CURSO

DE

LENGUAJE "C"

Angel Salas

Centro de Cálculo
Universidad de Zaragoza

Enero - 1991

SALAS, Angel

Curso de Lenguaje "C" / Angel Salas . - Zaragoza :
Secretariado de Publicaciones de la Universidad ,
1991 . - 238 p. (pag. var.) ; 30 cm . - (C.C.U.Z. ; 28)

ISBN 84-7733-232-0

CURSO DE LENGUAJE "C"
Angel Salas

Centro de Cálculo
de la Universidad de Zaragoza, 1991.
Edificio de Matemáticas
Ciudad Universitaria
50009 - ZARAGOZA
Tfno: 354100

Edita: Secretariado de Publicaciones
de la Universidad de Zaragoza

Depósito Legal: Z-416-91
I.S.B.N.: 84-7733-232-0

PRESENTACION

Esta publicación recoge la documentación que se entrega en el Curso de Lenguaje "C" impartido por el Centro de Cálculo de la Universidad de Zaragoza.

Contiene una reproducción de todas las transparencias que usa el profesor en sus exposiciones.

No es un manual de referencia, sino un material didáctico dirigido a facilitar la comprensión de los conceptos, elementos y reglas de la construcción de programas con lenguaje "C".

En el curso, se explica el lenguaje desde el principio, en términos que presuponen conocimientos básicos sobre la progamación de computadores. Se estudian, presentando abundantes ejemplos, todos los aspectos del lenguaje: tipos de datos, clases de almacenamiento, operadores, expresiones, sentencias de control, funciones, bibliotecas estándar.

Hay una Introducción donde se exponen algunos datos históricos y características generales del lenguaje.

En la lección 1, se presenta un panorama general de la estructura de un programa escrito en C sin profundizar, con el fin de adquirir desde el principio familiaridad con algunos aspectos importantes. Se habla del formato de escritura, de los comentarios, de las sentencias para el preprocesador, de la definición de funciones, de las reglas de alcance, de las expresiones, de la sentencias "if-else" y "for", de las funciones "scanf" y "printf".

En la lección 2 se tratan los elementos que forman el programa desde el punto de vista léxico: caracteres, identificadores, palabras reservadas, constantes, cadenas, operadores, separadores.

La lección 3 es muy breve. Se trata únicamente el operador de asignación para explicar la semántica de las expresiones de asignación.

En la lección 4 se presentan los tipos de datos predefinidos sin entrar a fondo. También se trata de las clases de almacenamiento y de la inicialización de variables en las declaraciones.

En la lección 5 se estudian los tipos fundamentales a fondo y los operadores más afines.

La lección 6 se dedica a mostrar todas las sentencias de control.

La lección 7 está dedicada a las funciones. Se estudian todos los aspectos relacionados con la definición y uso de las funciones. Se explican las reglas de alcance, las clases de almacenamiento.

En la lección 8 se muestra un panorama general de los tipos estructurados predefinidos sin entrar a fondo.

En la lección 9 se estudian a fondo el tipo "array", las cadenas y la relación con los punteros. Se presentan algunas funciones de biblioteca para manejar cadenas de caracteres.

La lección 10 se dedica explicar el tipo "struct", los campos de "bits", las uniones y los tipos definidos por el usuario.

En la lección 11 se presentan algunas funciones de biblioteca para leer y escribir en ficheros: abrir y cerrar fichero, posicionar, leer y escribir un carácter, leer y escribir una cadena, leer y escribir en binario, leer y escribir con formato.

En la lección 12 se explican las bases para construir y manejar estructuras de datos dinámicas. Se estudia el caso de una lista encadenada con disciplina FIFO.

CONTENIDO

El lenguaje C fue diseñado por Dennis Ritchie, de los Laboratorios Bell, y se instaló en un PDP-11 en 1972.

Se diseñó para ser el lenguaje de los sistemas operativos UNIX.

Se creó para superar las limitaciones del lenguaje B, utilizado por Ken Thompson para producir la versión original de UNIX en 1970.

El lenguaje B se basó en BCPL, lenguaje sin tipos desarrollado por Martin Richards, en 1967, para programación de sistemas.

Su definición apareció en 1978:

> **apéndice "C Reference Manual"**
> **del libro "The C programming Language"**
> **de Brian W. Kernighan y Dennis M. Ritchie**
> **(Ed. Prentice-Hall)**

En 1983, se publicó otro estándar:

> **"The C Programming Language-Reference Manual" (Lab.Bell)**
> **escrito por Dennis M. Ritchie**

ES UN LENGUAJE DE NIVEL MEDIO

Combina elementos de lenguajes de alto nivel (tipos, bloques, ...) con la funcionalidad de los ensambladores.

Permite manejar los elementos típicos de la programación de sistemas:

bits
bytes
direcciones

NO ESTA FUERTEMENTE ORIENTADO A TIPOS

Tiene cinco tipos de datos básicos, tipos estructurados y admite definición de tipos por el usuario.

Pero permite casi todas las conversiones (p.ej. se pueden mezclar los tipos "int" y "char" en casi todas las expresiones).

No hace comprobaciones de error en tiempo de ejecución (desbordamiento de arrays, ...)

"Deja hacer" al programador.

ES UN LENGUAJE SEMI-ESTRUCTURADO

No es completamente estructurado en bloques porque no permite declarar procedimientos o funciones dentro de otros procedimientos o funciones.

Pero tiene algunas características propias de los lenguajes estructurados:

- **Dos formas de estructuración del código:**

  Con funciones independientes
  Con bloques

- **Dispone de las sentencias típicas para construir estructuras de control:**

  while
  do-while
  for

ES UN LENGUAJE PARA PROGRAMADORES

Algunos otros lenguajes están hechos para no-programadores (BASIC, COBOL, ...)

El lenguaje C está influenciado, diseñado y probado por programadores profesionales.

Proporciona:

- **Una visión próxima a la máquina**
- **Pocas restricciones**
- **Pocas pegas**
- **Conjunto reducido de palabras clave**
- **Estructuración en bloques**
- **Funciones independientes**
- **Recursos para el encapsulamiento de datos**

Permite alcanzar casi la eficiencia del código ensamblador, junto con la estructuración propia de lenguajes como ALGOL, MODULA-2.

Se diseñó para la programación de sistemas

Los programas son muy transportables

Actualmente se usa para otros propósitos

BIBLIOGRAFIA

"The C programming language"
Brian W Kernighan y Dennis M. Ritchie
Ed. Prentice-Hall, segunda edición, 1988.

"Lenguaje C. Introducción a la programación"
Al Kelley e Ira Pohl
Ed. Addison-Wesley, 1987 (edición original, en 1984).

"C estándar. Guía de referencia para programadores"
P.J. Plauger y Jim Brodie
Ed. Anaya Multimedia, 1990 (primera edición en 1989)

"C. Manual de referencia. Segunda edición"
Herbert Schildt
Ed. McGraw-Hill, 1990.

Manual de Referencia de la implementación
que se use.

- 1 -

ESTRUCTURA Y FORMATO DEL PROGRAMA. PANORAMA GENERAL.

- **El formato de escritura**
- **Ejemplo de un programa completo**
- **Los comentarios**
- **Las sentencias para el pre-procesador**
- **Las funciones. Definición de una función**
- **Las declaraciones**
- **Reglas de alcance**
- **Las expresiones**
- **Las sentencias**
- **La sentencia "if-else"**
- **La sentencia "for"**
- **La función "printf"**
- **La función "scanf"**

```
#include <stdio.h>

main ( )
{
  saludo( );
  primer_mensaje( );
}



saludo()
{
  printf ("Buenos dias\n");
}



primer_mensaje()
{
        printf("Un programa esta formado ");
        printf("por funciones\n");
}
```

Los programas se construyen con:

Comentarios.

Ordenes para el preprocesador de macros.

Definiciones de funciones.

Expresiones formadas con constantes, variables, funciones y operadores.

Sentencias.

EL FORMATO DE ESCRITURA ES MUY FLEXIBLE:

Las constantes, identificadores y palabras clave deben separarse; pero ello puede hacerse con :

- **espacios en blanco**
- **marcas de tabulador**
- **marcas de salto de linea**
- **comentarios**

```
/*
  ej1.c
  Indica el menor de dos enteros leidos
*/

#include <stdio.h>

void main ( )
{
        int n1, n2, menor (int, int);
        printf ("Introducir dos enteros:\n");
        scanf ("%d%d", &n1, &n2);
        if ( n1 == n2 )
                printf ("Son iguales \n");
        else
                printf ("El menor es: %d\n",menor(n1,n2));
}


int menor (int a, int b)
{
        if ( a < b )  return (a );
        else  return ( b );
}
```

LOS COMENTARIOS

Son lineas que ignora el compilador pero sirven para documentar los programas:

Empiezan con: /*

terminan con: */

pueden abarcar varias lineas

no se pueden anidar

```
/*
        ej1.c
        Indica el menor de dos enteros leidos
*/
...
...
...
scanf("%d%d", &n1, &n2);  /*      aqui se leen dos
                                   valores enteros */
...
...
```

LAS SENTENCIAS PARA EL PREPROCESADOR

Son órdenes que el preprocesador interpreta antes de que el código fuente sea compilado.
El preprocesador produce un programa escrito en C que es lo que se compila después.

Deben empezar con el símbolo " # " en la primera columna.

```
# define  PI  3.1416
# define  EQ  ==
----------
# define  cuadrado(x)  ( (x) * (x) )
----------
# include <stdio.h>
# include <math.h>
----------
# define  PRUEBA  1
...
...
# if  PRUEBA
printf("prueba:  x = %d\n", x);
# endif
----------
# ifdef UNO
...
...
# else
...
...
# endif
( cc -DUNO fn.c )
----------
# ifndef
----------
```

LAS FUNCIONES

Un programa esta formado por funciones.

No se pueden definir anidadas.

Desde una función se puede llamar a cualquier otra.

Está permitida la recursividad.

Si no se indica otro tipo, las funciones son de tipo "int" por defecto

El mecanismo de paso es por valor.

DEVUELVEN UN VALOR.

La función "main"

Todo programa debe contener una función llamada "main".

Es la invocada desde el sistema operativo cuando comienza la ejecución del programa.

También devuelve un valor al medio de llamada.

DEFINICION DE UNA FUNCION

Encabezamiento . . . . . tipo nombre (p1, p2, p3)

Declaración de parámetros . . . . . tipo p1, p2, p3;

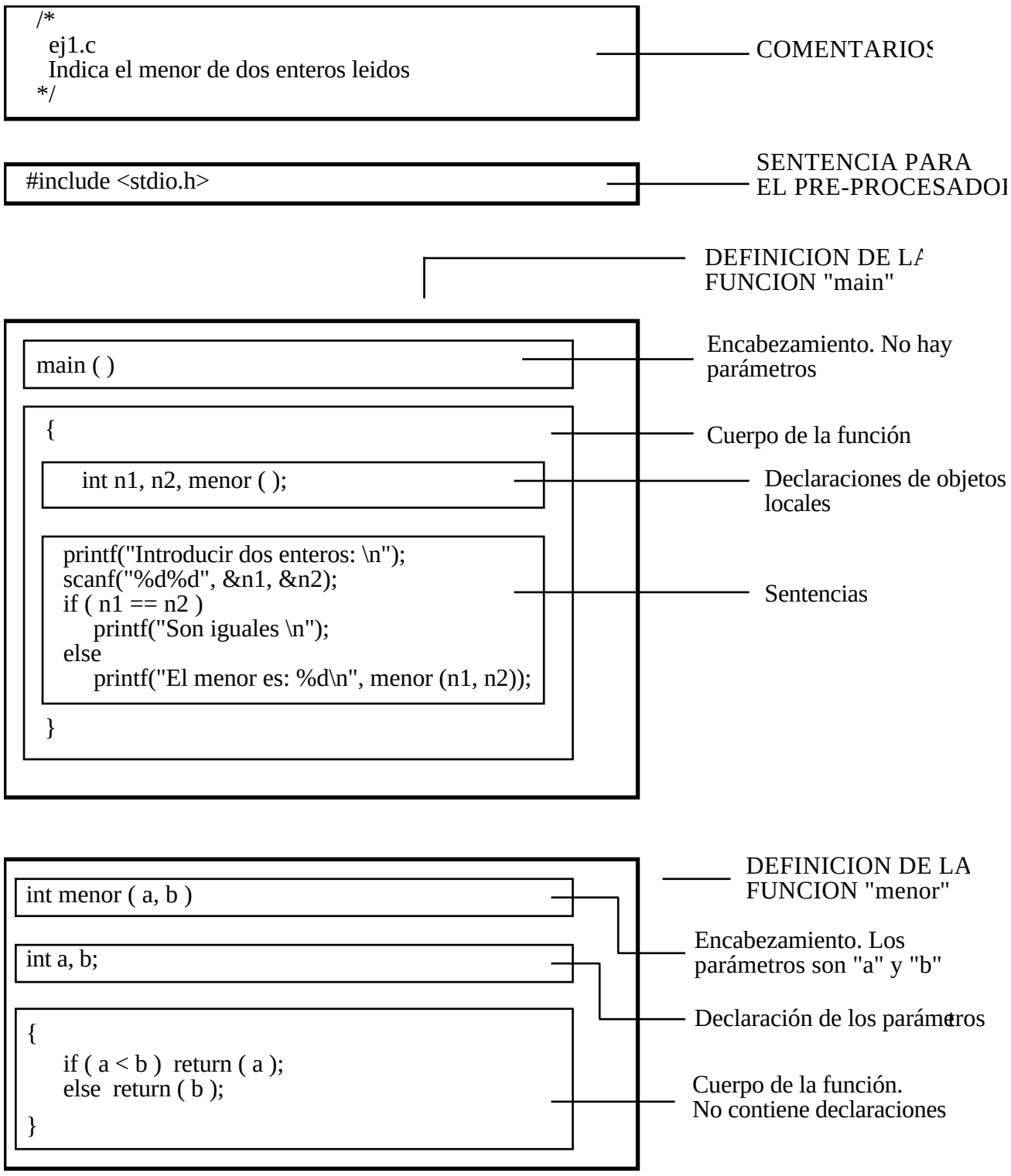
{

Cuerpo con: Declaraciones . . . . . tipo v1, v2, v3;

Sentencias . . . . . sentencia
...
sentencia

}

```
int menor(a, b)
int a, b;
{
    if ( a<b ) return( a );
    else return( b );
}
```


/*
ej1.c
Indica el menor de dos enteros leidos
*/
COMENTARIOS
#include <stdio.h>
SENTENCIA PARA
EL PRE-PROCESADOR
DEFINICION DE LA
FUNCION "main"
main ( )
Encabezamiento. No hay
parámetros
{
Cuerpo de la función
int n1, n2, menor ( );
Declaraciones de objetos
locales
printf("Introducir dos enteros: \n");
scanf("%d%d", &n1, &n2);
if (n1 == n2)
printf("Son iguales \n");
else
printf("El menor es: %d\n", menor (n1, n2));
Sentencias
}
int menor (a, b)
DEFINICION DE LA
FUNCION "menor"
Encabezamiento. Los
parámetros son "a" y "b"
int a, b;
Declaración de los parámetros
{
if (a < b) return (a);
else return (b);
Cuerpo de la función.
No contiene declaraciones
}

DEFINICIONES DE DATOS. DECLARACIONES

Todas las variables deben declararse antes de usuarlas:

Indicar el tipo de datos al que pertenece.

Ello determina : Su representación en memoria.
El conjunto de valores posibles.
Las operaciones permitidas.
La forma de realizar las operaciones.

También se puede indicar:

La clase de localización en memoria.
El ámbito de existencia.
Un valor inicial.

El usuario puede definir tipos nuevos, combinando tipos predefinidos

pero...

C no está orientado a los tipos estrictamente

REGLAS DE ALCANCE

Los objetos declarados fuera de la función son globales.

Al principio de cualquier bloque (proposiciones entre { y })
se pueden poner declaraciones.

Los objetos sólo son conocidos dentro del bloque en que han sido declarados, y en los bloques anidados dentro.

Ejemplos de declaraciones

```
int n, m, *p, v[100];

float r, a[10][10], *pf[100];

char c, *s, lin[80];

float superficie();


struct naipe {
        int valor;
        char palo;
} carta ;


enum colores{ rojo, azul,
                        amarillo } color;


typedef struct {
            float real;
            float imaginaria;
        } complejo;

complejo c1, c2;
```

Reglas de alcance

```
int g;                    /*  "g" es variable global  */

main( )
{
 int a, b;
 . . .
 . . .
 {
        float b, x, y;
        . . .        /* se conoce a "int a", pero no
        . . .           a "int b", que ha quedado
                              enmascarada.  */
 }

 {
        unsigned a;
        char c, d;
        . . .        /* se conoce a "int b", pero no a
        . . .           "int a".
                              No se conoce a "x" ni a "y"  */
 }
 ...
 ...                 /*      se conoce a "int a" y a "int b".
 ...                         No se conoce a "x", "y", "c"  */
 ...
}
```

La variable "g" es conocida en todos los bloques.

EXPRESIONES

Una expresion se forma combinando constantes, variables, operadores y llamadas a funciones.

```
s = s + i
n == 0
++i
```

Una expresión representa un valor que es el resultado de realizar las operaciones indicadas siguiendo las reglas de evaluación establecidas en el lenguaje.

Con expresiones se forman sentencias, con sentencias se forman funciones y con funciones se construye un programa completo.

Algunos operadores

De relación:	menor que:	<	**Aritméticos**:	suma:	+
	mayor que:	>		resta:	-
	menor o igual que:	<=		multiplicación:	*
	mayor o igual que:	>=		división:	/
				resto división entera:	%
De igualdad:	igual a:	==		(unario) incremento:	++
	distinto de:	!=		(unario) decremento:	--
				asignación:	=
Lógicos:	(unario) negación:	!			
	y (AND) lógico:	&&			
	o (OR) lógico:	||			

SENTENCIAS

Una sentencia se forma con expresiones:

Puede contener una, varias o ninguna expresión

Un bloque empieza con " { " y termina con " } ", pudiendo contener cualquier número de sentencias y declaraciones.

Todas las sentencias, excepto los bloques, terminan con el símbolo " ; ".

```
s = s + n ;
if ( letra == '.' ) terminar( );
printf ( "mensaje\n" );
```

En los bloques, las declaraciones de datos deben estar al principio del bloque delante de cualquier sentencia que los utilice.

```
. . .
. . .
{
    int j;
    j = i-1;
    while ( j >= 0 ) printf("%c", s[j--]);
    printf ("\n");
}
. . .
. . .
```

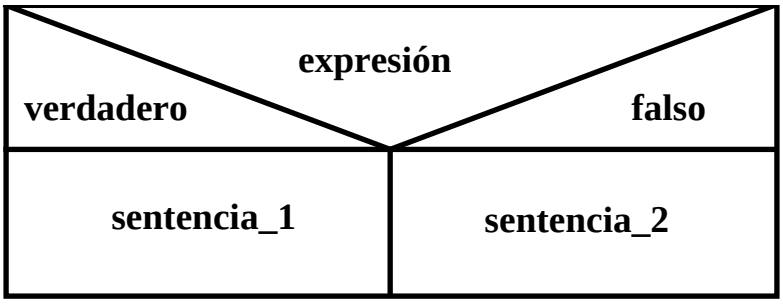
LA SENTENCIA "if - else"

if (*expresión* **)** *sentencia_1* **else** *sentencia_2*

La expresión debe escribirse entre paréntesis.
Las sentencias terminan con ";".

Si la expresión no es cero (verdadero), se ejecuta la "sentencia_1", en caso contrario, si es cero (falso), se ejecuta la "sentencia_2"

cero	es	**falso**
distinto de cero	es	**verdadero**

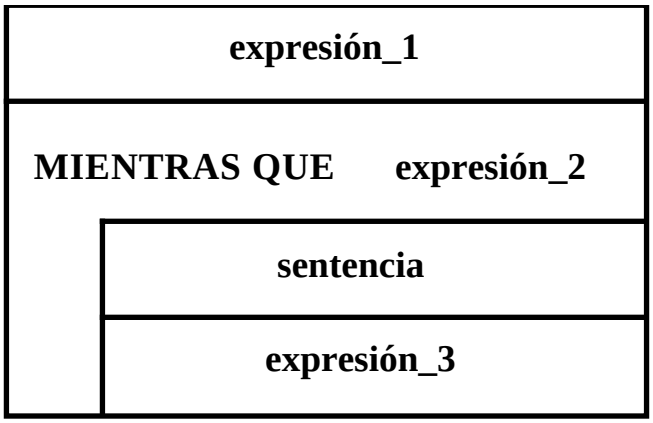



```
if ( a < b ) return (a);
else  return ( b );
```

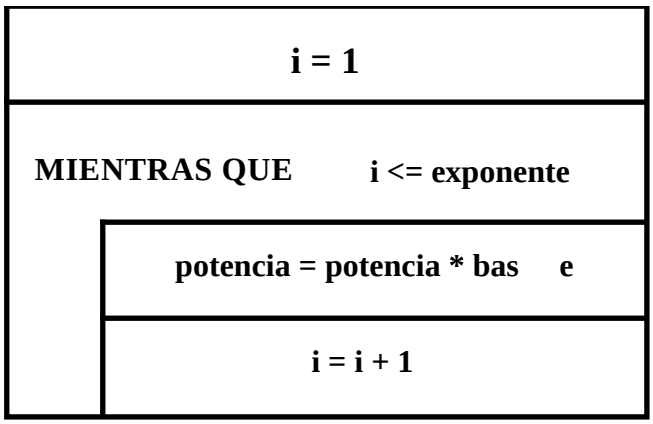
LA SENTENCIA "for"

for (*expresión_1* **;** *expresión_2* **;** *expresión_3* **)** *sentencia*

Se evalúa la "expresión_1", se evalúa la "expresión_2" y si produce el valor "verdadero" (distinto de cero) se ejecuta la "sentencia". Después se evalúa la "expresión_3" y el control pasa al principio del ciclo y se salta la evaluación de la "expresión_1". El proceso se repite hasta que la "expresión_2" tome el valor "falso" (cero).




```
potencia = 1.0;
for (i=1 ; i<=exponente ; i++)
potencia=potencia*base;
```



```
/*
  ej2.c
  Calcula potencias de base real positiva y exponente entero
*/


#include <stdio.h>


void main ( )
{
  int exponente;
  float base, potencia (float, int);
  printf ("Introducir BASE y EXPONENTE: \n");
  scanf ("%f%d", &base, &exponente);
  if ( base <= 0.0 ) printf ("Solo admito bases positivas \n");
  else {  if ( exponente > 0 )
        printf ("Potencia: %f \n", potencia(base, exponente));
           else
                printf ("Potencia: %f \n",
                                        1.0/potencia(base, -exponente));
        }
}


float potencia (float b, int e)
{
        if ( b == 1.0 ) return (1.0);
        else  {
                if ( e == 1 ) return ( b );
                else  {
                        int i;   float p=1.0;
                        for ( i=1; i<=e; i++ ) p=p*b;
                        return ( p );
                }
        }
}
```

La función "printf"

Permite escribir con formato por el dispositivo de salida estándar.

Admite dos tipos de parámetros:

Especificaciones de formato
Lista de valores

Se indican los valores que hay que escribir y el formato de escritura. La función "printf" convierte, formatea y escribe.

las sentencias:

```
float  x1, x2;
x1=3.1416;  x2=2.5;
printf("Resultados: \nx1=%f \nx2=%f \n", x1,x2);
printf("Otro formato: \nx1=%7.2f \nx2=%4.2f \n", x1,x2);
```

producen una salida así:

```
Resultados:
x1=3.141600
x2=2.500000
Otro formato:
x1=   3.14
x2=2.50
```

Especificaciones de formato

Se escriben entre comillas.
Pueden contener:

-Caracteres y secuencias de escape como "\n" (salto de linea)

"INDICE\n"

-Especificaciones de conversión:

Empiezan con "%" y lleva un código de conversión y un modificador opcional:

%d %4d %7.4f

Con el modificador se determina la presentación (longitud del campo, número de dígitos decimales, etc...)

Con el código de conversión se indica el tipo de transformación:

d	se convierte a notación decimal
o	se convierte a formato octal sin signo
x	se convierte a formato hexadecimal sin signo
f	se considera que el argumento es de tipo "float" y se convierte a notación decimal
e	se considera que el argumento es de tipo "float" y se convierte a notación exponencial
c	se considera que el argumento es un simple carácter
s	se considera que el argumento es una cadena de caracteres

Lista de valores

Pueden ponerse constantes, variables, llamadas a funciones y expresiones

```
printf ( "El numero %d es impar.\n", n );

for ( r=3.1416; r >= 0.0; r=r-0.001)  printf ( "%7.4f\n", r );

printf ( "Atencion!! %c \n", 7 );

printf ( "Potencia:%f\n", potencia(base, exponente));
```

La función "scanf"

Permite leer datos con formato por el dispositivo de entrada estándar.

