El lenguaje de Programación C

---

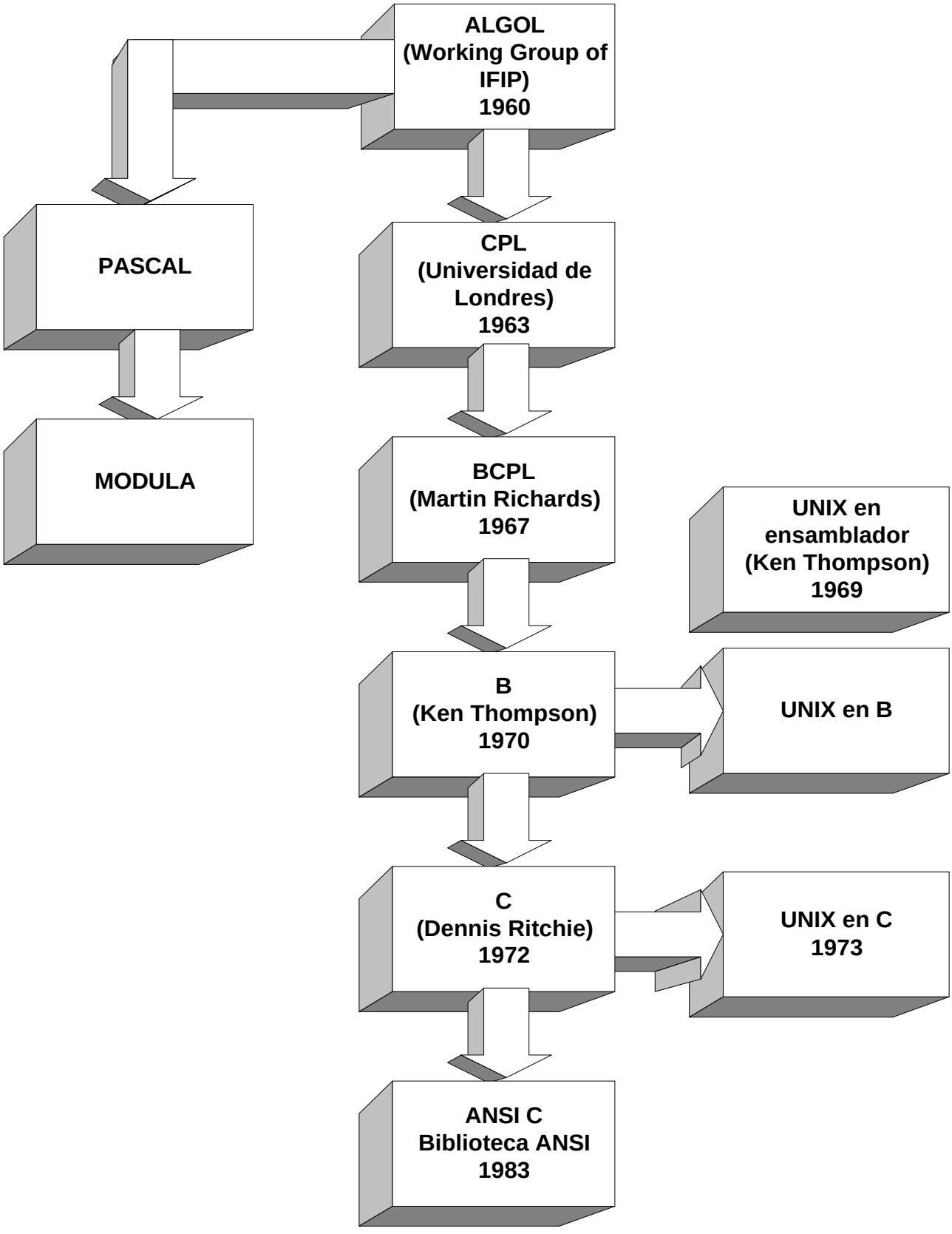
1. Introducción.
2. Elementos básicos del lenguaje.
3. Las estructuras de control.
4. Estructura de un programa.
5. Tipos de datos estructurados.
6. Las entradas y salidas de C.

1. Introducción

---

1. El origen del C.
2. La estructura de un programa en C.
3. Un primer programa en C.
4. Fases de la compilación.
5. La compilación separada.

El origen del lenguaje C



La estructura de un programa en C

- Un programa en C está constituido por una sucesión de funciones.
- Siempre existe una función llamada main.
- Una función se compone de:
  - Una cabecera

    Tipo nombre_de_función (argumentos)

  - Una secuencia de instrucciones agrupadas en un bloque

    {

    Instrucciones

    }

```
/*Este programa convierte una cantidad de minutos */
/* a su equivalente en horas y minutos */

#include <stdio.h>

void main(void)
{
        int x, horas, minutos;

        printf("Cuantos minutos? ");
        scanf("%d", &x);
        horas   = x / 60;
        minutos = x % 60;
        printf("%d minutos son %d hora(s) y %d
                minuto(s)",x,horas,minutos);
}
```

