, por tanto, escribir lo siguiente:

```
a + b = c;       // Incorrecto ya que a la izquierda debe haber una variable.
100 = 99 + 1;    // Incorrecto ya que 100 no es una variable.
```

Son ejemplos de asignaciones correctas, las siguientes:

```
valor = 8;
incremento = 2;
resultado = incremento + valor;
distancia = distancia + 1;
rango = rango /2.0;
x = x * (3.0 * y - 1.0);
```

3.3 Operadores relacionales

Una característica imprescindible de cualquier lenguaje de programación es la de ***considerar alternativas***, esto es, la de proceder de un modo u otro según se cumplan o no ciertas condiciones. Los ***operadores relacionales*** permiten estudiar si se cumplen o no esas condiciones.

En el lenguaje natural, existen varias palabras o formas de indicar si se cumple o no una determinada condición. En inglés estas formas son (***yes***, ***no***), (***on***, ***off***), (***true***, ***false***), etc. En Informática se ha hecho bastante general el utilizar la última de las formas citadas: (***true***, ***false***).

Si una condición se cumple, el resultado es verdadero; en caso contrario, el resultado es falso.

En C el valor verdadero se representa mediante el 1 y el falso mediante el 0.

Los ***operadores relacionales*** de C son los siguientes:

Igual que: **==**
Menor que: **<**
Mayor que: **>**
Menor o igual que: **<=**
Mayor o igual que: **>=**
Distinto que: **!=**

Todos ***operadores relacionales*** son operadores ***binarios*** (tienen dos operandos), y su forma general es la siguiente:

expresion1 **op** expresion2

donde ***op*** es uno de los operadores siguientes: **==**, **<**, **>**, **<=**, **>=**, **!=**

El funcionamiento de estos operadores es el siguiente: se evalúan **expresion1** y **expresion2**, y se comparan los valores resultantes. *Si la condición representada por el operador relacional se cumple, el resultado es verdadero; si la condición no se cumple, el resultado es falso*.

A continuación se incluyen algunos ejemplos de estos operadores aplicados a constantes:

```
(2==1)        /* resultado: 0, equivalente a falso porque la condición no se cumple */
('a' == 'a')  /* resultado: 1, equivalente a cierto porque la condición se cumple */
(3<3)         /* resultado: 0, equivalente a falso porque la condición no se cumple */
(1!=1)        /* resultado: 0, equivalente a falso porque la condición no se cumple */
```

3.4 Operadores lógicos

Los ***operadores lógicos*** son operadores binarios que permiten combinar los resultados de los operadores relacionales, comprobando que se cumplen simultáneamente varias condiciones, que se cumple una u otra, etc.

El lenguaje C tiene dos operadores lógicos binarios: el operador **&&** y el operador **||**. En inglés estos operadores se pronuncian ***and*** y ***or***. Su forma general es la siguiente:

expresion1 **||** expresion2
expresion1 **&&** expresion2

El operador **&&** devuelve ***1*** si tanto **expresion1** como **expresion2** son verdaderas, y ***0*** en caso contrario, es decir si una de las dos expresiones o ambas son falsas; por otra parte, el operador **||** devuelve ***1*** si al menos una de las expresiones es cierta.

Es importante tener en cuenta que los compiladores de C tratan de optimizar la ejecución de estas expresiones, lo cual puede tener a veces efectos no esperados. Por ejemplo: para que el resultado del operador **&&** sea verdadero, ambas expresiones tienen que ser verdaderas; si se evalúa **expresion1** y es falsa, ya no hace falta evaluar **expresion2**, y de hecho no se evalúa. Algo parecido pasa con el operador **||**: si **expresion1** es cierta, ya no hace falta evaluar **expresion2**.

