Universidad de Las Palmas de Gran Canaria

Introducción al lenguaje C

José Miguel Santos Espino

Escuela Universitaria de Informática
Facultad de Informática

Contenido

1. Introducción

En este documento 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.

1.1 Marco histórico

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 práctica totalidad de aplicaciones de sistema para microordenadores y para sistemas UNIX está escrita 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.

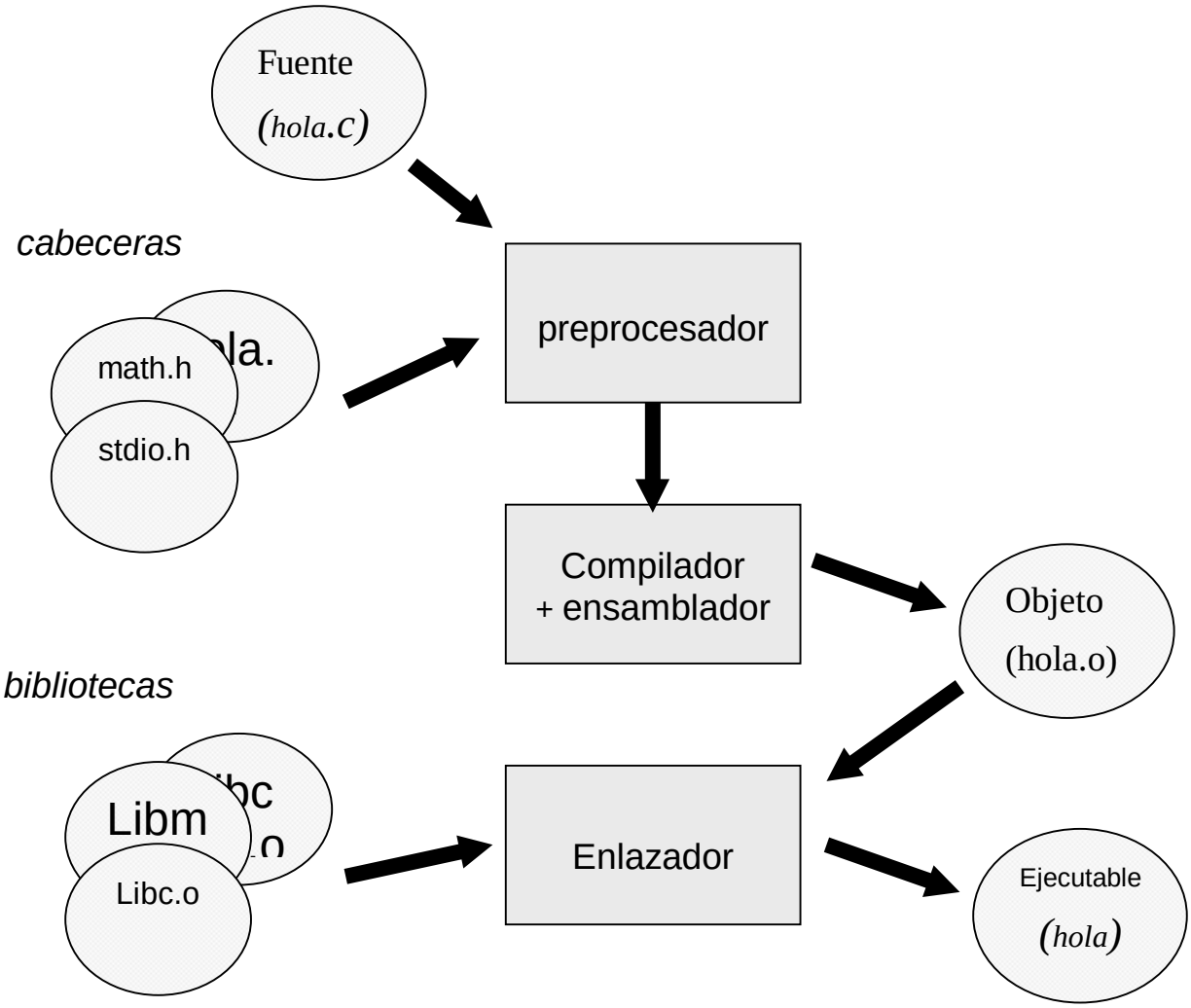
1.2 Características

- Orientado a la programación de sistemas

- 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 "predigerido". Las transformaciones incluyen:

- Eliminar los comentarios.
- Incluir en el fuente el contenido de los ficheros declarados con **`#include <fichero>`** (a estos ficheros se les suele llamar **cabeceras**)
- Sustituir en el fuente las macros declaradas con **`#define`** (ej. #define CIEN 100)

El compilador

Convierte el fuente entregado por el preprocesador en un archivo en lenguaje máquina: **fichero objeto.**

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

```
#include <stdio.h>

main()
{
     /* Escribe un mensaje */

     printf (“Hola, mundo\n”);
}
```

1.5 Bibliotecas estándares

El 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 **ficheros 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)

1.6 Componentes del lenguaje C

Sigue el paradigma de la programación estructurada:

Algoritmos+estructuras de datos = programas.

Estructuras de datos

- literales
- tipos básicos (todos numéricos)
- tipos enumerados
- tipos estructurados (struct, union)
- punteros y vectores

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.

1.7 Estructura de un fichero fuente

Un fichero 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

- Literales
- Tipos básicos
- Declaraciones de variables
- Rangos de valores y tipos modificados
- Nombres de variables (identificadores)
- Expresiones
- Asignaciones
- Operadores booleanos
- Operadores avanzados

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 c.f.	**`1.25`** **`3.456e6`** **`3.456e-6`**
Cadena	texto literal	**`"hola, mundo"`**

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.