Lee caracteres y los convierte a representaciones internas de variables según las especificaciones de formato.

Admite parámetros de dos clases:

Especificaciones de formato
Apuntadores a variables

```
scanf ( "%d%d", &n1, &n2 );
```

Las especificaciones de formato:

-se escriben en tre comillas
-empiezan con el símbolo " % "
-pueden contener códigos de conversión y modificadores de conversión

Algunos códigos de conversión:

d	**a entero decimal (int)**
f	**a coma flotante (float)**
c	**a carácter (char)**
s	**a cadena de caracteres**

La lista de apuntadores:

Contiene expresiones que apuntan (son direcciones) a las variables que recibirán los valores leidos según las especificaciones de formato correspondientes.

Pueden ser nombres de variables de tipo puntero o expresiones formadas con el símbolo " & " seguido de un nombre de variable:

```
int n1, n2;
scanf ("%d %d", &n1, &n2);

...
char letra; int n; float r ;
scanf ("%c %d %f", &letra, &n, &r);

...

char cad[80];
scanf ("%s", cad);
```

& es el operador de dirección

En C, el mecanismo de paso de valores a las funciones es por valor siempre. Por eso, cuando se quiere producir el efecto lateral de modificar el valor de una variable, hay que pasar su dirección usando el operador de dirección " & ".

```
/*
  ej3.c
  Lee un numero entero y determina si es par o impar
*/

#include <stdio.h>

#define MOD %   /*  %, es el operador que obtiene  el resto
                                    de la división entera   */
#define EQ ==
#define NE !=
#define SI 1
#define NO 0

void main ( )
{
int n, es_impar(int);
printf ("Introduzca un entero: \n");
scanf ("%d", &n);
if ( es_impar (n) EQ SI )
        printf ("El numero %d es impar. \n", n);
else
        printf ("El numero %d no es impar. \n", n);
}


int es_impar (int x)
{
        int respuesta;
        if ( x MOD 2  NE  0 ) respuesta=SI;
        else  respuesta=NO;
        return (respuesta);
}
```

- 2 -

ELEMENTOS LEXICOS

- **Caracteres**
- **Símbolos:**
  - **Identificadores**
  - **Palabras reservadas**
  - **Constantes**
  - **Cadenas**
  - **Operadores**
  - **Separadores**

Los caracteres

Un programa escrito en "C" es una secuencia de caracteres agrupados, formando símbolos que componen cadenas sintácticas válidas.

letras minúsculas:	`a b c d . . . z`
letras mayúsculas:	`A B C D . . . Z`
cifras:	`0 1 2 3 4 5 6 7 8 9`

caracteres especiales:	`+ = _ - ( ) * & % $ # !`
	`| < > . , ; : " ' / ?`
	`{ } ~ \ [ ] ^`

caracteres no imprimibles:	`espacio en blanco,`
	`salto de linea,`
	`marca de tabulador`

LOS SIMBOLOS

Son secuencias de uno o más caracteres, que se usan para representar entidades de distinta naturaleza:

identificadores:

n, suma, main, scanf, ...

palabras reservadas

int, float, while, if, else, return, ...

constantes: 3.1416, 27, 'a', ...

cadenas: "palabra", ...

operadores: +, -, *, /, ++, ...

separadores: **{, }, . . .**

Los espacios en blanco, las marcas de tabulador, los saltos de linea y los comentarios sirven como separadores, pero se ignoran a otros efectos por el compilador.

LOS IDENTIFICADORES

Son secuencias de caracteres que se pueden formar usando letras, cifras y el carácter de subrayado "_".

Se usan para dar nombre a los objetos que se manejan en un programa: tipos, variables, funciones, ...

Deben comenzar por letra o por "_".

Se distingue entre mayúsculas y minúsculas.

Se deben definir en sentencias de declaración antes de ser usados.

```
int i, j, k;

float largo, ancho, alto;

enum colores {rojo, azul, verde}
    color1, color2;
```

LAS PALABRAS RESERVADAS

Son algunos símbolos cuyo significado está predefinido y no se pueden usar para otro fin:

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

LAS CONSTANTES

Son entidades cuyo valor no se modifica durante la ejecución del programa.

Hay constantes de varios tipos.

Ejemplos:

numéricas: `-7 3.1416 -2.5e-3`

caracteres: `'a' '\n' '\0'`

cadenas: `"indice general"`

CONSTANTES SIMBOLICAS

Se definen con sentencias para el preprocesador.

Se define un identificador y se le asigna un valor constante. Después se puede usar el identificador.

Favorecen la realización de modificaciones en los programas.

```
#define PI  3.1416
```

LAS CADENAS

Una cadena de caracteres (string) en "C" es una secuencia de caracteres, almacenados en posiciones de memoria contiguas, que termina con el carácter nulo.

Una cadena se representa escribiendo una secuencia de caracteres encerrada entre comillas:

```
"buenos dias"
"asi: \" se pueden incluir las comillas"
```

Una cadena es un valor constante de una estructura de tipo "array de char".

```
char titulo[24];
strcpy(titulo, "Curso de C");
printf("%s", titulo);
```

Para realizar operaciones con cadenas hay que usar funciones de biblioteca. El lenguaje no dispone de operadores para cadenas.

LOS OPERADORES

Sirven para indicar la aplicación de operaciones sobre los datos:

operaciones aritméticas, de comparación, de desplazamiento de bits, de indirección, etc...

Casi todos se representan con símbolos especiales:

+ - / % & * etc...

Algunos tienen significados diferentes según el contexto:

printf("%d", a)	**aquí se usa para especificación de formato**
n % 5	**aquí es el operador módulo**
p = n * m	**aquí es para multiplicar**
n = *puntero	**aquí es la indirección**

LOS OPERADORES

Puestos en orden de prioridad descendente

Operadores	Asociatividad
( ) [ ] -> . (miembro)	**izquierda a derecha**
~ ! ++ -- sizeof (tipo) **-(unario) *(indirección) &(dirección)**	**derecha a izquierda**
*** / %**	**izquierda a derecha**
+ -	**izquierda a derecha**
<< >>	**izquierda a derecha**
< <= > >=	**izquierda a derecha**
== !=	**izquierda a derecha**
&	**izquierda a derecha**
^	**izquierda a derecha**
|	**izquierda a derecha**
&&	**izquierda a derecha**
||	**izquierda a derecha**
?:	**derecha a izquierda**
= += -= *= ...	**derecha a izquierda**
, (operador coma)	**izquierda a derecha**

LOS SEPARADORES

Los caracteres especiales actúan como separadores en general.

Hay algunos caracteres que sólo sirven para actuar de separadores:

- El espacio en blanco
- La marca de tabulador
- El salto de linea
- Los comentarios (se trata el conjunto como un espacio en blanco)

Hay que separar con espacios en blanco:

- Las palabras reservadas
- Las constantes
- Algunos identificadores adyacentes

Si la secuencia que lee el compilador ha sido analizada hasta un carácter determinado:

el siguiente elemento se toma incluyendo la cadena de caracteres más larga que pueda constituir un elemento sintáctico válido.

- 3 -

EXPRESIONES Y SENTENCIAS.

EXPRESIONES DE ASIGNACION.

LAS EXPRESIONES

Son combinaciones de constantes, variables, operadores y llamadas a funciones:

```
i++
a + b
3.1416*r*r
densidad*volumen(a, b, c)
```

En general, toda expresión tiene un valor que es el resultado de aplicar las operaciones indicadas a los valores de las variables siguiendo unas reglas de prioridad y asociatividad propias del lenguaje.

Una expresión seguida de un punto y coma es una sentencia.

```
3.1416;
n=0;
++n;
s=s+n;
```

LAS EXPRESIONES DE ASIGNACION

Se forman con el operador " = ".

variable = expresion

Cuando se ejecuta, se evalúa la expresión que está a la derecha y el resultado se asigna a la variable del lado izquierdo haciendo conversión de tipo si es necesario. Ese valor es el que toma la expresión en su conjunto también.

n=5 **la expresión toma el valor 5**

x=(y=5)+(z=7) **y toma el valor 5**
z toma el valor 7
x toma el valor 12
la expresión toma el valor 12

El operador de asignación " = " es asociativo de derecha a izquierda:

x=y=z=0 **es equivalente a x=(y=(z=0))**

- 4 -

LOS TIPOS DE DATOS.

VISION GENERAL.

- **El concepto de tipo de datos.**
- **Los tipos fundamentales. Esquema. Descripción breve.**
- **Declaraciones: tipo - clase - valor inicial.**
- **Las clases de almacenamiento.**
- **Ejemplo de declaraciones e inicialización.**
- **El operador "sizeof".**

LOS TIPOS DE DATOS

PROGRAMA = ESTRUCTURAS DE DATOS + ALGORITMOS

Las formas de organizar datos están determinadas por los TIPOS DE DATOS definidos en el lenguaje.

Todo objeto de datos pertenece a un tipo.

Un tipo de datos determina el rango de valores que puede tomar el objeto, las operaciones a que puede ser sometido y el formato de almacenamiento en memoria.

En "C":

- **Existen tipos predefinidos**
- **El usuario puede definir otros tipos, a partir de los básicos.**

pero...

No es un lenguaje orientado a tipos estrictamente

LOS TIPOS DE DATOS

ESCALARES:

Numéricos: **Enteros:** **int**
long
short
unsigned
unsigned long
unsigned short
char

Reales: **float**
double

pointer **(punteros)**

enum **(enumerativos)**

ESTRUCTURADOS:

array
string
struct
campos de bits
union

`int, long, short, unsigned, ...:`

Para manejar números enteros.

```
int n;  n=27;
```

`float, double:` **Para manejar números reales.**

```
float r;    r=3.1416;
```

`char:` **Representa a un carácter de la tabla de codificación.**

```
char c; c='a';
```

`pointer:` **Apuntadores para manejar estructuras dinámicas. Es fundamental en "C". Se permite la aritmética de punteros. Se usan para conseguir el paso por referencia en la comunicación entre funciones.**

```
int n, m, *p;   p=&x;
                m=*p;
                leer(&n);
```

`enum :` **Enumerativos.El usuario define cual es el conjunto de valores posibles.**

```
enum colores
    {rojo,azul,verde}color;

color=verde;
```

`array`: **Arreglos homogéneos. Pueden ser de una o varias dimensiones. Permiten el acceso directo a cualquiera de sus elementos.**

```
int n[100]; n[1]=0;

float r[10][20][30];
float a[100][100];
int i,j;
                    a[1][2]=2.5;
                    a[i][j]=0.0;
```

`string`: **Cadenas. Secuencias de caracteres terminando con el carácter nulo.**
Se manejan con funciones de biblioteca.
Son arreglos de "char".

```
char *c, s[24];

strcpy(c,"abc");     'a' 'b' 'c' '\0'
strcpy(s,"indice general");
```

`struct`: **Estructuras. Son arreglos heterogéneos.**

```
struct naipe {
        int valor;
        char palo;
} c1, c2;
                    c1.valor=3;
                    c1.palo='e';
```

union : **Uniones. Para compartir memoria.**

```
union numerico {
            int n;
            float x;
} numero;

                        numero.x=5.3;
```

DECLARACION DE VARIABLES

Todos los identificadores de variables deben ser declarados antes de usarse.

En la declaración de una variable se establece:

- **la clase de almacenamiento**
- **el tipo al que pertenece**
- **el valor inicial (opcionalmente)**

```
int n, m;
static char s[24]="cadena";
static float a[3][3] ={      1.0, 1.0, 1.0,
                             1.0, 1.0, 1.0,
                             1.0, 1.0, 1.0  };
```

CLASES DE ALMACENAMIENTO

automatic
static
external
register

CLASES DE ALMACENAMIENTO

automatic : **son variables locales de cada llamada a un bloque. Desaparecen al terminar la ejecución del bloque.**

static : **locales de un bloque. Conservan su valor entre ejecuciones sucesivas del bloque.**

external : **existen y mantienen sus valores durante toda la ejecución del prgrama. Pueden usarse para la comunicación entre funciones, incluso si han sido compiladas por separado.**

register : **se almacenan, si es posible, en registros de alta velocidad de acceso. Son locales al bloque y desaparecen al terminar su ejecución.**

INICIALIZACION DE VARIABLES

POR DEFECTO:

external, static	**se inicializan a cero**
automatic, register	**quedan indefinidas (valores aleatorios)**

PARA VARIABLES ESCALARES:

Se pueden asignar valores iniciales en la declaración:

tipo variable = valor;

con external y static:

- **se hace una vez, en tiempo de compilación**
- **se deben usar valores constantes**

con automatic y register:

- **se hace cada vez que se entra en el bloque**
- **se pueden usar variables y llamadas a funciones**

INICIALIZACION DE ARRAYS

Los automatic no se pueden inicializar.

Los external y los static se pueden inicializar escribiendo una lista de valores separados por comas y encerrada entre llaves:

```
static float a[5]={1.0,1.0,1.0,1.0,1.0};
```

- **si se ponen menos valores, el resto de los elementos se inicializa a cero**
- **si no se declaró tamaño, se toma el número de elementos de la lista de inicialización**
- **con los array de char, se permite una construcción especial:**

```
static char s[24]="los tipos de datos"
```

```
/*
    ej4.c
    Inicializacion y manejo de "arrays", cadenas y estructuras.
*/
# include <stdio.h>
void main()
{
    int i, j;
    static int  enteros [5] = { 3, 7, 1, 5, 2 };
    static char cadena1 [16] = "cadena";
    static char cadena2  [16] = { 'c','a','d','e','n','a','\0' };
    static int  a[2][5] = {
                            { 1, 22, 333, 4444, 55555 },
                            { 5, 4, 3, 2, 1 }
    };
    static int  b[2][5] = { 1,22,333,4444,55555,5,4,3,2,1 };
    static char *c = "cadena";
    static struct  {
                int i;
                float x;
    } sta = { 1, 3.1415e4}, stb = { 2, 1.5e4 };
    static struct  {
                char c;
                int i;
                float s;
    } st [2][3] = {
                    {{ 'a', 1, 3e3 }, { 'b', 2, 4e2 }, { 'c', 3, 5e3 }},
                    {     { 'd', 4, 6e2 },}
                };
    printf ("enteros:\n");
    for ( i=0; i<5; ++i ) printf ("%d  ", enteros[i]);
    printf ("\n\n");
    printf ("cadena1:\n");
    printf ("%s\n\n", cadena1);
    printf ("cadena2:\n");
    printf ("%s\n\n", cadena2);
    printf ("a:\n");
    for (i=0; i<2; ++i) for (j=0; j<5; ++j) printf ("%d  ", a[i][j]);
    printf("\n\n");
    printf ("b:\n");
    for (i=0; i<2; ++i) for (j=0; j<5; ++j) printf ("%d  ", b[i][j]);
```

```
    printf ("\n\n");
    printf ("c:\n");
    printf ("%s\n\n", c);
    printf ("sta:\n");
    printf ("%d  %f  \n\n", sta.i, sta.x);
    printf ("st:\n");
    for (i=0; i<2; ++i) for (j=0; j<3; ++j)
    printf ("%c  %d  %f\n", st[i][j].c, st[i][j].i, st[i][j].s);
}
```

```
enteros:
3  7  1  5  2

cadena1:
cadena

cadena2:
cadena

a:
1  22  333  4444  55555  5  4  3  2  1

b:
1  22  333  4444  55555  5  4  3  2  1

c:
cadena

sta:
1  31415.000000

st:
a  1  3000.000000
b  2  400.000000
c  3  5000.000000
d  4  600.000000
  0  0.000000
  0  0.000000
```

EL OPERADOR " sizeof "

Devuelve un entero que indica el número de bytes usados para almacenar un objeto.

```
int n;
printf("%d", sizeof(n));

printf("%d", sizeof( (int) ));
```

```
/*
  ej5.c
  Se indican los tamanios de las representaciones internas de
  algunos tipos de datos fundamentales.
*/
# include <stdio.h>
void main()
{
  char c;
  short s;
  int i;
  long l;
  float f;
  double d;
  printf ("Tipo char: %d  bytes\n", sizeof(c));
  printf ("Tipo short: %d  bytes\n", sizeof(s));
  printf ("Tipo int: %d  bytes\n", sizeof(i));
  printf ("Tipo long: %d  bytes\n", sizeof(l));
  printf ("Tipo float: %d  bytes\n", sizeof(f));
  printf ("Tipo double: %d  bytes\n", sizeof(d));
}
```

```
/*
  ej5a.c
  Aplicacion del operador "sizeof" a los tipos fundamentales.
*/
# include <stdio.h>
void main()
{
 printf ("\n                    char: %d bytes",     sizeof(char));
  printf ("\n                   short: %d  bytes",     sizeof(short));
  printf ("\n          unsigned short: %d  bytes",     sizeof(unsigned short));
  printf ("\n                     int: %d  bytes",     sizeof(int));
  printf ("\n                unsigned: %d  bytes",     sizeof(unsigned));
  printf ("\n                    long: %d  bytes",     sizeof(long));
  printf ("\n           unsigned long: %d  bytes",     sizeof(unsigned long));
  printf ("\n                   float: %d  bytes",     sizeof(float));
  printf ("\n                  double: %d  bytes",     sizeof(double));
  printf ("\n\n");
}
```

- 5 -

LOS TIPOS DE DATOS FUNDAMENTALES

- **Los tipos numéricos enteros**
- **Los tipos numéricos de coma flotante**
- **Operadores unarios**
- **Operadores aritméticos**
- **Funciones matemáticas**
- **El tipo char**
- **Las funciones "getchar" y "putchar" para leer y escribir un carácter**
- **Operadores de asignación**
- **Conversiones y "casts"**
- **El tipo " pointer "**
- **Aritmética de punteros**
- **Interpretación de las declaraciones complejas**
- **Los operadores de "bits"**
- **Máscaras**
- **Empaquetamiento**
- **Los tipos enumerativos**

LOS TIPOS NUMERICOS ENTEROS

int **long** **short**
unsigned **unsigned long** **unsigned short**
char

El tipo "int" es el más usado para trabajar con enteros.

Los demás se utilizan cuando se desea algún efecto especial como ahorrar memoria o poder operar con valores mayores.

short : **rango de valores menor para poder usar un número menor de bits**

unsigned : **sólo valores positivos, pero con mayor límite superior porque no se gasta un bit para el signo.**

La cantidad de memoria que ocupa cada tipo depende de la implementación pero determina, en todo caso, el rango de valores posibles.

El caso típico de las arquitecturas de 32 bits como la del VAX y la del HP-9000 se resume en la tabla siguiente.

Implementación de los tipos numéricos enteros en arquitecturas de 32 bits.

	Bits	Mínimo	Máximo
int	**32**	**-2^{31} (-2.147.483.648)**	**$2^{31}-1$ (+2.147.483.647)**
unsigned	**32**	**0**	**$2^{32}-1$ (+4.294.967.295)**
short	**16**	**-2^{15} (-32.768)**	**$2^{15}-1$ (+32.767)**
unsigned short	**16**	**0**	**$2^{16}-1$ (+65.535)**
long	**32**	**-2^{31} (-2.147.483.648)**	**$2^{31}-1$ (+2.147.483.647)**
unsigned long	**32**	**0**	**$2^{32}-1$ (+4.294.967.295)**
char	**8**	**'\0' (0)**	**'\377' (255)**

El tipo " int "

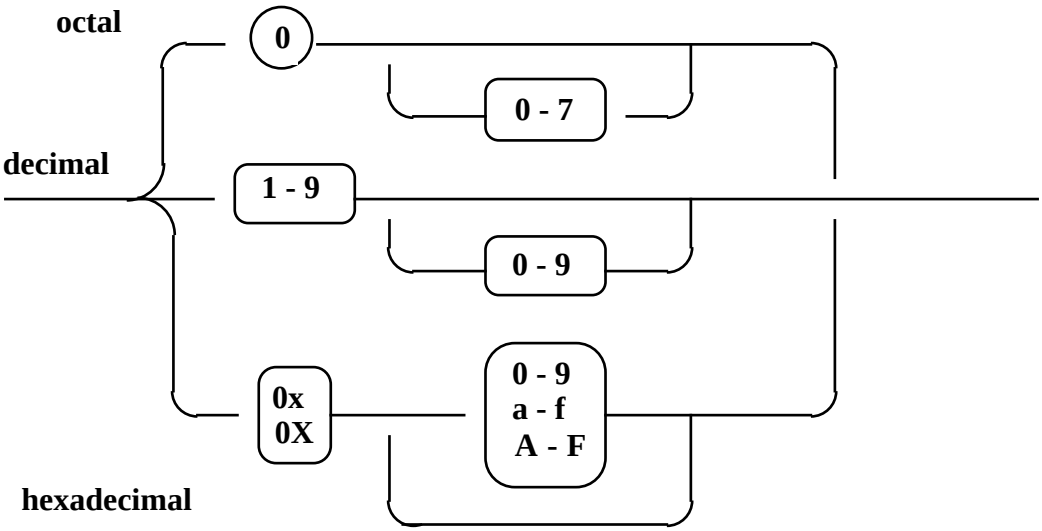
Es el tipo más importante:

- **Es el tipo que toman las funciones por defecto**
- **En la evaluación de expresiones aritméticas mixtas, los valores "char" y "short" se convierten a "int"**
- **Los direcciones de memoria se representan con enteros**
- **etc...**

Las variables de tipo "int" se usan:

- **Para realizar operaciones aritméticas**
- **Como contadores de bucles repetitivos**
- **Como subíndices de arrays**
- **etc...**

Constantes enteras



Ejemplos:

0
25
32767
0777
0x1A

Los enteros negativos son expresiones

Operador unario "menos" aplicado a constante entera

Rango de valores en arquitectura de 32 bits

Valor mínimo	Valor máximo
-2^{31}	$2^{31} - 1$
-2.147.483.648	2.147.483.647

Con 32 bits se pueden representar 2^{32} estados:

La mitad, son enteros positivos: 0, 1, 2, ..., 2^{31}-1

La mitad, son enteros negativos: -1, -2, . . ., -2^{31}

Se maneja la representación de complemento a 2 de 32 bits con "bytes" de 8 bits y codificación ASCII.

0	00000000000000000000000000000000
1	00000000000000000000000000000001
2	00000000000000000000000000000010
5	00000000000000000000000000000101
2.147.483.647	01111111111111111111111111111111
-2.147.483.648	10000000000000000000000000000000
-5	11111111111111111111111111111011
-2	11111111111111111111111111111110
-1	11111111111111111111111111111111

Los operadores aritméticos

+ - * / %

Corresponden a las operaciones matemáticas de suma, resta, multiplicación, división y módulo.

Son binarios porque cada uno tiene dos operandos.

Hay un operador unario menos "-", pero no hay operador unario más "+":

-3	**es una expresión correcta**
+3	**NO es una expresión correcta**

La división de enteros devuelve el cociente entero y desecha la fracción restante:

1/2	**tiene el valor 0**
3/2	**tiene el valor 1**
-7/3	**tiene el valor -2**

El operador módulo se aplica así:

- **con dos enteros positivos, devuelve el resto de la división**

12%3	**tiene el valor 0**
12%5	**tiene el valor 2**

- **si un operando o los dos son negativos, el resultado depende de la máquina.**

  En HP-UX:

printf ("%d\n", -12%5);	**escribe**	**-2**
printf ("%d\n", 12%-5);	**escribe**	**2**
printf ("%d\n", -12%-5);	**escribe**	**-2**

Los operadores de incremento y decremento

Son unarios.

Tienen la misma prioridad que el menos "-" unario.

Se asocian de derecha a izquierda.

Pueden aplicarse a variables, pero no a constantes ni a expresiones.

Se pueden presentar como prefijo o como sufijo.

Aplicados a variables enteras, su efecto es incrementar o decrementar el valor de la variable en una unidad:

++i; es equivalente a i=i+1;
--i; es equivalente a i=i-1;

Cuando se usan en una expresión, se produce un efecto secundario sobre la variable:

El valor de la variable se incrementa antes o despues de ser usado.

con ++a el valor de "a" se incrementa antes de evaluar la expresión.

con a++ el valor de "a" se incrementa después de evaluar la expresión.

con a el valor de "a" no se modifica antes ni después de evaluar la expresión.