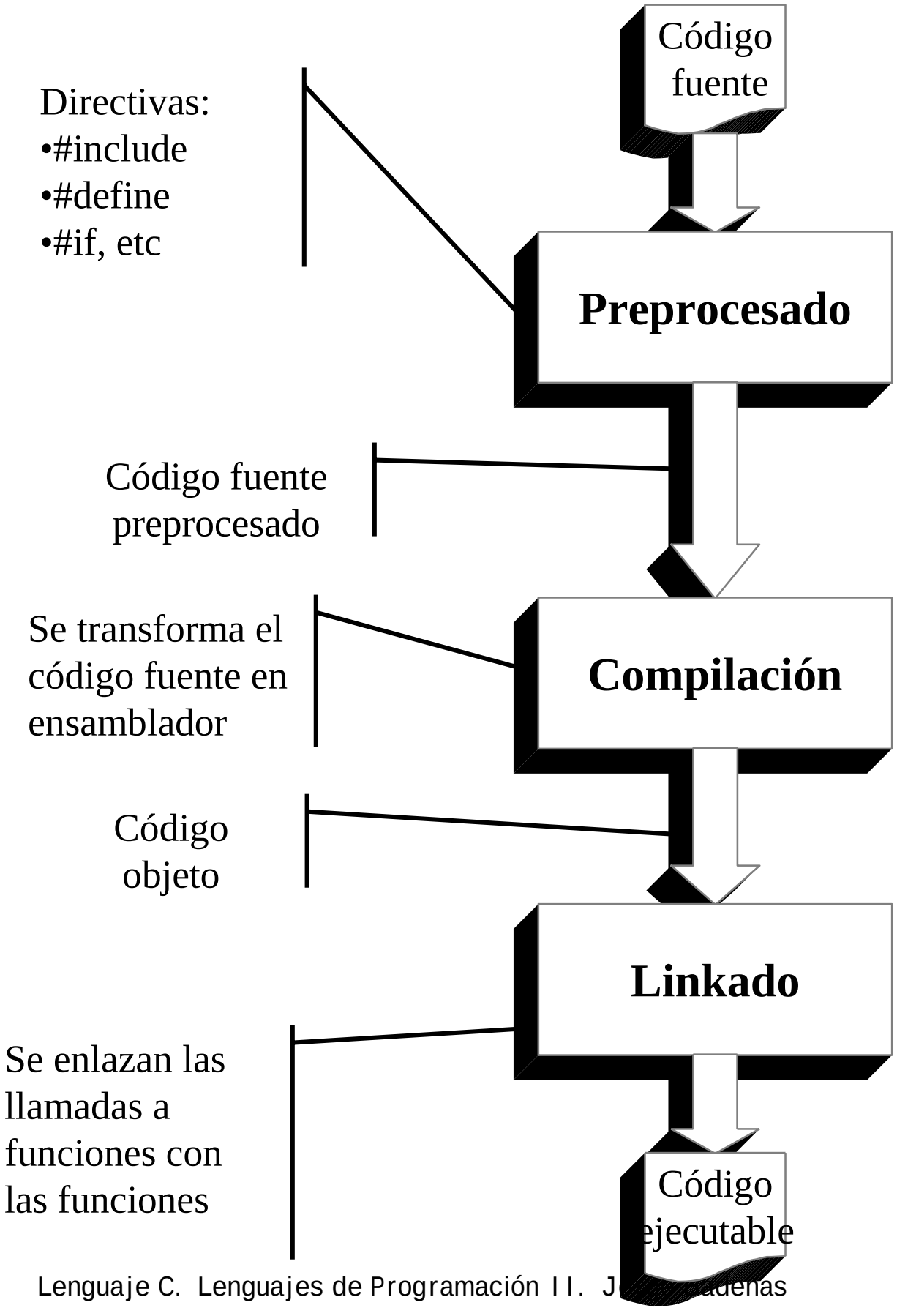
- La salida del anterior programa es:

