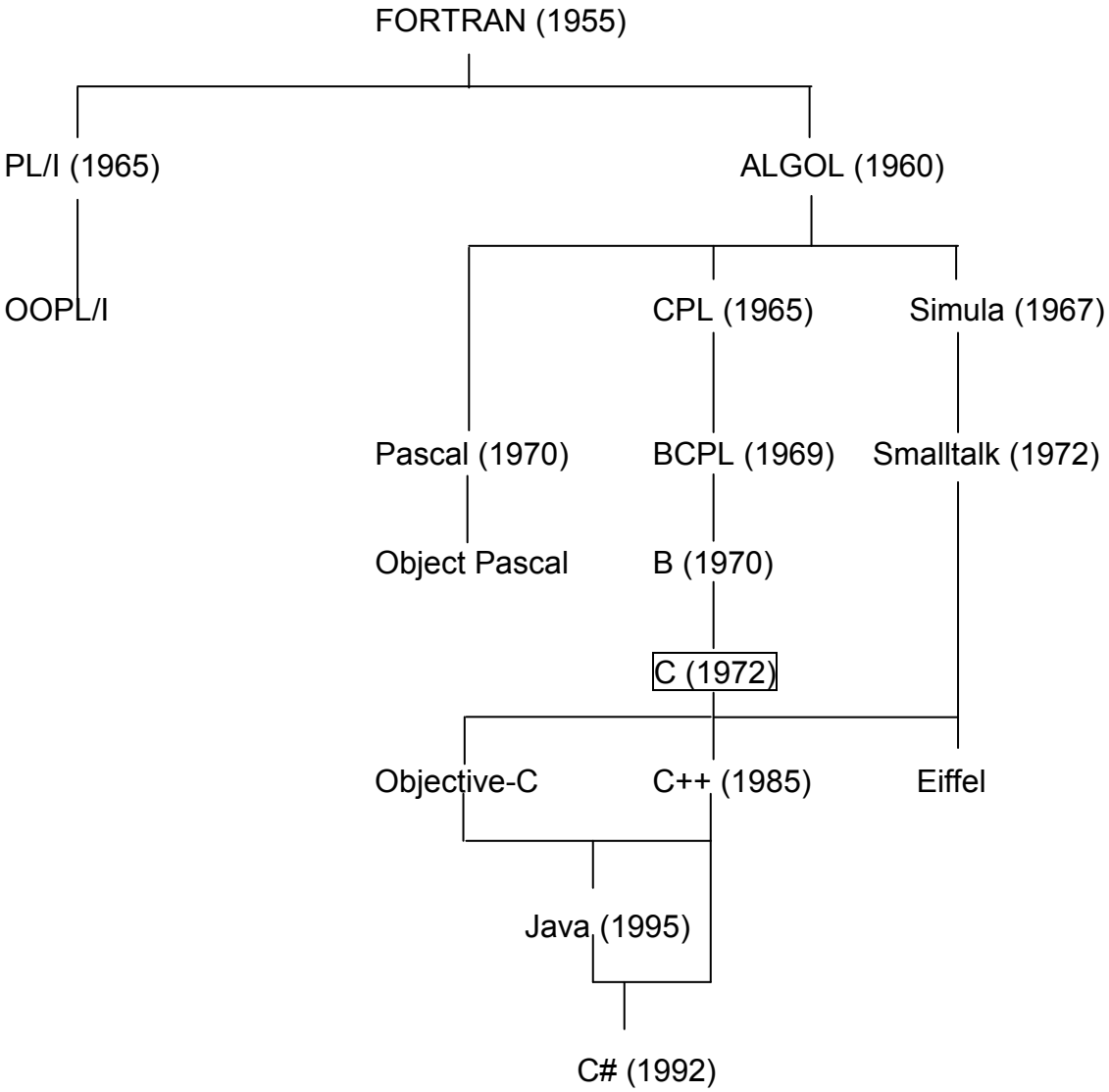
Introducción al lenguaje C

En este apunte se introducen los elementos principales de la programación en lenguaje C. Se cubre gran parte de las características del lenguaje, así como algunas funciones de las bibliotecas estándares, otros aspectos serán desarrollados en apuntes posteriores.

1.1 Marco histórico

El lenguaje C fue creado entre 1970 y 1972 por Brian Kernighan y Dennis Ritchie para escribir el código del sistema operativo UNIX.
Desde su nacimiento se fue implantando como el lenguaje de programación de sistemas favorito para muchos programadores, sobre todo por ser un lenguaje que conjugaba la abstracción de los lenguajes de alto nivel con la eficiencia del lenguaje máquina. Los programadores de sistemas que trabajaban sobre MS-DOS y Macintosh también utilizaban C, con lo cual la mayoría de las aplicaciones de sistema para microordenadores y para sistemas UNIX están escritas en este lenguaje.
A mediados de los ochenta el C se convierte en un estándar internacional ISO. Este estándar incluye tanto la definición del lenguaje como una enorme biblioteca de funciones para entrada/salida, tratamiento de textos, matemáticas, etc.
A mediados de los ochenta se crea el C++, extensión de C orientada a objetos. El C++ se convierte en estándar ISO en 1998. En el momento actual, el lenguaje C no va a modificarse más. Será el C++ el que incorporará nuevos cambios.

El lenguaje C (1972) proviene del lenguaje BCPL (1969), cuya influencia le llega a través del lenguaje B (1970). El siguiente árbol muestra los lenguajes antecesores y sus predecesores más destacados.



1.2 Características

Aunque las características del lenguaje solo se comprenderán a posteriori, cuando se lo conozca y emplee, aquí mencionamos:

- Orientado a la programación de sistemas,
- Es de un nivel intermedio entre la programación de bajo y alto nivel,
- Es altamente transportable,
- Es muy flexible,
- Genera código muy eficiente,
- Es muy expresivo (se pueden realizar muchas funciones escribiendo pocas líneas de código),
- Es muy poco modular,
- Hace pocas comprobaciones,
- Da poca disciplina al programador,
- Es difícil leer código escrito por otras personas.

1.3 Fases de desarrollo de un programa en C

El preprocesador

Transforma el programa fuente, convirtiéndolo en otro archivo fuente "preprocesado". Las transformaciones incluyen:

- Eliminar los comentarios.
- Incluir en el archivo fuente el contenido de los archivos declarados con:

  #include <archivos> (los archivos están en la carpeta **INCLUDE)**

  #include "archivos" (los archivos están en la carpeta del programa)

  a estos archivos se les suele llamar **cabeceras**, y hacen referencia a librerías de funciones preprogramadas que se desean emplear en el programa a desarrollar.

- Sustituir en el archivo fuente las macros declaradas con:

  #define.

  Esta directiva permite definir constantes (ej. #define CIEN 100).

El compilador

Convierte el archivo fuente entregado por el preprocesador en un archivo en lenguaje máquina: **archivo objeto** (extensión .obj).
Algunos compiladores pasan por una fase intermedia en lenguaje ensamblador.

El enlazador

Un fichero objeto es código máquina, pero no se puede ejecutar, porque le falta código que se encuentra en otros archivos binarios.
El **enlazador** genera el ejecutable binario, a partir del contenido de los ficheros objetos y de las **bibliotecas**.
Las bibliotecas contienen el código de funciones precompiladas, a las que el archivo fuente llama (por ejemplo **printf**).