El C dispone de estos tipos básicos:

`int`	enteros (números enteros positivos y negativos)
`char`	caracteres (letras)
`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.`**

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()`**.

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)

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 carácer de subrayado _, seguido de cualquier cantidad de letras, dígitos o subrayados.

OJO: 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 booleanos

Hay operadores 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:

Expresión	Resultado
E1 && E2	Cierta si E1 y E2 son ciertas (AND)
E1 || E2	Cierta si E1 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: a) devuelve un valor; b) 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
```

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 –29769 !! */
}
```

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

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.

2.14 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.

3. Entrada y salida de datos

- Función printf
- Función scanf

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:

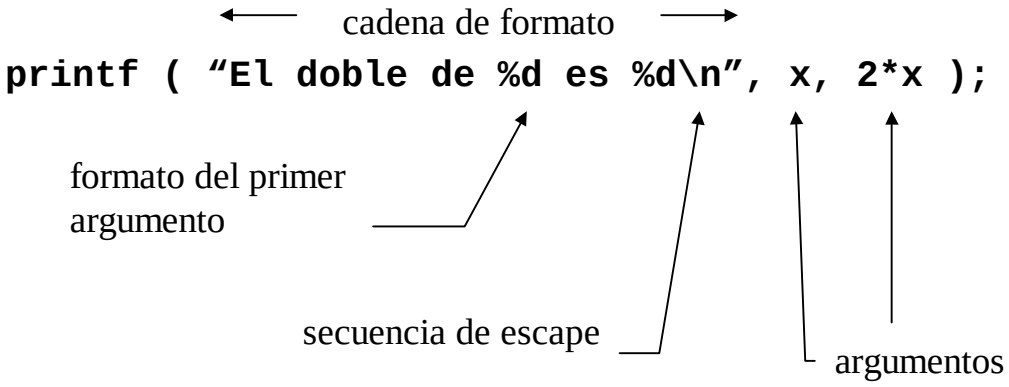
- el texto que se desea imprimir
- caracteres especiales ⇒ **secuencias de escape**
- indicaciones del formato de los argumentos

Los argumentos son expresiones cualesquiera.

Para usar printf, hay que escribir al principio del programa la directiva

`#include <stdio.h>`

Ejemplo:



3.2 Formatos de printf (básico)

%d	Entero decimal
%u	Entero decimal con signo
%x	Entero hexadecimal
%c	Carácter
%f	Coma flotante (**`float`**)
%lf	Coma flotante (**`double`**)

Ejemplos:

```
int una = 1234;
char otra = 'h';
main()
{
    printf( "una vale %d; otra vale %c\n",
            una, otra );
}
```

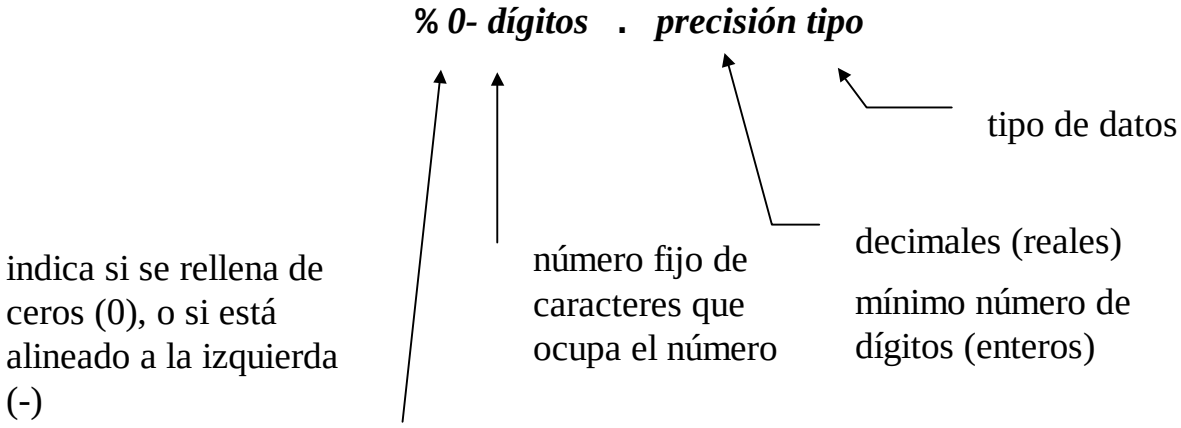
Secuencias de escape

\n	Salto de línea
\t	Tabulación
\a	Sonido

3.3 Formatos de printf (avanzado)

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



Formato	Significado
`%5d`	Entero decimal alineado a la izquierda, ocupando cinco espacios
`%04u`	Entero sin signo ocupando cuatro espacios, y rellenando de ceros a la izquierda si hace falta
`%.2lf`	Número real (doble precisión) con dos y sólo dos decimales
`%5.3d`	Entero ocupando cinco espacios; aparecen tres cifras como mínimo (se rellena de ceros)

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.

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.

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

***sentencia_simple*;**

// 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.