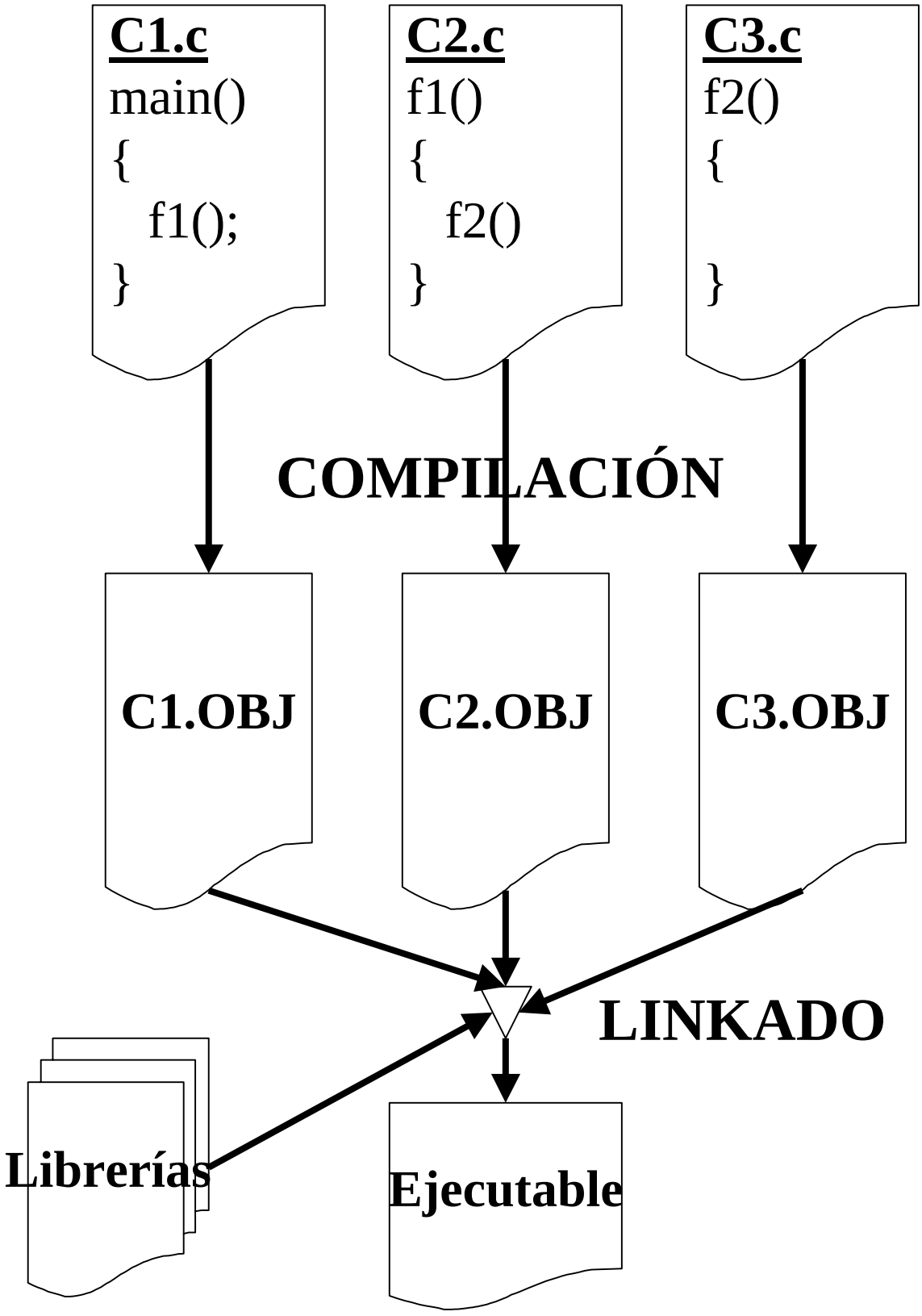
```
Cuantos minutos? 136
 136 minuto(s) son  2 hora(s) y 16 minuto(s)
```

Ejemplo: El área de un triángulo

```
/* Programa que calcula el área de un triángulo */

#include <stdio.h>

void main (void)
{
        float base, altura, area;

        printf ("Introduzca la base:");
        scanf ("%f", &base);
        printf ("Introduzca la altura:");
        scanf ("%f", &altura);
        area = base * altura / 2;
        printf ("El área del triángulo es: %f", area);
}
```

Fases de la compilación




C1.c
main()
{
f1();
}
C2.c
f1()
{
f2()
}
C3.c
f2()
{
}
COMPILACIÓN
C1.OBJ
C2.OBJ
C3.OBJ
LINKADO
Librerías
Ejecutable

2. Elementos fundamentales del lenguaje C

---

1. Identificadores.
2. Tipos de datos.
3. Constantes y variables.
4. Operadores, expresiones y sentencias.
5. Conversión de tipos.
6. Punteros.

- Un identificador es cualquier combinación de **letras**, **números** y **guiones de subrayado**.

  _Media8

  Calcular_Sumatorio

- No se puede comenzar un identificador con un número.

  8_Media ⇒ *No válido*

  – *Las palabras clave de C son en minúsculas.*

- Se distingue entre mayúsculas y minúsculas.

  CASA≠casa

- Se recomienda utilizar identificadores descriptivos.

- Los tipos de datos fundamentales son:
  - **char** carácter
  - **int** entero
  - **float** real
  - **double** real de doble precisión
- Modificadores de los tipos fundamentales:
  - **short** entero corto
  - **long** entero largo y double largo
  - **unsigned** entero sin signo
  - **signed** entero con signo

Rango de los tipos de datos

- El rango de los valores de los tipos de datos es dependiente de la implementación.
- Consultar limits.h y float.h
- No obstante, si que hay definidos unos mínimos:

Tipo	*Rango mínimo de valores*
char	-128 a 127
unsigned char	0 a 255
signed char	-128 a 127
int	-32768 a 32767
unsigned int	0 a 65535
signed int	-32768 a 32767
short int	-32768 a 32767
unsigned short int	0 a 65535
signed short int	-32768 a 32767
long int	-2147483648 a 2147483647
signed long int	-2147483648 a 2147483647
unsigned long int	0 a 4294967295
float	6 dígitos de precisión
double	10 dígitos de precisión
long double	10 dígitos de precisión[1]

Tabla 2.1. Rangos de los tipos de datos en C

- A diferencia de PASCAL, en C no existe un tipo booleano.
- Tampoco existen las constantes TRUE y FALSE, aunque podemos definirlas.
- Lo que se usa como valores booleanos son los tipos enteros:
  - FALSE es 0, '\0' o NULL
  - TRUE es cualquier valor diferente de 0.
- Una expresión como:

  a>5

  es evaluada en tiempo de ejecución por el programa y en caso de ser falsa el resultado es **cero**, en caso contrario **un valor distinto de cero** (puede ser cualquier valor).