A continuación se muestra una tabla que explica el funcionamiento del operador **&&** en todos los casos posibles. En ella indicaremos los valores obtenidos después de la evaluación de la expresión indicada en la primera fila. Utilizaremos un 0 para indicar que la expresión

se evalúa a 0. Utilizaremos un 1 para indicar que la evaluación ha producido cualquier valor distinto de 0. También indicaremos cuando no se evalúa la expresión.

Expresión1	Expresión2	Expresión1 **&&** Expresión2
0	**No evaluada**	0
1	1	1
1	0	0

De la tabla anterior se deduce que el operador **&&** no tiene la propiedad conmutativa.

Para el operador **||** tendremos la siguiente tabla:

Expresión1	Expresión2	Expresión1 **||** Expresión2
0	1	1
1	**No evaluada**	1
0	0	0

De la tabla anterior se deduce que el operador **||** no tiene la propiedad conmutativa.

Los operadores **&&** y **||** se pueden combinar entre sí (quizás agrupados entre paréntesis), dando a veces un código de difícil interpretación.

Por ejemplo:

```
(2==1) || (-1==-1) /* el resultado es 1 */
(2==2) && (3==-1) /* el resultado es 0 */
((2==2) && (3==3)) || (4==0) /* el resultado es 1 */
((6==6) || (8==0)) && ((5==5) && (3==2)) /* el resultado es 0 */
```

3.5 Algunos operadores unarios

Además de los operadores vistos hasta ahora, el lenguaje C dispone de otros operadores, entre los que se encuentran los ***operadores unarios*** que se muestran a continuación:

Operador ***menos*** (–).
El efecto de este operador en una expresión es cambiar el signo de la variable o expresión que le sigue. La forma general de este operador es:

- expresión

Operador ***mas*** (+).
Este operador unario tiene como finalidad servir de complemento al operador (–) visto anteriormente. Se puede anteponer a una variable o expresión como operador unario, pero en realidad no hace nada.

+ expresión

Operador negación lógica (**!**).
Este operador devuelve 0 (*falso)* si se aplica a un valor **distinto de 0** (*cierto)*, y devuelve un 1 (*true)* si se aplica a un valor 0 (*falso)*. Su forma general es:

! expresión

4 INSTRUCCIONES

Las operaciones se convierten en instrucciones al añadirles un punto y coma.

Por ejemplo: 2+3 es una expresión y 2+3; es una instrucción.

Las instrucciones se encadenan para formar los programas. La forma de encadenar las instrucciones es importante ya que según se haga la programación puede ser más o menos sencilla.

Es decir, un programa estará formado por una secuencia de instrucciones de forma similar a la siguiente:

```
…
Instrucción_1;
Instrucción_2;
…
Instrucción_N;
…
```

Y normalmente se ejecutarán una a continuación de la otra, empezando por la primera y terminando por la última, esto es lo que entendemos cuando decimos que el programa se ejecuta secuencialmente.

Dado que las instrucciones se ejecutan una detrás de otra. Para que esto sea útil se definen unas instrucciones especiales que permiten que, dentro de la misma instrucción, se ejecuten condicionalmente instrucciones diferentes o que se pueda ejecutar una instrucción de forma reiterada, repetida o repetitiva. Más adelante veremos cómo son estas instrucciones especiales, en concreto, las instrucciones de selección y las instrucciones de repetición.

También veremos algunas instrucciones que permiten modificar el flujo normal de ejecución del programa, es decir que en lugar de ejecutarse secuencialmente el flujo de ejecución pasa a otro lugar del programa.

5 BLOQUES

Un bloque es un conjunto de instrucciones encerradas entre llaves. Por ejemplo:

```
{
  Instrucción_1;
  Instrucción_2;
  …
  Instrucción_N;
}
```

Por ejemplo, a continuación se muestra un bloque formado por 3 instrucciones:

```
{ int x; ++x; x = sin(x); }
```

Un bloque se puede utilizar en cualquier lugar en que pueda utilizarse una instrucción.

Se puede utilizar un bloque con una única instrucción (algunas veces se hace para que la programación sea más sencilla).