Ejemplos

a=2*(++c) **se incrementa el valor de "c" y se evalúa la expresión después.**

Es equivalente a: **c=c+1;** **a=2*c;**

a[++i]=0 **se incrementa el valor de "i" y se realiza la asignación después.**

Es equivalente a: **i=i+1;** **a[i]=0;**

a[i++] **se realiza la asignación con el valor actual de "i", y se incrementa el valor de "i" después.**

Es equivalente a: **a[i]=0;** **i=i+1;**

Hay que evitar expresiones como ésta: a=++c+c porque el resultado depende de la máquina

Los tipos enteros "short" "long" "unsigned"

Se utilizan para obtener algún efecto especial: ahorrar espacio en memoria, disponer de un rango de valores mayor.

La implementación depende de la máquina.

short: **la representación interna ocupa un número menor de bytes.**

Si son 2 bytes (16 bits):

valor menor: $-2^{15}=$ **-32768**
valor mayor: $2^{15}-1=$ **32767**

long: **la representación interna ocupa un número mayor de bytes.**
En arquitecturas de 32 bits, suele ser igual que "int".

unsigned: **se prescinde del signo para disponer de un bit más y poder operar en un rango de valores mayor.**

Para una implementación de "int" en 32 bits, "unsigned" proporciona el rango:

valor mínimo: **0**
valor máximo: $2^{31}-1=$ **4.294.967.295**

El operador " sizeof "

Está incorporado al lenguaje para proporcionar el número de bytes usados para almacenar un objeto.

sizeof(int)
sizeof(short)
sizeof(long)
sizeof(unsigned)

Lo que C garantiza:

sizeof(char) = 1
sizeof(short) <= sizeof(int) <= sizeof(lon)
sizeof(unsigned) = sizeof(int)
sizeof(float) <= sizeof(double)

```
/*
  ej5.c
  Se indican los tamanios de las representaciones internas de
  algunos tipos de datos fundamentales.
*/
# include <stdio.h>
void main()
{
 char c;
 short s;
 int i;
 long l;
 float f;
 double d;
 printf ("Tipo char: %d  bytes\n", sizeof(c));
 printf ("Tipo short: %d  bytes\n", sizeof(s));
 printf ("Tipo int: %d  bytes\n", sizeof(i));
 printf ("Tipo long: %d  bytes\n", sizeof(l));
 printf ("Tipo float: %d  bytes\n", sizeof(f));
 printf ("Tipo double: %d  bytes\n", sizeof(d));
}
```

Formatos de lectura y escritura para enteros

d entero decimal.
o entero octal que no empieza con 0.
x entero hexadecimal que no empieza con x.
u entero decimal sin signo.

```
main( )
{
   int n;
   printf("Introducir una constante entera:\n");
   scanf("%d", &n);
   printf("El valor leido es: %d\n", n);
}
```

Los tipos numéricos de coma flotante

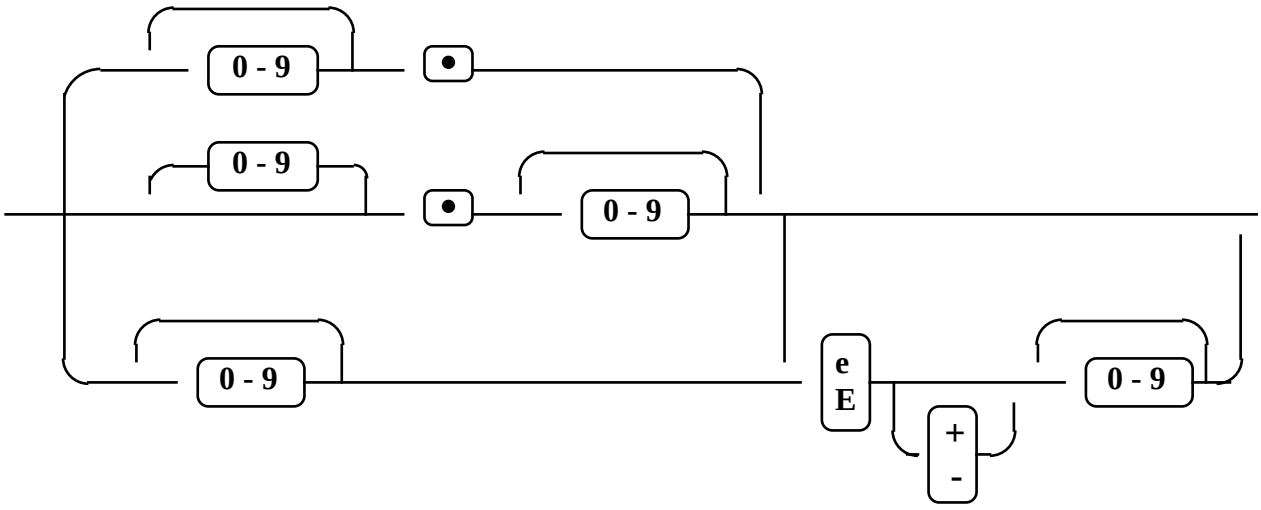
float double

Se usan para operar con números reales.

```
float  r, v[100], a[10][10], *p;
double       r, v[100], a[10][10], *p;
```

La precisión simple (float) se suele implementar con 32 bits (4 bytes), y la doble precisión con 64 bits (8 bytes).

Constantes reales



Ejemplos: **27.**
3.14
3.14e10
.612e-5
47e-3

En esta constante: **888.7777e-22**

888 es la parte entera
7777 es la parte fraccionaria
e-22 es la parte exponencial

Rango de valores y precisión

Son dependientes de la máquina. Unos valores típicos:

	float	double
precisión	**6 cifras**	**16 cifras**
rango	**10^{-38} a 10^{38}**	**10^{-38} a 10^{38}**
representación	**+/-.$d_1d_2...d_6x10^n$**	**+/-.$d_1d_2...d_{16}x10^n$**
	-38<=n<=+38	**-38<=n<=+38**

Los operadores aritméticos que se aplican a tipos de coma flotante son:

+ - * /

con el significado habitual.

```
/*
        ej6.c
        Precisión en el cálculo
*/
# include <stdio.h>
void main()
{
    float x=1.0;
    long i=1;
    x=x*(float)0.99937;
    while ( 1.0 != (1.0+x) )  {
        ++i;
        x=x*(float)0.99937;
    }
    printf ("%ld  %20.12f\n", i, x);
}
```

Las funciones matemáticas no están incorporadas en el lenguaje.

Se proporcionan en una biblioteca, que contiene funciones de uso frecuente:

sqrt( )
exp( )
log( )
sin( )
cos( )
tan( )
etc...

En la sección 3 del manual de Referencia de UNIX se describen las funciones de las bibliotecas.

Casi todas usan el tipo "double" para los parámetros.

Los programas que utilizan funciones de la biblioteca matemática, en el entorno UNIX, deben incluir ciertas declaraciones con la orden :

```
# include <math.h>
```

y deben ser compilados invocando a la librería:

```
cc fn.c -lm
```

```
{
        double x;
        scanf("%f", &x);
        printf("%20.12e\n", exp(x));
}
```

Formatos para lectura y escritura

Formatos para lectura con "scanf"

e	**número en coma flotante**
f	**equivalente a "e"**

Formatos para escritura con "printf"

e	**coma flotante en forma exponencial**
f	**coma flotante sin exponente**
g	**se elige el más corto entre "e" y "f"**

```
{
    float r;
    printf("Un número en coma flotante ?\n");
    scanf("%f", &r);
    printf("El valor leido es: %f\n", r);
}
```

El tipo " char "

Se usa para manejar caracteres aislados.

char c, s[10], *p;

Un "char" se almacena en un byte y lleva asociado un valor entero (el ordinal que indica su posición en la tabla de codificación ASCII, o la codificación en uso).

'A'

7	6	5	4	3	2	1	0
0	**1**	**0**	**0**	**0**	**0**	**0**	**1**

65

printf ("%d", 'a' + 'b' + 'c');

Los caracteres no imprimibles se pueden representar mediante secuencias de escape expresadas con caracteres alfabéticos o con valores numéricos en sistema octal.

Nombre del carácter	Escritura en C		Valor entero
nulo	'\0'	'\0'	0
retroceso	'\b'	'\10'	8
tabulador	'\t'	'\11'	9
salto de linea	'\n'	'\12'	10
salto de página	'\f'	'\14'	12
retorno de carro	'\r'	'\15'	13
comillas	'\"'	'\42'	34
apóstrofo	'\''	'\47'	39
barra invertida	'\\'	'\134'	92

La tabla de codificación ASCII

Izda. \ Derecha	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	np	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Significado de algunas abreviaturas

nul	nulo	nl	salto de linea
ht	tabulador horizontal	esc	escape
cr	retorno de carro	bs	retroceso
bel	campana	vs	tabulador vertical

Observaciones:

- **Los códigos de los caracteres 0 a 31 y 127 no son imprimibles**
- **Los códigos de las letras mayúsculas, los de las minúsculas y los de las cifras son contiguos entre sí.**
- **La diferencia entre una letra mayúscula y su correspondiente minúscula es 32.**

Ejemplos

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

{
 char s[80];   int lon;
 lon=0;
 while ( s[lon] != '\0' ) lon++;
}
```

Notar la diferencia entre el valor numérico que representa una cifra y el entero asociado al carácter:

el valor del carácter '3' no es 3

Distinguir entre 'a' y "a"

'a' es una constante de tipo char

"a" es una cadena constante,de dos caracteres:

'a' '\0'

(El carácter nulo indica siempre el final de una cadena)

Lectura y escritura de caracteres

getchar( ) putchar( )
scanf( ) printf( )

Son funciones de la biblioteca estándar.

En los programas que usan estas funciones y constantes útiles como EOF, se deben incluir ciertas definiciones y declaraciones poniendo la orden:

 # include <stdio.h>

getchar( ) **se obtiene el siguiente carácter del fichero de entrada estándar (el teclado, por defecto).**
El valor del carácter se devuelve como "int".
Si se encuentra el fin de fichero o si se produce error, se devuelve EOF.

```
#include <stdio.h>
main( )
 {      int c;
        c=getchar( );
        printf("El carácter leido es: %c\n", c);      }
```

putchar(c) **escribe el valor "char" de "c" en el fichero de salida estándar.**
Devuelve el valor "int" del carácter escrito.

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

Para la lectura y escritura de caracteres con formato por los dispositivos estándar, con

scanf( ) y printf( )

se usa el formato " %c ".

```
/*
        ej7.c
*/
# include <stdio.h>
void main( )
{
   int c;
   printf("Introducir un caracter:\n");
   scanf("%c", &c);
   printf("El caracter leido es: %c\n", c);
}

...

printf("%d", 'A');

printf("%c", 65);

printf("%c", 7);
```

Los operadores de asignación

=	+=	-=	*=	/=	%=
>>=	<<=	&=	^=	|=	

>>	**desplazamiento de bits a la derecha**
<<	**desplazamiento de bits a la izquierda**
&	**AND binario**
^	**OR exclusivo binario**
|	**OR inclusivo binario**

Abreviaturas

En C se admiten abreviaturas para simplificar la escritura de algunas sentencias de asignación:

x = x + 10; se puede escribir como x += 10;

Esta forma de abreviatura actúa con todos los operadores binarios.

Forma general:

variable = variable operador expresión
es lo mismo que
variable operador= expresión

k *= 3 + x es equivalente a k = k*(3 + x)
a[í] /= x + y es equivalente a a[i] = a[i] / (x + y)

Una expresión a la izquierda de un operador de asignación, se evalúa una sola vez:

a[++i] += 3;
no es equivalente a
a[++i] = a[++i]+3; (codificación indeseable porque el resultado depende del compilador)

Conversiones de tipo en las expresiones aritméticas

Cuando en una expresión se mezclan constantes y variables de distinto tipo, se convierten a un tipo único para evaluar la expresión.

Los valores de las variables almacenados en memoria no cambian. Se hacen copias temporales para evaluar la expresión.

short x;
int y;
...

para evaluar la expresión "x+y" :

- **el valor de "x" se convierte a un "int"**
- **el valor de la expresión será un "int"**

Reglas para la conversión automática en expresiones aritméticas como " x op y ":

1.
Todos los "char" y "short" se convierten a "int".
Todos los "unsigned char" o "unsigned short" se convierten a "unsigned".

2.
Si la expresión es de tipo mixto después del primer paso, el operando de menor rango se promueve al tipo del operando de mayor rango y se convierte toda la expresión en ese tipo.

int < unsigned < long < unsigned long < float < double

A este proceso se le llama promoción.

Ejemplos

```
char c;      double d;      float f;      int i;
long l;      short s;       unsigned u;
```

expresión	tipo
c - s / i	int
u * 3 - i	unsigned
u * 3.0 -i	double
f * 3 - i	float
c + 1	int
c + 1.0	double
3 * s * l	long

Los tipos de las constantes

El tipo de datos asociado a una constante depende de la máquina. Se asignan según la forma en que se escribe.

Lo normal es que no existan constantes de los tipos short, unsigned y float.

```
int:      0          77          5013
long:     0L         77L         5013L
double:   0.003      1.0         0.5013e-2
char:     'a'        'b'         'c'
cadena:   "esta es una constante de cadena"
```

Cuando una constante entera es demasiado grande para "int", se le asigna el tipo "long".

Las expresiones constantes se evalúan en tiempo de compilación.

Conversiones implícitas a través del signo " = "

Con el operador de asignación " = " también se producen conversiones de tipo implícitas.

El operando de la derecha se convierte al tipo del operando de la izquierda al hacerse la evaluación.

Supongamos: **int i; double d; char c;**

Al evaluarse la expresión: **d = i**

se convierte el valor de "i" a "double" y se asigna a "d", y éste será el valor de la expresión.

Si se trata de una degradación : **i = d**

se puede perder información porque se desecha la parte fraccionaria.

En los casos de degradación, el resultado depende de la máquina.

Al evaluarse: **c = i**

se descartan los bytes más significativos en "i".

Al evaluarse: **i = c**

si el valor ASCII de c es cero o mayor que cero, se llenan con ceros los bits de los bytes más significativos de "i".

Conversiones explícitas con el operador " casts "

Con el operador "casts" se pueden forzar conversiones de tipos indicándolas explícitamente.

(*t ipo* **)** *expresión*

Ejemplo: **int;**
...
.. (double) i ... **hace que se convierta el valor de "i" en uno de tipo "double" para la evaluación de la expresión.**

Estas conversiones no afectan al valor de la variable almacenado en memoria, que no se altera.

Los "casts" pueden aplicarse a cualquer expresión.

- **El operador "casts" es unario.**
- **Tiene la misma prioridad que otros unarios y asociatividad de derecha a izquierda.**

(float) i + 3 es equivalente a ((float) i) + 3
porque el operador "casts" tiene prioridad superior al operador "+"

Ejemplos:

```
(char) (3 - 3.14 * x)
k = (int)((int) x + (double) i + j )
(float) (x = 77)
```

Aquí se usa una variable entera ("i") para controlar un bucle y para realizar un cociente con parte fraccionaria:

```
int i;
for ( i=1; i<100; ++i )
        printf("%d / 2 es: %f\n", i, (float)i/2);
```

El tipo " pointer ". Los punteros.

En C, siempre se pueden manejar:

- **el valor asignado**
- **la dirección en memoria**

de cualquier variable

El operador " & " (dirección) proporciona la dirección de una variable

int n; **"n" representa al valor almacenado (un entero)**
"&n" representa la dirección de memoria donde se almacena el valor

Para manejar direcciones se usan los punteros (variables de tipo "pointer").

Los valores que toman las variables puntero son:

Direcciones de memoria.

"Apuntan" a dirección de memoria donde se almacena un valor del tipo base.

Para declarar un puntero se escribe:

tipo_base ***** *variable*

por ejemplo: **int *n;**

"n" es un puntero a valores de tipo "int"

- **los valores de "n" son direcciones**
- **los valores de la expresión " *n " son enteros**

Asignación de valores a punteros:

```
int *p, i;
p = 0
p = NULL   /* equivalente a  p=0  */
p = &i
p = ( int * ) 1501
```

El operador de indirección " * "

Proporciona el valor de la variable apuntada:

Si "p" es un puntero, " *p " representa al valor almacenado en la dirección " p ".

El operador " * " es unario y tiene asociatividad de derecha a izquierda.

con la declaración: `int a, b, *p;`

son equivalentes: `p = &a;` `b = *&a;` `b = a;` `b = *p;`

```
main( )
{
    float i=3.1416, *p;
    p=&i;
    printf("el valor de i es %f\n", *p);
    printf("la dirección de i es %d\n", p);      }
```

Asignación de un valor inicial: `int i=7, *p=&i;`

```
int n;
printf("Introducir un entero\n");
scanf("%d", &n);
printf("El valor leido es: %d\n", n);
```

Se puede apuntar a:

- **variables:** **int n;**
  &n

- **elementos de "array":** **int v[10], n;**
  &v[0] &v[1]
  &v[n]

No se puede apuntar a:

- **constantes:** **&7 ERROR**

- **"arrays":** **int v[10];**
  &v ERROR
  El nombre de un "array" es una constante

- **expresiones:** **&(n+1) ERROR**

- **variables de clase "register":** **register x;**
  &x ERROR

La aritmética de punteros

Es una de las características más eficaces del lenguaje.

Si la variable "p" es un puntero a un tipo determinado, la expresión " p+1 " da la dirección de máquina correspondiente a la siguiente variable de ese tipo.

Son válidas: **p + 1**
++p
p += i

La unidad de desplazamiento es propia del tipo base y depende de la máquina:

1 byte	**para**	**char**
4 bytes	**para**	**int**
8 bytes	**para**	**double**

int *p;
al evaluarse " ++p " el puntero pasa a apuntar al valor almacenado 4 bytes más adelante

Esta técnica se usa mucho para acceder a los elementos de las estructuras "array".

Interpretación de las declaraciones complejas

char *(*x( ))[ ]

A)
Hay que tener en cuenta las reglas de prioridad y asociatividad de los operadores: ( ) [ ] *

1)	**( ) función y**	**[ ] array**	**izquierda a derecha**
2)	*** puntero**		**derecha a izquierda**

B)
Se busca la variable y se aplican los operadores siguiendo las reglas de precedencia y asociatividad.

C)
Se hace la siguiente lectura:

( )	**función que devuelve ...**
[ ]	**array de ...**
*****	**puntero a ...**

Ejemplo:

char *(*x( )) [ ]; x es una funcíon que devuelve punteros
1 a "arrays" de punteros a caracteres.
2
3
4
5
6

Ejemplos

int i, *i, f( ), *f( ), (*p)( ), *p[ ];

int i **i es un entero**

int *p **p es un puntero a entero**
2 1

int f( ) **f es función que devuelve entero**
2 1

int *f( ) **f es función que devuelve apuntador a entero**
3 2 1

int (*p)( ) **p es puntero a función que devuelve entero**
4 1 2 3

int *p[ ] **p es array de punteros a entero**
3 2 1

No están permitidas todas las combinaciones posibles:

- **En algunas implementaciones, las funciones no pueden devolver "arrays", estructuras, uniones ni funciones. (Pueden devolver punteros a esos tipos).**
- **No se permiten "arrays" de funciones. (Se permiten "arrays" de punteros a funciones).**
- **Las estructuras y las uniones no pueden contener a una función. (Pueden contener punteros a funciones).**

Los operadores de "bits"

Actúan sobre expresiones enteras representadas como cadenas de dígitos binarios.

El resultado de las operaciones depende de la máquina.

Aquí se supone una representación de enteros en complemento a 2, de 32 bits, con bytes de 8 "bits" y codificación ASCII.

0	**00000000000000000000000000000000**
1	**00000000000000000000000000000001**
2	**00000000000000000000000000000010**
3	**00000000000000000000000000000011**
4	**00000000000000000000000000000100**
5	**00000000000000000000000000000101**
2147483647	**01111111111111111111111111111111**
-2147483648	**10000000000000000000000000000000**
-5	**11111111111111111111111111111011**
-4	**11111111111111111111111111111100**
-3	**11111111111111111111111111111101**
-2	**11111111111111111111111111111110**
-1	**11111111111111111111111111111111**

En complemento a 2, una cadena binaria y su complementaria suman -1.

Los operadores de "bits"

Op. lógicos:	**(unario) complemento de "bits"**	**~**
	operador AND de "bits"	**&**
	OR exclusivo de "bits"	**^**
	OR de "bits"	**|**
Op. de desplazamiento:	**desp. hacia la izquierda**	**<<**
	desp. hacia la derecha	**>>**

LOS OPERADORES

Puestos en orden de prioridad descendente

Operadores	Asociatividad
( ) [ ] -> . (miembro)	**izquierda a derecha**
~ ! ++ -- sizeof (tipo) **- (unario) *(indirección) &(dirección)**	**derecha a izquierda**
*** / %**	**izquierda a derecha**
+ -	**izquierda a derecha**
<< >>	**izquierda a derecha**
< <= > >=	**izquierda a derecha**
== !=	**izquierda a derecha**
&	**izquierda a derecha**
^	**izquierda a derecha**
|	**izquierda a derecha**
&&	**izquierda a derecha**
||	**izquierda a derecha**
?:	**derecha a izquierda**
= += -= *= ...	**derecha a izquierda**
, (operador coma)	**izquierda a derecha**

El operador de complemento "~" es unario, los demás son binarios.

Todos operan con variables enteras.

El operador complemento de "bits"

Invierte la representación de la cadena de "bits" de su argumento, convirtiendo los ceros en unos y los unos en ceros.

Si x, tiene la representación: 000...1100001
entonces, ~x tiene la representación: 111...0011110

Operadores binarios lógicos de "bits"

Operan de "bit" en "bit" con todos los "bits" de los operandos.

Semántica operando sobre un "bit"

x	y	x&y	x^y	x|y
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Operadores de desplazamiento de "bits" << >>

- **Son operadores binarios.**

- **Los operandos deben ser expresiones enteras.**

- **El operando derecho se convierte a "int".**
- **El operando derecho no puede ser negativo. Su valor no puede ser mayor que el número de "bits" de la representación del operando izquierdo.**
- **El tipo de la expresión es el del operando izquierdo.**

Desplazamiento hacia la izquierda: <<

La representación de "bits" del operando izquierdo se mueve hacia la izquierda el número de lugares que indica el operando de la derecha.
Los "bits" de orden menor se sustituyen por ceros.

char x='Z';

expresión	representación
x	01011010
x<<1	10110100
x<<4	10100000

Desplazamiento hacia la derecha: >>

La representación de "bits" del operando izquierdo se mueve hacia la derecha el número de lugares que indica el operando de la derecha.

Con expresiones enteras "unsigned", se introducen ceros.
Con expresiones "char", "short", "int" y "long", el resultado depende de la máquina. En algunas se introducen ceros, en otras se introducen bits de signo.

Máscaras

Una máscara es una expresión que se utiliza para extraer "bits" de otra variable o expresión.

La constante 1 tiene esta representación:

00000000000000000000000000000001

y se puede utilizar para conocer como es el "bit" de orden menor en una expresión de tipo "int":

```
int n;  . . .;  printf ( "%d", n&1 );
```

Para saber como está un "bit" determinado de un valor entero, se puede usar una máscara que tenga un 1 en esa posición y 0 en las demás.

La constante 0x10 sirve como máscara para el quinto "bit" contando desde la derecha:

0x10 se representa como 0...00010000

Si "x" tiene tipo "int",
la expresión (x&0x10) ? 1 : 0
vale 0 si el quinto "bit" en "x" es 0
vale 1 si el quinto "bit" en "x" es 1

La constante 0xff sirve como máscara para el "byte" de orden menor:

0xff se representa como 0...011111111

Si "x" tiene tipo "int",
la expresión x&0xff
tiene un valor con todos los "bytes" de orden mayor a cero y con el "byte" de orden menor igual que el correspondiente de "x".

Ejemplos

```
/*
        Función para escribir la representación binaria
        de un valor "int"
*/

bitprint ( n )
int n;
{
        int i, mascara;
        mascara=1;          /*  0...001  */
        mascara=mascara << 31;       /* se desplaza el "bit" 1 hacia
                                        el extremo más significativo */
        for ( i=1; i<=32; ++i ) {
                putchar( ((n&mascara)==0) ? '0' : '1' );
                n=n<<1;
        }
}
```

```
        /*
                Esta función cuenta los "bits" que tienen valor 1
        */

        cuenta_bits ( n )
        unsigned n;
        {
                int unos;
                for ( unos=0; n!=0; n>>1 ) if ( n&01 ) unos++;
                return (unos)
        }
```

Empaquetamiento

Los operadores de "bits" permiten empaquetar datos en los límites de los "bytes" para ahorrar espacio en memoria y tiempo de ejecución.

Ejemplo:

En una máquina con arquitectura de 32 "bits" se puede hacer que se procesen 4 caracteres en un ciclo.

Esta función los empaqueta:

```
pack ( a, b, c, d )
char a, b, c, d;
{
        int p;
        p=a;
        p=(p<<8)|b;
        p=(p<<8)|c;
        p=(p<<8)|d;
        return(p);   }
```

Esta función los desempaqueta:

```
unpack ( p )
int p;
{
        extern a, b, c, d;
        d=p&0xff;
        c=(p&0xff00)>>8;
        b=(p&0xff0000)>>16;
        a=(p&0xff000000)>>24;        }
```

Los tipos enumerativos

En la definición se establece un conjunto de valores posibles representados por identificadores.

enum colores { rojo, azul, amarillo } color;

- **La variable "color" es de tipo "enum colores".**
- **Los valores posibles son 3 y están identificados por esos nombres que se llaman enumeradores: rojo, azul, amarillo**

```
color=azul;
. . .
if ( color==amarillo ) . . .
```

El compilador asigna un valor "int" a cada enumerador, empezando en cero.

enum laborables { lunes,martes,miercoles,jueves,viernes } dia;

enumerador	valor
lunes	0
martes	1
miercoles	2
jueves	3
viernes	4

Se puede establecer otra asignación indicándola explícitamente:

enum {pino=7, chopo, alamo=3, abeto, abedul} arbol;

enumerador	valor
pino	7
chopo	8
alamo	3
abeto	4
abedul	5

Ejemplo

```
...
enum dia { lun, mar, mie, jue, vie, sab, dom };
...