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 );
```

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.

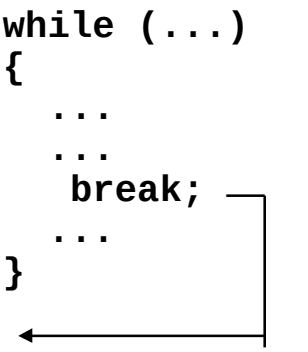
Se recomienda no utilizar esta construcción, porque las construcciones **`while`** y **`for`** bastan para diseñar cualquier clase de bucles. Muy pocos programas hoy día usan esta construccíón.

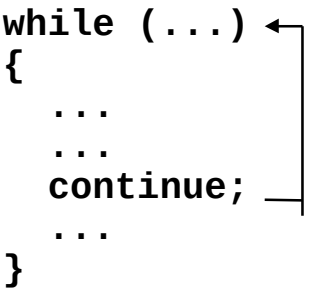
4.8 Control de bucles: break y continue

`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.





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
```

4.10 Construcción switch

Se utiliza para ejecutar acciones diferentes según el valor de una expresión.

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");

   }
```

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. 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 se especifica como en una declaración de variable.

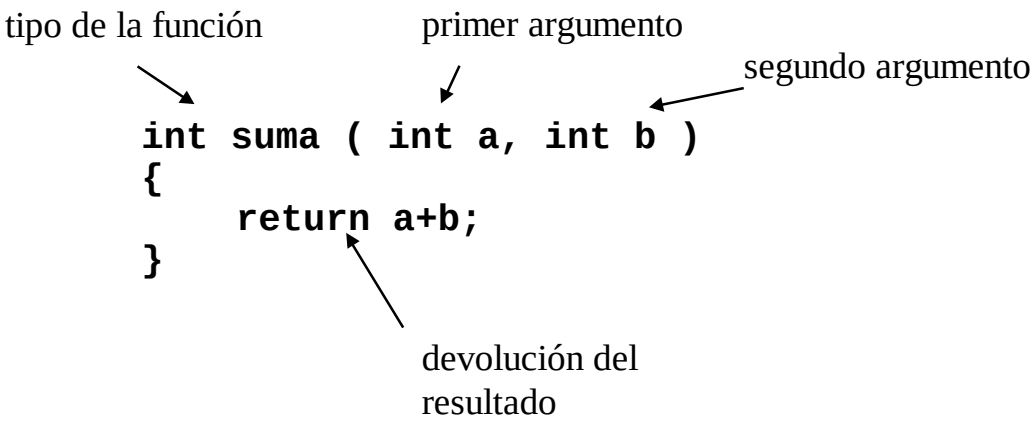
El cuerpo de la función debería contener una sentencia donde se devuelve el resultado de la función, que se hace de esta forma:

```
return expresión;
```

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

5.1 Ejemplo de función

Función que devuelve la suma de dos enteros.



5.2 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 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).

5.3 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`**.

5.4 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;
  ...
}
```

5.5 Argumentos de entrada/salida o paso por referencia

Una función en C no puede alterar las variables pasadas como parámetros. Los parámetros se pasan *por valor*.

```
#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 );

}
```

Para conseguir alterar una variable pasada como parámetro, hay que recurrir a los **punteros**. (Se verá más adelante).

5.6 Otras consideraciones

Funciones anidadas

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

Recursividad

Se permite hacer llamadas recursivas:

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

6. Tipos de datos

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

- Cadenas de caracteres (*strings*)
- Tipos estructurados
- Tipos enumerados
- Uniones

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

6.1 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 (cero).

El C no tiene otra manera de detectar el final de una cadena.

6.2 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.

6.3 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);
```

6.4 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.

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 );

}
```

6.5 Lectura de cadenas

En teoría, podría utilizarse la opción **%s** de **scanf**, pero
CASI NUNCA FUNCIONA.

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>**)

6.6 Tipos estructurados

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 estructurada,
se utiliza esta sintaxis:

```
variable.campo
```

6.7 Ejemplo de tipo estructurado

```
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 };
```

6.8 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;
```

6.9 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 */
```

6.10 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:

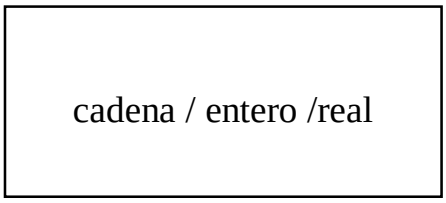
```
enum codigo_postal
{
     LasPalmas=35, Madrid=28, Barcelona=8
};
```

6.11 Uniones

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.

6.12 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 }
};
```

6.13 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" */
```

6.14 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.

NOTA: 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).

6.15 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.

7. Punteros

El tipo de datos más característico del C son los punteros. Un puntero contiene un valor que es la dirección en memoria de un dato de cierto tipo. Los punteros se emplean en C para muchas cosas, por ejemplo recorrer vectores, manipular estructuras creadas dinámicamente, pasar parámetros por referencia a funciones, etc.

Cuando se declara una variable, se reserva un espacio en la memoria para almacenar el valor de la variable.

Ese espacio en memoria tiene una **dirección**.

Un **puntero** es una dirección dentro de la memoria, o sea, un apuntador a donde se encuentra una variable.

7.1 Operaciones básicas

Declaración

Los punteros se declaran con un asterisco, de esta forma:

***tipo* * *variable*;**

Por ejemplo:

```
int* puntero;
```

Se dice que la variable **`puntero`** es un *puntero a enteros* (apunta a un entero).

Asignación

El valor que puede adquirir un puntero es, por ejemplo, la dirección de una variable.

El operador **&** devuelve la dirección de una variable:

```
puntero = &variable;
```

Desreferencia de un puntero

Se puede alterar la variable a la que apunta un puntero.
Para ello se emplea el operador de **desreferencia**, que es el asterisco:

```
*puntero = 45;
```

En este caso, se está introduciendo un 45 en la posición de memoria a la que apunta **`puntero`**.