6 FUNCIONES

Con el fin de simplificar los programas un bloque de instrucciones se agrupan bajo un nombre pasando a formar una nueva función. Es decir, una función es una instrucción definida por el programador.

La ejecución de una función supone la ejecución de las instrucciones asociadas, es decir, si al ordenador se le solicita la ejecución de una función, ésta se llevará a cabo ejecutando el bloque de instrucciones existentes en el bloque de definición de la función.

A continuación veremos algunas funciones ya definidas por programadores expertos que nos permiten realizar tareas importantes y explicaremos cómo utilizarlas.

7 FUNCIONES DE ENTRADA/SALIDA

Las funciones de entrada/salida (Input/Output) son un conjunto de funciones, incluidas con el compilador, que permiten a un programa recibir y enviar datos al exterior. Aunque el lenguaje de programación C contiene varias funciones de entrada/salida, nos vamos a centrar sólo en dos: *printf()* para la salida y *scanf()* para la entrada.

7.1 Salida o presentación de datos

Los programas interaccionan con los usuarios mostrando información. A la presentación de datos al usuario se le denomina salida de datos del programa.

Por muchos cálculos que hagamos con el ordenador, si estos no los mostramos o los escribimos en algún sitio no tienen demasiada utilidad.

Será necesario mostrar datos al usuario cuando queramos que el programa proporcione al usuario cualquier tipo de información tal como mensajes de ayuda para manejo del programa, mensajes de ayuda para indicar qué información debe introducir el usuario, mensajes que contengan las respuestas del programa y cualquier otra información de interés para el usuario del programa.

Con el fin de mostrar datos al usuario se utiliza la función printf(). Esta función se encuentra definida en la biblioteca del sistema (que depende del sistema operativo en el que se ejecute el programa) y declarada en el fichero: <stdio.h>

7.2 Función printf()

Los programas deben comunicarse con el exterior, con su entorno más inmediato, con el fin de mostrar al usuario las informaciones adecuadas. Para ello se utiliza la función printf() que suponemos definida y que nosotros podemos utilizar a nuestra conveniencia.

La función printf() imprime en la unidad de salida (por defecto, la pantalla), el texto, las constantes y las variables que se indiquen. Se van a distinguir dos formas distintas de usar la función printf():

•printf("cadena de caracteres");
•printf("cadena de formato", arg1, arg2,…..argn);

La primera de las formas de uso de la función printf() permite mostrar por pantalla mensajes formados por cadenas de caracteres. Los mensajes se muestran de manera secuencial (en la misma línea) a no ser que se indique que se desea imprimir una nueva línea, lo cual se indica mediante el uso del *indicador de formato* **\n**

Los indicadores de formato nunca se muestran por pantalla y sirven para que el compilador sepa cómo tiene que imprimir la información y se pueden colocar en cualquier posición dentro de la primera cadena de caracteres.

printf ("Hola mundo");

printf("Hola");
printf("Mundo");
printf("\nHola\n");
printf("Adios");

Veamos un ejemplo completo que saluda al usuario y finaliza el programa. No es un programa demasiado útil, pero nos puede servir de base para realizar nuestros propios programas y que estos envíen mensajes al exterior programa.

```
#include <stdio.h>

int main(void)
{
    printf ("Bienvenido a la EUITI");

    return 0;
}
```

El programa anterior es un programa completo, se utiliza una instrucción del preprocesador del lenguaje C (la única que utilizaremos nosotros) que indica al preprocesador que incluya el fichero stdio.h que se encuentra en un directorio especial y que depende del sistema operativo que estemos utilizando (esto lo sabe el preprocesador porque se están utilizando paréntesis angulares; si se utilizara "stdio.h" entendería que el fichero se encuentra en el mismo directorio que el programa)

En este fichero se encuentra la declaración (la definición se encuentra en la biblioteca) de la función printf(), para que pueda utilizarse dentro del programa.