/*
     Esta funcion devuelve el dia siguiente
*/
enum dia dia_siguiente ( d )
enum dia d;
{
return ( (enum dia) ( (( int)d+1 ) % 7 );
}
```

La utilidad de los tipos enumerativos se basa en dos aspectos:

- **Mejoran la claridad del programa, por su carácter autodocumentador**
- **Obligan al compilador a hacer test de consistencia de tipo.**

- 6 -

LAS SENTENCIAS DE CONTROL

- **Operadores de relación, igualdad y lógica.**
- **La sentencia vacía.**
- **La sentencia compuesta.**
- **Las sentencias "if", "if-else". El operador condicional "?"**
- **La sentencia "while"**
- **La sentencia "for". El operador coma ";".**
- **La sentencia "do-while"**
- **La sentencia "switch"**
- **La sentencia "break"**
- **La sentencia "continue"**
- **La sentencia "goto"**

OPERADORES DE RELACION, IGUALDAD Y LOGICA

Se usan para modificar el flujo de la ejecución.

Op. de Relación:	**<**	**menor que**
	>	**mayor que**
	<=	**menor o igual que**
	>=	**mayor o igual que**
Op de Igualdad:	**==**	**igual a**
	!=	**distinto de**
Op. de Lógica:	**!**	**negación**
	&&	**AND lógico**
	||	**OR lógico**

El operador " ! " es unario, los demás son binarios.

Todos operan sobre expresiones dando como resultado uno de dos:

0 FALSO
1 VERDADERO

En C:	**FALSO**	**es 0 ó 0.0**
	VERDADERO	**es distinto de 0,**
		o distinto de 0.0

Prioridad (orden descendente) y asociatividad de operadores de relación y algunos otros

!	**-(unario)**	**++**	**--**	**sizeof(tipo)**	**derecha a izquierda**
*****	**/**	**%**			**izquierda a derecha**
+	**-**				**izquierda a derecha**
<	**<=**	**>**	**>=**		**izquierda a derecha**
==	**!=**				**izquierda a derecha**
&&					**izquierda a derecha**
||					**izquierda a derecha**
=	**+=**	**-=**	***= ...**		**derecha a izquierda**
, (coma)					**izquierda a derecha**

Tabla semántica de los operadores de relación

valor de e1-e2	**e1<e2**	**e1>e2**	**e1<=e2**	**e1>=e2**
positivo	**0**	**1**	**0**	**1**
cero	**0**	**0**	**1**	**1**
negativo	**1**	**0**	**1**	**1**

Analizar el valor de la expresión: **3<j<5**

si j tiene el valor 7

(3<j)<5 ---> 1<5 ---> 1

```
j=7;
printf("%d\n", 3<j<5);
```

se escribirá: 1

Si se trata de chequear si el valor de "j" está comprendido entre 3 y 5, hay que escribir:

3<j && j<5

Los operadores de igualdad: == !=

- **son binarios**
- **actúan sobre expresiones**
- **producen un valor lógico:** **int 0** **FALSO**
  int 1 **VERDADERO**

Semántica:

valor de e1-e2	e1==e2	e1!=e2
cero	1	0
no cero	0	1

Para la expresión a==b,
a nivel de máquina se realiza: **a-b==0**

Hay que distinguir entre: **==** **y** **=**

if (i=1) .... **la expresión "i=1" siempre devuelve el valor VERDADERO**

if (i==1) ..... **la expresión "i==1" devuelve el valor VERDADERO si la variable "i" tiene el valor 1.**

Los operadores lógicos: ! && ||

- **Se aplican a expresiones**
- **El operador "!" es unario, los demás son binarios**
- **Producen uno de estos valores:** **int 0** **FALSO**
  int 1 **VERDADERO**

Semántica del operador " ! "

expresión	! expresión
cero	1
no cero	0

Diferencia con el operador de negación en Lógica:

no (no x) es x
! (! 5) es 1

Semántica de los operadores "&&" y "||"

valores de exp1	exp2	exp1 && exp2	exp1 || exp2
cero	cero	0	0
cero	no cero	0	1
no cero	cero	0	1
no cero	no cero	1	1

Al evaluar expresiones que son operandos de "&&" o de "||", el proceso de evaluación se detiene tan pronto como se conoce el resultado (verdadero o falso) final de la expresión:

en exp1 && exp2
si exp1 tiene el valor 0, no se evalúa exp2

en exp1 || exp2
si exp1 tiene el valor 1, no se evalúa exp2.

Ejemplo

```
i=0;
while (i++<3 && (c=getchar()) != EOF) acción...
```

la acción se ejecuta para los tres primeros caracteres del fichero, mientras que no se alcance EOF.

Cuando "i" toma el valor 3, la expresión "i++<3" toma el valor FALSO y deja de evaluarse la siguiente. Por tanto, en otra sentencia de lectura posterior, se obtendría el cuarto caracter, no el quinto.

La sentencia vacía

" ; "

```
for ( n=0; getchar() != EOF; ++n) ;
```

Sentencia compuesta

- **Una serie de sentencias encerradas entre llaves.**
- **Si hay declaraciones al principio, entonces se llama BLOQUE.**
- **Una sentencia compuesta se puede poner en cualquier lugar donde pueda ir una simple**

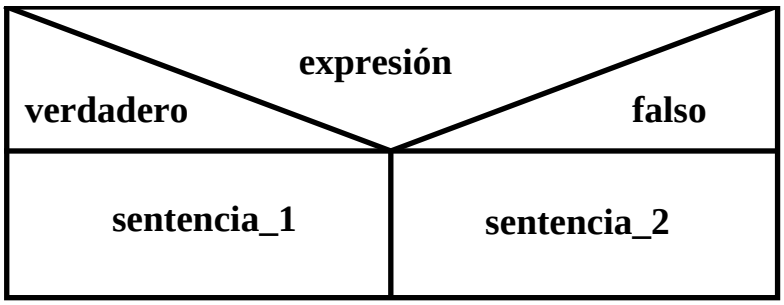
```
{
        int i;
        float p;
        p=1.0;
        for (i=1; i<=e; ++i) p=p*b;
        return(p);
}
```

Las sentencias " if " " if - else "

if (*expresion* **)** ***s****entencia1* **else** *sentencia2*

if (*expresion* **)** *sentencia*

Se evalúa la expresión. Si tiene un valor distinto de cero se ejecuta la "sentencia1" y se salta la "sentencia2". Si tiene el valor cero, se salta la "sentencia1" y se ejecuta la "sentencia2".



La expresión puede ser de cualquier tipo.

```
if ( x<min ) min=x;

------

if ( c==' ' ) {
        ++contador;
        printf("hallado otro espacio en blanco\n");
}

------

i=0;
if ( (c=getchar()) != EOF ) s[i++]=c;
else  s[i]='\0';
```

```
int n;
if ( n%2 == 0 ) printf("%d es par\n", n);
else printf("%d es impar\n", n);
```

```
/*
        ej8.c
*/
# include <stdio.h>
void main( )
{
char c;
c=getchar( );
if ( ( c>=65 ) && ( c<=90 ) )
        printf("Es letra mayuscula\n");
else {
                if ( (c>=97) && (c<=122) )
                        printf("Es letra minuscula\n");
                else
                        printf("No es un caracter alfabetico\n");
        }
}
```

Cuando el compilador lee varias sentencias " if " anidadas:

asocia cada " else " con el " if " más próximo.

```
if ( c == ' ')
    ++blancos;
else  if ( '0'<=c && c<='9' )
    ++cifras;
else  if ('a'<=c && c<='z' || 'A'<=c && c<='Z')
    ++letras;
else  if (c =='\n')
    ++saltos;
else
    ++otros;
```

El operador condicional " ? "

Es un operador ternario.

exp1 **?** *exp2* **:** *exp3*

Se evalúa la expresión "exp1". Si tiene valor distinto de cero, se evalúa la expresión "exp2" y su resultado será el valor de la expresión.

Si "exp1" tiene valor cero, se evalúa "exp3" y su resultado será el valor de la expresión.

La sentencia: **x = (y<z) ? y : z ;**

Tiene un efecto equivalente a:

```
if ( y < z )
        x=y;
else
        x=z;
```

Ejemplo:

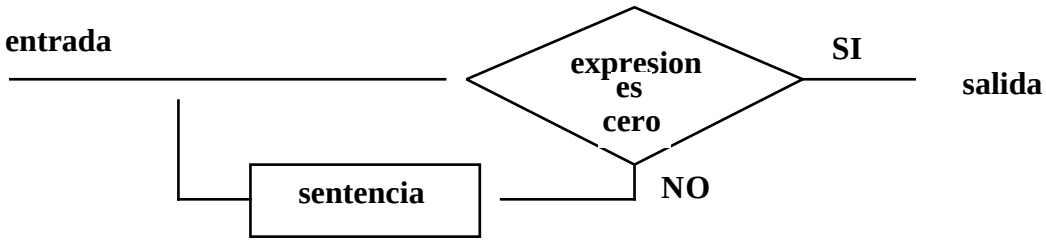
Escribir los elementos de un "array" de "int" poniendo cinco en cada linea :

```
#define TAM 100
...
int a[TAM];
...
for ( i=0; i<TAM; ++i )
        printf("%c%12d", ( i%5 == 0 ) ? '\n' : '\0', a[i] );
```

La sentencia " while "

while (*expresion* **)** *sentencia*

Se evalúa la "expersion". Si tiene valor distinto de cero (verdadero), se ejecuta la "sentencia" y el control vuelve al principio del ciclo. Cuando la "expresión" tiene el valor cero, termina la ejecución del bucle y se pasa a la sentencia siguiente.



La "sentencia" se puede ejecutar cero o más veces.

```
while ( (c=getchar() ) == ' ')
                ;       /* sentencia vacia */
```

```
int i=2, max=100;
while ( i<max) {
        printf("%d\n", i);
        i=i+2;
}
```

```
/*      contar el numero de caracteres almacenados en un
        fichero, incluyendo los saltos de linea
*/
int n=0;
while ( getchar() != EOF ) ++n;
```

```
# define TOTAL 100
. . .
float v[TOTAL];
int n;
n=0;
while ( n<TOTAL ) {
    v[n]=0.0;
    ++n;
}
```

```
while ( scanf("%d", &entero) != EOF )
            printf("El valor leido es: %d\n", entero);
```

La sentencia " for "

for (*expresion1* **;** *expresion2* **;** *expresion3* **)** *sentencia*

Su semántica es equivalente a la de:

```
expresion1;
while ( expresion2 ) {
        sentencia
        expresion3;
}
```

```
factorial=1;
for (i=1; i<=n; ++i) factorial *= i;
```

Pueden faltar alguna o todas las expresiones, pero deben permanecer los signos de punto y coma:

```
i=1; suma=0;
for ( ; i<=10; ++i) suma += i;
```

es equivalente a:

```
i=1; suma=0;
for ( ; i<=10 ; ) suma += i++;
```

Cuando falta la "expresion2", se le supone valor verdadero siempre, formandose un ciclo sin salida:

```
i=1;  suma=0;
for ( ; ; ) {
                suma += i++;
                printf("%d\n", suma);
}
```

```
{
   float v[100];
   int num=25, i;
   for ( i=0; i<num; ++i ) v[i]=0.0;
}
```

```
/*
   ej9.c
   escribir una tabla de verdad
*/
# include <stdio.h>
# define EXPRESION a&&b
# define T 1
# define F 0
void main( )
{
   int a, b;
   for ( a=F; a<=T; ++a)  {
         for ( b=F; b<=T; ++b)  {
               if ( EXPRESION) printf("T  ");
               else printf("F  ");
         }
   printf("\n");
   }
}
```

El operador coma " , "

Es un operador binario para expresiones :

expresion1 **,** *expresion2*

Se evalúa primero "expresion1" y después se evalúa "expresion2". La expresión coma tiene el valor y tipo de su operando derecho.

suma=0 , i=1

Se suele usar en las sentencias " for " porque permite hacer asignación múltiple de valores iniciales y procesamiento múltiple de índices.

```
for ( suma=0, i=1; i<=n; ++i ) suma += i;
printf("%d\n", suma);
```

```
for ( suma=0, i=1; i<=n; suma += i, ++i) ;
printf("%d\n", suma);
```

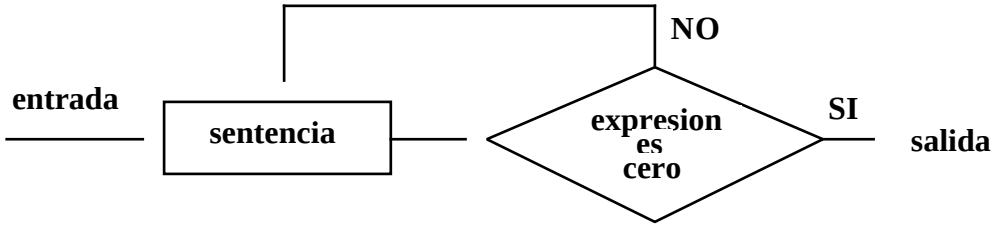
Para imprimir la suma de los primeros "n" enteros pares y la de los primeros "n" enteros impares, puede escribirse:

```
for (cont=0, j=2, k=1; cont<n; ++cont, j += 2, k += 2)  {
        suma_par += j;
        suma_impar += k;
}
printf("%7d %7d\n", suma_par, suma_impar);
```

La sentencia " do "

Es una variante de "while". En "do", el test se hace al final del bucle.

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




```
/*
     Se repite la orden de lectura mientras
     el valor introducido sea menor o igual
     que cero
*/
{
     int n;
     do {
          printf("Entero positivo ? ");
          scanf("%d", &n);
     }  while ( n<=0 );
}
```

```
int i=o, exponente;
float base, potencia;
. . .
potencia=1.0;
do  { potencia *= base; ++i } while ( i<exponente );
```

```
/*
        ej10.c
        Calculo del  numero "e" . (  x_n  = ( 1+ 1/n)^n  )
        Contiene funciones para obtener el valor absoluto de un
        numero y para obtener potencias de exponente entero
        positivo.
*/
# include <stdio.h>
void main()
{
   int n=1;
   float eant, e=2, precision;
   float abs(float), potencia(float, int);
   printf("Que precision?\n");
   scanf("%f", &precision);
   do  {
        eant=e;
        ++n;
        e = potencia (1.0+1.0/ (float) n, n );
   }
   while ( abs(e-eant) >= precision );
   printf("e= %f\n",e);
   printf("precision= %f\n", precision);
}

/*  Devuelve el valor absoluto de un numero real.  */
float abs(float r)
{
        if ( r < 0.0 ) return (-r);
        else return (r);
}

/*      Calcula potencias de exponente entero positivo      */
float potencia (float b, int e)
{
        int i;
        float p;
        p=1.0;
        for (i=1; i<=e; ++i) p=p*b;
        return(p);
}
```

La sentencia " switch "

Es una sentencia condicional múltiple que generaliza a la sentencia "if-else".

switch (*expresion_entera* **) {**

case *expresion_constante_entera* **:** *sentencia*
break;
case *expresion_constante_entera* **:** *sentencia*
break;
. . .
default : *sentencia*
}

1. **Se evalúa la "expresion_entera"**
2. **Se ejecuta la claúsula "case" que se corresponda con el valor obtenido en el paso 1. Si no se encuentra correspondencia, se ejecuta el caso "default"; y si no lo hay, termina la ejecución de "switch".**
3. **Termina cuando se encuentra una sentencia "break" o por caída al vacío.**

```
c=getchar( );
while ( c != '.' && c != EOF ) do {
        switch ( c ) {
              case 'c'    :   consultas( );
                              break;
              case 'a'    :   altas( );
                              break;
              case 'b'    :   bajas( );
                              break;
              case 'm'    :   modificaciones( );
                              break;
              default     :   error( );
        }
c=getchar( );
}
```

```
/*
        ej11.c
        Se leen caracteres por STDIN contando la cantidad de
        cifras, espacios (tabulador o salto de linea) y otros.
*/
#include <stdio.h>
void main()
{
        int c, i;
        int blancos=0;
        int otros=0;
        static int cifs[10]={0,0,0,0,0,0,0,0,0,0};
        while ( (c=getchar()) != EOF )
                switch (c) {
                        case '0':
                        case '1':
                        case '2':
                        case '3':
                        case '4':
                        case '5':
                        case '6':
                        case '7':
                        case '8':
                        case '9':
                                        cifs[c-'0']++;
                                        break;
                        case ' ':
                        case '\n':
                        case '\t':
                                        blancos++;
                                        break;
                        default:
                                        otros++;
                                        break;
                }
        printf ("cifras: ");
        for (i=0; i<10; i++) printf("%d",cifs[i]);
        printf("\nblancos: %d,  otros: %d\n", blancos, otros);
}
```

La sentencia " break "

- **Para forzar la salida inmediata de un bucle, saltando la evaluación condicional normal.**
- **Para terminar una claúsula "case" de sentencia "switch" y finalizar su ejecución.**

```
# include <math.h>
...
while ( 1 ) {
        scanf("%lf", &x);
        if ( x<0.0 )  break;
        printf("%f\n", sqrt(x));
}
```

```
switch ( c ) {
      case '.'    :   mensaje("fin");
                      break;
      case '?'    :   ayuda( );
                      break;
      default     :   procesar (c);
}
```

La sentencia " continue "

Se detiene la iteración actual de un bucle y se inicia la siguiente de inmediato.

```
for ( i=0; i<TOTAL ;++i ) {
        c=getchar( );
        if ( '0'<=c && c<'9' ) continue;
        procesar(c);
}
```

El caso:

```
for ( exp1; exp2; exp3 ) {
. . .
continue
. . .
}
```

Es equivalente a:

```
exp1;
while ( exp2 ) {
. . .
goto siguiente
siguiente:  exp3;
        }
```

```
/*    ej12.c */
# include <stdio.h>
void main ( )
{
      char fin, c;
      fin=0;
      while ( !fin ) {
            c=getchar( );
            if ( c == '$') {        fin=1;
                                    continue;
            }
            putchar(c+1);
      }
}
```

La sentencia " goto "

Bifurcación incondicional a una sentencia etiquetada arbitrariamente dentro de la función.

La sentencia "goto" y su sentencia etiquetada correspondiente deben estar en el cuerpo de la misma función.

```
            . . .
            {
                    . . .
                    goto error;
                    . . .
            }
error:      mensaje( );
```

Su uso está desaconsejado porque suele romper la estructuración de los algoritmos.

CONVIENE NO USARLA MUCHO

- 7 -

LAS FUNCIONES

- **El diseño descendente. Las funciones. La función "main".**
- **Definición o declaración de funciones.**
- **El tipo "void".**
- **Valor de retorno.**
- **Llamada a una función.**
- **La transmisión de valores. Paso por valor.**
- **Reglas de alcance. Clases de almacenamiento.**
- **Recursividad.**

Las funciones

Un programa escrito en C está formado por una o varias funciones.

Cada función expresa la realización del algoritmo que resuelve una de las partes en que se ha descompuesto el problema completo en cuestión.

> **En el método de análisis descendente, un problema complejo se descompone sucesivamente en partes cada vez más simples hasta que resulta trivial su programación.**
>
> **Con el lenguaje C, cada una de esas partes se resuelve con una función.**

Formalmente, una función tiene un nombre y está formada por un conjunto de sentencias que se ejecutan devolviendo un valor al medio que la invoca.

Todo programa debe contener la función "main", que es la invocada desde el sistema operativo cuando comienza la ejecución del programa.

> **La función "main" controla la ejecución del resto del programa.**
>
> **La función "main" devuelve un valor entero al sistema cuando termina la ejecución. Dicho valor se puede controlar con la función de biblioteca "exit".**

```
# include <stdio.h>
# include <stdlib.h>
/*      ej13.c */
void main()
{       int b, e, p, i;
        printf("Calculo de potencias de exponente entero > 0\n");
        printf("Introducir base y exponente\n");
        scanf("%d %d", &b, &e);
        if ( e<=0 ) exit(-1);
        else {  for ( p=1, i=1; i<=e; ++i ) p *=b;
                        printf("%d elevado a %d es %d\n", b, e, p);
                        exit(0);
        }
}
```

Las funciones:
- **se definen**
- **se asocian a un tipo de datos**
- **se invocan**
- **se ejecutan**
- **devuelven un valor**

Definición de una función

Las funciones que componen un programa pueden definirse en ficheros diferentes o en el mismo fichero.

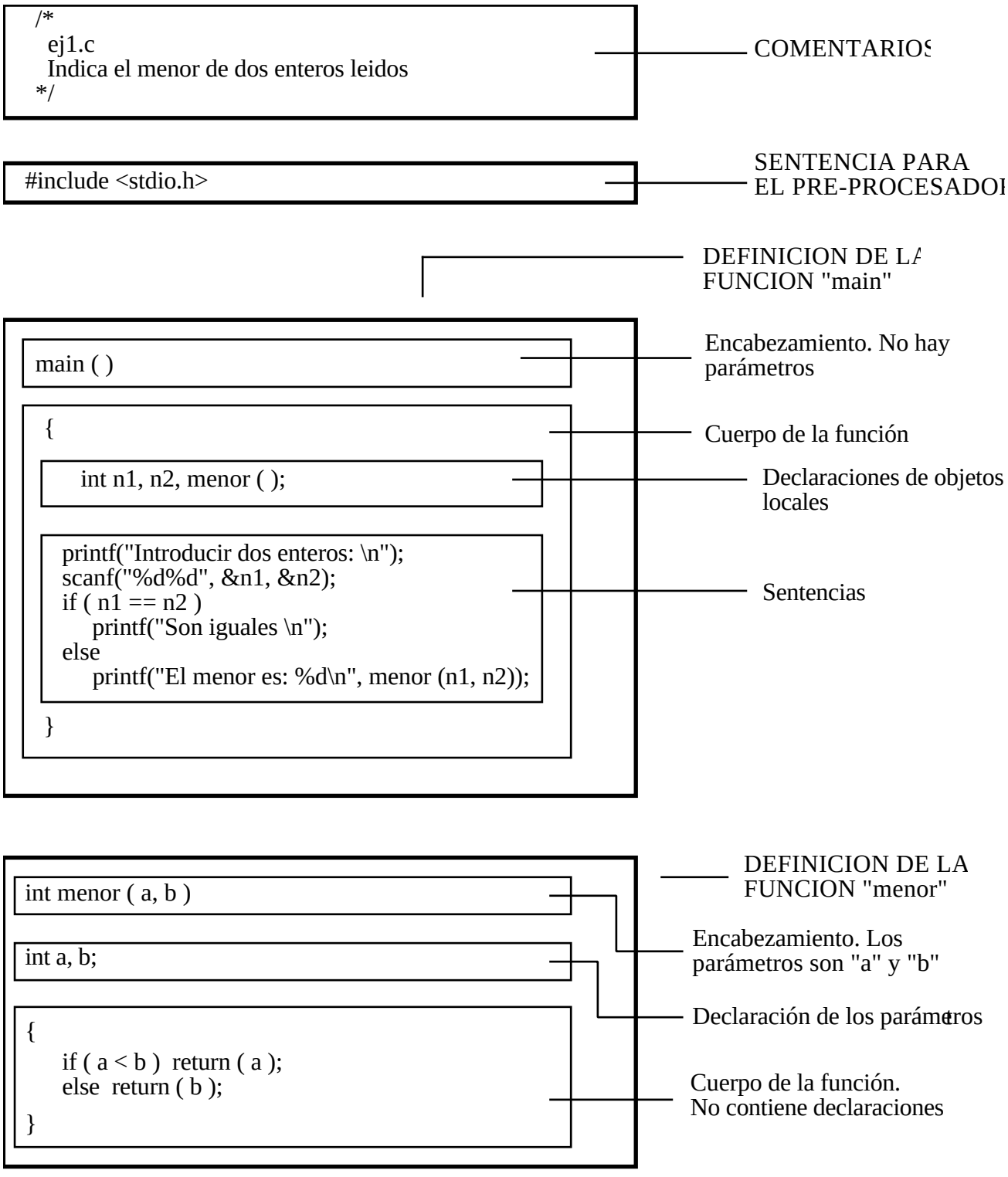
tipo nombre **(** *lista_de_parámetros* **)**
{
cuerpo
}

Sus elementos, con más detalle:

Encabezamiento . . . . . . *tipo nombre* **(p1, p2, p3,...)**
Declaración de parámetros . . *tipo* **p1, p2, p3...;**
{
Cuerpo con: Declaraciones . . . . . *tipo* **v1,v2,v3,...;**
Sentencias . . . . . . *sentencia*
sentencia
. . .
return (*expresión***)**
}

- **Si no se declara tipo, adquiere tipo "int".**
- **La lista de parámetros permite la transmisión de valores entre funciones.**
- **Hay que declarar el tipo de cada parámetro. Si no se hace, adquieren el tipo "int".**
- **El cuerpo puede contener declaraciones que deben estar al principio de los bloques.**
- **No se pueden definir otras funciones dentro de una función**

```
/*
ej14.c
Captura el primer argumento, se convierte a entero ("int")
y se multiplica por 2.