Definición de tipos

- C dispone de una declaración denominada **typedef** para la creación de nuevos nombres de tipos de datos.

```
typedef tipo nombre_del_nuevo_tipo;
```

- Por ejemplo:

```
typedef unsigned short int UNSHORT;
```

define el tipo UNSHORT. Este tipo de dato nuevo será un entero corto sin signo, utilizable como si fuese un tipo básico.

- En C existen cuatro tipos básicos de constantes:
  - enteras
  - reales
  - carácter
  - cadena de caracteres.
- Constantes enteras
  - Decimal: 19, -213, 3
  - Octal: 023, -0325, -03
  - Hexadecimal: 0x13, 0Xd5, 0x3
  - Decimal sin signo: 19U, 213U, 3U
  - Octal sin signo: 023U, 0325U
  - Hexadecimal sin signo: 0x13U, 0Xd5U
  - Decimal largo: -213L, 3000000L
  - Octal largo: 023L, -0325L
  - Hexadecimal largo: 0x13L, 0Xd5L
  - Decimal largo sin signo: 190000UL

- Constantes reales:

  4.7, -5.65, 4E3, -7e-2, 4.8E4, -7.2e-2

  - La letra E substituye al número 10. Así, el número 4.8 x 10^4 es el 4.8E4.
  - Las siguientes constantes representan el mismo valor:

    0.00075E1, 0.0075, 0.075E-1, 0.75E-2, 75E-4

- Constantes carácter:
  - **Representación**: Un único carácter entre comillas simples (‘).
  - Una constante carácter tiene asociado un valor numérico: su código ASCII.
  - Una secuencia de escape es una pareja de caracteres que representan uno solo. El primero es siempre la barra inclinada \

Significado	*Mnemónico*	*Símbolo*
Nueva línea	NL (LF)	\n
Tabulación horizontal	HT	\t
Espacio atrás	BS	\b
Retorno de carro	CR	\r
Salto de página	FF	\f
Antislash	\	\\
Apóstrofe	'	\'
Comilla doble	“	\”
Configuración de bits	ddd (d en base 8)	\ddd

Tabla 2.2. Secuencias de escape

- Constantes cadena:
  - Representación: una secuencia de caracteres delimitados por comillas dobles “.

    “Esto es una cadena”

    “INICIO\n\nFIN”

    “Comillas dobles \” y simples \’”

    “C:\\Directorio\\Fichero.txt”
  - Toda cadena tiene un carácter final de código ASCII 0 (‘\0’) que el compilador sitúa automáticamente al final de la cadena.

- Una variable es un identificador que hace referencia a una posición de memoria.
- Todas las variables deben ser declaradas antes de ser usadas.
- Las variables pueden declararse en:
  - Dentro de una función. **Variables locales**.
  - En la cabecera de una función. **Argumentos.**
  - Fuera de todas las funciones. **Variables globales**.
- Es posible dar un valor inicial a las variables.

```
int i=0;
char a='a';
```

- La declaración de constantes:

```
const float pi=3.14;
const char A = 'A';
```

- **Expresión**: combinación de operadores y expresiones.
- La expresión más simple puede ser una constante, una variable o una llamada a una función.
- Una expresión acabada en punto y coma (;) se convierte en una **sentencia**.
- Una secuencia de sentencias dentro de llaves { } forma una sentencia (**sentencia compuesta**).

Operadores aritméticos

Operador	Acción
-	Sustracción y signo unario
+	Adición
*	Multiplicación
/	División
%	Módulo (resto de la división entera)
--	Decremento
++	Incremento

Sentencia	Equivalencia
z++;	z=z+1;
++z;	z=z+1;
x=++z;	z=z+1; x=z;
x=z++;	x=z; z=z+1;

Operadores relacionales

Operador	Acción
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
==	Igual
!=	Distinto

- El resultado de estos operadores es un valor entero (0 FALSE, otro valor TRUE).
- En los primeros programas que se escriben en C es frecuente confundir los operadores == y != al usar sus equivalentes en PASCAL, = y <>.
- <> producirá un error de compilación.
- = no produce error de compilación, pues es un operador válido en C.

Operadores lógicos

Operador	Acción
&&	AND (Y lógico)
||	OR (O lógico)
!	NOT (negación)

Condición	Resultado lógico
5	Cierto
!5	Falso
!0	Cierto
5<6 && 7>=5	Cierto
5>6 && 7>=5	Falso
6>5 || 7>=5	Cierto
5>6 || 7<=5	Falso
!(5>6 || 7<=5)	Cierto

Operadores de manejo de bits

Operador	Acción
~	Complemento a 1 unario
&	AND bit a bit
|	OR inclusivo bit a bit
^	OR exclusivo (XOR) bit a bit
<<	Desplazamiento a la izquierda
>>	Desplazamiento a la derecha

Operadores de asignación

Operador	Acción
=	Asignación
operador=	Asignación especial

Sentencia	Equivalencia
a=b=c;	a=c; b=c;
a+=b;	a=a+b;
a*=b;	a=a*b;

- El operador ?: es un operador de asignación condicional.
- Su sintaxis es:

```
(expresión_1)?expresión_2:expresión_3
```

- Ejemplo

```
x=(a>5)? 1 : 0;
```

- Es equivalente a:

```
Si a>5 entonces x=1
       sino x=0
```

Operador	Acción
&	Dirección de un objeto (unario).
*	Direccionamiento indirecto (unario).
[]	Direccionamiento indexado.
sizeof(_)	Da la talla de la variable o el tipo entre paréntesis.
,	Sirve para reunir varias expresiones en una instrucción.
.	Da acceso a los campos de las estructuras.
->	Da acceso a los campos de las estructuras accedidas por punteros.