A continuación se encuentra la definición de la función main() el pefijo **int** es una abreviatura de **integer** y significa que la función main() debe devolver un valor entero para que el sistema operativo conozca cuál ha sido el resultado del programa con el fin de encadenar la ejecución de unos programas con otros en función del resultado obtenido (nosotros no lo utilizaremos, pero es conveniente saberlo porque algunos compiladores lo exigen y por ello hemos utilizado la instrucción **return 0;**

El valor 0 significa que el programa no ha producido ningún error y cualquier otro valor indica algún error en el programa (cada error identificado mediante un número)

En la definición de la función se ha utilizado la palabra reservada **void.** Esto significa que estamos definiendo una función que no utilizará argumentos cuando

sea solicitada su ejecución, es decir, para ejecutar el programa sólo introduciremos su nombre. Por ejemplo, si el programa se denomina "saludo.exe" para ejecutarlo sólo introduciremos la palabra "saludo" o "saludo.exe". Como esta será la forma normal de ejecutar nuestros programas no explicaremos cómo utilizar argumentos en la línea de órdenes, es decir, no explicaremos cómo utilizar los argumentos si la forma de llamar al programa fuera la siguiente: "saludo José Javier" ya que para nosotros no habrá argumentos en la línea de órdenes (si alguno tiene interés en utilizar argumentos en la línea de órdenes, es mejor esperar a estudiar los arrays y las variables de tipo cadena)

La línea que provoca que el proceso (programa en ejecución) proporcione un mensaje al usuario en la pantalla del ordenador es la siguiente:

```
printf ("Bienvenido a la EUITI");
```

Esta línea contiene el nombre de la función **printf** y a continuación, entre paréntesis, una cadena de caracteres. Esta cadena de caracteres es lo que se mostrará al usuario.

Del mismo modo que hemos puesto una línea, podíamos escribir en pantalla todas las líneas que necesitáramos para informar al usuario.

La otra forma de usar la función printf() permitirá mostrar mensajes cuyo contenido no conocemos (pero que los tiene almacenados el proceso) en el momento de ejecutar el programa.

Por ejemplo, si deseamos obtener un conjunto de operaciones sobre un par de números conocidos, podíamos hacer lo siguiente:

// tema2_io\ej_salida.c

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

int main(void)
{
    float a, b, r;

    a = 3.2457;
    b = 1.2345;

    printf(" a = %f", a);
    printf("\n");
    printf(" b = %f", b);
    printf("\n");

    r = a + b;
    printf("a + b = %f", r);
    printf("\n");

    r = a - b;
    printf("a - b = %f", r);
    printf("\n");

    r = a * b;
    printf("a x b = %f", r);
    printf("\n");

    r = a / b;
    printf("a / b = %f", r);
    printf("\n");

    system("pause");
```

```
    return 0;
}
```

El programa necesita algunas explicaciones; vayamos por partes.

En primer lugar hemos introducido otro fichero, el <stdlib.h>, debido a que contiene la declaración de la función system(). Esta función es capaz de ejecutar cualquier programa del sistema, en este caso se ejecuta la orden pause que permite detener la ejecución del programa hasta que el usuario pulsa la tecla intro.

El resto del programa lo que hace es asignar unos valores conocidos a las variables a y b y realizar unos cálculos con ellas con el fin de mostrarlos al usuario.

Analicemos más en detalle las 12 instrucciones printf(), como sabemos, estas instrucciones lo que hacen es mostrar sus argumentos en pantalla.

En primer lugar analicemos las 6 instrucciones iguales:

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

Estas instrucciones (cuando se ejecutan) lo que hacen es saltar al inicio de la siguiente línea, es decir, saltan a la nueva línea en la pantalla del ordenador. Podemos observar que la instrucción printf() sólo utiliza un argumento que es una cadena de caracteres, pero se han utilizado dos caracteres que tienen un significado especial. En este caso, la pareja de caracteres \n (se lee "contrabarra n") equivale a un único carácter que se interpreta como "saltar al inicio de la nueva línea".

Ahora nos quedan el resto de instrucciones printf():