        Uso:  a.out cifras
*/

# include <stdio.h>
void main(int argc, char *argv[ ])
{
        int num(char *);
        if (argc == 1)
        printf("Uso: a.out cifras\n");
        else printf("El doble es: %d\n", 2*num(argv[1]));
}


int num (char s[ ])
{
        int i,n;
        n=0;
        for (i=0; s[i] >= '0' && s[i] <= '9'; ++i)
                n=10*n+s[i]-'0';
        return(n);
}
```


/*
ej1.c
Indica el menor de dos enteros leidos
*/
COMENTARIOS
#include <stdio.h>
SENTENCIA PARA
EL PRE-PROCESADOR
DEFINICION DE LA
FUNCION "main"
main ( )
Encabezamiento. No hay
parámetros
{
Cuerpo de la función
int n1, n2, menor ( );
Declaraciones de objetos
locales
printf("Introducir dos enteros: \n");
scanf("%d%d", &n1, &n2);
if (n1 == n2)
printf("Son iguales \n");
else
printf("El menor es: %d\n", menor (n1, n2));
Sentencias
}
int menor (a, b)
DEFINICION DE LA
FUNCION "menor"
int a, b;
Encabezamiento. Los
parámetros son "a" y "b"
Declaración de los parámetros
{
if (a < b) return (a);
else return (b);
}
Cuerpo de la función.
No contiene declaraciones

En el standard ANSI, se pueden declarar los tipos de los argumentos.

Ello permite que el compilador realice tests de consistencia entre los parámetros actuales y los formales.

```
main( )
{
        int i, n;
        float r, a[10];
        int funcion(int, float, float[ ]);
        n=5;
        r=3.14;
        for ( i=0; i<n; ++i ) a[i]=(float)i+r;
        funcion(n, r, a);
}


int funcion ( int n, float x, float v[ ] )
{
        int i;
        printf("primer parametro: %d\n", n);
        printf("segundo parametro: %12.4f\n", x);
        for (i=0; i<n; ++i)  printf("v[%d]: %12.4f\n", i, v[i]);
}
```

El tipo " void "

Se usa en los casos en que no se pretende que la función devuelva un valor.

```
void  saltar_lineas ( n )
int n;
{
        int i;
        for ( i=1; i<=n; ++i ) printf("\n");
}
```

Cuando se declara tipo "void":

- **Se puede invocar a la función como si fuera una sentencia**

  ```
  satar_lineas (10);
  ```

- **No se puede poner la función en una expresión donde se requiera un valor.**

Cuando no se declara tipo explícitamente:

- **El sistema asigna el tipo "int"**
- **La función devuelve un valor aleatorio aunque no contenga una sentencia "return"**

El valor de retorno

El valor que devuelve una función al medio de llamada se puede determinar con la sentencia "return".

return;
return (*expresión* **);**

Semántica:

concluye la ejecución de la función y se devuelve el control al medio que la invocó. El valor de la expresión se convierte al tipo de la función y se devuelve como valor de la función.

Se puede omitir la sentencia "return". Entonces:

el control se devuelve cuando se alcanza la llave "}" que delimita el cuerpo de la función.

El valor devuelto queda indefinido si falta la sentencia "return" o si aparece sin indicación de expresión.

- **En algunas implementaciones las funciones no pueden devolver**

arrays
structs
unions
funciones

pero pueden devolver apuntadores a esos objetos

- **No se permiten los "arrays" de funciones, pero sí los "arrays" de apuntadores a funcion.**

- **Una estructura (struct) no puede contener una función, pero si puede contener apuntadores a funciones.**

int i, *ip, f(), *fip(), (*pfi)();

i	**entero**
ip	**puntero a entero**
f	**funcion que devuelve un entero**
fip	**función que devuelve puntero a un entero**
pfi	**puntero a función que devuelve entero (la indirección respecto a un puntero a función, es una función que devuelve un entero).**

Llamada a una función

Una función se puede usar de dos formas.

• Invocándola normalmente:

escribiendo su nombre seguido de una lista de parámetros escritos entre paréntesis.

```
c = getchar ( );
m = raiz ( m );
s = menor ( lista, num );
peso = volumen ( a, b, c) * densidad (material);
```

• Tomando su dirección:

Cuando el nombre de una función aparece dentro de una expresión en un lugar que no es el que le corresponde según el formato normal indicado antes, se genera un puntero a la función.

Esto permite pasar una función a otra.

```
main()
{
    int f( ), g( );
   ...
    ...
    g ( f );
    ...
}

int g(funcion)
int (*funcion) ( );
{
    ...
    (*funcion) ( );
    ...
}
```

Semántica de la llamada a una función

1. **Se evalúa cada expresión de la lista de argumentos.**

2. **Al principio del cuerpo de la función, se asigna el valor de cada expresión a su parámetro formal correspondiente.**

3. **Se ejecuta el cuerpo de la función.**

4. **Si se ejecuta una sentencia "return", el control regresa al medio que hizo la llamada.**

5. **Si la sentencia "return" incluye una expresión, el valor de la expresión se convierte (si es necesario) al tipo de la función, y ese valor se devuelve al medio que hizo la llamada.**

6. **Si no hay una sentencia "return", el control regresa al medio que hizo la llamada cuando se llega al final del cuerpo de la función.**

7. **Si se ejecuta una sentencia "return" que no tiene expresión o si no hay sentencia "return", entonces no se devuelve ningún valor determinado al medio de llamada (el valor de retorno queda indefinido).**

8. **Todos los argumentos pasan con una "llamada por valor".**

```
/*
        ej15.c
        Paso de funcion por lista de parametros
*/
# include <stdio.h>
# include <stdlib.h>
# include <math.h>
void main ( )
{
        float r, rad;
        float raiz(float (*f)(float), float);
        float estandar(float), aprox(float);
        char opcion;
        printf("Introducir radicando y opcion(1/2)\n");
        scanf("%f %c", &rad, &opcion);
        switch ( opcion ) {
                case '1' :      r=raiz(estandar, rad);
                                break;
                case '2' :      r=raiz(aprox, rad);
                                break;
                default :       printf("opcion incorrecta\n");
                                exit(-1);
        }
        printf("radicando: %f  opcion: %c  raiz: %f\n",
                        rad, opcion, r);
}

float raiz( float (*f)(float), float v )
{       return( (*f)(v) );    }

float estandar(float rad)
{       return( sqrt(rad) );         }

float aprox(float rad)
{
        float r1, r2=1.0;
        do {
                r1=r2;
                r2=( rad/r1 + r1 )/(float)(2);
        } while ( abs(r2-r1) > 1e-3*r1 );         return(r2);
}
```

La transmisión de valores. El paso por valor.

Al llamar a una función se le pueden pasar pasar valores por la lista de parámetros.

Los parámetros se pasan "por valor":

> **Se evalúan las expresiones de la lista de parámetros de paso y los valores resultantes se asignan a sus parámetros formales correspondientes.**
>
> **Las modificaciones en los valores de los parámetros formales de la función, no afectan a las variables del medio que hizo la llamada.**

Si se quiere conseguir el efecto lateral de afectar a las variables del medio de llamada, hay que inducir el paso por referencia, pasando direcciones de variables en vez de sus valores:

> **Entonces, se usan los parámetros formales desreferenciados dentro de la función.**

Paso por valor

```
{
    int n,m;
    n=5;
    printf("%d\n", n);
    m=doble( n );
    printf ("%d\n", n);
}

int doble(i)
int i;
{
    i *= 2;
    return ( i );
}
```

Paso por referencia

```
{
    int n,m;
    n=5;
    printf("%d\n", n);
    m=doble( &n );
    printf ("%d\n", n);
}

int doble(i)
int *i;
{
    *i *= 2;
    return( *i );
}
```

Variables locales

Todas las variables declaradas en el cuerpo de una función son "locales" a esa función:

> **Sólo existen durante la ejecución del bloque en el que se han declarado.**

Variables globales

Las variables que no han sido declaradas ni como argumentos, ni en el cuerpo de una función son "globales" a la función y deben estar definidas externamente:

> **Existen durante la ejecución de todo el programa.**

```
int g;                  /*  "g" es variable global  */

main( )
{
 int a, b;
 . . .
 . . .
 {
        float b, x, y;
        . . .           /* se conoce a "int a", pero no
        . . .           a "int b", que ha quedado
                           enmascarada.  */
 }

 {
        unsigned a;
        char c, d;
        . . .           /* se conoce a "int b", pero no a
        . . .           "int a".
                        No se conoce a "x" ni a "y"  */
 }
 ...
 ...                    /* se conoce a "int a" y a "int b".
 ...                    No se conoce a "x", "y", "c"  */
 ...
}
```

La variable "g" es conocida en todos los bloques.

Clases de almacenamiento

Las variables y las funciones tienen dos atributos:

- **tipo**
- **clase de almacenamiento**

Hay cuatro clases de almacenamiento:

- **auto**
- **extern**
- **register**
- **static**

Variables de clase "auto"

Las variables declaradas dentro del cuerpo de las funciones son automáticas por omisión.

Son equivalentes estas dos formas:

```
{
    char c;
    int i, j, k;
    ...
}
```

```
{
    auto char c;
    auto int i, j, k;
    ...
}
```

Tienen almacenamiento dinámico:

- **Cuando se entra en el bloque, se reserva memoria automáticamente y se libera al abandonarlo.**

- **Sólo existen mientras dura la ejecución del bloque donde han sido declaradas.**

Variables de clase "extern"

A las variables declaradas fuera de una función se les asigna almacenamiento permanente. Son de clase "extern".

Se consideran globales para todas las funciones declaradas después de ellas.

Existen durante toda la vida del programa.

Se pueden usar para compartir valores entre funciones, pero se recomienda no hacerlo. Es preferible canalizar la transmisión de valores por la lista de parámetros siempre para evitar efectos laterales.

La especificación "extern" se puede utilizar para indicar que la variable se espera de otro contexto, aunque la declaración no esté contenida en el mismo fichero.

Variables de clase "register"

Se almacenan en registros de memoria de alta velocidad si es posible física y semánticamente.

Se utiliza para las variables de uso muy frecuente (control de bucles, por ejemplo).

Su uso es muy limitado porque suele haber muy pocos registros de alta velocidad.

Su declaración es una sugerencia para el compilador.

Variables de clase "static"

Locales estáticas

El almacenamiento "static" permite que una variable local retenga su valor previo cuando se entra de nuevo en el bloque donde reside.

Globales estáticas

Las variables externas estáticas proporcionan una propiedad de privacidad muy interesante para la programación modular:

- **Son de alcance restringido al resto del fichero fuente en el que se han declarado.**
- **Se puede conseguir que sean privadas a un conjunto de funciones**

Funciones estáticas

El almacenamiento "static" se puede aplicar a funciones.

Las funciones estáticas son visibles sólo dentro del fichero en el que se han declarado (útil para desarrollar módulos privados).

La recursividad

En C, las funciones tienen la propiedad de poderse invocar a sí mismas.

```
/*
      ej16.c
      Calculo de potencias con algoritmo recursivo.
*/
potencia(int b, int e)
{
      if (e == 0) return(1);
      else return(b*potencia(b, e-1));
}
```

```
/*
      ej17.c
      Calculo del factorial con algoritmo recursivo
*/
factorial ( int n )
{
      if ( (n==1) || ( n==0) ) return(1);
      else  return ( n*factorial(n-1) );
}
```

- 8 -

LOS TIPOS DE DATOS ESTRUCTURADOS.

VISION GENERAL.

- **array**
- **string**
- **struct**
- **bits**
- **union**

Los tipos de datos estructurados

Pueden contener más de un componente simple o estructurado a la vez.

Se caracterizan por:

- **El tipo o los los tipos de los componentes**
- **La forma de la organización**
- **La forma de acceso a los componentes**

array **(arreglo homogéneo de acceso directo)**

string **(cadena de caracteres)**

struct **(estructura heterogénea)**

bits **(campos de bits)**

union **(compartir memoria)**

El tipo " array "

Una organización de datos caracterizada por:

- **Todos los componentes son del mismo tipo (homogéneo).**
- **Acceso directo a sus componentes. Todos sus componentes se pueden selecionar arbitrariamente y son igualmente accesibles**

```
int v[100], a[100][100];

v[10]=5;          v[i]=0;          v[i++]=3;

for ( i=0; i<100; i++ ) v[i]=0;

for ( i=0; i<100; i++ )
   for ( j=0; j<100; j++ ) a[i][j]=0;
```

El tipo string (cadena de caracteres)

Una cadena de caracteres es:

Un "array" unidimensional de tipo "char" que termina con el caracter nulo ('\0').

```
char lin[80];

'c' 'a' 'd' 'e' 'n' 'a' '\0'
```

El lenguaje no tiene operadores para manejar cadenas de caractes.

Se manejan con funciones de biblioteca

El tipo "struct"

Las estructuras ("struct") son organizaciones de datos cuyos miembros pueden ser de tipos diferentes.

```
struct naipe {
        int valor;
        char palo;
};
struct naipe carta, c;
...
carta.valor=10;
carta.palo='e';
c=carta;
```

```
enum palos {oros, copas, espadas, bastos};
struct naipe {
  int valor;
  enum palos palo;
};
carta.palo=espadas;
```

Campos de bits

Son miembros consecutivos de una estructura ("struct") que contienen un número constante no negativo de bits.

```
struct byte {
    unsigned    bit0:  1,
                bit1:  1,
                bit2:  1,
                bit3:  1,
                bit4:  1,
                bit5:  1,
                bit6:  1,
                bit7:  1;
};
byte c;
c.bit0=1;
c.bit1=0;
c.bit2=0;
```

```
struct naipe {
        unsigned valor     : 4;
        unsigned palo      : 2;
}
struct naipe carta;
carta.valor=9;
carta.palo=2;
```

El tipo " union "

Las uniones son estructuras ("struct") cuyos miembros comparten memoria.

```
struct octetos {
        unsigned    byte0: 8, byte1: 8, byte2: 8, byte3: 8;
};

struct bits {
        unsigned
          bit0    :1, bit1    :1, bit2    :1, bit3    :1,
          bit4    :1, bit5    :1, bit6    :1, bit7    :1,
          bit8    :1, bit9    :1, bit10   :1, bit11   :1,
          bit12   :1, bit13   :1, bit14   :1, bit15   :1,
          bit16   :1, bit17   :1, bit18   :1, bit19   :1,
          bit20   :1, bit21   :1, bit22   :1, bit23   :1,
          bit24   :1, bit25   :1, bit26   :1, bit27   :1,
          bit28   :1, bit29   :1, bit30   :1, bit31   :1;
};

union palabra {
        int                 x;
        struct octetos      y;
        struct bits   z;
} w;
w.x=7;
w.y.byte3='a';
w.z.bit5=1;
```

```
union equivalencia {
        int i;
        char c;
};
union equivalencia x;
x.i=5;
x.c='A';
```

- 9 -

"ARRAYS", CADENAS Y PUNTEROS.

- **El tipo "array". "Arrays" unidimensionales. Declaración. Descriptor de elemento.**
- **Inicialización de "arrays".**
- **El índice.**
- **Punteros y "arrays". La aritmética de punteros.**
- **Arrays multidimensionales.**
- **Paso de "arrays" a funciones.**
- **Cadenas de caracteres.**
- **Funciones de biblioteca para manejar cadenas.**
- **"Arrays" de punteros. Programa para ordenas palabras . Paso de argumentos a la función "main".**
- **"Arrays" no uniformes.**

El tipo " array "

Organización de datos que se caracteriza porque todos los componentes:

- **Son del mismo tipo (homogénea).**
- **Se pueden acceder arbitrariamente y son igualmente accesibles (acceso directo).**

Declaración: *tipo identificador* **[** *tamaño* **];**

```
int v[10];
```

- **El "tamaño" tiene que ser una expresión entera positiva que indica el número de elementos del arreglo.**

- **Los elementos se identifican escribiendo el nombre de la variable y un índice escrito entre corchetes:**

v[0] v[1] v[2] . . . v[8] v[9]

límite inferior del índice = 0
límite superior del índice = tamaño-1

El uso de constantes simbólicas facilita la modificación de los programas:

```
#define TAM 10
int a[TAM];
```

Pueden tener clase de almacenamiento:
- **automática**
- **estática**
- **externa**

No está permitida la clase : • **register**

Inicialización de "arrays"

Asignación de valor en la sentencia de declaración.

- **Se pueden inicializar los de clase "static" o "extern".**
- **No se pueden inicializar los de clase "auto".**

Sintaxis:

clase tipo identificador **[***tamaño***] = {***lista de valores* **};**

```
static float v[5]={0.1, 0.2, -1.7, 0.0, 3.14};
```

• Si se escriben menos valores, el resto de elementos se inicializan a cero.

```
static int a[20] = { 1,2,3,4,5,6 };
```

• Si no se declaró tamaño, se deduce de la inicialización.

```
static int a[ ] = { 0,0,0 };
```

El índice

Los elementos de un "array" responden a un nombre de variable comun y se identifican por el valor de una expresión entera, escrita entre corchetes, llamada índice.

Modificando el valor del índice se puede recorrer toda la estructura.

```
# define TAM 100
int i, a[TAM], suma;
suma=0;
for ( i=0; i<TAM; ++i )  suma += a[i];
```

El índice debe tomar valores positivos e inferiores al indicado en la declaración.

Si el índice sobrepasa el límite superior, se obtiene un valor incorrecto porque se está haciendo referencia a una posición de memoria ajena al "array".

Los desbordamientos no se suelen avisar durante la ejecución.

Punteros y "arrays"

Los punteros y los "arrays" se usan de forma parecida para acceder a las direcciones de memoria.

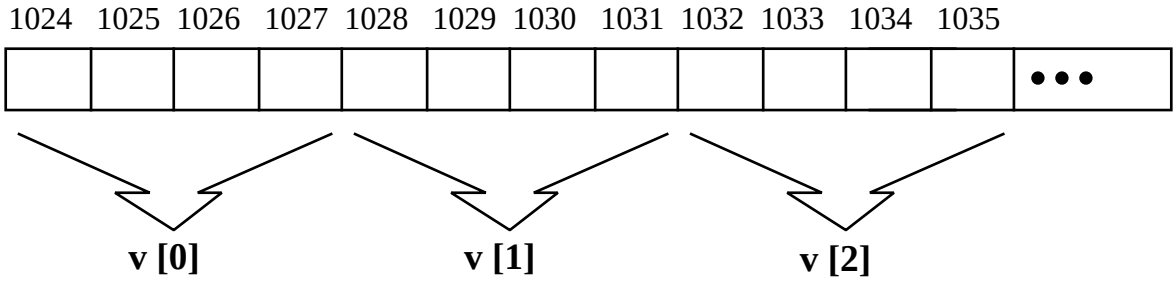
El nombre de un "array" es un puntero constante. (Representa una dirección constante).

Es la posición en memoria donde comienza el "array" (la que ocupa el primer elemento).

La declaración de un "array" produce:

- **La asignación de una dirección base**
- **La reserva de un espacio suficiente para almacenar los valores de todos los elementos.**

 # define TAM 3
int v [TAM]



Un valor de tipo "int" ocupa 4 bytes en arquitecturas de 32 bits

- **El nombre del "array" ("v") representa la dirección base.**

- **Un descriptor de elemento representa un valor de tipo "int" (v[1]).**

Según el ejemplo anterior:

Si: **int *p;**

Son equivalentes: **p = v;** **p = &v[0];**
asignan 1024 a "p"

Son equivalentes: **p = v+1;** **p = &v[1];**
asignan 1028 a "p"

Aritmética de punteros

Si "p" es un puntero a un tipo determinado, entonces la expresión "p+1" representa la dirección en memoria del siguiente valor del mismo tipo. El desplazamiento depende del tipo:

es 1 byte para char
es 4 bytes para int

```
for ( p=v; p<&v[TAM]; ++p ) *p=0;

for ( i=0; i<TAM; ++i ) *(v+i) = 0;

for ( p=v; p<&v[TAM]; ++p ) suma=suma+ *p;

for ( suma=0, i=0; i<TAM; ++i ) suma += *(v+i);
```

Como el nombre de "array" es una constante, las expresiones siguientes son incorrectas:

v=p **++v** **v += 2**

"arrays" multidimensionales

Son "arrays" de "arrays".

```
int a[3][5];
int b[3][3][3];
```

Los elementos se almacenan en posiciones de memoria contiguas desde la dirección base (primer elemento).

La transformación entre el puntero y los índices del arreglo se llama función de transformación del almacenamiento.

Para el "array a" la función de transformación se especifica considerando que:

a[i][j] es equivalente a *(&a[0][0]+5*i+j)

a[1][2] es equivalente a *(&a[0][0]+5+2) *(&a[0][0]+7)

a[0][0] a[0][1] a[0][2] a[0][3] a[0][4] a[1][0] a[1][1] a[1][2] ...

Se puede pensar que el índice que varía más deprisa es el que está más a la derecha; o que las matrices se almacenan por filas.

a+3 (dirección del cuarto elemento) equivale a &a[0][3]

Es más rápido el acceso a un elemento si se utiliza la aritmética de punteros, porque se evita el cálculo de la función de transformación.

Paso de "arrays" a funciones

Cuando se pasa un "array", pasa su dirección base por valor. Los elementos del "array" no se copian.

```
int sumar ( v, n )
int v[ ];
int n;
{
        int i, s=0;
        for ( i=0; i<n; ++i )
                s += v[i];
        return (s);
}
```

```
int sumar (v, n )
        int *v;
        . . .
        {
                . . .
                . . .
                . . .
                . . .
        }
```

llamada	lo que se calcula
sumar(a, 100);	a[0]+a[1]+...+a[99]
sumar(a, 40);	a[0]+a[1]+...+a[39]
sumar(a, m);	a[0]+a[1]+...+a[m-1]
sumar(&a[5], m-5);	a[5]+a[6]+...+a[m-1]
sumar(v+7, 2*k);	a[7]+a[8]+...+a[2*k+6]

Otra forma de escribir la función:

```
int sumar (v, n)
int v[ ], n;
{
        int i, s;
        for ( i=0, s=0; i<n; ++i ) s += *(v+i);
}
```

Si un "array" multidimensional es parámetro formal de una función, deben declararse los tamaños de todas sus dimensiones excepto el de la primera, para que se pueda calcular la función de transformación de almacenamiento.

```
int a[3][5], n, m;
. . .
. . .
sumar (b, n, m)
int b[ ][5], n, m;
{
      etc...
}
```

En el encabezamiento de una función, son equivalentes:

```
int b[ ][5];          int (*b)[5];
```

apuntador a "arrays" de 5 enteros

Las cadenas de caracteres ("strings")

Una cadena es un "array" unidimensional de tipo "char".

Con el carácter nulo ('\0') se indica el final de la cadena.

Una cadena es una secuencia de caracteres almacenados en posiciones contiguas, que termina con el carácter nulo.