1.4 Ejemplo de programa en C

Veamos unos ejemplos simples como introducción:

Ejemplo 1:

```
#include <stdio.h>

main()
{
        /* Escribe un mensaje */

        printf ("Hola, mundo\n");
}
```

Ejemplo 2:

```
/*ingresa y muestra el nombre*/
#include<stdio.h>

void main()
{
    char nombre[20];
    printf("\nComo te llamas? ");
    scanf("%s",  nombre);
    printf("\nHola %s!", nombre);
}
```

Explicación:

/* */	cometario
#include	directiva para incluir bibliotecas de funciones
stdio.h	biblioteca externa que contiene las funciones de entrada y salida
main	función principal, es la primera en ejecutarse
()	sin parámetros
void	la función main no devuelve resultado
{ }	definición o cuerpo de la función
char	definición de variable tipo caracter,
nombre[20]	array de 20 datos char
" "	constante literal o string
printf	función externa para salida de datos. Para salida de un caracter existe la función putchar().
\n	directiva de impresión (o secuencia de escape) indica nueva línea
scanf	función externa de entrada de datos. (Si se emplean tipos básicos como parámetro de funciones externas, los nombres de las variables se preceden con el operador &) Para entrada de un caracter existe la función getchar().
%s	formato de entrada o salida: string. Cada indicador de formato corresponde al 2º, 3º, etc. parámetro de la misma función

1.5 Bibliotecas estándares

El núcleo del lenguaje C es muy simple. Carece de tipos y servicios que forman parte de otros lenguajes. No tiene tipo booleano, ni manejo de cadenas, ni manejo de memoria dinámica.
No obstante, el estándar de C define un conjunto de **bibliotecas** de funciones, que necesariamente vienen con todo entorno de compilación de C y que satisfacen estos servicios elementales.
Las interfaces de estos servicios vienen definidas en unos **archivos cabeceras** (*header files*). El nombre de estos ficheros suele terminar en **.h**
Algunos de los servicios proporcionados por las bibliotecas estándares son:

- entrada y salida de datos (stdio.h)
- manejo de cadenas (string.h)
- memoria dinámica (stdlib.h)
- rutinas matemáticas (math.h)

posteriormente estudiaremos las funciones que proveen estas bibliotecas y otras más.

1.6 Componentes del lenguaje C

C sigue el paradigma de la programación estructurada:

Algoritmos + estructuras de datos = programas.

Un programa en C puede tener diversos componentes que serán estudiados más adelante, como:

Estructuras de datos

- literales
- tipos básicos (todos numéricos)
- arrays o vectores
- tipos estructura (struct, union)
- tipos enumerados
- punteros

Construcciones algorítmicas

- construcciones condicionales (if,switch)
- construcciones iterativas(while,for,do...while)
- subrutinas (funciones)

Además de lo anterior, el C tiene **otros elementos**:

- comentarios
- inclusión de ficheros
- macros
- compilación condicional

El preprocesador es quien normalmente se encarga de interpretar estas construcciones.

Sintaxis: C **distingue mayúsculas de minúsculas**. Las **palabras reservadas se escriben en minúsculas**. Por convención las mayúsculas se suelen reservar para los identificadores de las constantes.

1.7 Estructura de un archivo fuente

Un archivo fuente en lenguaje C tendrá esta estructura típica:

```
#include <biblioteca1.h>
#include <biblioteca2.h>

... declaraciones de funciones ...

... definiciones (cuerpos de funciones) ...
... declaraciones de variables globales ...
```

```
main()
{
... cuerpo del main ...
}

... otras definiciones de funciones ...
```

Las declaraciones y definiciones se pueden hacer en cualquier orden, aunque es preferible declarar las funciones al principio del programa (por legibilidad).

main es simplemente una función más del programa, con la particularidad de que es el punto de entrada al programa.

1.8 Comentarios

En el C original, tienen la forma
/* cualquier texto */
Los comentarios se pueden extender varias líneas
No se pueden anidar comentarios (comentarios dentro de otros)

En C++ se usan también comentarios de una sola línea. La sintaxis es:
// cualquier texto
Todo lo que se escriba a partir de las dos barras es un comentario. El comentario termina con el final de la línea.

Ejemplos:

```
{
        /*      Esto es un comentario
                que ocupa varias líneas
        */

        // esto es un comentario de C++
        // y esto es otro comentario
}
```

2. Manipulación básica de datos

2.1 Literales

Literal: un dato escrito directamente (ej. **1234**, **"hola"**, etc.)

Nombre	**Descripción**	**Ejemplos**
Decimal	entero en base 10	**1234**
Hexadecimal	entero en base 16	**0x1234**
Octal	entero en base 8	**01234**
Carácter	byte en ASCII	**'A'**

Coma flotante	número real en coma flotante	**1.25** **3.456e6** **3.456e-6**
Cadena	texto literal	**"hola, mundo"**

- Los datos numéricos con un punto al final o parte decimal son tomados como float. Ej.: 2.0
- Los literales long se escriben con una L final, los unsigned con una U final y los unsigned long con UL final.
- Los literales de tipo char se escriben entre apóstrofes. Son enteros de 1 byte código ASCCI (u otro según la máquina).
- Las cadenas de caracteres se delimitan entre comillas y se almacenan en arrays de char teminados por el carácter \0.
- Los enteros que comienzan en 0 (cero) están en octal.
- Los enteros precedidos por 0x ó 0X están en hexadecimal.

2.2 Tipos básicos

Los datos en C han de tener un **tipo**. Las **variables** contienen datos, y se han de declarar del tipo adecuado a los valores que van a contener. Esto le permite al compilador reservar la cantidad de memoria necesaria para albergar el dato.

El C dispone de estos tipos básicos:

int	enteros (números enteros positivos y negativos)
char	caracteres (letras, signos)
float	números en coma flotante (números reales)
double	números en coma flotante de doble precisión
void	no-tipo (se emplea con punteros)

Todos estos tipos -salvo **void**- son tipos numéricos. Incluso el tipo **char** (código ASCII)**.**

Se pueden construir tipos de datos más elaborados a partir de estos tipos básicos:

- Vectores y matrices
- Punteros
- Tipos estructurados (registros)

2.3 Declaraciones de variables

Las **variables** se utilizan para guardar datos dentro del programa.
Hay que **declarar** las variables antes de usarlas.
Cada variable tiene un **tipo**.

Declaración:
tipo nombre **;**

Ejemplo:

```
int pepe;
```

Las variables globales se declaran justo antes de **main()**.
Las declaraciones de variables locales se colocan después de la llave de apertura que introduce cualquier sentencia compuesta, no solamente al comienzo de una función.

2.4 Rangos de valores y tipos modificados

Rango de los enteros
Una variable entera acepta valores positivos y negativos dentro de un rango determinado, que depende de la plataforma y del compilador (en pecés bajo MS-DOS suele estar entre -32768 y 32767; en Linux son enteros de 32 bits).