```
    printf(" a = %f", a);

    printf(" b = %f", b);

    printf("a + b = %f", r);

    printf("a - b = %f", r);

    printf("a x b = %f", r);

    printf("a / b = %f", r);
```

Vemos que estas instrucciones son diferentes de las anteriores, ya que, en este caso, tienen dos argumentos. El primero es una cadena de caracteres y el segundo una variable.

Dentro de la cadena de caracteres podemos observar el símbolo de porcentaje que, en este caso, está seguido por la letra f. Esto significa que le debe seguir una variable (o expresión) de tipo float, es decir, un número con decimales.

Resumiendo, al usar esta forma de printf(), en la *cadena de formato* además del texto que queramos que salga literalmente en la pantalla, intercalaremos los **indicadores (o especificadores) de formato** necesarios para mostrar los valores que deseemos.

En la siguiente tabla se ven algunos de los distintos indicadores de formato que podemos usar en función del tipo de datos que queramos mostrar por pantalla.

Indicador de formato	Tipo de dato
%d	Entero en base diez (int)
%c	Carácter (char)
%f	Coma flotante (float)
%.2f	Coma flotante con dos dígitos de precisión
%ld	Entero largo en base diez (long)
%lf	Coma flotante (double)
%x	Entero hexadecimal
%s	Cadena de caracteres.

En un misma llamada a la función **printf()** podemos usar tantos indicadores de formato como deseemos, pero tenemos que asegurarnos que cada indicador de formato tiene su variable correspondiente y que además se refiere a un dato del tipo de datos que corresponda al indicador de formato.

Por ejemplo, las instrucciones anteriores se podían haber escrito del siguiente modo:

```
printf("%s %f", " a =", a);

printf("%s %f", " b =", b);

printf("%s %f", "a + b =", r);

printf("%s %f", "a - b =", r);

printf("%s %f", "a x b =", r);