```
char s[36];
```

- **Se declara la longitud máxima.**
- **El carácter nulo ('\0') consume una posición siempre.**
- **Se pueden almacenar cadenas de longitud variable sobre el tamaño máximo declarado.**
- **El programador tiene que preocuparse de evitar el desbordamiento.**

Las constantes de cadenas se escriben entre comillas:

```
"ejemplo de cadena"
```

Se puede asignar valor inicial en la declaración:

```
static char cad[40]="valor inicial";
static char cad[40]={ 'c', 'a', 'd', 'e', 'n', 'a', '\0' };
```

No hay operadores para manejar cadenas.

Las cadenas se manejan con funciones de biblioteca incluyendo, en el programa, la cabecera:

<string.h>

Funciones de biblioteca para manejar cadenas

Están descritas en la sección 3 del manual de referencia.

Necesitan incluir el fichero "string.h" en el programa:

```
# include <string.h>
```

Funciones que devuelven un "int":

strcmp	**comparar**
strncmp	**comparar "n" elementos**
strlen	**longitud**
etc...	

Funciones que devuelven un puntero a "char":

strcat	**encadenar**
strncat	**encadenar "n" elementos**
strcpy	**copiar**
strncpy	**copiar "n" elementos**
strchr	**localizar un elemento**
etc...	

int strcmp(char *s1, char *s2)

Compara elementos sucesivos de dos cadenas, s1 y s2, hasta que se encuentran elementos diferentes. Si todos los elementos son iguales, la función devuelve cero. Si el elemento diferente de s1 es mayor que el de s2, la función devuelve un valor positivo, en caso contrario devuelve un valor negativo.

```
/*
    ej18.c
*/
# define TAM 40
# include <stdio.h>
# include <string.h>
void main()
{
    char s1[TAM], s2[TAM];
    int n;
    printf("Introducir dos cadenas: ");
    scanf("%s %s", s1, s2);
    n = strcmp(s1, s2);
    if  ( n == 0 ) printf("\nSon iguales");
    else    if ( n<0 ) printf("\ns1 es menor que s2");
            else printf("\ns1 es mayor que s2\n");
}
```

int strncmp(char *s1, char *s2, int n)

Es semejante a "strcmp", pero no se comparan más de "n" caracteres.

int strlen(char *s)

Devuelve el número de caracteres de la cadena s, sin incluir su carácter nulo de terminación.

Una versión posible:

```
/*
     ej19.c
*/
int strlen(s)
char *s;
{
     int i;
     for ( i=0; *s != '\0'; s++ ) ++i;
     return( i );
}
```

char *strcat(char *s1, char *s2)

Copia la cadena s2, incluyendo el carácter nulo, en los elementos de la cadena s1, empezando en el elemento de s1 que contiene el carácter nulo. Devuelve s1.

char *strncat(char *s1, char *s2, int n)

Es semejante a "strcat", pero se agregan no más de "n" elementos no nulos seguidos opr '\0'. Devuelve s1.

char *strcpy(char *s1, char *s2)

La función copia la cadena s2, incluyendo su carácter nulo , en elementos sucesivos del "array" de "char" cuyo primer elemento tiene la dirección s1. Devuelve s1.

```
{
        char s[24];
        strcpy(s, "valor asignado");
        printf("%s\n", s);
}
```

char *strncpy(char *s1, char *s2, int n)

La funcion copia la cadena s2, sin incluir el carácter nulo,en elementos sucesivos del "array" de "char" cuyo primer elemento tiene la dirección s1. Se copian "n" caracteres de s2 como máximo. Después la función almacena cero o más caracteres nulos en los siguientes elementos de s1 hasta almacenar un total de "n" caracteres. Devuelve s1.

char *strchr(char *s, char c)

La función busca el primer elemento de la cadena s que sea igual a "(char)c" . Considera el carácter nulo de terminación como parte de la cadena. En caso de encontrarlo, se devuelve la dirección del elemento; en caso contrario, se devuelve un puntero nulo.

"arrays" de punteros

Ejemplo donde se usa un "array" de punteros a "char" en un programa para ordenar palabras en sentido creciente.

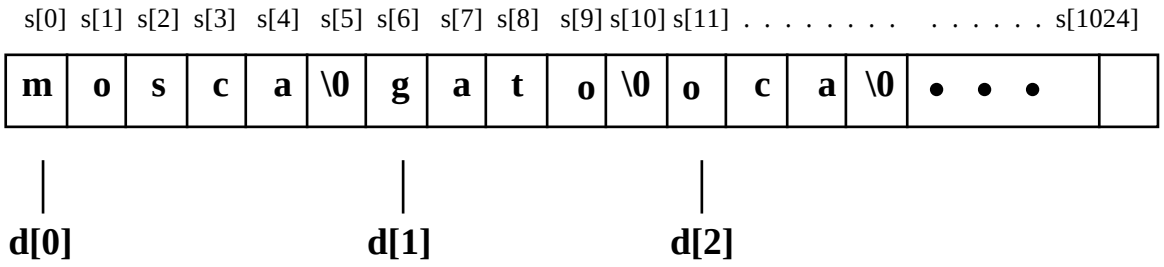
Se utiliza una técnica para almacenar cadenas de caracteres de forma compacta:

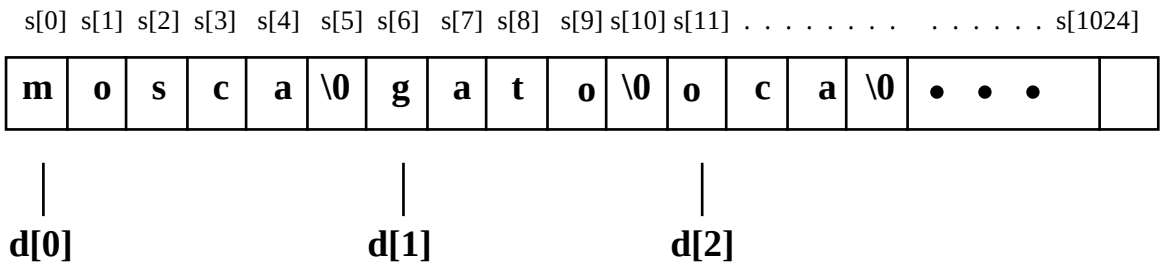
- **Se declara un "array" de "char" de tamaño suficiente para almacenar todas las palabras como cadenas de caracteres contiguas:**

  char s[1024];

- **Se usa un "array" de punteros auxiliar donde se almacenan las direcciones base de las cadenas en el "array" anterior:**

  char *d[128];





char s[1024];		s[0] ='m'
char *p;		s[1] ='o'
char *d[128];		s[2] ='s'
		s[3] ='c'
p=s;	p=&s[0]	s[4] ='a'
		s[5] ='\0'
scanf("%s", p);		s[6] ='g'
		s[7] ='a'
	s[0]='m'	s[8] ='t'
	s[1]='o'	s[9] ='o'
	s[2]='s'	s[10]='\0'
	s[3]='c'	s[11]='o'
	s[4]='a'	s[12]='c'
	s[5]='\0'	s[13]='a'
		s[14]='\0'
d[0]=p;	d[0]=&s[0]	s[15]
		etc....
p=p+strlen(p)+1;	p=&s[0]+5+1 p=&s[6]	
scanf("%s", p);		
	s[6] ='g'	
	s[7] ='a'	
	s[8] ='t'	
	s[9] ='o'	
	s[10]='\0'	
d[1]=p;	d[1]=&s[6]	
p=p+strlen(p)+1	p=&s[6]+4+1 p=&s[11]	
etc....		

La función principal

```
/*
    ej20.c
*/
# include <stdio.h>
# include <string.h>
# define BUFFER 1024
# define MAXPAL 128

void main( )
{
    int leer_palabras(int max, char s[ ], char *d[ ]);
    void escribir_palabras(char s[ ], char *d[ ], int n);
    void ordenar(char s[ ], char *d[ ], int n);
    char s[BUFFER], *d[MAXPAL];
    int n;
    if ( n=leer_palabras(MAXPAL, s, d) ) {
        ordenar(s, d, n);
        escribir_palabras(s, d, n);
    }
}
```

La función "leer_palabras"

```
int leer_palabras(int max, char s[ ], char *d[ ])
{
        char *p;
        int i, num;
        p=s;
        printf("Cuantas palabras se van a ordenar? \n");
        scanf("%d", &num);
        if ( num <= max ) {
                printf("Introducir %d palabras:\n", num);
                for ( i=0; i<num; ++i ) {
                                                scanf("%s", p);
                                                d[i]=p;
                                                p += strlen(p)+1;
                }
                return(num);
        }
        else {
                printf("ERROR. Valor maximo: %d\n", max);
                return(0);
        }
}
```

La función "escribir_palabras"

```
        void escribir_palabras(char s[ ], char *d[ ], int n)
        {
                int i;
                printf("Lista ordenada:\n");
                for ( i=0; i<n; ++i ) printf("%s\n", d[i]);
        }
```

La función "ordenar"

```
void ordenar(char s[ ], char *d[ ], int n)
{
    char *aux;
    int i, j;
    for ( i=0; i<n-1; ++i )   for ( j=n-1; i<j; --j)
            if ( strcmp(d[j-1], d[j]) > 0 ) {
                    aux=d[j-1];
                    d[j-1]=d[j];
                    d[j]=aux;
            }
}
```

Paso de argumentos a "main"

En el medio UNIX, se pueden pasar valores a un programa en la orden de llamada.

$ factorial 4

Esos valores se pueden recuperar desde la función "main" mediante parámetros.

main(argc, argv)

argc : **es el número de argumentos que se pasan, contando el nombre del programa**

argv: **es un "array" de punteros a cada uno de los parámetros tomado como cadena de caracteres**

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

```
/*
        ej21.c
        uso:            $ a.out 4
                        factorial de 4 = 24
*/
# include <stdio.h>
void main(argc, argv)
int argc;
char *argv[ ];
{
        int num(char *), factorial(int);
        int n;
        if ( argc == 1 ) {
                        printf("Introducir un entero\n");
                        scanf("%d", &n);
        }
        else            n=num(argv[1]);
        printf("factorial de %d = %d\n", n, factorial(n));
}


int num(char *s)
{
        int i, n;
        n=0;
        for ( i=0; s[i]>='0' && s[i]<='9'; ++i )
                n=10*n+s[i]-'0';
        return(n);
}


int factorial(int n)
{
        int i, f;
        if ( n == 0 || n == 1 )  return(1);
        else  for (i=1, f=1; i<=n; ++i)  f *= i;
        return(f);
}
```

"arrays" no uniformes

"arrays" de punteros que apuntan a otros "arrays" de tamaños diferentes.

```
static char *p[3] = {"Jose Perez", "Coso, 25", "252423"};
```

Otra forma menos eficaz:

```
static char a[3][20] = {"Jose Perez", "Coso, 25", "252423"};
```

El "array a":

- **Es bidimensional**
- **Puede almacenar hasta 60 caracteres**
- **Contiene 3 "arrays" de "arrays" de 20 caracteres**

El "array p":

- **Es un "array" de punteros a "char"**
- **Puede almacenar 3 punteros (3x4=12 bytes)**

p[0] contiene la dirección base de la cadena "Jose Perez"
p[1] contiene la dirección base de la cadena "Coso, 25"
p[2] contiene la dirección base de la cadena "252423"

Es mejor usar el puntero "p":

- **Se ocupa menos memoria (sólo la estrictamente necesaria).**
- **El acceso es más rápido porque el compilador no tiene que generar código para la función de transformación de almacenamiento, como se hace para acceder a los elementos a[i][j]**

- 10 -

ESTRUCTURAS, UNIONES Y TIPOS DEFINIDOS POR EL USUARIO.

- **Estructuras. El tipo "struct".**
- **Inicialización de estructuras.**
- **El operador "miembro de estructura".**
- **El operador puntero a estructura.**
- **Estructuras y funciones.**
- **Campos de bits.**
- **Uniones.**
- **Tipos definidos por el usuario.**

Declaración de estructuras

Una estructura ("struct") es una organización de datos heterogénea donde sus miembros pueden ser de tipos diferentes.

```
struct naipe {
        int valor;
        char palo;
} c1, c2;
```

aquí han quedado definidos un tipo y dos variables:

- **el tipo es "struct naipe"**
- **las variables son "c1" y "c2"**
- **los miembros de la estructura son "valor" y "palo"**

```
struct {
        int valor;
        char palo;
} carta, baraja[40];
```

aquí, no se ha definido identificador para el tipo.

Inicialización

Las estructuras externas ("extern") y estáticas ("static") pueden recibir valores en la declaración misma, de forma análoga a los "arrays".

```
static struct naipe {
        int valor;
        char palo[8];
} carta = { 3, "bastos" };
```

```
static struct complejo {
        float re;
        float im;
};
static struct complejo x[3][3] = {
                { {1.0, -0.5},      {3.5, 7.3},      {0.2, 0.2} },
                { {6.0, -2.3},      {-0.7, 2.4},     {23.1, 7.2} }
};  /* a x[2][ ] se le asignan ceros */
```

```
static struct {
    char c;
    int i;
    float s;
} x[2][3]= {    { 'a', 1, 3e3 },
                { 'b', 2, 4e2 },
                { 'c', 3, 5e3 },
                { 'd', 4, 6e2 }
            };
```

El operador miembro de estructura "."

El descriptor de un miembro de estructura tiene la forma:

variable_estructura•*nombre_miembro*

- **Un nombre de miembro no se puede repetir en una estructura**
- **Puede haber estructuras diferentes que tengan miembros del mismo nombre**

```
struct complejo {
        float real;
        float imaginaria;
} x, y, z, a[10][10];

x.real=2.7;
x.imaginaria=-0.5;
y.real=x.real;

z=x;

a[2][4].real=x.real;
```

El operador puntero a estructura " -> "

Es frecuente usar punteros a estructuras.

Por ejemplo, en las implementaciones donde no se permite que las funciones devuelvan valor de tipo "struct".

Por eso existe un símbolo especial para los punteros a estructuras:

-> ("menos" seguido de "mayor que")

La descripción de un miembro tiene la forma:

variable_puntero_a_estructura **->***variable_miembro*

Recordar que los operadores

apuntador a estructura	**->**
miembro de estructura	**.**
paréntesis	**( )**
corchetes	**[ ]**

tienen
- **la prioridad más alta**
- **asociatividad de izquierda a derecha**

Ejemplo

```
struct alumno {
    int curso;
    char grupo;
    char nombre[40];
    float notas[10];
};
struct alumno buf, *p=&buf;

buf.curso=2;
buf.grupo='A';
strcpy( buf.nombre, "Pedro Perez");
buf.notas[5]=7.5;
```

expresión	expresión equivalente	valor
buf.curso	p->curso	2
buf.grupo	p->grupo	A
buf.nombre	p->nombre	Pedro Perez
*(p->nombre+2)	p->nombre[2]	d
*p->nombre+2	*(p->nombre)+2	R

Estructuras y funciones

- **Las estructuras se pueden pasar a las funciones por valor.**

- **Las funciones pueden devolver un valor de tipo "struct" (depende de la implementación).**

Aunque la implementación no permita declarar funciones de tipo "struct", siempre se pueden declarar como apuntadores a "struct".

```
struct complejo {
        float re;
        float im;
};

main( )
{
        struct complejo x, y, z;
        x.re=2.5;   x.im=-5.4;
        y.re=0.4;   y.im=12.1;
        z = *sumar(x, y);
        . . .
}

struct complejo *sumar(a, b)
struct complejo a, b;
{
        struct complejo *z;
        z->re = a.re + b.re;
        z->im = a.im + b.im;
        return(z);
}
```

Campos de bits

Son miembros consecutivos de una estructura ("struct") que contienen un número constante no negativo de bits:

tipo identificador **:** *expresión_constante* **;**

```
struct word {
        unsigned    byte0 : 8,
                    byte1 : 8,
                    byte2 : 8,
                    byte3 : 8;
};
struct word w;
w.byte0='A';
w.byte1='B';
```

Los campos permiten:

- **Acceso cómodo al bit**
- **Ahorrar espacio en memoria**

- **Un campo no puede ser mayor que una palabra (32 bits).**
- **Los campos deben ser de tipo "int" o "unsigned".**
- **No están permitidos los "arrays" de campos.**
- **No puede haber punteros a campos.**
- **A un campo no se le puede aplicar el operador dirección.**
- **Se pueden usar campos sin nombre, a efectos de relleno.**

Ejemplos

```
struct naipe {
  unsigned valor : 4;        /*  para representar el entero 12
                                 necesito más de 3 bits  */
  unsigned palo : 2;         /*  con 2 bits puedo representar
                                 4 valores diferentes   */
};
struct naipe carta;
carta.valor=3;
carta.palo=1;
```

```
  struct rs232c {
      unsigned                pin1:   1,
                  transmision_datos:   1,
                    recepcion_datos:   1,
                                rts:   1,
                                cts:   1,
                                dsr:   1,
                                 sg:   1,
                                  :  12,
                                dtr:   1,
                                  :   5;
};
struct rs232c estado;
. . .
if (estado.cts) printf("listo para enviar");
. . .
```

Las uniones

Son organizaciones de datos semejantes a las estructuras, que se caracterizan porque sus miembros comparten memoria.

- **Permiten ahorrar memoria.**
- **El compilador asigna una cantidad de memoria suficiente para poder almacenar al miembro mayor.**
- **Se accede a los miembros como en las estructuras.**

```
/*
    ej22.c
*/
# include <stdio.h>
void main( )
{
    union numero {
            int i;
            float f;
    } x;
    x.i=222;
    printf("i: %15d   f: %15.8e\n", x.i, x.f);
    x.f=222.0;
    printf("i: %15d   f: %15.8e\n", x.i, x.f);
}
```

```
/*
        ej23.c
*/
# include <stdio.h>
typedef struct {
        unsigned        b0:1, b1:1, b2:1, b3:1,
                        b4:1, b5:1, b6:1, b7:1,
                        b8:1, b9:1, b10:1, b11:1,
                        b12:1, b13:1, b14:1, b15:1;
} word;

void main()
{
        void escribir(word);
        union equivalencia {
        int n;
        char c;
        word bits;
        } x;

        scanf("%d", &x.n);
        while ( x.n != 0 ){
                        escribir(x.bits);
                        scanf("%d", &x.n);
        }

        x.n=0;
        x.c='A';
        escribir(x.bits);
}

void escribir (word b)
{
printf("%1d%1d%1d%1d%1d%1d%1d%1d",
        b.b0,b.b1,b.b2,b.b3,b.b4,b.b5,b.b6,b.b7);
printf("%1d%1d%1d%1d%1d%1d%1d%1d",
        b.b8,b.b9,b.b10,b.b11,b.b12,b.b13,b.b14,b.b15);
printf("\n");
}
```

Tipos definidos por el usuario

Con la declaración "typedef" se puede asociar explícitamente un identificador a un tipo.

```
typedef int entero;
typedef int vector[10];
typedef char *cadena;
typedef float matriz[100][100];
```

Después, los identificadores se pueden usar para declarar variables y funciones:

```
vector v;
matriz a, b, c;
```

Se pueden construir jerarquías de tipos:

```
typedef double escalar;
typedef escalar vector[N];
typedef vector matriz[N];
```

El uso de "typedef" tiene algunas ventajas:

- **Permite abreviar expresiones largas.**
- **Facilita la documentación del programa.**
- **Facilita la modificación del programa y, por tanto, su transporte entre máquinas diferentes.**

- **Permite al programador pensar en términos de la aplicación (alto nivel) y no en términos de la representación interna.**

Las definiciones de tipos se suelen almacenar en ficheros independientes que se incorporan a los programas con órdenes "include":

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

Con "typedef", el usuario puede ampliar el lenguaje de manera natural incorporando tipos nuevos como dominio. Después, se pueden definir funciones que proporcionen operaciones sobre estos dominios.

```
/*
    ej24.c
*/
#define N 100
typedef double escalar;
typedef escalar matriz[N][N];

void mat_producto(matriz a, matriz b, matriz c)
{
    int i, j, k;
    for ( i=0; i<N; ++i )
            for ( j=0; j<N; ++j )
                    for ( c[i][j]=0, k=0; k<N; ++k )
                            c[i][j] += a[i][k] * b[k][j];
}
```

```
/*
ej25.c
Se maneja un tipo definido por el usuario para implementar
operaciones con numeros complejos.
Caso en que las funciones pueden devolver "struct".
*/
#include <math.h>
#include <stdio.h>
typedef float tipo_componente;
typedef struct {
        tipo_componente re;
        tipo_componente im;
} complex;

complex cpxasig(tipo_componente, tipo_componente);
complex cpxsuma(complex, complex);
complex cpxprod(complex, complex);
complex cpxconj(complex);
tipo_componente cpxlon(complex *, int);
tipo_componente cpxabs(complex);
void cpxprint(complex);

#define N 5

void main()
{
        static tipo_componente  a[N]={1.0, 2.0, 3.0, 4.0, 5.0};
        static tipo_componente  b[N]={1.0, 2.0, 3.0, 4.0, 5.0};
        complex c[N];
        int i;
        for (i=0; i<N; ++i) c[i] = cpxasig(a[i], b[i]);
        for (i=0; i<N; ++i) {
                printf("%d: ", i);
                cpxprint(c[i]);
        }
        printf("Suma c[1]+c[2]: ");
                cpxprint(cpxsuma(c[1],c[2]));
        printf("Producto c[1]*c[2]: ");
                cpxprint(cpxprod(c[1], c[2]));
        printf("Conjugado de c[1]: ");
                cpxprint(cpxconj(c[1]));
        printf("Val.absoluto de c[1]: %f\n", cpxabs(c[1]));
        printf("Longitud del vector: %f\n", cpxlon(c, N));
```

```
complex cpxasig(   tipo_componente parte_real,
                                tipo_componente parte_imaginaria)
{
        complex x;
        x.re=parte_real;
        x.im=parte_imaginaria;
        return(x);
}


void cpxprint(complex x)
{
        printf("%f + %f i\n", x.re, x.im);
}


complex cpxsuma(complex x, complex y)
{
        complex z;
        z.re=x.re+y.re;
        z.im=x.im+y.im;
        return(z);
}


complex cpxprod(complex x, complex y)
{
        complex z;
        z.re=x.re*y.re-x.im*y.im;
        z.im=x.re*y.im+x.im*y.re;
        return(z);
}
```

```
complex cpxconj(complex x)
{
	complex z;
	z.re= x.re;
	z.im= -x.im;
	return(z);
}


tipo_componente cpxabs(complex x)
{
	complex z;
	z=cpxprod(x, cpxconj(x));
	return(sqrt(z.re));
}


tipo_componente cpxlon(complex *v, int n)
{
	int i;
	tipo_componente s;
	complex z;
	for (s=0.0, i=0; i<n; ++i) {
		z=cpxprod(v[i], cpxconj(v[i]));
		s += z.re;
	}
	return(sqrt(s));
}
```

```
/*    ej26.c
      Se maneja un tipo definido por el usuario para
      implementar operaciones con numeros complejos.
      Caso en que las funciones no pueden devolver "struct"
      y se usan funciones del tipo "int".  */
#include <math.h>
#include <stdio.h>
typedef float tipo_componente;
typedef struct {
        tipo_componente re;
        tipo_componente im;
} complex;

void cpxasig(tipo_componente, tipo_componente, complex *);
void cpxsuma(complex, complex, complex *);
void cpxprod(complex, complex, complex *);
void cpxconj(complex, complex *);
void cpxprint(complex);
tipo_componente cpxlon(complex *, int);
tipo_componente cpxabs(complex);

#define N 5

void main()
{
    static tipo_componente  a[N]={1.0, 2.0, 3.0, 4.0, 5.0};
    static tipo_componente  b[N]={1.0, 2.0, 3.0, 4.0, 5.0};
    complex c[N], x;
    int i;
    for (i=0; i<N; ++i) cpxasig(a[i], b[i], &c[i]);
    for (i=0; i<N; ++i) {
        printf("%d: ", i);
        cpxprint(c[i]);
    }
    cpxsuma(c[1],c[2], &x);
    printf("Suma de c[1]+c[2]: ");     cpxprint(x);
    cpxprod(c[1],c[2], &x);
    printf("Producto de c[1]*c[2]: "); cpxprint(x);
    cpxconj(c[1], &x);
    printf("Conjugado de c[1]: ");     cpxprint(x);
    printf("Val.absoluto de c[1]: %f\n", cpxabs(c[1]));
    printf("Longitud del vector: %f\n", cpxlon(c, N));
}
```

```
void cpxasig(tipo_componente a, tipo_componente b,
                    complex *x)
{
        x->re=a;
        x->im=b;
}


void cpxprint(complex x)
{
        printf("(%f + %f i)\n", x.re, x.im);
}


void cpxsuma(complex x, complex y, complex *s)
{
        s->re=x.re+y.re;
        s->im=x.im+y.im;
}


void cpxprod(complex x, complex y, complex *p)
{
        p->re=x.re*y.re-x.im*y.im;
        p->im=x.re*y.im+x.im*y.re;
}


void cpxconj(complex x, complex *c)
{
        c->re=x.re;
        c->im=-x.im;
}
```