Existen modificaciones para el tipo **int**, para alterar el rango de valores sobre el que trabaja:

Modificador	**Significado**
short	entero corto (rango más pequeño)
long	entero largo (rango más amplio)
unsigned	entero sin signo (0..N)
signed	entero con signo (-N-1 .. +N)

tipo	**tamaño en bytes**	**rango**
int	2	-32.768 a 32767
short	2	-32.768 a 32767
long	4	-2.147.483.648 a 2.147.483.647
unsigned	2	0 a 65.535
unsigned short	2	0 a 65.535
unsigned long	4	0 a 4.294.967.296
char	1	-128 a 127
unisgned char	1	0 a 255
float	4	3.4x10^-38 a 3.4x10^38
double	8	1.7x10^-308 a 1.7x10^308
long double	10	3.4x10^-4.932 a 1.1x10^4.932

La palabra **int** se puede omitir en la declaración de la variable.
Los modificadores de tamaño (**short**, **long**) y de signo (**signed**, **unsigned**) se pueden combinar.
Por omisión, un entero es **signed** (en la práctica, esta palabra reservada casi nunca se emplea).
Ejemplos:

```
unsigned sin_signo;        /* entero sin signo */
```

```
long saldo_en_cuenta;     /* entero largo con signo */
unsigned long telefono;   /* entero largo sin signo */
```

Tipo char

El tipo **char** permite manejar caracteres (letras), aunque se trata de un tipo numérico.
Normalmente el rango de valores va de –128 a +127 (signed char), o bien de 0 a 255 (unsigned char).
Los literales de tipo carácter se pueden utilizar como números.

```
char caracter;
int entero;
main()
{
        caracter = 65; // valdría como una 'A'
        entero = 'A';  // valdría como un 65
}
```

2.5 Nombres de variables (identificadores)

Un **identificador** es un nombre que define a una variable, una función o un tipo de datos.
Un identificador válido ha de empezar por una letra o por el carácter de subrayado **_**, seguido de cualquier cantidad de letras, dígitos o subrayados.
Atención: Se distinguen mayúsculas de minúsculas.
No se pueden utilizar palabras reservadas como **int**, **char** o **while**.
Muchos compiladores no permiten letras acentuadas o eñes.

Ejemplos válidos:

```
char letra;
int Letra;
float CHAR;
int __variable__;
int cantidad_envases;
double precio123;
int __;
```

Ejemplos no válidos:

```
int 123var;       /* Empieza por dígitos */
char int;         /* Palabra reservada */
int una sola;     /* Contiene espacios */
int US$;          /* Contiene $ */
int var.nueva;    /* Contiene el punto /
int eñe;          /* Puede no funcionar */
```

2.6 Expresiones

Los datos se manipulan mediante **expresiones**, que sirven para calcular valores. En C hay varios **operadores** para construir expresiones.
Estos son los operadores elementales sobre números:

Operador	Significado
+	suma
-	resta
*	producto
/	división
%	módulo (resto de la división)

Una expresión combina varias operaciones y devuelve un valor.
Los operadores *****, **/** y **%** tienen precedencia sobre la suma y la resta.
Se pueden utilizar paréntesis para agrupar subexpresiones.

Ejemplos de expresiones:

```
1
2+2
4 + 6/2
(4+6) / 2
( 3*5 + 12 ) % 7
```

2.7 Asignaciones

La forma de dar valor a una variable es
***variable* = *expresión* ;**
Se le llama **asignación**.

También se puede dar valor a una variable en el mismo momento en que se declara (**inicialización**).
***tipo variable* = *expresión* ;**
Una variable que se declara sin inicializar contiene un valor indeterminado.

Ejemplo:

```
int valor1 = 0;     /* variable inicializada a cero */
int valor2;         /* variable no inicializada */

main()
{
        valor1 = 4 + 3;    /* asignación */
        valor2 = 5;        /* otra asignación */
}
```

2.8 Expresiones: uso de variables

Una expresión puede ser el nombre de una variable.
En ese caso, el resultado de la expresión es el valor de la variable.

Ejemplo:

```
int valor1 = 5;
int valor2 = 1;
```

```
main()
{
        valor2 = ( valor1 * 4 ) - valor2;
}
```

2.9 Operadores relacionales y booleanos

Hay operadores relacionales para evaluar condiciones:
En C no existe tipo booleano, así que el resultado de la expresión utiliza números enteros: si la condición es cierta, estas expresiones devuelven un 1; si no es cierta, devuelven un cero.

Operador	**Resultado**
A == B	1 si A es igual a B; 0 en caso contrario
A != B	1 si A es distinto de B
A > B	1 si A es mayor que B
A < B	1 si A es menor que B
A >= B	1 si A es mayor o igual que B

Para elaborar condiciones complejas, existen estos operadores lógicos:

Expresión	**Resultado**
E1 && E2	Cierta si E1 y E2 son ciertas (AND)
E1 || E2	Cierta si E1 y/o E2 son ciertas (OR)
! E	Cierta si E es falsa; falsa si E es cierta (NOT)

Se pueden agrupar expresiones booleanas con paréntesis.
Ejemplo:
(a>0 && a<10) || a==20
cierto si “a” está entre 1 y 9 (ambos inclusive), o vale 20.

2.10 Las asignaciones son expresiones

Una asignación es una expresión. Esto quiere decir que:
- devuelve un valor;
- una asignación puede incrustarse dentro de una expresión más compleja.
El valor devuelto por la asignación ***a=b*** es el resultado de evaluar ***b***.

Ejemplo:

```
        C =  20 - ( B = 2*(A=5)+4 ) ;
```

A valdrá **5** (por la expresión A=5)
B valdrá 2*(5)+4= **14**
C valdrá 20-(14)= **6**

En consecuencia, una asignación se puede colocar en cualquier sitio donde se puede emplear una expresión.

2.11 Operadores avanzados

Los operadores de **incremento, decremento** y **asignación compuesta** permiten modificar el contenido de una variable de forma eficiente y abreviada.

Operadores	Significado
A++, ++A	Incrementa en 1 el valor de A (A=A+1)
A--, --A	Disminuye en 1 el valor de A (A=A-1)
A+=x	A=A+x
A-=x	A=A-x
A*=x	A=A*x
A/=x	A=A/x

Operadores "pre" y "post" y valor devuelto

Si el operador **++** o **--** se coloca a la izquierda, se llama **preincremento** o **predecremento**, respectivamente. Si se coloca a la derecha, se llama **postincremento** o **postdecremento**. Cuando se escriben estas expresiones dentro de expresiones más complejas, el valor que se devuelve es:

- Operaciones "pre": El valor *nuevo* de la variable afectada
- Operaciones "post": el valor *anterior* de la variable afectada

Ejemplo:

```
x=1;
A = ++x;        // preincremento:
                // A valdrá 2, x valdrá 2
x=1;
A = x++;        // postincremento:
                // A valdrá 1, x valdrá 2
```

Las asignaciones compuestas devuelven el nuevo valor de la variable:
x=2; A=(x*=3)+1; // x valdrá 6, A valdrá 7

Operador condicional o triádico

Tiene la forma:
expresión ? expresión1 : expresión2
Se utiliza como un **if** dentro de expresiones.
Su resultado es: si ***expresión*** es no nula, se evalúa y devuelve ***expresión1***. Si ***expresión*** es nula, se evalúa y devuelve ***expresión2***.

Ejemplo:

```
minimo = ( x<y ? x : y );
```

El uso de este operador es superfluo, dado que el **if** ya resuelve la ejecución condicional. Es conveniente, si se emplea, utilizar paréntesis para evitar ambigüedades.

Operadores de aritmética de bits

Efectúan operaciones aritmético-lógicas (AND, OR, desplazamientos) sobre los bits de datos enteros.
- No se aplican a operandos de tipo float o double.
- Actúan sobre la representación binaria de los operandos. Estos operadores trabajan a nivel de bits, no hay que confundirlos con los operadores lógicos (como **&&** o **||**).