7.2 Ejemplo de uso

```
{
   int* puntero;
   int variable;

   puntero = &variable;
   *puntero = 33;  /* mismo efecto que variable=33 */
}
```

Varios punteros pueden apuntar a la misma variable:

```
int* puntero1;
int* puntero2;
int var;

puntero1 = &var;
puntero2 = &var;
*puntero1 = 50;      /* mismo efecto que var=50  */
var = *puntero2 + 13;/* var=50+13 */
```

7.3 Otros usos

Declaración múltiple de punteros

Si en una misma declaración de variables aparecen varios punteros, hay que escribir el asterisco a la izquierda de cada uno de ellos:

```
int *puntero1, var, *puntero2;
```

Se declaran dos punteros a enteros (**`puntero1`** y **`puntero2`**) y un entero (**`var`**).

El puntero nulo

El **puntero nulo** es un valor especial de puntero que no apunta a ninguna parte. Su valor es cero.

En **`<stdio.h>`** se define la constante **`NULL`** para representar el puntero nulo.

7.4 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
- Si se pasa como parámetro una estructura, se realiza un duplicado de ella, con lo que se pierde tiempo y memoria

Sin embargo, se puede pasar un puntero como argumento a una función. El puntero no se puede alterar, pero sí el valor al que apunta:

```
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.

7.5 Precauciones con los punteros

Punteros no inicializados

Si se altera el valor al que apunta un puntero no inicializado, se estará modificando cualquier posición de la memoria.

```
main()
{
     int* puntero;
     *puntero = 1200; /* Se machaca una zona
                         cualquiera de la memoria */
}
```

Confusión de tipos

Un puntero a un tipo determinado puede apuntar a una variable de cualquier otro tipo. Aunque el compilador lo puede advertir, no es un error.

(Afortunadamente, esto no ocurre en C++).

```
main()
{
     int p;
     double numero;
     int* puntero;

     p = &numero; /* incorrecto,
                     pero el compilador no aborta */
     *p = 33;     /* Un desastre */
}
```

Punteros a variables locales fuera de ámbito

Si un puntero apunta a una variable local, cuando la variable desaparezca el puntero apuntará a una zona de memoria que se estará usando para otros fines. Si se desreferencia el puntero, los resultados son imprevisibles y a menudo catastróficos.

```
main()
{
     int* puntero;

     while (...)
     {
       int local;
       puntero = &local;
       /* 'puntero' apunta a una variable local */
       ...
       *puntero = 33;     /* correcto */
     }
     ...

     /* ahora 'puntero' apunta a una zona de memoria
        inválida */
     puntero = 50; /* catástrofe */
}
```

7.6 Aritmética de punteros

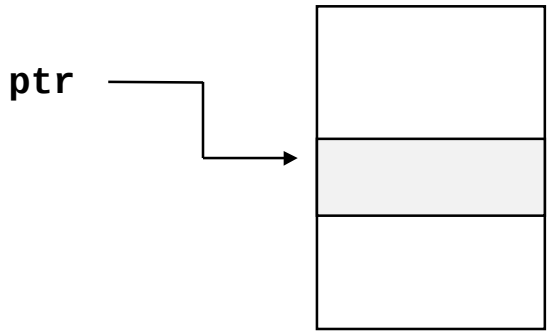
Un puntero apunta a una dirección de memoria.

El lenguaje C permite sumar o restar cantidades enteras al puntero, para que apunte a una dirección diferente: **aritmética de punteros**.

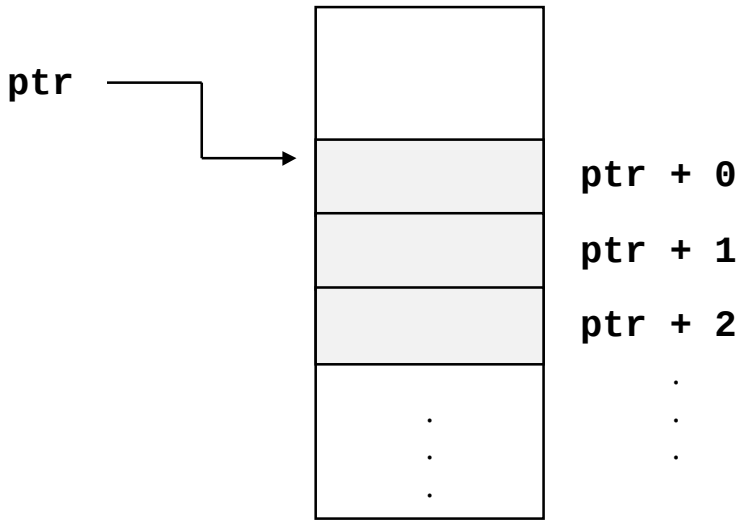
Consideremos un puntero a enteros:

```
int* ptr;
```

`ptr` apuntará a cierta dirección de memoria:



Pero también tendrán sentido las expresiones **`ptr+1`**, **`ptr+2`**, etc. La expresión **`ptr+`*k*** es un puntero que apunta a la dirección de **`ptr`** sumándole ***k*** veces el espacio ocupado por un elemento del tipo al que apunta (en este caso un **`int`**):



Ejemplo de aritmética de punteros

```
int vector [100];   /* un vector de enteros */
int *ptr;           /* un puntero a enteros */
...
ptr = &vector[0];   /* ptr apunta al principio del vector */
*ptr = 33;          /* igual que vector[0] = 33 */
*(ptr+1) = 44;      /* igual que vector[1] = 44 */
*(ptr+2) = 90;      /* igual que vector[2] = 90 */
```

La expresión que se suma al puntero ha de ser entera y no tiene por qué ser constante. Obsérvese que **`ptr`** es lo mismo que **`ptr+0`**.

La expresión sumada NO es el número de bytes que se suman a la dirección, es el número de elementos del tipo al que apunta el puntero:

```
/* Supongamos que un "char" ocupa 1 byte */
/* y que un "double" ocupa 8 bytes */

char* ptrchar;
double* ptrdouble;

...