```
tipo_componente cpxabs(complex x)
{
        complex c, p;
        cpxconj(x, &c);
        cpxprod(x, c, &p);
        return(sqrt(p.re));
}


tipo_componente cpxlon(complex *v, int n)
{
        int i;
        tipo_componente s;
        complex c, p;
        for (s=0.0, i=0; i<n; ++i) {
                cpxconj(v[i], &c);
                cpxprod(v[i], c, &p);
                s += p.re;
        }
        return(sqrt(s));
}
```

```
/*
      ej27.c
      Se maneja un tipo definido por el usuario para implementar
      operaciones con numeros complejos.
      Se manejan funciones que devuelven punteros a estructuras
*/

#include <math.h>
#include <stdio.h>
typedef float tipo_componente;
typedef struct {
        tipo_componente re;
        tipo_componente im;
} complex;
complex *cpxasig(), *cpxsuma(), *cpxprod(), *cpxconj();
tipo_componente cpxabs(), cpxlon();
#define N 5

main()
{
    static tipo_componente  a[N]={1.0, 2.0, 3.0, 4.0, 5.0};
    static tipo_componente  b[N]={1.0, 2.0, 3.0, 4.0, 5.0};
    complex c[N];
    int i;
    for (i=0; i<N; ++i) c[i] = *cpxasig(a[i], b[i]);
    for (i=0; i<N; ++i) {
       printf("%d: ", i);
       cpxprint(c[i]);
    }
    printf("Suma c[1]+c[2]: ");     cpxprint(*cpxsuma(c[1],c[2]));
    printf("Producto c[1]*c[2]: "); cpxprint(*cpxprod(c[1], c[2]));
    printf("Conjugado de c[1]: ");  cpxprint(*cpxconj(c[1]));
    printf("Val.absoluto de c[1]: %f\n", cpxabs(c[1]));
    printf("Longitud del vector: %f\n", cpxlon(c, N));
}
```

```
complex *cpxasig(parte_real, parte_imaginaria)
tipo_componente parte_real, parte_imaginaria;
{
    complex x;
    x.re=parte_real;
    x.im=parte_imaginaria;
    return(&x);
}


cpxprint(x)
complex x;
{
        printf("%f + %f i\n", x.re, x.im);
}


complex *cpxsuma(x, y)
complex x, y;
{
        complex z;
        z.re=x.re+y.re;
        z.im=x.im+y.im;
        return(&z);
}


complex *cpxprod(x, y)
complex x, y;
{
        complex z;
        z.re=x.re*y.re-x.im*y.im;
        z.im=x.re*y.im+x.im*y.re;
        return(&z);
}
```

```
complex *cpxconj(x)
complex x;
{
        complex z;
        z.re= x.re;
        z.im= -x.im;
        return(&z);
}


tipo_componente cpxabs(x)
complex x;
{
        complex z;
        z= *cpxprod(x, *cpxconj(x));
        return(sqrt(z.re));
}


tipo_componente cpxlon(v, n)
complex v[];
int n;
{
        int i;
        tipo_componente s;
        complex z;
        for (s=0.0, i=0; i<n; ++i) {
                z= *cpxprod(v[i], *cpxconj(v[i]));
                s += z.re;
        }
        return(sqrt(s));
}
```

- 11 -

FUNCIONES PARA MANEJO DE FICHEROS

- **Fichero, flujo de datos y puntero a flujo.**
- **Resumen de funciones de la biblioteca estándar.**
- **Abrir fichero con "fopen".**
- **Cerrar fichero con "fclose".**
- **Posicionar en fichero con "fseek", "rewind".**
- **Leer un carácter con "fget", "getc", "getchar".**
- **Escribir un carácter con "fput", "putc", "putchar".**
- **Leer una cadena con "fgets", "gets".**
- **Escribir una cadena con "fputs", "puts".**
- **Leer en binario usando "buffer" con "fread".**
- **Escribir en binario usando "buffer" con "fwrite".**
- **Leer con formato usando "fscanf", "scanf", "sscanf"**
- **Escribir con formato usando "fprintf", "printf", "sprintf"**

Los programas se comunican con el entorno leyendo y escribiendo ficheros.

Un fichero se considera como flujo de datos que se procesan secuencialmente.

El concepto de fichero es amplio:

- **Fichero de disco:** **conjunto de datos que se puede leer y escribir repetidamente.**
- **Tubería, canal:** **flujo de bytes generados por un programa.**
- **Dispositivos:** **flujo de "bytes" recibidos desde o enviados hacia un dispositivo periférico (teclado, monitor, ...).**

Todas las clases de ficheros se manejan casi de la misma forma.

Los ficheros se manejan con funciones de biblioteca cuyas declaraciones se incorporan al programa con la orden:

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

Las funciones para manejo de ficheros usan el soporte que proporciona el sistema operativo.

Antes de realizar cualquier operación sobre un fichero, hay que abrirlo.

En el entorno se dispone siempre de tres ficheros abiertos:

identificador	nombre	asignación inicial
=========	==========	========
stdin	**fichero estándar de entrada**	**teclado**
stdout	**fichero estándar de salida**	**pantalla**
stderr	**fichero estándar para mensajes de error**	**pantalla**

Al abrir un fichero, la función "fopen" lo asocia con un flujo de datos y devuelve un puntero a un objeto de tipo FILE que se usa para mantener el estado del flujo.

Después ese puntero se utiliza para las operaciones sobre el fichero.

- **Todas las entradas se producen como si cada carácter fuera leido llamando a "fgetc".**
- **Todas las salidas se producen como si cada carácter fuera escrito llamando a "fputc".**
- **El fichero se puede cerrar con "fclose".**

El puntero a FILE mantiene información sobre el estado del flujo de datos:

- **Un indicador de error, que toma valor distinto de cero cuando la función encuentra un error de lectura o de escritura.**

- **Un indicador de fin de fichero, que resulta distinto de cero cuando la función encuentra el final de fichero cuando está leyendo.**

- **Un indicador de posición, que indica el siguiente "byte" que se lee o se escribe en el flujo de datos, si el fichero puede soportar preguntas sobre la posición.**

- **Un "buffer" de fichero, que indica la dirección y tamaño de un "array" que utilizan las funciones para leer y escribir.**

```
# include <stdio.h>
main( )
{
        FILE *fp;
        . . .
        fp = fopen ( "datos", "r" );
        . . .
        fclose ( fp );
        . . .
}
```

Algunas de las funciones estándar

Control**:**	**fopen**	**fclose**	
	fseek	**rewind**	
Leer y escribir un carácter:	**fgetc**	**getc**	**getchar**
	fputc	**putc**	**putchar**
Leer y escribir una cadena:	**fgets**	**gets**	
	fputs	**puts**	
Leer y escribir con "buffer":	**fread**	**fwrite**	
Leer y escribir con formato:	**fprintf**	**printf**	**sprintf**
	fscanf	**scanf**	**sscanf**

Los entornos operativos también proporcionan bibliotecas de funciones que implementan diversos modos de acceso.

Abrir fichero con "fopen"

FILE *fopen (char *nombre, char *modo);

Abre un fichero, lo asocia con un flujo de datos y devuelve un puntero a FILE. Si el fichero no es accesible, se devuelve el puntero de valor NULL.

Los modos "r", "w" y "a" corresponden a leer, escribir y añadir.

Si el modo es "r" o "w", el puntero de fichero se coloca al principio. Si el modo es "a", el puntero se coloca al final.

Si el modo es "w" o "a", se crea el fichero si no existe.

```
# include <stdio.h>
main()
{
        FILE *fp;
        char nombre[36];
        printf("nombre de fichero ? ");
        scanf("%s",nombre);
        if ( ! (fp=fopen(nombre,"r")) )
                printf("error\n");
        else
                procesar(fp);
. . .
. . .
}
```

Cerrar fichero con "fclose"

int fclose (FILE *fp);

Si el puntero no está asociado a un fichero, se devuelve EOF.
En caso de éxito, se devuelve cero (0).

Posicionar con "fseek"

int fseek (FILE *fp, long desplaz, int posicion);

Se establece la posición para la siguiente operación de lectura o escritura.

El parámetro "posicion" puede tomar los valores 0, 1, 2.
Con "posicion" se identifica una posición en el fichero: 0 para el principio, 1 para la posición actual y 2 para el final de fichero.

El puntero se desplaza un número de bytes indicado por "desplaz" a partir de "posición".

En caso de error, devuelve un valor distinto de cero.

Posicionar al principio con "rewind"

void rewind (FILE *fp)

Tiene el mismo efecto que " fseek (fp, 0L, 0) ";

Leer un carácter con "fgetc", "getc", "getchar"

int fgetc (FILE *fp);

Se obtiene el siguiente carácter en el flujo de entrada al que apunta "fp", avanza el indicador de posición y se devuelve el carácter leido como un "int".

Si se produce error o se alcanza el fin de fichero, se devuelve EOF.

int getc (FILE *fp);

Es una macro que tiene el mismo efecto que "fgetc".

La función "fgetc" es más lenta que la macro "getc", pero ocupa menos espacio por llamada y su nombre se puede pasar como argumento a otra función.

int getchar ( );

Es una macro que se define como "getc(stdin)".

Uso de "fopen()", "fclose()", "getc()", "putchar()"

```
/*
    ej28.c
    Se leen caracteres en un fichero y se escriben
    por "stdout"
*/

# include <stdio.h>

void main()
{
    FILE *fp;
    char nombre[36], c;
    printf("nombre de fichero ? ");
    scanf("%s",nombre);
    if ( ! (fp=fopen(nombre,"r")) ) printf("error\n");
    else {
            c = getc(fp);
            while ( c != EOF ) {
                    putchar(c);
                    c = getc(fp);
            }
    fclose(fp);
    }
}
```

Escribir un carácter con "fput", "putc", "putchar"

int fputc (int c , FILE *fp);

Escribe el carácter "c" en el flujo de salida "fp", avanza la posición y devuelve el carácter escrito como "int".

Si se produce error, devuelve EOF.

int putc (int c, FILE *fp);

Es una macro con el mismo efecto que "fputc".

La función "fputc" es más lenta, pero ocupa menos espacio y se puede pasar como argumento a otra función.

int putchar (int c);

Es una macro equivalente a "putc(stdout)".

Uso de "fopen()", "fclose()", "getc()", "putc()"

```
/*
        ej29.c
        Se copia el contenido de un fichero a otro, operando
        sobre caracteres con "getc()" y "putc()".
*/

# include <stdio.h>

void main()
{
        FILE *in, *out;
        char nomin[36], nomout[36], c;
        printf("Fichero de entrada ? "); scanf("%s",nomin);
        in=fopen(nomin, "r");
        if ( ! in )
                printf("error al abrir fichero de entrada\n");
        else {
                printf("Fichero de salida ? "); scanf("%s",nomout);
                out=fopen(nomout, "w");
                if ( ! out ) {
                        printf("error al abrir fichero de salida\n");
                        fclose(in);
                }
                else {
                        c=getc(in);
                        while ( c != EOF ) {
                                putc(c, out);
                                c = getc(in);
                        }
                        fclose(in); fclose(out);
                }
        }
}
```

Uso de "fseek()", "getc()", "putchar()"

```
/*    ej30.c
      Se lee un número de caracteres a partir de una posición
      del fichero que se determina con  "fseek()" y se escriben
      por pantalla.   */
# include <stdio.h>
int leer_posicion(void);
int leer_num(void);
int test(int);
void main()
{
      FILE *fp;
      char nombre[36], c;
      int pos, num;   long desp;
      printf("Nombre de Fichero ? "); scanf("%s",nombre);
      fp=fopen(nombre, "rb");
      if ( ! fp )
            printf("error al abrir fichero de entrada\n");
      else {
            while ( 1 ) {
                  pos = leer_posicion();
                  printf("Desplazamiento (num. de bytes) ? ");
                  scanf("%ld", &desp);
                  num = leer_num();
                  if ( fseek(fp, desp, pos) != 0 )
                        printf("error en posicionamiento\n");
                  else {
                        int i=0;
                        c = getc(fp);
                        while ( i<=num && c!=EOF ) {
                              putchar(c);
                              i=i+1;
                              c = getc(fp);
                        }
                  putchar('\n'); putchar('\n');
                  }
            }
      }
```

```
int leer_posicion()
{
	int n;
	printf("Posicion (0:principio,  1:actual,  2:EOF) ? ");
	scanf("%d", &n);
	while ( ! test(n) ) {
		printf("Posicion (0:principio,  1:actual,  2:EOF) ?");
		scanf("%d", &n);
	}
	return(n);
}



int test ( int n )
{
	if ( n<0 || n>2 ) return(0);
	else return(1);
}



int leer_num()
{
	int n;
	printf("Numero de bytes ( mayor que cero ) ? ");
	scanf ("%d", &n);
	while ( n<0 ) {
		printf("Numero de bytes ( mayor que cero ) ? ");
		scanf ("%d", &n);
	}
	return(n);
}
```

Leer una cadena con "fgets()" y "gets()"

char *fgets (char *s, int n, FILE *fp);

Lee caracteres por el flujo al que apunta "fp" y los coloca en el "array" al que apunta "s", hasta que se han leido "n-1" caracteres o se lee el carácter "nueva_línea" o se alcanza la condición de fin de fichero. Después se pone el carácter nulo ('\0') como marca de fin de cadena.

El carácter "nueva_línea" leido también entra en la cadena.

En caso de error se devuelve el puntero nulo.
En caso de alcanzar fin de fichero sin haber leido caracteres, se devuelve el puntero nulo.
En otro caso, se devuelve "s".

char *gets (char *s);

Lee caracteres por el flujo de entrada estándar (stdin) y los coloca en el "array" apuntado por "s", hasta que se alcance el carácter de "nueva_línea" o la condición de fin de fichero.

El carácter "nueva_línea" se descarta y se pone el carácter nulo ('\0').

Se comporta igual que "fgets" en cuanto al valor que devuelve.

Escribir una cadena con "fputs()" y "puts()"

int fputs (char *s, FILE *fp);

Escribe la cadena apuntada por "s" en el flujo de salida "fp".

No se escribe el carácter nulo que marca el fin de cadena.

Devuelve EOF en caso de error.

int puts (char *s);

Escribe la cadena apuntada por "s" y un carácter de "nueva_línea" por el flujo de salida estándar (stdout).

No se escribe el carácter nulo que marca el fin de cadena.

Devuelve EOF en caso de error.

Leer en binario usando "buffer", con "fread"

int fread(char *buf, int tam, int num, FILE *fp);

Se lee un número "num" de items por el flujo "fp" y se almacenan en el "array" apuntado por "buf".

Cada item es una secuencia de "bytes" de longitud dada por "tam".

Termina cuando se alcanza la condición de fin de fichero o cuando se ha leido un número "num" de items. El puntero del flujo de entrada queda apuntando al "byte" que sigue al último leido.

No se modifica el contenido del flujo de entrada.

Se devuelve en número de items leidos.

Escribir en binario usando "buffer" con "fwrite"

int fwrite (char *buf, int tam, int num, FILE *fp);

Añade al menos un número "num" de items de datos del "array" apuntado por "buf", en el flujo de salida "fp". Se termina de añadir cuando ya se ha añadido un número "num" de items o cuando se produce condición de error en el flujo de salida.

No se modifica el contenido del "array" apuntado por "buf".

Se devuelve el número de items escritos.

Uso de "fread()", "fwrite()"

```
/*
        ej31.c
        Se copia el contenido de un fichero, operando
        en binario con buffer de 512 bytes
*/

# define LON 512
# include <stdio.h>

void main()
{
        FILE *in, *out;
        char nomin[36], nomout[36], buf[LON];
        int n;
        printf("Fichero de entrada ? ");
        scanf("%s", nomin);
        in=fopen(nomin, "r");
        if ( ! in )
                printf("error al abrir fichero de lectura\n");
        else{
                printf("Fichero de salida ? ");
                scanf("%s", nomout);
                out=fopen(nomout, "w");
                if ( ! out ) {
                        printf("error al abrir fichero de salida\n");
                        fclose(in);
                }
                else {
                        while ( (n=fread(buf, 1, LON, in)) > 0 )
                                fwrite(buf, 1, n, out);
                        fclose(in); fclose(out);
                }
        }
}
```

Leer con formato usando "fscanf", "scanf", "sscanf".

int fscanf (FILE *fp, char *formato, lista . . .);

Se leen caracteres por el flujo de entrada "fp", se convierten en valores de acuerdo con las especificaciones de "formato" y se almacenan en la dirección dada por el puntero correspondiente de la lista de argumentos.

Se devuelve el número de conversiones conseguidas. Se devuelve EOF si se alcanza la condición de fin de fichero.

En caso de error, si el formato no es numérico, devuelve el número de argumentos leidos correctamente antes de producirse el error. Si el formato es numérico, devuelve cero.

La cadena de control "formato" puede contener:

- **Espacios en blanco, que se corresponden con el espacio en blanco opcional del flujo de entrada.**

- **Un carácter ordinario, que no sea '%' ni espacio en blanco y que se corresponda con el siguiente carácter en el flujo de entrada.**

- **Especificaciones de conversión conteniendo:**

  - **El carácter '%', el carácter opcional '*' para suprimir la conversión.**
  - **Un entero opcional que indica la anchura máxima del campo.**
  - **Uno de los modificadores 'l' o 'h' para indicar tamaño "long".**
  - **Un código de conversión.**

Carácter de Conversión	Interpretación del flujo de entrada
d	Entero decimal
o	Entero octal
x	Entero hexadecimal
u	Entero decimal sin signo
e	Número de coma flotante
f	Equivalente a "e"
c	Un carácter
s	Cadena de caracteres
[cadena]	Cadena especial

Los caracteres 'd', 'o', 'x', 'e' y 'f' pueden ir precedidos por 'l' para indicar conversión a "long" o a "double".

Los caracteres 'd', 'o', 'x' pueden ir precedidos por 'h' para convertir a "short".

- **Un campo de entrada es una secuencia de caracteres distintos del espacio en blanco.**

- **Para todos los descriptores, excepto "[" y "c", se ignoran los espacios en blanco que preceden a un campo.**

- **El campo de entrada termina cuando se alcanza un carácter inapropiado o cuando acaba la anchura del campo.**

- **Cuando se leen caracteres, el espacio en blanco no se salta.**

- **El formato "%1s" puede usarse para leer el siguiente carácter distinto de espacio en blanco.**

- **La especificación "%[cadena]" se usa para leer una cadena especial.**

  Si el primer carácter de la cadena es '^', entonces la cadna leida estará formada por caracteres diferentes de los que forman la especificación:

  Con "%[^abc], la lectura termina al encontrar 'a', 'b' ó 'c'; pero no un espacio en blanco.

  Si el primer carácter es distinto de '^', la cadena leida sólo puede contener los caracteres de la especificación.

scanf ("%[ab \n\t]", s)

Leerá una cadena que contenga únicamente los caracteres 'a', 'b', espacio en blanco, nueva linea, tabulador.

scanf ("%5s", s)

Se ignoran espacios en blanco y se leen los cinco caracteres siguientes.

scanf ("%s", s)

Se lee una cadena arbitraria hasta encontrar espacio en blanco.

```
int n, a, b, c;
float x;
char nombre[50];
```

n=scanf("%d%f%s", &a, &x, nombre);

25 54.32E-1 eugenio

variable	valor asignado
n	3
a	25
x	5.432
nombre	eugenio

scanf("%d,%d,%d", &a, &b, &c);

12,5,7

variable	valor asignado
a	12
b	5
c	7

scanf("%2d%f%*d%[0-9]", &a, &x, nombre);

56789 0123 56a72

variable	valor asignado
a	56
x	789.0
nombre	56

int scanf (char *formato, lista . . .);

Se lee por el flujo de entrada "stdin", con el mismo efecto que "fscanf".

int sscanf (char *s, char *formato, lista . . .);

Se lee en la cadena apuntada por "s", con el mismo efecto que "fscanf".

```
/*
        ej32.c
        Se calculan las raices cuadradas de los valores
        pasados en la orden de llamada.
        Uso:        $ raices 4 5.6 7 8
*/
# include <stdio.h>
# include <math.h>
void main(argc, argv)
int argc;
char *argv[];
{
        int i;
        float x;
        if ( argc == 1 )
        printf("Error. Hay que pasar valores en la llamada\n");
        else {
                for ( i=1; i<argc; ++i ) {
                sscanf(argv[i], "%f", &x);      /*      la entrada se toma de
                                                        una cadena */

                printf("La raiz cuadrada de %f es %lf\n",
                        x, sqrt(x) );
                }
        }
}
```

Escribir con formato usando "fprintf", "printf", "sprintf".

int fprintf (FILE *fp, char *formato, lista . . .);

Esta función convierte, formatea y escribe por un flujo de salida.

Las expresiones que forman la "lista" de argumentos se evalúan, se convierten de acuerdo con los formatos de la cadena de control "formato", y se escriben por el flujo de salida "fp".

Se devuelve el número de caracteres escritos o un valor negativo en caso de error.

El formato se indica con una cadena de caracteres que puede contener especificaciones de conversión y caracteres normales.

Las especificaciones de conversión empiezan po '%' o por "%n$", donde "n" es un entero que indica la posición del argumento al que se aplicará. Después se puede poner:

• Ninguno o más modificadores: - + blanco #

• Una secuencia de cifras para indicar la anchura del campo.

• Una secuencia de cifras para indicar la precisión:
- **número mínimo de dígitos para las conversiones "d, i, o, u, x, X"**
- **número de cifras decimales para las conversiones "e, f"**
- **máximo número de cifras significativas para la conversión "g"**
- **máximo número de "bytes" que se escriben para la conversión "s"**

La anchura del campo y la precisión se separan con un punto. Si se pone "*", el valor se calcula usando los siguientes argumentos:

printf("%*.*d\n", ancho, precision, valor)

• Una "l" o "h" opcionales para convertir a "long" o "short" los tipos enteros.

• Un carácter que indica el tipo de conversión que se solicita.

Ejemplos: %6d %8.4f %2$4d

Modificadores

- ajustar a la izquierda
+ escribir el signo siempre
etc...

Caracteres de conversión

d, i	**se convierte a decimal con signo.**
o	**se convierte a octal sin signo.**
u	**se convierte a decimal sin signo**
x, X	**se convierte a hexadecimal sin signo**
f	**se supone argumento "float" o "double y se convierte a notación decimal.**
e, E	**se supone argumento "float" o "double" y se convierte a notación exponencial.**
g, G	**se supone argumento "float" o "double" y se elige el más breve entre entre "f" o "e".**
c	**un carácter.**
s	**se toma el argumento como cadena de caracteres**

int printf (char *formato, lista . . .);

Escribe por el flujo de salida "stdout", con el mismo efecto que "fprintf".

int sprintf (char *s, char *formato, lista . . .);

Escribe en la cadena apuntada por "s", con el mismo efecto que "fprintf".

```
/*
    ej33.c
    Escribir con formato definido en tiempo de ejecucion,
    que se introduce como dato.
*/

# include <stdio.h>

void main()
{
    int n=12;
    float x=3.1416;
    static char s[18]="25-ENE-91";
    char formato[80];
    printf("Introducir un formato para: ");
    printf("entero float cadena\n");
    while ( gets(formato) ) {
            printf ( formato, n, x, s );
            printf("\nIntroducir un formato para: ");
            printf("entero float cadena\n");
    }
}
```

- 12 -

ESTRUCTURAS DINAMICAS DE DATOS

- **Estructuras estáticas y dinámicas.**
- **Definición recursiva de datos.**
- **Estructuras dinámicas típicas.**
- **Asignar memoria con "malloc( )"**
- **Liberar memoria con "free( )".**
- **Construcción de una lista encadenada en forma lineal.**
- **Programa para manejar una cola.**

Estructuras estáticas

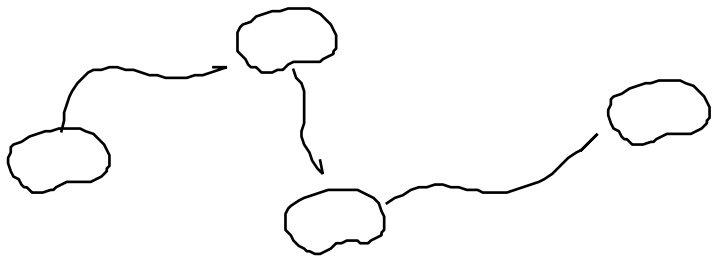
- **Se les asigna memoria en tiempo de compilación**
- **Ocupan un número fijo de posiciones de memoria.**
- **Existen durante toda la ejecución del bloque.**

Estructuras dinámicas

- **Se les asigna memoria en tiempo de ejecución.**
- **Pueden expandirse y contraerse.**
- **Se crean durante la ejecución invocando a la función "malloc".**

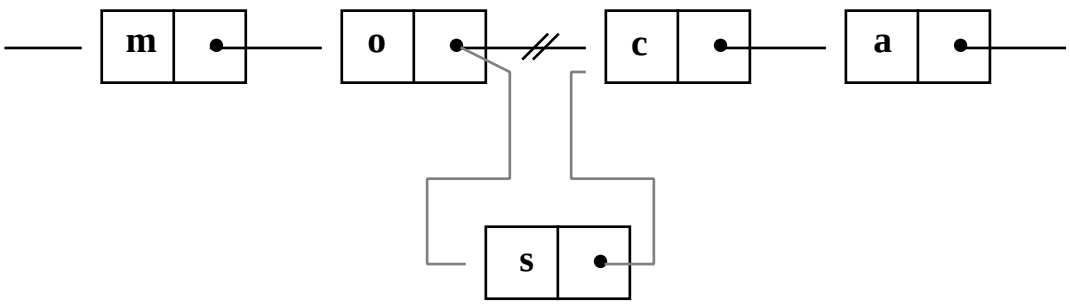
Las estructuras dinámicas se construyen con punteros

- **Existe una función que crea una realización del objeto.**
- **Un componente del objeto es un puntero que "apunta" a otro objeto del mismo tipo.**
- **Se consigue tener un conjunto de objetos relacionados que puede aumentar y disminuir en tiempo de ejecución.**



Ejemplo

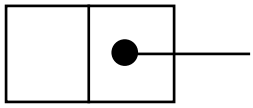
Un texto se puede representar con una lista de caracteres encadenados. Así, las operaciones de inserción y eliminación son muy económicas.



Si se representa el texto con un "array", entonces las operaciones de inserción y eliminación son más costosas porque obligan a reasignar muchos elementos.

Las estructuras dinámicas se manejan con definiciones de datos recursivas.