Son estos:

Operador	Operación
A & B	AND de los bits de A y B
A | B	OR de los bits de A y B
A ^ B	XOR (O exclusivo) de los bits de A y B
A >> B	desplazamiento a la derecha B posiciones de los bits de A
A << B	desplazamiento a la izquierda B posiciones de los bits de A
~A	negación (NOT) de los bits de A

También existen operadores abreviados, como
A >>=B equivale a: A = A>>B
A &= B equivale a: A = A&=B

Precedencia y Asociatividad para los operadores vistos hasta aquí:

Precedencia	**Operador**	**Asociatividad**
1	() []	izquierda a derecha
2	!,~,++, --, - (monario)	derecha a izquierda
3	*,/,%	izquierda a derecha
4	+,-	"
5	<<, >>	"
6	>,>=,<,<=	"
7	==, !=	"
8	&	"
9	^	"
10	|	"
11	&&	"
12	||	"
13	? :	derecha a izquierda
14	=, +=, -=, etc.	"

2.12 Desbordamientos y redondeos

Desbordamientos (overflows)

En lenguaje C *no* se detectan desbordamientos. Si el resultado de una expresión está fuera del rango, el valor resultante es erróneo, pero no se interrumpe el programa ni se señaliza de ninguna forma.

Ejemplo:

```
/* supongamos enteros de 16 bits */
/* su rango va de –32768 a +32767 */
int x, y;
main()
{
        x = 30000;
        y = x + 3000;      /* ¡¡ y valdrá –32536 !! */
}
```

Redondeos

Si una expresión entera da un resultado fraccionario, se redondea al entero más cercano a cero (redondeo inferior).
Esto ocurre aunque el destino de la expresión sea una variable en coma flotante.

Ejemplo:

```
x = 13 / 3;  // x valdrá 4
```

Números en coma flotante

Los números en coma flotante son necesarios para trabajar con fracciones o con números de un rango mayor que los enteros.

```
float x = 123.456;
float y = 10000000.0;
```

2.13 Conversión de tipo

Conversión automática de tipos:

- Cuando los operandos de un operador aritmético son de distintos tipos, C homogeneiza ambos al tipo más general.
- Todo char en una expresión aritmética se convierte en int.
- Los valores float de una expresión se convierten en double porque toda la aritmética de punto flotante se realiza en doble precisión.
- Las conversiones también tienen lugar en las asignaciones: el valor de la parte derecha se convierte al tipo de la parte izquierda, según el caso se trunca la parte fraccionaria o los bits de orden superior o double se convierte a float mediante redondeo.
- Las reglas de conversión de tipos se aplican también a los argumentos de una función.

Conversión forzada de tipo (casting):

Se puede cambiar el tipo de una expresión de esta forma:
(*nuevo_tipo*) *expresión*

Por ejemplo, para forzar a que una división de enteros se realice en coma flotante, podemos escribir:

```
int x=5,y=3;
float f;
f = (float)x/y;
```

En este ejemplo, el valor de x, que es entero, se transforma a **float**. Así la división se realizará en coma flotante.

3. Entrada y salida de datos

3.1 Salida por pantalla: printf

La función **printf** se utiliza según este formato:

```
printf ( “cadena de formato”, arg1, arg2, ... argN );
```

En la **cadena de formato** aparecen:

- el texto que se desea imprimir
- caracteres especiales o **secuencias de escape**
- indicaciones del formato de los argumentos

Los argumentos son expresiones cualesquiera.
Para usar printf, hay que escribir al principio del programa la directiva
#include <stdio.h>

3.2 Formatos de printf (básicos)

%f	real con signo
%d	enteros decimal con signo
%u	entero decimal sin signo
%ld	entero long
%o	enteros en octal sin signo
%x	enteros en hexadecimal sin signo
%e	real con signo en notación científica con caracter e
%E	real con signo en notación científica con caracter E
%g	real con signo en formato f o e, el que resulte más compacto
%G	real con signo en formato f o E, el que resulte más compacto
%c	un sólo caracter.
%s	cadena de caracteres.
%%	escribir %

Ejemplos:

```
int una = 1234;
char otra = ‘h’;
main()
{
        printf( “una vale %d; otra vale %c\n”,
                        una, otra );
}
```

Secuencias de escape

Son en realidad un sólo carácter (en ASCII) aunque se representan en el programa con dos caracteres:

\n nueva línea
\t tabulador
\b retroceso
\' imprime '
\" imprime "
\\ imprime \
\a sonido

3.3 Formatos de printf para números reales

Se puede modificar el formato de salida, indicando cuantos decimales llevará el número, si se rellena de ceros por la izquierda, etc.

La estructura (casi) completa de un formato de printf es

% <Nº de dígitos>.<precisión> < tipo>

Ejemplo: printf(“ %6.2f”, x) imprimirá el valor de la variable x abarcando 6 espacios: 1 para el punto decimal, 3 para la parte entera y 2 para la parte decimal.

3.4 Entrada de datos: scanf

Se pueden recoger datos desde el teclado con la función **scanf**.
Sintaxis:

scanf (*formato*, & *arg1*, & *arg2*, ...);

En ***formato*** se especifica qué tipo de datos se quieren leer. Se utiliza la misma descripción de formato que en **printf**. También hay que incluir la cabecera **<stdio.h>**

Ejemplo:

```
int x,y;
...
scanf ( “%d %d”, &x, &y );
```

Notas:
Si no se anteponen los *ampersands* (**&**), el resultado puede ser desastroso (el porqué se explicará más adelante)
En **scanf** sólo van descripciones de formato, nunca texto normal. Si se quiere escribir antes un texto, hay que utilizar **printf**.

4. Construcciones algorítmicas

En C existen estas construcciones para implementar algoritmos:

- Sentencias simples y sentencias compuestas (con las llaves).

- Construcciones condicionales:

  if (*expresión*) *sentencia* [**else *sentencia***]
  switch (*expresión*) { *caso1 caso2 ... casoN* }

- Construcciones iterativas:

  while (*expresión*) *sentencia*
  do *sentencia* while (*expresión*);
  for (*expresión*; *expresión*; *expresión*) *sentencia*

- Instrucciones de control de flujo: **break, continue** y **goto**.

- Subprogramas: funciones.

4.1 Sentencias (statements)

Una sentencia es un fragmento de código.
Una **sentencia simple** es una expresión terminada en punto y coma.

```
sentencia_simple;
```

Una **sentencia compuesta** es una serie de sentencias entre llaves.

```
// sentencia compuesta: varias sentencias entre llaves.
        {
                sentencia
                sentencia
                ...
        }
```

Ejemplos:

```
/* sentencia simple */
x = y * 5 + sqrt(z);

        /* sentencia compuesta con llaves */
{
        a = b;
        b = x + 1;
        printf ( “hay %d productos”, num_prod );
}

        /* sentencias compuestas dentro de otras */
{
        { x=1; y=2; }
        { z=0; printf(“hola\n”); }
}
```

4.2 Sentencia if

La construcción **if** sirve para ejecutar código sólo si una condición es cierta:

```
if  ( condición )
   sentencia
```

La **condición** es una expresión de cualquier clase.

- Si el resultado de la expresión es CERO, se considera una condición FALSA.
- Si el resultado de la expresión NO ES CERO, se considera una condición CIERTA.