- Cuando en una expresión aparecen operandos de distinto tipo:
  - Conversión automática sin pérdida de información.
    - los datos se convierten al más general (*char* a *int*, *int* a *float*, etc.)

```
int i=20; float f;
f=i * 3.1416;
```

  - Conversión explícita o *type casting*
    - Se obliga a una expresión a evaluarse con un tipo que se indica explícitamente.

```
x=(unsigned int) (y/655)%32;
```

- Un **puntero** es una variable que contiene direcciones.
- Una variable puntero es una variable como cualquier otra, lo que cambia es el tipo de los datos que almacena.
  - Debemos inicializarla antes de usarla.
  - Lo que le asignemos debe corresponder con el tipo de la variable (una dirección).
- Declaración:

```
int *px, *py; /* Punteros a int */
char *pc;     /* Puntero a char */
void *pv;  /* Puntero a cualquier
  cosa */
```

- Asignación:
  - Asignación de otro puntero:

```
px=py;
```

  - Asignación de una dirección:

```
int x, *px;
px=&x;
px=0xFF0E;
px=NULL; /* Dirección 0*/
```

- **Operador asterisco**: permite acceder a un dato apuntado por un puntero.

```
int x = 5;
int *px;
px=&x;
printf ("%d", *px);/*Escribirá 5*/
```

  - No es lo mismo px=py que *px=*py.
    - px=py asigna la dirección de py a px.

```
int x=5, y=10;
int *px, *py;
px=&x; py=&y;
px=py;
```

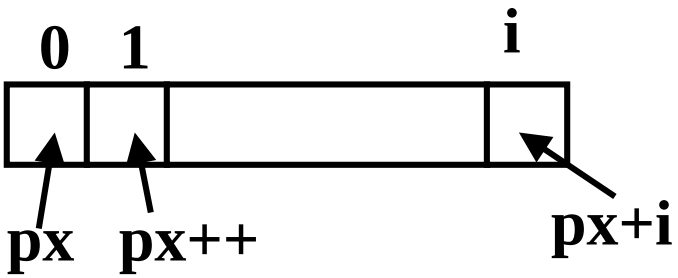
      - **Resultado**: x→ 5, y→10, *px→10, *py→10

    - *px=*py asigna el contenido de donde apunta py al lugar donde apunta px.

```
int x=5, y=10;
int *px, *py;
px=&x; py=&y;
*px=*py;
```

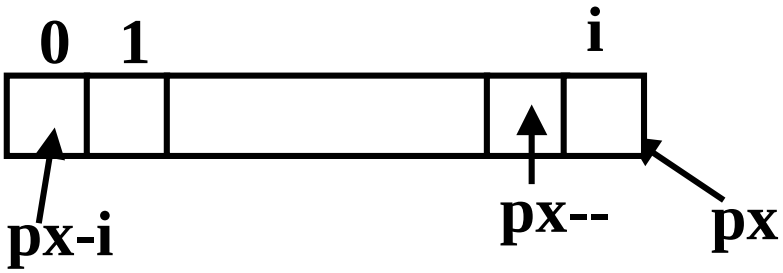
      - **Resultado**: x→ 10, y→10, *px→10, *py→10

- Suma y resta de punteros.
  - Puntero + entero:

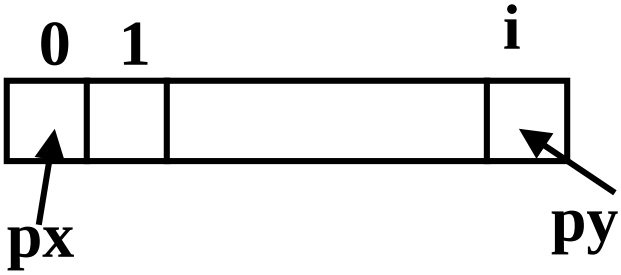


  - Puntero + puntero: No válido.

  - Puntero – entero:



  - Puntero – puntero:



py-px → i

- Comparación (==, >, !=, etc.)
  - Se comparan los valores numéricos de las direcciones.
- Punteros a puntero.
  - Dado que un puntero es una variable, su dirección también puede ser contenida por un puntero.

```
int x=5;
int *px;
int **ppx; /*puntero a puntero*/
px=&x;
ppx=&px;
```

- **Resultado**: x→5,*px→5,**ppx→5

```
**ppx=6;
```

- **Resultado**: x→6, *px→6, **ppx→6

```
ppx=&x;
```

- **Error**: Distinto nivel de indirección

Ejercicio de punteros

- ¿Qué resultado muestra en pantalla el siguiente programa?

```
int x=5, y=2;
int *px, *py, *pz;

px=&x;
printf("%d", *px);
py=&y;
printf("%d", *py);

*px=*py;
printf("%d", x);

*px=5;
px=py;
printf("%d", x);
printf("%d", *px);

*pz=6;
pz=NULL;
*pz=6;
```

3. Las estructuras de control

---

1. Estructura de selección simple.
2. Estructura de selección múltiple.
3. Estructuras de repetición.
4. Control de las estructuras de repetición.