```
struct lista {
        char   dato;
        struct lista  *siguiente;
} nodo;
```



La flecha representa "estar apuntando a . . ."

- **El compilador no reserva memoria para ese tipo de objetos.**

- **La creación del objeto y asigación de memoria se hace en tiempo de ejecución, invocando la función de biblioteca "malloc":**

malloc (*tamaño*)

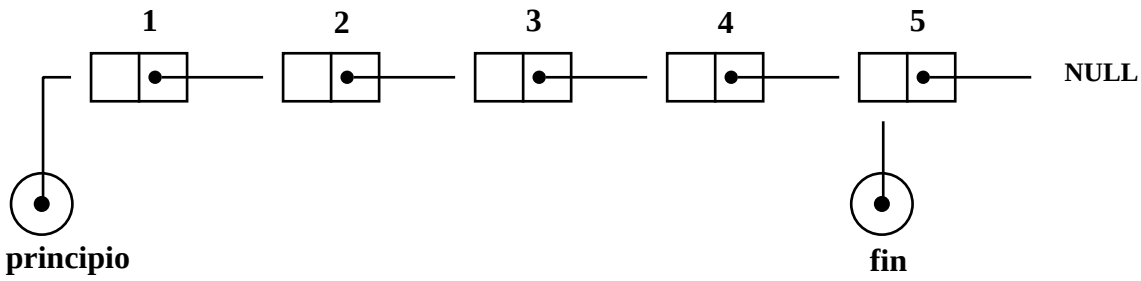
- **Los valores de ese tipo que se crean durante la ejecución no se pueden identificar por el nombre de variable, que es sólo un identificador genérico.**

Se asigna memoria con "malloc()"

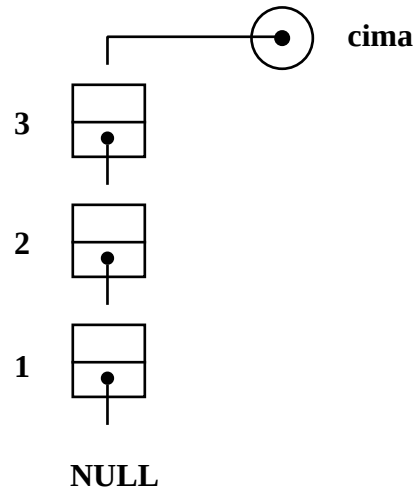
Se libera memoria con "free( )"

Estructuras dinámicas típicas

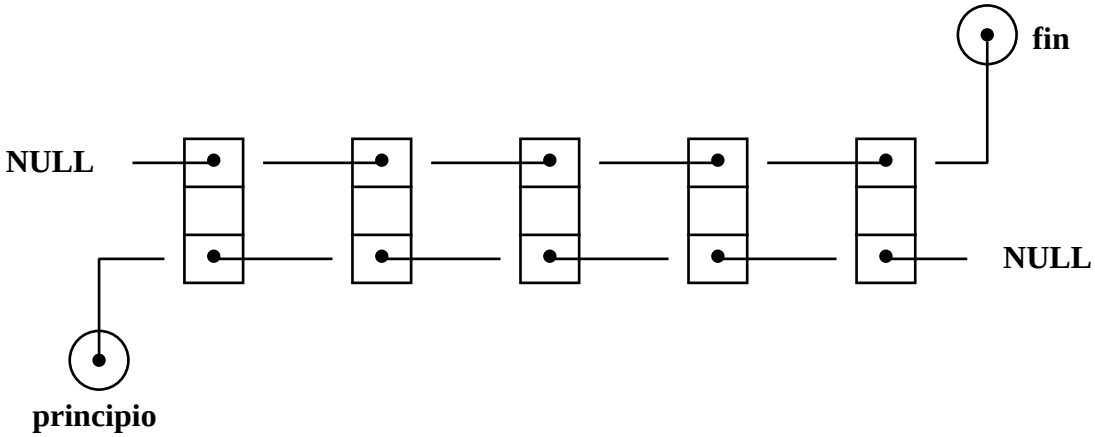
COLA. Lista encadenada, con disciplina FIFO



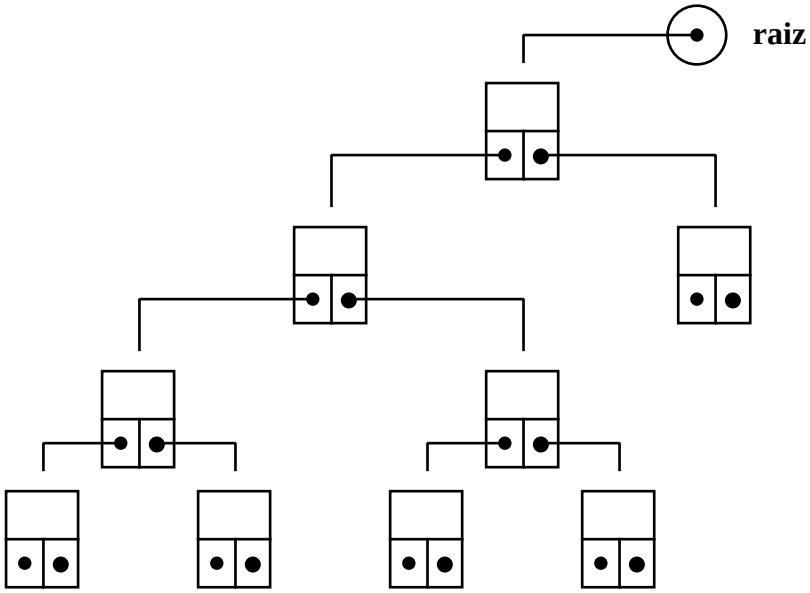
PILA. Lista encadenada, con disciplina LIFO



LISTA CON ENCADENAMIENTO DOBLE



ARBOL BINARIO



La función "malloc"

void *malloc (tipo_size longitud);

La función asigna una región de memoria para un objeto de datos de tamaño "longitud", y devuelve la dirección del primer "byte" de esa región ("tipo_size" es el tipo del valor que devuelve el operador "sizeof").

Los valores almacenados en el objeto de datos quedan indefinidos.

En caso de error (no hay memoria suficiente, ...) devuelve el puntero nulo.

```
struct lista {
     char dato;
     struct lista *siguiente;
} nodo, *p;
. . .
p = malloc ( sizeof(lista) );
. . .
p = (struct lista *) malloc (sizeof(lista) );
. . .
if ( p=malloc(sizeof(lista)) ) procesar( );
else printf("error\n");
```

La función "free"

void free (void *puntero);

Se libera la asignación de memoria correspondiente al objeto de datos cuya dirección indica "puntero".

Si el puntero tiene valor nulo, no hace nada.

Se puede liberar asignación de memoria de cualquier objeto de datos al que se haya asignado previamente con "malloc", "calloc" o "realloc".

```
struct lista {
        char dato;
        struct lista *siguiente;
} nodo, *p;
. . .
p = malloc ( sizeof(lista) );
. . .
free ( p );
. . .
```

Encadenamiento de datos

```
struct lista {
        char dato;
        struct lista *siguiente;
} x, y, z, *p;
```

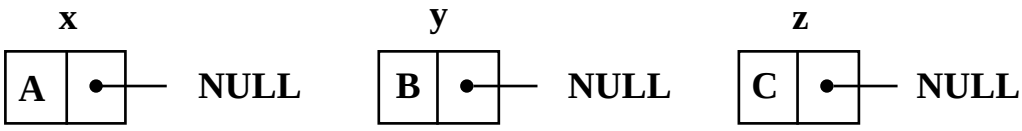
x, y, z son variables de tipo "struct lista"

p es un puntero a datos de tipo "struct lista"

Hay un campo para almacenar datos y otro para hacer conexiones.

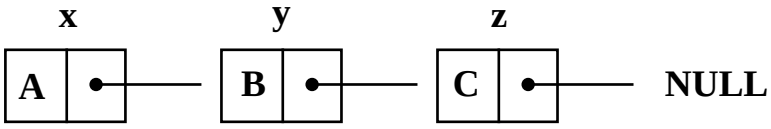
Asignación de valores:

```
x.dato='A';
y.dato='B';
z.dato='C';
x.siguiente = y.siguiente = z.siguiente = NULL;
```



Encadenamiento:

```
x.siguiente = &y;
y.siguiente = &z;
```

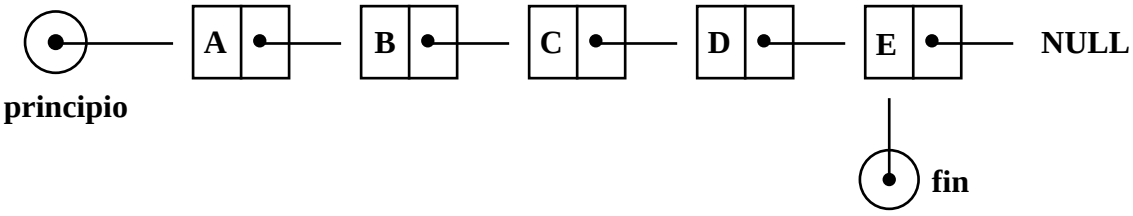


El campo de enlace permite acceder al siguiente elemento hasta llegar a una marca de final convenida.

x.siguiente -> dato	**es 'B'**
x.siguiente -> siguiente -> dato	**es 'C'**

Pero sería muy poco útil usar variables diferentes para manejar estructuras dinámicas.

Construcción de una lista encadenada en forma lineal



- **El final de la lista se marca asignando el valor nulo al campo puntero del último elemento.**

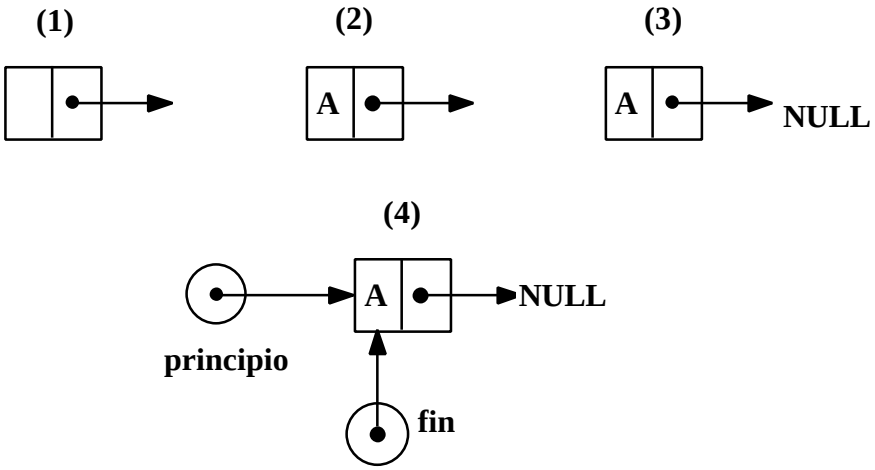
- **Se manejan dos punteros auxiliares para poder acceder al primer elemento y al último.**

- **Se usa una variable genérica para representar los elementos de la lista.**

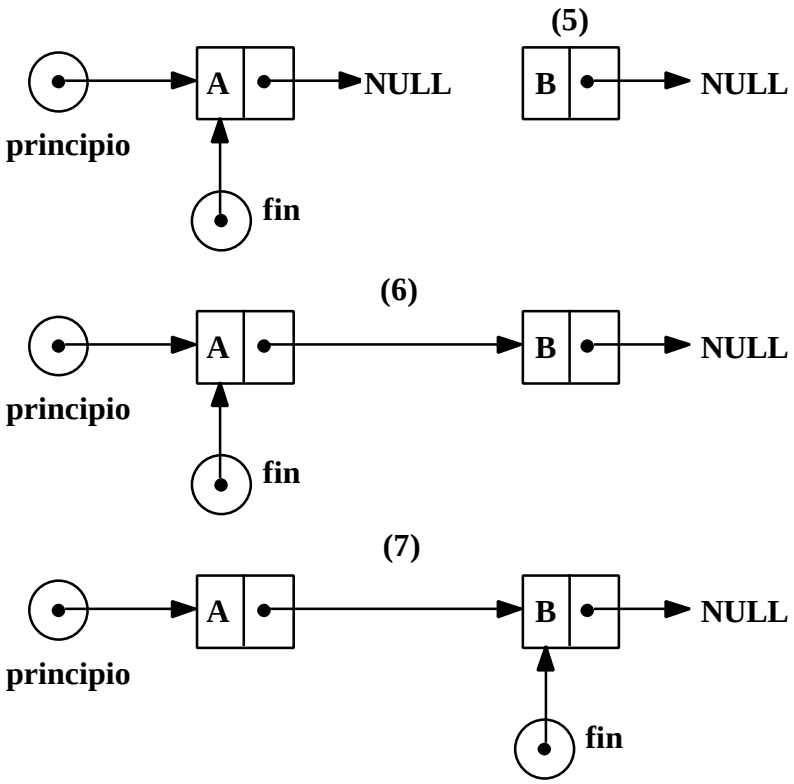
Definición de datos:

```
# define NULL 0

typedef char TIPO_DATOS;
struct lista {
    TIPO_DATOS dato;
    struct lista *siguiente;
};

typedef struct lista ELEMENTO;
typedef ELEMENTO *ENLACE;

ENLACE nodo, principio, fin;
```

```
nodo = malloc( sizeof(ELEMENTO) );        (1)
nodo -> dato = 'A';                       (2)
nodo = NULL;                              (3)
principio = fin = nodo;                   (4)
```

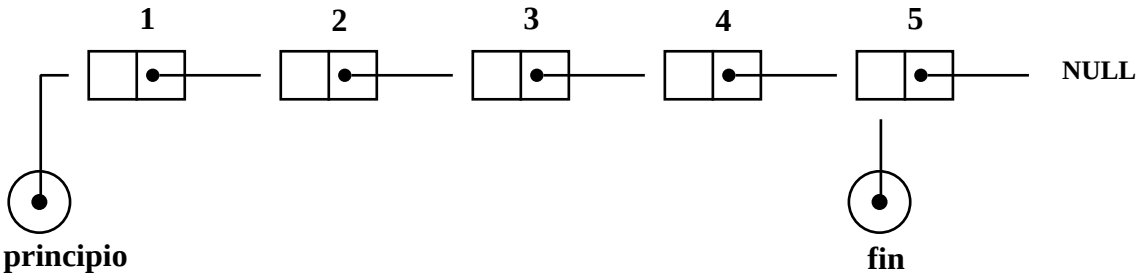



```
nodo = malloc ( sizeof(ELEMENTO) );
nodo -> dato = 'B';
nodo -> siguiente = NULL;       (5)
fin -> siguiente = nodo;        (6)
fin = fin -> siguiente;         (7)
```



Ejemplo

Se propone un programa para manejar una Cola (Lista encadenada con disciplina FIFO).



- **El campo de datos de cada elemento es de tipo "char".**
- **Los datos se introducen por pantalla en forma de cadena de caracteres.**
- **Desde el teclado se pueden ejecutar en orden arbitrario las operaciones de adición , eliminación y listado de elementos.**
- **La adición de un elemento nuevo se hace por el final de cola.**
- **La eliminación de un elemento se hace por el principio de cola.**
- **El listado se hace recorriendo la cola desde el principio hasta el final.**

```
/*
    ej34.c
    Manejo de una estructura dinamica con disciplina de
    cola (FIFO).
*/
# include <stdio.h>
# include <stdlib.h>
typedef char TIPO_DATO;
struct lista {
    TIPO_DATO dato ;
    struct lista *sig;
};
typedef struct lista ELEMENTO;
typedef ELEMENTO *ENLACE;

void Opcion_1(ENLACE *, ENLACE *);
void aniadir(ENLACE *, ENLACE *, ENLACE);
void Opcion_2(ENLACE *, ENLACE *);
ENLACE retirar(ENLACE *, ENLACE *);
void Opcion_3(ENLACE);
void listar(ENLACE);
void leer_cadena(char *, int);
void mensaje(void), preguntar(void);

# define NULL 0
# define LONCAD 80
```

```
void main()
{
    ENLACE principio, fin;
    int seguir=1; char  c;
    mensaje();
    principio=(ENLACE)malloc(sizeof(ELEMENTO));
    fin=(ENLACE)malloc(sizeof(ELEMENTO));
    principio=fin=NULL;
    while ( seguir ) {
        preguntar();
        while ( (c=getchar()) == '\n' ) ; getchar();
        switch ( c ) {
            case '1' :  Opcion_1 ( &principio, &fin );
                        break;
            case '2' :  Opcion_2 ( &principio, &fin );
                        break;
            case '3' :  Opcion_3 ( principio );
                        break;
            case '4' :  seguir=0;
                        break;
            default :   printf("Elija una opcion correcta %c\n",
                                7);
        }
    }
    printf("FIN DE LA EJECUCION\n");
}
```

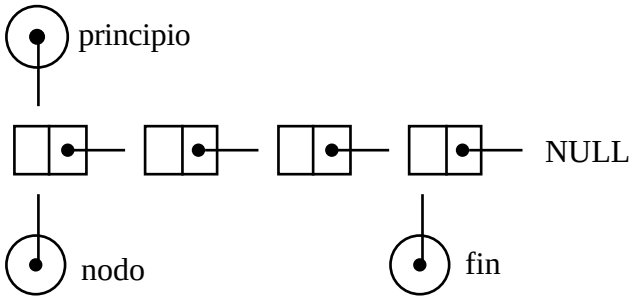
```
/*
     Opcion_1
     Se lee una cadena de caracteres y se añaden a la cola
     por el final con algoritmo iterativo.
*/
void Opcion_1 (ENLACE *principio, ENLACE *fin)
{
     ENLACE nuevo;
     int i;
     char s[80];
     printf("Introducir una cadena: ");
     leer_cadena(s, LONCAD);
     for (i=0; s[i] != '\0'; ++i) {
          nuevo=(ENLACE)malloc(sizeof(ELEMENTO));
          nuevo->dato=s[i];
          nuevo->sig=NULL;
          aniadir(principio, fin, nuevo);
     }
}
```

```
void aniadir(   ENLACE *principio, ENLACE *fin,
                        ENLACE nuevo     )
{
     if ( *fin ) {
          (*fin)->sig=nuevo;
          *fin = (*fin)->sig;
     }
     else *principio = *fin = nuevo;
}
```

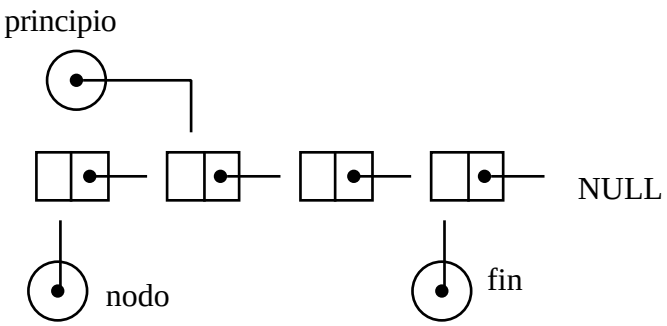
```
/*
    Opcion_2
    Eliminar un elemento por el principio de cola.
*/
void Opcion_2 (ENLACE *principio, ENLACE *fin)
{
    ENLACE nodo;
    if ( *principio ) {
        nodo = retirar (principio, fin);
        printf ("Elemento retirado: %c\n", nodo->dato);
        free(nodo);
    }
    else printf("No hay elementos en cola\n");
}
```

```
ENLACE retirar ( ENLACE *principio, ENLACE *fin )
{
    ENLACE tope;
    tope= *principio;
    *principio= (*principio)->sig;
    if (*principio) tope->sig=NULL;
    else *fin=NULL;
    return (tope);
}
```

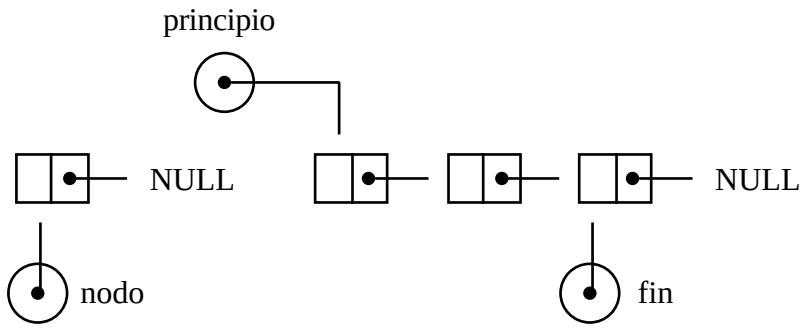
Retirar un elemento



nodo = principio;



principio = principio -> sig;



nodo -> siguiente = NULL;

```
void Opcion_3 (ENLACE p)
{
     if ( p==NULL ) printf("No hay elementos en cola\n");
     else listar(p);
}
```

```
void listar (ENLACE p)
{
     while ( p->sig != NULL ) {
          printf("%c", p->dato);
          p=p->sig;
     }
     printf("%c", p->dato);
     printf("\n");
}
```

```
void leer_cadena(char *s, int lon)
{
     int i=0;
     while ( (s[i]=getchar()) != '\n' ) if ( i<(lon-1) ) i++;
     s[i]='\0';
}



void mensaje()
{
     printf("Programa COLA\n");
     printf("Este programa permite manejar una Cola: ");
     printf("Lista encadenada\n");
     printf("con disciplina FIFO (primero en entrar, ");
     printf("primero en salir)\n");
}



void preguntar()
{
     printf("\nQue opcion ( 1-Aniadir, 2-Retirar, ");
     printf("3-Listar, 4-   Terminar) ? \n");

}
```

Ejemplo de algoritmo recursivo para manejar listas encadenadas.

Es la forma natural de manejar datos definidos recursivamente.

En el siguiente programa se construye una lista encadenada con los caracteres de una cadena que se lee por pantalla, y se recorre dos veces escribiendo los campos de de datos.

```
/*
        ej35.c
        Creacion y recorrido de una lista encadenada con
        algoritmo recursivo.
*/

# include <stdio.h>
# include <stdlib.h>
# define NULL 0

struct lista {
        char dato;
        struct lista *siguiente;
};
typedef struct lista ELEMENTO;
typedef ELEMENTO *ENLACE;

ENLACE cadena(char *);
void listar(ENLACE);

void main()
{
        ENLACE principio;
        char s[80];
        printf("Introducir una cadena: \n");
        scanf("%s", s);
        principio=cadena(s);
        listar(principio);
        printf("\n");
        listar(principio);
        printf("\n");
}
```

```
/*
        Creacion de lista encadenada con algoritmo recursivo
*/

ENLACE cadena(char s[ ])
{
        ENLACE principio;
        if ( s[0] == '\0' ) return(NULL);
        else {
                principio=(ENLACE)malloc(sizeof(ELEMENTO));
                principio->dato=s[0];
                principio->siguiente=cadena(s+1);
                return(principio);
        }
}




/*
        Se recorre una lista con algoritmo recursivo escribiendo
        los campos de datos.
*/
void listar(ENLACE principio)
{
        if ( principio == NULL ) printf("NULL");
        else {
                printf("%c --> ", principio->dato);
                listar(principio->siguiente);
        }
}
```