Ejemplo:

```
int x = 1;
main()
{
  if ( x == 1 )
          printf (“la variable x vale uno\n”);
  if ( x>=2 && x<=10 )
          printf (“x está entre 2 y 10\n”);
}
```

4.3 Construcción else

Con la construcción **else** se pueden definir acciones para cuando la condición del **if** sea falsa. La sintaxis es

```
if ( condición )
   sentencia
else
   sentencia
```

Ejemplo:

```
if ( x==1 )
          printf (“la variable x vale uno\n”);
else
          printf (“x es distinta de uno\n”);
```

4.4 Bucle while

Para ejecutar el mismo código varias veces, se puede utilizar:

```
while ( condición )
          sentencia
```

La ***sentencia*** se ejecuta una y otra vez mientras la ***condición*** sea cierta.

Ejemplos:

```
main()
{
        int x=1;
        while ( x < 100 )
        {
                printf("Línea número %d\n",x);
                x++;
        }
}
```

Ejemplo usando el operador de predecremento:

```
main()
{
        int x=10;
        while ( --x )
        {
                printf("una línea\n");
        }
}
```

En cada iteración se decrementa la variable **x** y se comprueba el valor devuelto por **--x**. Cuando esta expresión devuelva un cero, se abandonará el bucle. Esto ocurre después de la iteración en la que **x** vale uno.

```
/*convierte celcius a fahrenheit e imprime tabla*/

#include<stdio.h>
#define INICIO 0
#define FINAL 300
#define PASO 20

main()
{
    float fahr, celcius;
    fahr = INICIO;
    while (fahr <= FINAL)
    {
        celsius = (5.0 / 9.0) * (fahr - 32.0);
        printf("%4.0f %6.1f\n", fahr, celsius);
        fahr += PASO
    }
}
```

4.5 Bucle for

También se pueden ejecutar bucles con **for**, según esta sintaxis:

```
for (expresión_inicial;condición;expresión_de_paso)
        sentencia
```

La ***expresión_inicial*** se ejecuta antes de entrar en el bucle.
Si la ***condición*** es cierta, se ejecuta ***sentencia*** y después ***expresión_de_paso***.
Luego se vuelve a evaluar la ***condición***, y así se ejecuta la sentencia una y otra vez hasta que la condición sea falsa.

El bucle **for** es (casi) equivalente a

```
expresión_inicial;
while ( condición )
{
   sentencia
   expresión_de_paso;
}
```

Ejemplo típico de uso:

```
int i;
...
for ( i=0; i<10; i++ )
   printf (“%d “, i );
```

Ejemplo:

```
/*convierte celcius a fahrenheit e imprime tabla*/

#include<stdio.h>
#define INICIO 0
#define FINAL 300
#define PASO 20

main()
{
    float fahr;
    for (fahr = INICIO; fahr <= FINAL; fahr += PASO)
        printf("%4f %6.1f\n", fahr, (5.0 / 9.0) + (fahr - 32));
}
```

4.6 Bucle for: omisión de expresiones

Las tres expresiones del bucle for se pueden omitir, con el siguiente resultado.

Se omite	**Resultado**
expresión_inicial	no se hace nada antes del bucle
condición	la condición es siempre cierta (1)
expresión_de_paso	no se hace nada tras cada iteración

Ejemplos:

```
for ( ; resultado!=-1 ; ) { ... }

for ( ; ; ) { /* Bucle infinito */ }
```

4.7 Bucle do...while

Parecido al bucle **while**, pero iterando al menos una vez.

Sintaxis:

```
do {
   sentencia
} while ( condición );
```

La ***sentencia*** se ejecuta al menos la primera vez; luego, mientras la ***condición*** sea cierta, se itera la sentencia.

Ejemplo de uso de las estructuras de control:

```
/* cuenta cada digito, espacios en blanco u otros caracteres */
#include <stdio.h>
#define FIN '#'

main()
{
    int c, i, nblanco, notros;

    int ndigito[10];
    nblanco = notros = 0;

    for (i=0; i<10; ++i)
        ndigito[i] = 0;
    while ((c = getchar()) != FIN)
        if (c >= '1' && c <= '9')
            ++ndigito[c - '0'];
                    else if (c == ' ' || c == '\n' || c == '\t')
            ++nblanco;
        else

            ++notros;
    printf("digitos =\n");
    for (i = 0; i < 10; ++i)
        printf(" %d\n", ndigito[i]);
    printf("\nespacios en blanco = %d, otros = % d\n", nblanco, notros);
}
```

4.8 Control de bucles: break y continue

break y **continue** permiten controlar la ejecución de bucles **while**, **for** y **do...while**

Se puede salir de un bucle directamente con **break**.
continue va directamente al final del bucle y emprende una nueva iteración.

```
while (...)
{
  ...
  ...
   break;
  ...
}

while (...)
{
  ...
  ...
  continue;
  ...
}
```

4.9 Instrucción goto

Sirve para saltar incondicionalmente a un punto cualquiera del programa. La sintaxis es

goto *etiqueta*;

etiqueta es un identificador que indica el punto al que queremos saltar. La etiqueta se define colocándola en el punto adecuado seguida de dos puntos.
Sólo se puede saltar a una etiqueta que se encuentre en la misma función donde se invoca a **goto**.

Ejemplo:

```
parriba:        // declaración de etiqueta
        ...
        // salto directo a una etiqueta
        if (error) goto pabajo;
        ...
        if (repetir) goto parriba;
pabajo:         // declaración de etiqueta
```

El uso de la sentencia GOTO NO ESTA PERMITIDO en la programación estructurada.

4.10 Construcción switch

Se utiliza para ejecutar acciones diferentes según el valor de una expresión.
switch evalúa la expresión entera entre y compara su valor con todos los casos. Cada opción debe ser etiquetada con un entero, constante de caracter o expresión constante. Si algún case

corresponde al valor de la expresión en el switch, la ejecución comenzará en la proposición etiquetada por ese switch. La opción etiquetada default se ejecuta si no se satisface ninguna de las otras. Los case y default pueden aparecer en cualquier orden. Las opciones deben ser todas diferentes.
La proposición break hace que se produzca una salida inmediata de la instrucción switch.

Ejemplo de sintaxis:

```
switch ( expresión )
{
        case valor1:
          ... sentenciasA...
        break;

        case valor2:
          ... sentenciasB ...
        break;

        case valor3:
        case valor4:
          ... sentenciasC ...
        break;

        default:
          ... sentenciasD ...
}
```

Las sentenciasA se ejecutarán si ***expresión*** adquiere el ***valor1***.
Las sentenciasB se ejecutarán si adquiere el ***valor2***.
Las sentenciasC se ejecutarán si adquiere el ***valor3*** o el ***valor4***, indistintamente.
Cualquier otro valor de ***expresión*** conduce a la ejecución de las sentenciasD; eso viene indicado por la palabra reservada **default**.

Ejemplo de switch:

```
int opcion;
printf ( "Escriba 1 si desea continuar; 2 si desea terminar: " );
scanf ( "%d", &opcion );
  switch ( opcion )
  {
            case 1:
                    printf ("Vale, continúo\n");
              break;

            case 2:
                    salir = 1;
              break;

            default:
                      printf ("opción no reconocida\n");
```

```
  }
```

Ejemplo de switch combinado con las otras estructuras:

```
/* cuenta cada digito, espacios en blanco u otros caracteres
#include <stdio.h>
#define FIN '#'
main()
{
    int c, i, nblanco, notros;
    int ndigito[10];
    nblanco = notros = 0;

    for (i=0; i<10; ++i)
        ndigito[i] = 0;
    while ((c = getchar()) != FIN)
        switch (c)
        {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                ndigito[c - '0']++;
                break;
            case ' ':
            case '\n':
            case '\t':

                nblanco++;
            default:
                notros++;
                break;
        }
    printf("digitos =");
    for (i = 0; i < 10; ++i)
        printf(" %d", ndigito[i]);
    printf("\nespacios en blanco = %d, otros = % d\n", nblanco,
          notros);
}
```

4.11 Precauciones con if y bucles

Asignaciones en los if y los bucles

Hemos visto que una asignación es una expresión. Por ello se puede colocar dentro de cualquier construcción **if**, **while** o similar:

```
if ( variable = valor ) { ... }
```

Este uso muchas veces es erróneo, porque casi siempre pretendemos escribir:

```
if ( variable == valor ) { ... }
```

Pero como es correcto, el compilador no abortará si se encuentra estas construcciones (de todas formas, muchos compiladores emiten una advertencia si encuentran asignaciones dentro de **if**s).