*(ptrchar+3) = 33; /* la dirección es ptrchar + 3 bytes */

*(ptrdouble+3) = 33.0;
                   /* la dirección es ptrdouble + 24 bytes,
                      ya que cada double ocupa 8 bytes */
```

El compilador "sabe" cómo calcular la dirección según el tipo.

Aritmética de punteros (más)

A un puntero se le puede aplicar cualquier clase de operación de suma o resta (incluyendo los incrementos y decrementos).

```
/* Rellenar de unos los elementos del 10 al 20 */

int* ptr;           /* el puntero */
int vector [100];   /* el vector */
int i;              /* variable contadora */

ptr = &vector[0];   /* ptr apunta al origen del vector */
ptr+=10;            /* ptr apunta a vector[10] */

for ( i=0; i<=10; i++ )
{
  *ptr = 1;    /*   asigna 1 a la posición de memoria apuntada
                     por "ptr" */
  ptr++;       /*   ptr pasa al siguiente elemento */
}
```

7.7 Punteros y vectores

Si **`ptr`** es un puntero, la expresión

```
ptr[k]
```

es equivalente a

```
*(ptr+k)
```

con lo que se puede trabajar con un puntero como si se tratara de un vector:

```
int* ptr;
int vector [100];

ptr = &vector[10];

for ( i=0; i<=10; i++ )
  ptr[i] = 1;    /* equivalente a *(ptr+i) = 1 */
```

Un vector es en realidad un puntero constante. El nombre de un vector se puede utilizar como un puntero, al que se le puede aplicar la aritmética de punteros (salvo que no se puede alterar).

Por ello la expresión

```
ptr = vector;
```

es equivalente a

```
ptr = &vector[0];
```

7.8 Paso de vectores como parámetros a funciones

La aritmética de punteros permite trabajar con vectores pasados como parámetros a funciones en forma de punteros:

```
/*  Rellena de ceros los "n_elem"
    primeros elementos de "vector"
*/

void rellena_de_ceros ( int n_elem, int* vector )
{
  int i;
  for ( i=0; i<n_elem; i++ )
    *(vector++) = 0;    /* operador de post-incremento */
}

main()
{
  int ejemplo [300];
  int otro_vector [100];

  /* pasa la dirección del vector "ejemplo" */
  rellena_de_ceros ( 300, ejemplo );

  /* rellena los elems. del 150 al 199 */
  rellena_de_ceros ( 50, otro_vector+150 );
}
```

7.9 Punteros y estructuras

Un puntero puede apuntar a una estructura y acceder a sus campos:

```
struct Dato
{
  int campo1, campo2;
  char campo3 [30];
};

struct Dato x;
struct Dato *ptr;
...
ptr = &x;
(*ptr).campo1 = 33;
strcpy ( (*ptr).campo3, "hola" );
```

El operador ->

Para hacer menos incómodo el trabajo con punteros a estructuras, el C tiene el operador flecha **->** que se utiliza de esta forma:

```
ptr->campo
```

que es equivalente a escribir

```
(*ptr).campo
```

Así, el ejemplo anterior quedaría de esta forma:

```
ptr = &x;
ptr->campo1 = 33;
strcpy ( ptr->campo3, "hola" );
```

7.10 Memoria dinámica: malloc y free

Los variables y vectores en C ocupan un tamaño prefijado, no pueden variarlo durante la ejecución del programa.

Por medio de punteros se puede reservar o liberar memoria **dinámicamente**, es decir, según se necesite. Para ello existen varias funciones estándares, de la biblioteca **`<stdlib.h>`**

La función **`malloc`** sirve para solicitar un bloque de memoria del tamaño suministrado como parámetro. Devuelve un puntero a la zona de memoria concedida:

```
void* malloc ( unsigned numero_de_bytes );
```

El tamaño se especifica en bytes. Se garantiza que la zona de memoria concedida no está ocupada por ninguna otra variable ni otra zona devuelta por **`malloc`**.

Si **`malloc`** es incapaz de conceder el bloque (p.ej. no hay memoria suficiente), devuelve un puntero nulo.

Punteros void*

La función **`malloc`** devuelve un puntero inespecífico, que no apunta a un tipo de datos determinado. En C, estos punteros sin tipo se declaran como **`void*`**

Muchas funciones que devuelven direcciones de memoria utilizan los punteros *void**. Un puntero *void** puede convertirse a cualquier otra clase de puntero:

```
char* ptr = (char*)malloc(1000);
```

Operador sizeof

El problema de **`malloc`** es conocer cuántos bytes se quieren reservar. Si se quiere reservar una zona para diez enteros, habrá que multiplicar diez por el tamaño de un entero.

El tamaño en bytes de un elemento de tipo ***T*** se obtiene con la expresión

```
sizeof (T)
```

El tamaño de un **`char`** siempre es 1 (uno).

Función free

Cuando una zona de memoria reservada con **`malloc`** ya no se necesita, puede ser *liberada* mediante la función **`free`**.