printf("%s %f", "a / b =", r);
```

Como podemos observar este programa tiene poca utilidad ya que los valores de a y de b son valores fijos que sólo podemos cambiarlos si disponemos del código fuente. Además, no tiene mucho sentido modificar sus valores para luego tener que recompilar el programa cada vez que queremos trabajar con distintos valores para a y b.

Lo que necesitamos es que el programa nos permita introducir los valores de a y de b desde el teclado (o desde un fichero) con el fin de realizar los cálculos con dichos valores sin tener que modificar el programa y, por lo tanto, sin tener que compilarlo.

La función scanf(), que veremos a continuación, permite que el programa lea valores de la entrada y los asigne a las variables que se proporcionan como argumentos.

7.3 Entrada de datos a la aplicación

Cuando se ejecuta cualquier aplicación, es muy común que esta nos pida datos. Por ejemplo, nos puede pedir nuestra edad, nuestra fecha de nacimiento, nuestra dirección de correo electrónico o cualquier otro dato dependiendo de lo que necesite realizar la aplicación.

La única forma que tiene un programa de obtener datos, es pedirlos, es decir, obtenerlos del exterior del propio programa. A estos datos se les suele denominar inputs o entradas al, o del, programa.

Por supuesto, también es posible que el programa obtenga los datos de otras fuentes tales como de otros ficheros, líneas de comunicaciones o dispositivos conectados al ordenador. En general, a la obtención de datos externos para ser utilizados por el programa se le denomina captura de datos u obtención de datos, y los datos obtenidos por el programa se utilizan para que el ordenador realice los cálculos para los cuales ha sido diseñado.

En muchas ocasiones, la entrada, de los datos al programa, se realiza mediante el teclado del ordenador.

Por ejemplo, si disponemos de un programa que realiza el cálculo de la raíz cuadrada, tendremos que proporcionar el número del cual queremos obtener su raíz con el fin de que sea calculada por el programa. La entrada la proporcionaremos nosotros al programa y el programa proporcionará el resultado en la salida (normalmente la pantalla del ordenador)

La función que obtiene los datos de la entrada y los guarda en variables del programa se denomina scanf() y es una función, de la biblioteca de entrada y salida del sistema.

Para utilizar la función scanf() es suficiente con incluir el fichero <stdio.h> al inicio del programa del siguiente modo:

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

Con esta línea le estamos indicando al compilador que incluya el fichero stdio.h al inicio de nuestro programa. De este modo todas las funciones declaradas en este fichero de cabecera estarán disponibles para nuestro programa.

7.4 Función scanf()

La función scanf() es análoga en muchos aspectos a printf(), y se utiliza para leer datos de la entrada estándar (que por defecto es el teclado). La forma general de esta función es la siguiente:

•scanf("%x1 %x2...", &arg1, &arg2, ...);

Donde %x1, %x2,… son los indicadores de formato explicados en la tabla anterior y que especifican el tipo de datos a leer; y arg1, arg2,… son nombres de variables cuyo tipo de datos tiene que tener correspondencia uno a uno (el 1º con el 1º, el 2º con el 2º, etc.) con los indicadores de formato.

Una vez finalizada la ejecución de la función scanf(), las variables arg1, arg2, etc. contendrán los valores que haya introducido el usuario.

Por ejemplo, si en la entrada tenemos el número 113 y utilizamos scanf("%d", &a) después de la ejecución la variable a contendrá el valor 113. Si, por el contrario, utilizamos la scanf("%c%c%c", &car1, &car2, &car3) las variables car1, car2 y car3 tendrán asociados los caracteres '1', '1' y '2' respectivamente. Evidentemente, esto será así si hemos definido la variable a de tipo int y las variables car1, car2 y car3 de tipo char.

Las lecturas de datos consecutivos se pueden realizar en un único scanf() o en varios consecutivos, por ejemplo, las siguientes instrucciones:

```
printf("\nEscribe dos números enteros y un caracter: ");
scanf("%d %d %c", &a, &b, &c);
```

Son parecidas a las siguientes:

```
printf("\nDame un número entero: ");
scanf("%d", &a);
printf("\nDame otro número entero: ");
scanf("%d", &b);
printf("\nDame un caracter: ");
scanf("%c", &c);
```

y también a las siguientes :

```
printf("\nDame dos números enteros y un caracter: ");
scanf("%d", &a);
scanf("%d", &b);
scanf("%c", &c);
```

A continuación, hemos modificado el ejemplo anterior de modo que los valores de a y de b no estén asignados en el programa de forma directa, sino que sea el propio usuario el que los introduzca al ejecutar el programa.

```
// tema2_io\ej_entrada.c

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    float a, b, r;

    printf("\nEscribe dos números: ");
    scanf("%f %f", &a, &b);

    printf(" a = %f", a);
    printf("\n");
    printf(" b = %f", b);
    printf("\n");

    r = a + b;
    printf("a + b = %f", r);
    printf("\n");

    r = a - b;
    printf("a - b = %f", r);
    printf("\n");

    r = a * b;
    printf("a x b = %f", r);
    printf("\n");

    r = a / b;
    printf("a / b = %f", r);
    printf("\n");

    system("pause");
    return 0;

}
```

Ahora, cuando se ejecute el programa, el usuario puede introducir los valores que quiera y el programa realizará los cálculos con dichos valores.

También podíamos ayudar más al usuario, haciendo que los mensajes fueran algo más claros de modo que sepa en todo momento en qué variables está introduciendo los valores. Por ejemplo utilizando las siguientes instrucciones :