Bucles for

Aunque el C lo permite, es conveniente no modificar la variable contadora dentro del bucle.

5. Tipos de datos

En este apartado se verán estructuras de datos más complejas que las descritas hasta ahora. Se hablará de:

- Vectores (Arrays) y matrices
- Cadenas de caracteres (*strings*)
- Tipos estructurados (struct, unión)
- Tipos enumerados
- Uniones

También se presentarán algunas conceptos sobre el uso de identificadores, como el ámbito y la existencia.

5.1 Vectores y matrices (arrays)

Se pueden crear variables que sean conjuntos de elementos del mismo tipo (vectores o matrices).
Sintaxis:

***tipo nombre_del_vector* [*dimensión*] ;**

Ejemplo:

```
int vector [5] ;      /* Crea un vector de cinco enteros */
```

Los elementos de un vector empiezan en cero y terminan en ***dimensión*** - 1.
Para acceder al elemento ***i*** de un vector, se utiliza la expresión
***vector* [*i*]**

Múltiples dimensiones

Se pueden declarar matrices de dos o más dimensiones, según esta sintaxis:

***tipo matriz* [*dimensión1*] [*dimensión2*] ... ;**

Ejemplo:

```
int matriz [3][8] ;
```

Se accede a los elementos con esta expresión:

```
matriz [i][j]
```

NOTA: la expresión **matriz[i,j]** no es válida, pero es una expresión correcta en C y no dará error de compilación (equivale a haber escrito **matriz[j]**).

Precauciones con los vectores

- El compilador de C reconoce la expresión vector[i,j], pero es un error.
- El C numera los elementos de un vector desde CERO.
- El C no detecta índices fuera de rango.
- Si **A** y **B** son vectores, la expresión **A = B** es ilegal.

5.2 Cadenas de caracteres

En C no existe un tipo predefinido para manipular cadenas de caracteres (*strings*). Sin embargo, el estándar de C define algunas funciones de biblioteca para tratamiento de cadenas.

La forma de declarar una cadena en C es mediante un vector de caracteres:

```
char hola [5];
```

Toda cadena ha de terminar con el carácter especial \0.
El C no tiene otra manera de detectar el final de una cadena.

5.3 Literales e inicialización de cadenas

Los literales tipo cadena son de la forma

```
"texto entre comillas"
```

Al declarar una vector de caracteres, se le puede inicializar con un literal:

```
char texto [4] = "abc";
```

Pero NO se puede hacer una asignación de ese tipo en una sentencia:

```
texto = "xyz";      /* ERROR */
```

En su lugar, hay que emplear ciertas funciones de biblioteca.
Tres formas equivalentes de inicializar una cadena:

```
char hola [5] = { 'h', 'o', 'l', 'a', 0 };
```

```
char hola [5] = "hola";

main()
{
        char hola [5];
        hola[0] = 'h';
        hola[1] = 'o';
        hola[2] = 'l';
        hola[3] = 'a';
        hola[4] = 0;
}
```

Obsérvese que una cadena de *N* elementos es capaz de almacenar un texto de *N*-1 caracteres (el último siempre ha de ser un cero).

No importa que un vector de caracteres contenga una cadena de menos letras, el carácter cero marca el final de la cadena.

Lo que sí es un error peligroso (y además no lo detecta siempre el compilador) es intentar asignarle a un vector de caracteres una cadena de mayor tamaño.
Hay que cuidar mucho que las cadenas queden dentro del espacio reservado para ellas.

5.4 Visualización de cadenas

La función **printf** admite formato de cadenas, con **%s**

```
char cadena[80];
...
printf ( "El texto es %s\n", cadena );
```

Para usar **printf**, debería incluirse la cabecera **<stdio.h>**

El formato **%s** admite modificadores. Por ejemplo:
%20s Texto a la derecha, ocupando siempre 20 caracteres
%-15s Texto alineado a la izquierda, ocupando 15 caracteres

Para imprimir sólo la cadena, seguida de un salto de línea, se puede emplear **puts** (también pertenece a **<stdio.h>**):

```
puts (cadena);
```

5.5 Biblioteca de manejo de cadenas (string.h)

La biblioteca **<string.h>** contiene un conjunto de funciones para manipular cadenas: copiar, cambiar caracteres, comparar cadenas, etc. Aquí mencionamos las más usadas, pero en un apunte posterior se ampliará su estudio:
Las funciones más elementales son:
strcpy (*c1*, *c2*); Copia ***c2*** en ***c1***
strcat (*c1*, *c2*); Añade ***c2*** al final de ***c1***

int strlen (*cadena*); Devuelve la longitud de la ***cadena***
int strcmp (*c1*, *c2*); Devuelve cero si ***c1*** es igual a ***c2***; no-cero en caso contrario

Para trabajar con estas funciones, al comienzo del programa hay que escribir
#include <string.h>

Ejemplo:

```
#include <stdio.h>
#include <string.h>
char completo [80];
char nombre[32] = "Pedro";
char apellidos [32] = "Medario Arenas";
main()
{
  /* Construye el nombre completo */
  strcpy ( completo, nombre );       /* completo <- "Pedro" */
  strcat ( completo, " ");           /* completo <- "Pedro " */
  strcat ( completo, apellidos );    /* completo <- "Pedro Medario Arenas" */
  printf ( "El nombre completo es %s\n", completo );
}
```

5.6 Lectura de cadenas

En teoría, podría utilizarse la opción **%s** de **scanf**, pero tiene algunos inconvenientes.

Una mejor alternativa es emplear **gets**, que también viene en **stdio.h**

```
#include <stdio.h>
main ()
{
        char nombre [80];
        printf ( "¿Cuál es su nombre? " );
        gets ( nombre );
        printf ( "Parece que su nombre es %s\n", nombre );
}
```

NOTA: **gets** no comprueba el tamaño de la cadena. Si el texto tecleado tuviera más de 80 caracteres, se destruirían posiciones de memoria incorrectas.
Para leer caracteres hasta un máximo, hay que usar **fgets**:

```
        fgets ( nombre, 80, stdin );
```

(el identificador **stdin** se refiere a la *entrada estándar* , normalmente el teclado. Está definido en **<stdio.h>**)

5.7 Tipo estructura

Se pueden definir tipos compuestos de varios elementos o **campos** de tipos más simples.