Estructura de selección simple

- La estructura de selección simple (**if-else**) tiene la siguiente sintaxis:

  if (expresión condicional)

  sentencia

  else

  sentencia

- La cláusula **else** es opcional.
- Los paréntesis son obligatorios.
- La sentencia puede ser simple o compuesta.
- Ejemplo:

```
if (a>5)
    printf("a es mayor que 5");
else
    printf("a no es mayor que  5");
```

- El **else** se asocia al **if** más cercano.
- Se pueden anidar if's.

```
switch ( expresión entera)
{
   case constante1: secuencia de sentencias;
                    break;
   case constante2: secuencia de sentencias;
                    break;
   ...
   default: secuencia de sentencias;
}
```

- Ejemplo:

```
switch(a)
{
case 1:     printf("vale 1");
            break;
case 2:
case 3:     printf("Vale 2 o 3");
            break;
default:    printf("Vale otra cosa");
}
```

- Existen tres estructuras de repetición:
  - while
  - do-while
  - for
- Estructura **while**:

while (condición)
    sentencia;

```
while (p!=7)
    p++;

while (x<5)
{
    printf("%d", x);
    x++;
}
```

- Estructura **do-while**:

```
do
{
   secuencia de sentencias;
}
while (condición);
```

```
do
{
   printf("Deme un número del 1 al 10:\n");
   scanf("%d", &i);
}
while (i<1 || i>10);
```

- Estructura **for**:

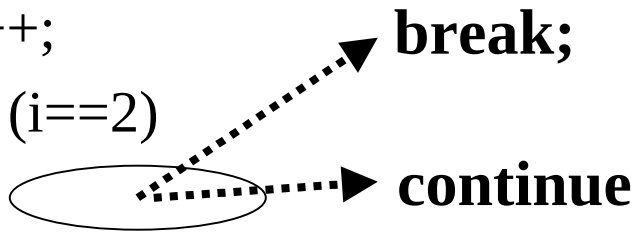
for (inicialización; condición; progresión)
    sentencia;

```
for (i=0; i<10; i++)
    printf("%d", vector[i];

for (i=0, j=0; i<10 && j<10; i++, j++)
    printf("%d", matriz[i][j]);

for (encontrado=0, i=0; i<10 && !encontrado; i++)
    if (vector[i]==valor)
      encontrado=1;

encontrado=0;
i=0;
for(;i<10 && !encontrado; )
{
    if (vector[i]==valor)
      encontrado=1;
    i++;
}
```

Sentencias de control en las estructuras de repetición

- Existen dos palabras clave en C que provocan un salto dentro de una estructura de repetición:
  - **break**: interrumpe la ejecución de un bucle saltando a la instrucción siguiente.
  - **continue**: produce un salto a la evaluación de la condición.

```
i=0;
while (i<5)
{
   i++;
   if (i==2)
      break;
   printf("%d", i);
}
```



Salida con continue: 1, 3, 4, 5

Salida con break: 1

4. Estructura de un programa en C

---

1. Funciones.
2. Paso de parámetros por referencia.
3. Tipos de variables.
4. El preprocesador.

- Un programa está constituido por funciones.
- El programa principal lo constituye la función **main**.
- No existe una diferenciación entre procedimientos y funciones, como en PASCAL.
- Una función es un sub-programa al cual se le transmite una lista de argumentos y que devuelve un valor.
- Los valores en una función se retornan por medio de la palabra **return**.

- Existen dos formas de declarar funciones:
  - Declaración clásica (K&R) considerada obsoleta:

```
tipo_retorno nombre_función (parámetros)
declaración de parámetros;
{
   declaración de variables;

   instrucciones;
}
```

  - Declaración moderna (ANSI):

```
tipo_retorno nombre_función(parámetros
   con sus tipos)
{
   declaración de variables;

   instrucciones;
}
```

- Declaración K&R:

```
float maximo( a, b)
float a,b;
{
        return (a>b)? a: b;
}
```

No se puede poner:
float a, b

- Declaración ANSI:

```
float maximo( float a, float b)
{
        return (a>b)?a:b;
}
```

- En ambos casos, la forma de llamar a la función es la misma:

```
float x=3.3,y=2.2,z;

z=maximo(x,y);
```

- Los **prototipos de funciones** tienen como finalidad que el compilador conozca el tipo de los parámetros de una función cuando se produce la llamada a ésta.
- Los **prototipos** son declaraciones de funciones, cuya definición aparecerá posteriormente, en el mismo fichero u otro que vaya a ser enlazado a éste.
- Sólo son necesarios cuando la llamada a una función aparece antes que su declaración.
- Los ficheros a los que se llama con **#include** contienen los prototipos de las funciones que incorpora el compilador.

```
#include <stdio.h>

float maximo( float, float); /* Prototipo*/

void main ( void )
{
   float x, y, z;

   printf("Dame x e y:");
   scanf ("%f,%f", &x,&y);
   z=maximo(x,y);
   printf("El máximo es %f", z);
}
float maximo(float x, float y)
{
   return (a>b)?a:b;
}
```