```
    printf("\nEscribe el valor de a: ");
    scanf("%f", &a);

    printf("\nEscribe el valor de b: ");
    scanf("%f", &b);
```

De este modo el programa puede ser más útil para el usuario.

8 FORMA BÁSICA DEL PROGRAMA

Un programa, escrito en, C es una colección de funciones y variables. Las funciones contienen instrucciones y se utilizan para realizar cálculos y mover datos en memoria. Las variables se utilizan para almacenar resultados. Todo programa C debe contener una función denominada **main** y que constituye el cuerpo principal del programa. Esta es la primera función que se ejecuta y de esta se pueden invocar al resto de las funciones. Todas las variables deben declararse, es decir, se debe indicar expresamente el nombre de las variables que van a utilizarse en el programa.

Las partes principales de un programa C son:

1. Cabecera del programa.
2. Cuerpo principal del programa.
3. Resto del programa.

Lo único que es esencial es el cuerpo del programa.

8.1 El Cuerpo Principal

El cuerpo principal del un programa es la parte imprescindible de un programa y consta de la función main().

La evolución del lenguaje ha originado que existan diferentes modos de escribir los programas y aunque no existen grandes diferencias puede resultar confuso estudiar las diferentes variaciones que pueden darse.

El cuerpo de un programa antiguo no es igual al de uno moderno. Los programas modernos son un poco más evolucionados en el sentido de que son más seguros y menos propensos a errores.

Nosotros nos limitaremos a explicar la forma moderna de programar en C.

En el caso de que se tenga que estudiar algún programa antiguo se puede consultar la bibliografía.

A continuación se describe el cuerpo de un programa que realiza el menor trabajo posible:

```
int main(void)
{
    return 0;
}
```

Como podemos apreciar, después de indicar el tipo del valor que proporcionará la función y después del nombre de la función y de los argumentos, viene un bloque que constituye el conjunto de instrucciones que definen la función.

En realidad el programa se puede escribir todo seguido, es decir sin caracteres de nueva línea, lo que ocurre es que normalmente se utiliza esta forma con el fin de poder añadir y eliminar fácilmente partes del programa, es decir, se podría haber escrito del siguiente modo y seguiría siendo equivalente:

```
int main(void) { return 0; }
```

Incluso se podrían eliminar algunos (no todos) de los espacios en blanco y el programa seguiría siendo equivalente:

```
int main(void){return 0;}
```

Solamente son necesarios los dos espacios en blanco, el primero separa la palabra **int** de la palabra **main** y el segundo separa la palabra **return** del número **0**.

El C es un lenguaje de formato libre de manera que el programa se puede escribir de la forma que se considere más adecuada siempre que se mantenga el orden adecuado, es decir, las instrucciones pueden empezar en cualquier parte de la línea cosa que no ocurre en otros lenguajes de programación utilizados en ingeniería; por ejemplen en FORTRAN. Además, los espacios en blanco, tabuladores y caracteres de nueva línea pueden utilizarse libremente.

Los elementos de un programa se separan mediante los denominados separadores y estos pueden ser: un espacio en blanco, un carácter tabulador y un carácter de nueva línea. Donde se debe escribir un separador se pueden escribir cualquier número de separadores.

El cuerpo principal de un programa C consta de la función main(). La función main() el la primera función que se llama cuando se solicita la ejecución del programa. Desde la función principal se realiza la llamada al resto de funciones que realizan el trabajo del programa y se ejecutan las instrucciones propias del programa.

En el primer ejemplo que estamos considerando sólo tenemos una función, la función main(). La idea de función en informática es algo diferente que la idea de función en matemáticas, pero tienen muchas cosas en común. En este caso el nombre de la función es: main Es importante que sea así ya que todos los programas deben tener una función denominada **main** con el fin de que sea la primera que sea ejecutada cuando se solicita la ejecución del fichero que contiene el programa. Esto es así en la mayoría de los sistemas operativos y en concreto en los sistemas tipo windows y linux.

Además de tener un nombre, la función tiene unos parámetros que indican cómo se debe llamar a la función. En este caso se utiliza **void** Esta palabra significa que la función se llamará sin utilizar ningún argumento, es decir, el sistema operativo solicitará la ejecución de la función:

```
main()
```

y con esto se ejecutará el programa.

De todo esto nosotros no nos damos cuenta ya que normalmente cuando invocamos un programa lo único que hacemos es proporcionar al sistema operativo el nombre del fichero ejecutable correspondiente, es decir, si el programa se llama **media.c** y el ejecutable se llama **media.exe** nosotros solicitaremos la ejecución del fichero exe y es el sistema operativo el encargado de localizar la función main() y suministrarle los argumentos si los tuviera (ya digo que nosotros no proporcionaremos argumentos detrás del fichero exe)

Veamos qué es lo que hace este programa. Como puede observarse, el programa consta de

una única instrucción: la instrucción **return 0**;

Esta instrucción lo que hace es finalizar la ejecución de la función main() y proporcionar como resultado de su ejecución el valor 0.

Una vez que la función main() devuelve el valor 0, el sistema operativo lo recibe y finaliza la ejecución del programa.

El cuerpo principal de un programa consta normalmente de dos partes, la parte de declaración de variables y las instrucciones propiamente dichas. Veamos la función main() de uno de los ejemplos utilizados anteriormente :

```
int main(void)
{
    float a, b, r;                              // declaración de variables

    a = 3.2457;                                 // primera instrucción
    b = 1.2345;

    printf(" a = %f", a);
    printf("\n");
    printf(" b = %f", b);
    printf("\n");

    r = a + b;
    printf("a + b = %f", r);
    printf("\n");

    r = a - b;
    printf("a - b = %f", r);
    printf("\n");

    r = a * b;
    printf("a x b = %f", r);
    printf("\n");

    r = a / b;
    printf("a / b = %f", r);
    printf("\n");

    system("pause");
    return 0;                                   // última instrucción
}
```

8.2 La Cabecera del Programa

La cabecera del programa está pensada para incluir las declaraciones de las funciones que vamos a utilizar en el programa. Estas declaraciones hacen mención a funciones definidas y ya compiladas que se encuentran en la correspondiente biblioteca.

Normalmente, en la cabecera del programa se encuentran las includes necesarias para la utilización de las funciones de las bibliotecas del sistema que necesitemos en nuestro programa. Por ejemplo, para utilizar las funciones matemáticas se utiliza la include <math.h> y para utilizar las funciones de entrada y salida se utiliza la include <stdio.h>

A continuación mostramos una cabecera con la mayoría de las includes que vamos a utilizar a lo largo del curso (según las vayamos utilizando explicaremos algunas de las funciones que contienen) :

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <ctype.h>
#include <time.h>
```

El resto de las includes, que nosotros no utilizaremos, de la biblioteca standard son las siguientes :

```
#include <assert.h>   // establece comprobaciones en partes del programa
#include <stdarg.h>   // simplifica la utilización de argumentos del programa
#include <setjmp.h>   // permite el tratamiento de errores
#include <signal.h>   // permite establecer alarmas y captura de señales
#include <limits.h>   // establece los valores límites de los tipos utilizados
#include <localas.h>  // permite internacionalizar los programas
```

Además, en la cabecera se pueden declarar y definir constantes y variables globales que vayan a utilizarse en el resto del programa.

Por ejemplo, si tenemos que realizar cálculos trigonométricos y vamos a utilizar el valor de pi, podemos definir la constante pi del siguiente modo:

```
const float pi = 3.1416 ;
```

y si queremos mayor precisión utilizaremos

```
const  double pi = 3.1415926535898 ;
```

También es posible definir nuevas funciones que pensemos utilizar en el resto del programa.

8.3 El Resto del Programa

En resto del programa normalmente se utiliza para definir las funciones que vamos a utilizar, aunque también se pueden definir en la cabecera del programa.