```
void free (void* ptr);
```

`ptr` es un puntero de cualquier tipo que apunta a un área de memoria reservada previamente con **`malloc`**.

Si **`ptr`** apunta a una zona de memoria indebida, los efectos pueden ser desastrosos, igual que si se libera dos veces la misma zona.

Ejemplo de uso de malloc, free y sizeof

```
#include <stdlib.h>

int* ptr;    /* puntero a enteros */
int* ptr2;   /* otro puntero */

...

/* reserva hueco para 300 enteros */
ptr = (int*)malloc ( 300*sizeof(int) );
...
  ptr[33] = 15;       /* trabaja con el área de memoria */

  rellena_de_ceros (10,ptr);  /* otro ejemplo */

  ptr2 = ptr + 15;         /* asignación a otro puntero */

/* finalmente, libera la zona de memoria */
free(ptr);
```

Obsérvese que hay que convertir el puntero **`void*`** devuelto por **`malloc`** a un **`int*`**, para que no haya incompatibilidad de tipos.

7.11 Precauciones con la memoria dinámica

Si una zona de memoria reservada con **`malloc`** se pierde, no se puede recuperar ni se libera automáticamente (no se hace *recolección de basura*). El programador es el responsable de liberar las áreas de memoria cuando sea debido.

Si una zona de memoria liberada con **`free`** estaba siendo apuntada por otros punteros, esos otros punteros apuntarán a una zona ahora incorrecta.

7.12 Otras funciones de manejo de memoria dinámica

`void* calloc ( unsigned nbytes );`

Como **`malloc`**, pero rellena de ceros la zona de memoria.

`char* strdup ( char* cadena );`

Crea un duplicado de la ***cadena***. Se reservan **`strlen(`*cadena*`)+1`** bytes.

7.13 Punteros a funciones

Un puntero puede apuntar a código. La declaración es similar a esta:

```
void (*ptr_fun) (int,int);
```

En este caso se está declarando una variable, ***ptr_fun***, que apunta a una función "void" que admite dos parámetros enteros.

Para tomar la dirección de una función, se escribe su nombre sin paréntesis ni parámetros.

Para llamar a una función apuntada por un puntero, se usa el nombre del puntero como si fuera una función.

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

int resta (int a, int b)
{
  return a-b;
}

// declaramos un puntero a funciones con dos
parámetros
// enteros que devuelven un entero
int (*funcion) (int,int);


{
  ...
  funcion = suma;       // 'funcion' apunta a 'suma'
  x = funcion(4,3);  // x=suma(4,3)
  funcion = resta;   // 'funcion' apunta a 'resta'
  x = funcion(4,3);  // x=resta(4,3)
}
```

8. Operadores avanzados

El lenguaje C dispone de varias clases de operadores, algunos para operaciones a nivel de bits, otros para funciones menos convencionales.

8.1 Operadores de aritmética de bits

Efectúan operaciones aritmético-lógicas (AND, OR, desplazamientos) sobre los bits de datos enteros. Son estos:

A & B	AND de los bits de A y B
A | B	OR de los bits de A y B
A ^ B	XOR 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

Estos operadores trabajan a nivel de bits, no hay que confundirlos con los operadores lógicos (como **&&** o **||**).

También existen operadores abreviados, como

A >>=B	equivale a:	A = A>>B
A &= B	equivale a:	A = A&=B

8.2 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.

8.3 Operador coma

Varias expresiones se pueden agrupar separadas por comas:

expr1, expr2, expr3

El resultado es que se evalúan todas las expresiones y se devuelve como valor la de más a la derecha.

El único uso razonable de este operador es en sentencias **`for`**:

```
for ( x=0, y=0; x<100; x++, y++ )
{ ... }
```

9. Manejo de ficheros básico con stdio.h

La biblioteca STDIO contiene funciones para trabajar con ficheros: básicamente leer y escribir datos de diferentes formas.

Antes de trabajar con un fichero, hay que **abrirlo**. Cuando se abre un fichero, se devuelve un puntero a una estructura de tipo **`FILE`** (definido en STDIO). Esta estructura, llamada **descriptor de fichero**, servirá para manipular posteriormente el fichero.

Tras su uso, hay que **cerrar** el fichero.

Modelo de uso:

```
FILE* fd;        /* variable para apuntar al
descriptor de fichero */

...

/* abre el fichero en modo lectura */
fd = fopen ( "pepe.txt", "rt" );

    ... trabaja con el fichero ...

fclose (fd);     /* cierra el fichero */
```

Muchas de las funciones de STDIO para trabajar con ficheros comienzan por la letra "f" (fopen, fclose, fprintf, fwrite, etc.)

9.2 Abrir y cerrar un fichero

Abrir un fichero

```
fd = fopen ( nombre, modo );
```

Devuelve **`NULL`** si no se pudo abrir correctamente el fichero.

nombre es el nombre del fichero.

modo es una cadena donde se define el modo de apertura:

r	sólo lectura
w	escritura
a	apéndice (escribir desde el final)
+	(combinado con r,w ó a) lectura/escritura
t	fichero de texto
b	fichero binario

Ejemplo:

```
fd = fopen ( "c:\\ejemplos\\fichero.txt", "r+t" );
```

abre el fichero **`c:\ejemplos\fichero.txt`** en modo lectura/escritura y en modo texto. Obsérvese que las barras en el nombre del fichero tienen que escribirse duplicadas.