Si no estuviera el prototipo, el compilador asumiría que z es de tipo **int**

- El equivalente al PROCEDURE de PASCAL se implementa mediante la definición de una función que no devuelve nada.

```
#include<stdio.h>

void mensaje(void);

void main(void)
{
   mensaje();
}

void mensaje (void)
{
   printf("Esto es un mensaje\n");
}
```

Se deben poner paréntesis, aunque no se pasen parámetros

El tipo **void** indica que no devuelve nada

Para hacer que la función concluya se puede escribir: **return;**

Paso de parámetros a funciones

- En la mayoría de lenguajes de programación existen dos tipos de pasos de parámetros:
  - Paso de parámetros por valor.
  - Paso de parámetros por referencia.
- En C no existe el paso de parámetros por referencia, auque se imita por medio de los punteros.
- En C++ sí existen ambos tipos de pasos de parámetros.

- **Paso de parámetros por valor**. Lo que se pasa a la función es únicamente un valor, sin que exista una relación entre la variable cuyo valor es pasado a la función y el parámetro que recibe el valor.

```
/* Cuerpo de una función cualquiera*/
int a=5;
...
a=5;
f(a); /* Llamada a la función f*/
...
}
void f( int p)
{
      p=30;
}
```

Al ser un paso por valor, a **p** se la ha asignado el valor de **a**, pero no hay ninguna relación entre ambas variables.

Aunque se modifique **p**, la variable **a** sigue valiendo 5.

Paso de parámetros por referencia

- En el paso de parámetros por referencia el parámetro es un nombre alternativo de la variable que le corresponde en la llamada a la función. Es decir, son la misma variable, así si se modifica el parámetro la variable queda modificada.

```
/* Cuerpo de una
función cualquiera*/
    int a=5;
    ...
    a=5;
    f(a);
    ...
    }
    void f( int & p)
    {
          p=30;
    }
```

En C++, el símbolo & en un parámetro indica que es un paso por referencia.

- Se verá en la parte de C++.
- No se puede usar en las prácticas de C.

Al ser un paso por referencia, lo que se le haga a **p** es como si se le hiciese a **a**.

Se está modificando **a**, pues **p** hace referencia a **a**.

- En C sólo existe el paso por valor.
- Para conseguir que una función pueda modificar una variable que se le pasa en una llamada, debemos pasar su dirección.

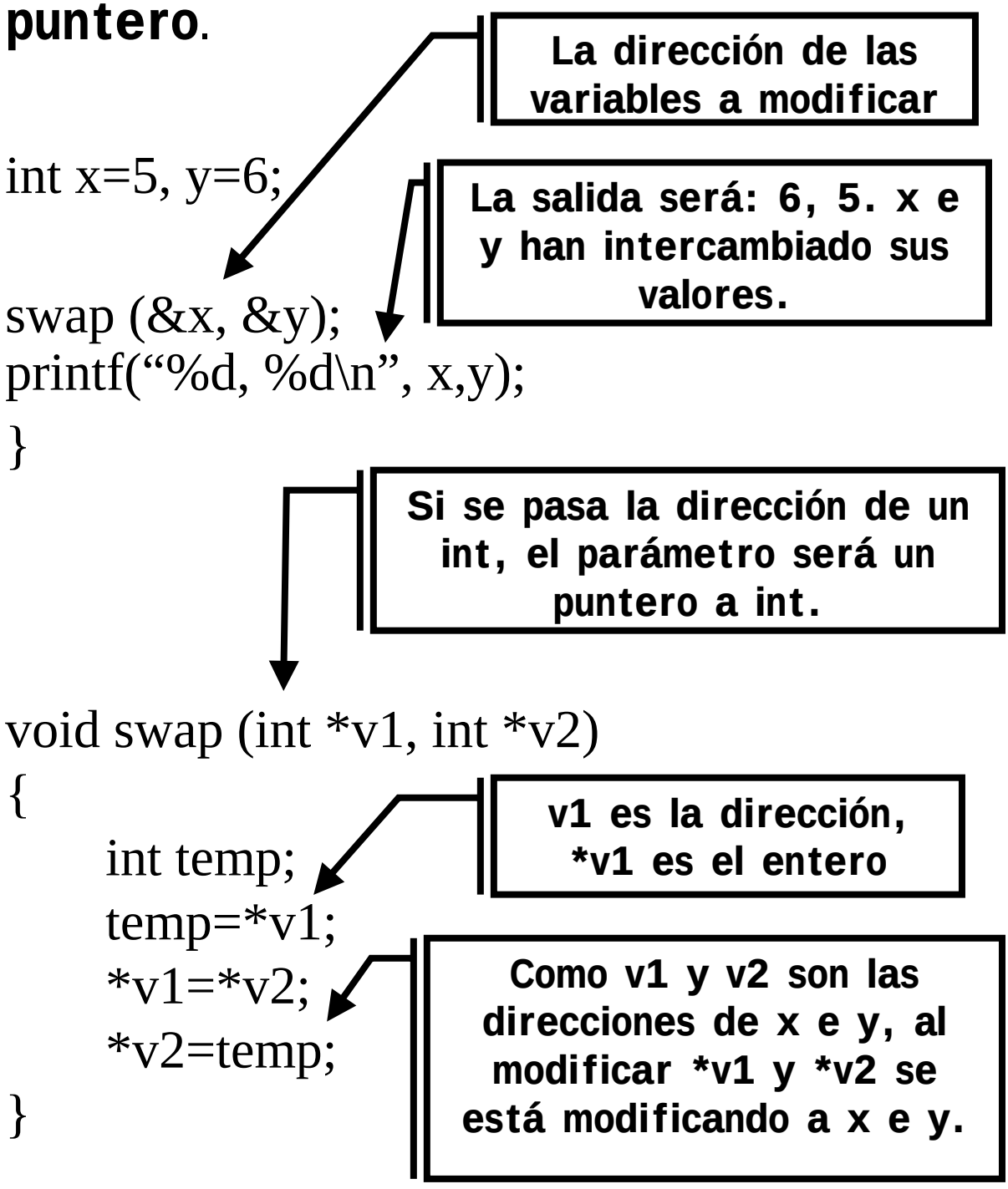
- int a = 6;
  f(a);