La sintaxis es:

```
struct nombre_del_tipo
{
        campo1;
        campo2;
        ...
        campoN;
};
```

Las variables de ese tipo se declaran así:

struct *nombre_de_tipo variable*;

y para acceder a un campo de una variable estructura,
se utiliza esta sintaxis:

variable.campo

5.8 Ejemplo de tipo estructura

```
struct Persona
{
        char nombre [40];
        char apellidos [80];
        long telefono;
        long dni;
        char sexo;
};
struct Persona usuario;

main ()
{
        usuario.dni = 43742809;
        strcpy ( usuario.apellidos, "Santana Melián" );
        strcpy ( usuario.nombre, "Jacinto Lerante" );
        usuario.telefono = 908330033;
        usuario.sexo = 'V';
}
```

Si hay campos del mismo tipo, se pueden declarar en la misma línea, separándolos por comas.
Se pueden declarar variables de tipo estructurado a la vez que se declara el tipo.

```
struct T
{

  int campo1, campo2, campo3; /* varios campos */

} v1, v2; /* declaración de variables de tipo struct T */

...
```

```
v1.campo1 = 33;
v2.campo3 = 45;
```

También se permite omitir el tipo de la estructura (*estructura anónima*):

```
struct {
  char nombre [8];
  char extension [3];
} nombre_fichero;
```

En este caso, no se puede crear nuevas variables de ese tipo. No se recomienda usar este tipo de declaraciones.

Inicialización de estructuras

Una variable de tipo **struct** se puede inicializar en su declaración, escribiendo entre llaves los valores de sus campos en el mismo orden en que se declararon estos.

```
struct Persona
{
        char nombre [32];
        char apellidos [64];
        unsigned edad;
};

struct Persona variable =
{ "Javier",  "Tocerrado", 32 };
```

5.9 Definición de tipos: typedef

Se puede dar un nombre nuevo a cualquier tipo de datos mediante **typedef**.
La sintaxis es

```
typedef declaración;
```

donde ***declaración*** tiene la forma de una declaración de variable,
sólo que se está definiendo un tipo de datos.

```
typedef long pareja [2];
```

define un tipo **pareja** que se puede usar en declaraciones de variables:

```
pareja p;
```

es equivalente a

```
long p [2];
```

Ejemplos de typedef con estructuras

```
typedef struct Persona PERSONA;

PERSONA dato; /* igual que struct Persona dato; */
```

Un uso típico es la redefinición de tipos estructurados:

```
typedef struct      /* estructura anónima */
{
  char nombre[80];
  char sexo;
  int edad;
} Persona;                    /* se declara el tipo Persona */
...
Persona p;
...
p.edad = 44;
```

5.10 Tipos enumerados: enum

Con la construcción **enum** se pueden definir tipos de datos enteros que tengan un rango limitado de valores, y darle un nombre a cada uno de los posibles valores.

```
enum dia_de_la_semana
{
        lunes, martes, miercoles, jueves, viernes,   sabado, domingo
};
...
enum dia_de_la_semana hoy;
...
hoy = sabado;
```

Los valores definidos en **enum** son constantes enteras que se pueden usar en cualquier punto del programa, usando un operador de moldeo (ver ejemplo).
Se empiezan a numerar de cero en adelante (en el ejemplo, **lunes** vale cero, **martes** vale uno, etc.)

```
int dia = (int)sabado;        /* dia = 5 */
```

5.11 Valores de la lista en enum

Se puede dar un valor inicial a la lista de valores dados en **enum**:

```
enum dia
{
        lunes=1, martes, miercoles, jueves, viernes,          sabado, domingo
};
```

En este caso los valores van del 1 al 7.

También se pueden dar valores individuales:

```
enum codigo_postal
{
        LasPalmas=35, Madrid=28, Barcelona=8
};
```

5.12 Tipo Union

El tipo estructurado **union** es similar al **struct**, salvo que en las mismas posiciones de memoria almacena todos sus campos.

```
struct rectangulo
{
  int ancho;
  int largo;
};

union todo_en_uno
{
  char cadena [8];
  int entero;
  double real;
};
```

En el caso del **struct**, cada campo ocupa unas posiciones de memoria diferentes. En la **union** los tres campos comparten las mismas posiciones de memoria.
Eso significa que si se altera un campo de una **union** se afecta a todos los demás campos.
La **union** sólo tiene sentido si se sabe que sólo se va a emplear un campo en cada ocasión.

5.13 Combinaciones de tipos

Los tipos estructurados y los vectores pueden combinarse entre sí: dentro de un **struct** se pueden declarar campos que sean **struct**, vectores, **enum**, etc.; se puede declarar un vector de **struct**, etc.

```
struct Producto
{
  unsigned identificador;
  char nombre [40];
  long disponible;
};


struct Producto catalogo [1000];
...
catalogo[133].disponible = 2467;

/* Ejemplo de inicialización */
```

```
struct Producto dos_productos [2] =
{
  { 12, "Brutopía", 28000 },
  { 33, "Papas el Canario", 35000 }
};
```

6. Funciones

Las funciones son porciones de código que devuelven un valor.
Permiten descomponer el programa en módulos que se llaman entre ellos.
En C no existe diferencia entre funciones y procedimientos: a todas las subrutinas se las llama *funciones*.

La **definición** de una función especifica lo siguiente:

- nombre de la función
- número de **argumentos** que lleva y tipo de cada uno de ellos
- tipo de datos que devuelve
- Cuerpo de la función (el código que ejecuta)

Sintaxis:
***tipo nombre* (*arg1*, *arg2*, *...*)**
{
... cuerpo ...
}

Cada argumento (parámetros formales) se especifica como en una declaración de variable.
El cuerpo de la función debería contener una sentencia donde se devuelve el resultado de la función, que se hace de esta forma:
return *expresión*;

La función devolverá el resultado de la ***expresión***

6.1 Ejemplo de función

Función que devuelve la suma de dos enteros.

```
int suma ( int a, int b )
{
        return a+b;
}
```

6.2 Llamadas a función

Para llamar a una función, se escribe su nombre y entre paréntesis los valores que se deseen dar a los argumentos:

```
función ( expr1, expr2, ... )
```

Cada expresión se evalúa y su resultado se pasa como argumento (parámetros reales) a la función. Las expresiones han de tener el mismo tipo del argumento correspondiente, o al menos un tipo compatible.

```
x = suma ( 1, a+5 );                    /* correcto */
y = suma ( “hola”, 5 );      /* arg. 1 incorrecto */
```

Una llamada a función es una expresión, con todo lo que ello implica.

No es necesario recoger en una variable el valor devuelto por la función.
(Por ejemplo, **printf** y **scanf** son funciones que devuelven un entero).

```
main()
{
    int i;
    printf ("n\t pot 2\t pot 3\n");
    for (i = 0; i < 10; ++i)
        printf ("%d\t %d\t\ %d\n",i,power(2,i),power(3,i));
         printf ("%d\t %d\t\ %d\n",i,power(2,i),power(3,i));
}

 power (int x, int n) /*potencias de x*/
{
     int i,p;
     p = 1;
     for (i = 1; i <= n; ++i)
          p = p * x;
     return (p);
}
```

6.3 Funciones sin argumentos

Se declaran con **void** entre paréntesis (sólo en C).

```
int fecha (void)
{ ... }
```

Se las llama así:

```
dato = fecha();
```

es decir, siempre hay que escribir los paréntesis aunque no haya argumentos.
En C++ se declaran sólo con los paréntesis, sin el **void**.

6.4 Funciones que no devuelven valor (Procedimientos)

En C no se distingue entre procedimientos y funciones. Un procedimiento sería una función que no devuelve ningún valor, lo que se define de esta forma:

```
void función ( arg1, arg2, ... )
{ ... }
```

A estas funciones se las suele llamar *funciones void* .