Un fichero abierto en modo texto convierte la combinación CR+LF en un carácter de salto de línea. Si es binario, CR+LF se consideran dos caracteres independientes.

Cerrar un fichero

```
fclose ( fd );
```

donde ***fd*** es un descriptor de fichero ya abierto.

9.3 Leer una cadena desde un fichero

`fgets (` ***cadena, N, fd*** **`);`**

Lee una línea de texto.

cadena es el vector de caracteres donde se depositará la línea.

N es el máximo número de caracteres para leer. Si la línea ocupa más de ***N*** caracteres, sólo se lee esa cantidad.

fd es el descriptor del fichero.

Cuando se abre un fichero, se comienza a leer desde la primera línea. Llamadas consecutivas a **`fgets`** van leyendo línea por línea. Si se cierra el fichero y se vuelve a abrir, se lee otra vez desde el principio.

Internamente existe un **puntero del fichero** que apunta a la posición relativa al fichero donde se va a leer el siguiente dato.

9.4 Escribir una cadena en un fichero

```
fputs ( cadena, fd );
```

cadena es el texto que se escribe. **`fputs`** incluye además un salto de línea.

fd es el descriptor del fichero.

Las líneas escritas se van añadiendo una detrás de otra. Se comienza por el principio del fichero (**`w`**) o por el final (**`a`**). También existe internamente un puntero de fichero para escritura.

```
fprintf ( fd, formato, arg1, arg2, ... );
```

Escribe un texto con formato en un fichero. La sintaxis es idéntica a **`printf`**, salvo que el primer argumento es un descriptor de fichero.

9.5 Detectar el final de fichero

```
x = feof ( fd );
```

Devuelve un 1 si se ha llegado al final del fichero. Un cero en caso contrario. Útil para saber cuándo dejar de leer información, por ejemplo.

9.6 Reposicionar el puntero del fichero

```
rewind ( fd );
```

Mueve el puntero de lectura/escritura del fichero al comienzo del mismo (rebobina).

```
fseek ( fd, posición, de_dónde );
```

Mueve el puntero del fichero a la ***posición*** indicada, relativa a un punto especificado en ***de_dónde***.

posición es de tipo **`long`**.

de_dónde puede tener los siguientes valores:

SEEK_SET	desde el origen del fichero
SEEK_CUR	desde la posición actual del puntero
SEEK_END	desde el final del fichero

Ejemplo

```
FILE* fich;
char cadena [16];

/* lee los últimos 15 caracteres del fichero */

fich = fopen ( "fichero.txt", "rt" );
fseek ( fd, -15L, SEEK_END);
fgets ( texto, 15, fich );
fclose (fich);
```

9.7 Flujos (*streams*) estándares

Los ficheros de donde se lee o escribe información con las rutinas de STDIO se llaman **flujos** o *streams*. Un flujo puede ser un fichero, pero también el teclado (se puede leer de él) o la pantalla (se puede escribir en ella).

Los llamados **flujos estándares** son estos:

`stdin`	entrada estándar (normalmente el teclado)
`stdout`	salida estándar (normalmente la pantalla)
`stderr`	error estándar (normalmente la pantalla)

Se pueden utilizar con las funciones de STDIO:

```
fgets ( cadena, 80, stdin );
fprintf (stderr,"Se ha producido un error" );
```

Los flujos estándares se pueden redirigir desde el MS-DOS.

9.8 Gestión de errores: errno

Muchas funciones de biblioteca devuelven un -1 o un NULL si ocurre un error. Esto es habitual en las funciones de tratamiento de ficheros.

La variable **`errno`** contiene un código de error relativo al último error producido. Para trabajar con **`errno`** hay que incluir el fichero ***errno.h***

En ***errno.h*** están definidos todos los códigos de error posibles.

Aparte, existe la función **`perror()`**, que da un mensaje (en inglés) sobre el error producido.

10. El preprocesador del C

En C existe un **preprocesador,** previo al compilador, que reconoce unas órdenes básicas para manipular constantes y macros, incluir varios ficheros en el fuente y dirigir la compilación posterior.

Este preprocesador tiene su propio lenguaje, muy sencillo, y es *independiente del lenguaje C*, aunque es absolutamente estándar.

IMPORTANTE: el preprocesador es independiente y ajeno al C, tan sólo es un editor de textos.

Todas las órdenes al preprocesador comienzan con un carácter **#** en la primera columna del texto.

10.1 Orden #define

La orden **`#define`** permite definir **macros**. Entre otras cosas, se utiliza para declarar constantes simbólicas:

```
#define TAM_VECTOR  100
...
struct Datos vector [TAM_VECTOR];
```

La sintaxis general de este uso de **`#define`** es

`#define` ***símbolo texto_sustituto***

símbolo puede ser cualquier identificador utilizable en C y ***texto_sustituto*** es cualquier ristra de caracteres, sin restricción.

El preprocesador trocea el fichero fuente en palabras. Cada vez que se encuentre con la palabra ***símbolo*** la sustituye por ***texto_sustituto***, sin tener en cuenta nada más.

Si en el ***texto_sustituto*** aparecen palabras que en realidad son macros, se sustituyen.

```
#define BEGIN    {
#define PROGRAM  main() BEGIN
#define END      }
#define BUCLE    for (i=0;i<N;i++)

...
PROGRAM
    int i;
    BUCLE
      BEGIN
        printf ( "paso número %d\n", i );
      END
END
```

Este ejemplo refleja un abuso del preprocesador, dado que el código no parece C y resulta poco legible.