No es obligatorio que una *función void* contenga sentencias **return**, puesto que no devuelve nada. No obstante, si se desea salir prematuramente de una función void, se puede usar return:

```
void rutina ()
{
  ...
  if (error) return;
  ...
}
```

6.5 Ámbitos y existencia de variables y tipos

Las variables y los tipos de datos se pueden declarar en dos puntos:

- Al principio de un bloque (después del "abre llave" **{**)
- Fuera de bloques

Las variables declaradas fuera de bloques se llaman **globales**.
Las variables en el interior de un bloque son **locales** a ese bloque.

Ámbito

Una variable global se puede utilizar en todo el programa, desde el punto en que se declara.

Una variable local se puede utilizar desde el punto en que se declara hasta el final del bloque donde se declaró.

Si se declara una variable local con el mismo nombre de otra variable accesible en el bloque actual, la nueva variable **oculta** a la otra variable.

Existencia

Una variable global existe permanentemente mientras el programa se ejecuta.

Una variable local sólo existe mientras se está ejecutando código del bloque donde se ha declarado.
Así, una variable local a una función se crea cuando se llama a la función y desaparece cuando la función retorna

Ejemplo de ámbitos

```
int global;
double area ( double base, double altura )
{
        double temp = base*altura;
        return temp;
}

main()
{
        int local;
```

```
        ...
        while (1)
        {
          int mas_local;
          ...
          {
                int mas_local_todavia;
                int local;
                ...
          }
        }
}
```

Ámbitos y existencia de tipos de datos

Los ámbitos y existencia de los tipos son similares a los de las variables.
Los tipos declarados fuera de bloques son globales al programa.
Los tipos declarados dentro de bloques son locales a ese bloque.

```
struct pepe { int a,b,c; }; /* ámbito global */

int funcion (void)
{
struct local { int x1,x2; }; /* ámbito local */
struct pepe una;
struct local otra;
}

struct pepe variable;                   /* correcto */
struct local otra;  /* incorrecto: no estamos
 en el ámbito de “local” */
```

6.6 Variables static

Se pueden definir variables que tengan ámbito local pero existencia permanente.
Para declararlas se utiliza la palabra **static**.

static *declaración*;

Por ejemplo, si se declara una variable **static** dentro de una función, aunque la función retorne la variable permanece con el último valor que se asignó:

```
int contador (void)
{
        static int cuenta = 0;
        return cuenta++;
}
```

Esta función devuelve un número que se va incrementando cada vez que se la llama. La variable **cuenta** es local a la función **contador**, pero no desaparece con la salida de la función.

Inicialización

La inicialización de una variable *static* se realiza una sola vez, al comienzo de la ejecución del programa. Por eso el ejemplo anterior funciona (*cuenta* se inicializa a cero una sola vez).
En las variables externas y estáticas la inicialización se realiza una sola vez.
Para las automáticas y registro se efectúa cada vez que se entra en la función o bloque.
En ausencia de inicialización explícita, las variables externas y estáticas tendrán inicialmente valor cero. Las automáticas y registros tendrán valor indefinido o "basura".

Los arreglos automáticos no pueden ser inicializados. Los arreglos externos y estáticos se pueden inicializar haciendo que a la declaración le siga una lista de valores entre llave, separados por comas.

6.7 Argumentos de entrada o paso de parámetros por valor

Una función en C no puede alterar las variables pasadas como parámetros reales. Por eso se dice que los parámetros se pasan *por valor*. Los parámetros formales son variables locales inicializadas con el valor con que se llamó a la función. Estas variables permanecen en de memoria sólo durante el tiempo de ejecución de la función. Los parámetros formales actúan como argumentos de entrada.

```
#include <stdio.h>

/* función inútil */
void incrementa ( int variable )
{
        variable ++;
}
main()
{

        int x = 33;
        incrementa (x);

        /* x no resulta afectada, sigue valiendo 33 */

        printf ( "la variable x vale ahora %d\n", x );

}
```

6.8 Argumentos de entrada/salida o paso de parámetros por referencia a funciones

En C todos los parámetros se pasan por valor. Esto tiene en principio dos inconvenientes:

- No se pueden modificar variables pasadas como argumentos reales.
- Si se pasa como parámetro una estructura, se realiza un duplicado de ella, con lo que se pierde tiempo y memoria.

- Se logra anteponiendo el & al parámetro formal, en la definición de la función. Esto solo es aplicable solo en C++.

```
void incrementa_variable (int & var)
{
  var++;
}

main()
{
  int y = 1;
  incrementa_variable (y); /* y pasa a valer 2 */
}
```

Paso por referencia empleando punteros:

Otra forma de conseguir alterar una variable pasada como parámetro real, es decir para que actúe como argumento de salida, es pasar la dirección del mismo, lo cual se hace anteponiendo el **símbolo &** al identificador del parámetro real en la llamada a la función. En la definición de la función se emplea el símbolo * que indica que el parámetro formal es un **puntero** (tema que se desarrollará más adelante), lo cual significa que recibirá un valor que es una dirección de memoria. De este modo, las modificaciones que reciban los parámetros formales afectarán a los parámetros reales (que ahora son las misma localización de memoria) después de finalizada la ejecución de la función.
Esto aclara la razón de emplear el & en las llamadas a la función scanf.

```
void incrementa_variable (int* var)
{
  (*var)++;
}

main()
{
  int x = 1;
  incrementa_variable (&x);          /* x pasa a valer 2 */
}
```

En el ejemplo anterior, habia que poner paréntesis en **(*var)++** porque el operador **++** tiene más precedencia que la desreferencia (el asterisco). Entonces ***var++** sería como escribir ***(var++)**, que no sería lo que queremos.

Cuando aparece el nombre de un **arreglo** como argumento de una función, se pasa la dirección de comienzo del mismo. Por omisión los arreglos pasados como parámetros de funciones se pasan por referencia (sin necesidad de indicarlo). En tal caso la función **puede modificar los elementos del arreglo**.

6.9 Recursividad

En C no se pueden declarar funciones dentro de otras (funciones anidadas o locales). Todas las funciones son globales.

Se permite hacer llamadas recursivas:

```
float factorial (int n)
{
if (n<=1) return 1.0;
else return n*factorial(n-1);
}
```

Una función pude llamarse a sí misma. Cada invocación crea una nueva copia del código y de las variables automáticas. Por eso puede requerir demasiada memoria y es de ejecución lenta.

6.8 Declaraciones de funciones

Las funciones son siempre **globales**, esto es, no se permite declarar una función dentro de otra.
Las funciones son visibles sólo después de que se han declarado.

Se pueden **declarar** funciones, especificando sólo su formato, pero no su cuerpo:

```
int suma ( int a, int b );
```

lo anterior es una declaración de la función **suma**, que queda disponible para su uso, a pesar de no haber sido definido su cuerpo.

La declaración de una función de esta forma se llama **prototipo**.

Es buena práctica declarar al comienzo del programa los prototipos de las funciones que vamos a definir, incluyendo comentarios sobre su finalidad.

```
#include <stdio.h>
void printd(int n);
main()    /* convierte numero a cadena */
        {
            int n;
            printf("\nIngrese un numero: ");
            scanf("%d \n",&n);
            printd(n);
         }

void printd(int n)
         {
            int i;
            if(n < 0) {
                putchar('-');
                n = -n;}
            if((i = n/10) != 0)
                printd(i);
            putchar(n % 10 + '0');
         }
```