Hay que tener cuidado con las expresiones en el texto sustituto:

```
#define DIEZ     10
#define VEINTE   20
#define TREINTA  DIEZ + VEINTE

main()
{
    printf ( "El resultado es: %d\n",
    TREINTA*TREINTA );       /* ¿Qué se obtiene? */
}
```

En el caso anterior, la solución era definir así la macro:

```
#define TREINTA  (DIEZ + VEINTE)
```

10.2 Macros con parámetros

La orden **#define** admite también pasar parámetros a las macros, para que se comporten de forma similar a las funciones.

La sintaxis es:

#define ***macro*(*arg1*,*arg2*,...)** ***texto_sustituto***

No puede haber espacios entre ***macro*** y el paréntesis. Los nombres entre paréntesis y separados por comas se tratan como si fueran argumentos pasados a una función. Todas las apariciones de ***arg1*** en el ***texto_sustituto*** serán reemplazadas en el texto definitivo por el argumento realmente escrito en ***arg1*** cuando se llama a la macro en el fichero fuente.

Con unos ejemplos:

```
#define incrementa(x)    (x)++
#define max(a,b)     (a)>(b)?(a):(b)
#define ERROR(n) printf("Error: %s\n", mensaje[n])
#define ERR_FICH 0
#define ERR_DIV  1

char* mensaje [] =
{
    "no se pudo abrir el fichero",
    "división por cero"
}

...
int x = 0;
incrementa(x);   /* x vale ahora 1 */
x=max(3,5);      /* x vale ahora 5 */

if ( !fopen("pepe.txt","r") )
  ERROR (ERR_FICH);
```

Precauciones con los argumentos

Las macros con parámetros son *parecidas* a funciones, pero sólo se limitan a sustituir texto. Supongamos la macro

```
#define producto(a,b) a*b
```

Si escribimos esta llamada:

```
u=1; v=3;
x = producto(2+u,v+6);
```

la macro se expande a esta expresión:

```
x = 2+u*v+6;
```

El valor depositado en ***x*** es 11, lo cual probablemente no era el resultado deseado (27). Para que no ocurran estos problemas, hay que encerrar en paréntesis los argumentos en las definiciones de las macros:

```
#define producto(a,b) (a)*(b)
```

En conclusión, hay que tener precaución con los argumentos, y usar paréntesis en las declaraciones de macros (incluso abusar de ellos).

10.3 Compilación condicional

Algunas órdenes del preprocesador permiten escribir código que sólo se compila si se dan ciertas condiciones. Se utiliza **`#ifdef`** y **`#endif`**

En este ejemplo,

```
#ifdef DEBUG
     printf ( "He pasado por esta línea\ncuenta=%d\n",
              cuenta );
#endif
```

sólo se compila la sentencia con **`printf`** si está definido el símbolo **`DEBUG`**.

Al inicio del programa, se puede definir el símbolo con

```
#define DEBUG
```

si se quiere compilar este fragmento de código.

También se pueden utilizar varias opciones de compilación:

```
#ifdef MSDOS
     ... código para MSDOS ...
#else
#  ifdef UNIX•
     ... código para UNIX ...
#  else
     ... código para otro sistema ...
#  endif
#endif
```

Existe también la orden **`#ifndef`** ***símbolo***, que sólo compila si no está definido el símbolo.

10.4 Eliminación de macros

A veces se quiere eliminar una macro o símbolo previamente definido. Esto se hace con

```
#undef símbolo
```

10.5 Inclusión de ficheros en el fuente

Las órdenes

#include <*fichero*>
#include "*fichero*"

insertan el contenido del ***fichero*** dentro del fuente actual.

si el nombre del fichero se escribe entre **< >**, el fichero se busca en directorios prefijados para ficheros cabeceras (como **stdio.h**). Por ejemplo, en UNIX se usa el directorio **/usr/include**. En ese caso, las directivas:

#include <stdio.h> y
#include "/usr/include/stdio.h"

tendrían el mismo efecto.

11. Programación modular

Un aplicación escrita en C puede estar compuesta de múltiples **módulos**, cada uno encargado de una función (manejo de archivos, gestión de errores, visualización de datos, interfaz de usuario, etc.)

Un módulo es un conjunto de funciones, tipos de datos, variables, etc., en definitiva, recursos que pueden utilizar otros módulos.

Cada módulo es un fichero fuente de extensión ***.c*** que se compila por separado.

11.1 Interfaces: ficheros cabecera

La **interfaz** de un módulo es el conjunto de funciones, variables, tipos de datos y constantes que se hacen públicas al resto de los módulos.

Por cada módulo ***fichero.c*** existirá un fichero cabecera ***fichero.h***, el cual contendrá la declaración de la interfaz de ese módulo.

Las funciones, variables, etc. que son de uso interno del módulo no se declaran en el fichero cabecera.

Los módulos que vayan a utilizar recursos de un módulo llamado ***mod.c*** tendrían que incluir su fichero cabecera con

```
#include "mod.h"
```

con lo que se incluyen las declaraciones públicas de ese módulo.

Proyectos. Programa make.

Para generar una aplicación compuesta de varios módulos, hace falta compilar todos los módulos y luego enlazarlos (*link*) para producir el ejecutable.

Los compiladores de C disponen de herramientas para el manejo cómodo de aplicaciones modulares. En UNIX y en MS-Windows existe un programa llamado `make` que permite mantener una aplicación modular, recompilando sólo lo necesario cuando haga falta.