Aquí lo que en realidad se está pasando es el valor 6, por lo que la función no podrá modificar a *a*

- int a = 6;
  f(&a);

Aquí se está pasando la dirección de a. A través de la dirección, la función puede modificar a a.

- Si se pasa una dirección a una función, el parámetro correspondiente tendrá que ser un **puntero**.




```
int x=5, y=6;

swap (&x, &y);
printf("%d, %d\n", x,y);
}

void swap (int *v1, int *v2)
{
        int temp;
        temp=*v1;
        *v1=*v2;
        *v2=temp;
}
```

```
/* Programa que pide dos valores y devuelve su suma */
#include <stdio.h>

void main(void)
{
    int x, y, resultado;

    PedirValor(&x);
    PedirValor(&y);

    SumarValores (x,y, &z);
    printf("La suma de %d y %d es %d",x,y,resultado);
}
void PedirValor(int *v)
{
    printf("Dame un valor:");
    scanf("%d", v);
}
void SumarValores(int v1, int v2, int *r)
{
    *r=v1+v2;
}
```

- En C se distinguen cuatro tipos de variables:
  - Externas
  - Automáticas
  - Estáticas
  - Registro
- Las variables **externas** (globales) se definen fuera de todas las funciones.
- Pueden ser leídas y modificadas desde cualquier función que se encuentre detrás de su declaración.
- Si se quiere utilizar una variable externa en un fichero diferente a donde está declarada, se la debe volver a declarar en el segundo fichero, pero precedida por la palabra reservada **extern**.

Tipos de variables: **extern**

```
/* Fichero 1 */

int variable_externa1;

void main( void )
{
    variable_externa1=1;
    fun();
}
int
  variable_externa2;
...
```

Variable externa (global)

Variable externa, se podrá usar en las funciones definidas a partir de aquí.

```
/* Fichero 2 */

extern int
    variable_externa1;
void fun( void )
{
   extern int variable_externa2;
   variable_externa1=2;
   variable_externa2=2;
}
...
```

La palabra extern indica que esta variable está definida en otro fichero

Esta variable sólo podrá usarse en esta función

- Las variables **automáticas** se definen dentro de las funciones.
- Su vida empieza cada vez que empieza a ejecutarse la función donde están.
- Desaparecen cada vez que la función concluye.
- Se crean y destruyen de manera automática.
- A lo largo de una ejecución de un programa, una variable automática se creará tantas veces como se ejecute la función donde ha sido declarada.

```
void funcion( void)
{
    char a;
    ...
}
```

- Las variables estáticas (static) son un caso intermedio entre las variables extern y auto:
  - Se declaran dentro de las funciones al igual que las auto. Sólo son accesibles desde la función donde se declaran.
  - Se crean y se destruyen sólo una vez en cada programa al igual que las extern. Su espacio de memoria se mantiene durante toda la ejecución del programa. Sólo se inicializan una vez.

```
int CalculoRaro(void)
{
  static int var = 0;

  var += 10;
  return var;
}
```

La primera vez que se ejecuta la función *CalculoRaro* devuelve 10, la segunda 20, la tercera 30, etc.

Tipos de variables: **register**

- Si una variable se declara como **register**, será situada en un registro de la CPU, siempre y cuando haya alguno disponible.
- Se emplea para variables a las que se acceda muy frecuentemente, ya que es más rápido acceder a un registro que a la memoria del ordenador.

```
register int i;

for (i=0; i<10000; i++)
        ...
```

- Las directivas del preprocesador son procesadas en un primer paso de la compilación.
- Tres tipos de directivas:
  - Inclusión de ficheros **#include**
  - Substitución simbólica **#define**
  - Compilación condicional **#if, #else, etc**.
- **#define** define una cadena que substituirá a otra:

```
#define   TALLA   100
#define   PI      3.14
#define   BEGIN   {
#define   END     }
#define   max(a,b)  ((a>b)?a:b)
```

- **#include** incluye un fichero de texto.
  - `#include <stdio.h>` En el directorio del compilador.
  - `#include "mio.h"` En mi directorio.

- **#if, #else, #endif**, etc.
  - Permiten definir partes del programa que serán compiladas o no dependiendo de una condición que se evalúa en tiempo de compilación.

```
#if defined(HDR)
   /* Código a compilar en el primer caso*/
...
#else
   /* Código a compilar en el segundo caso*/
...
#endif
```

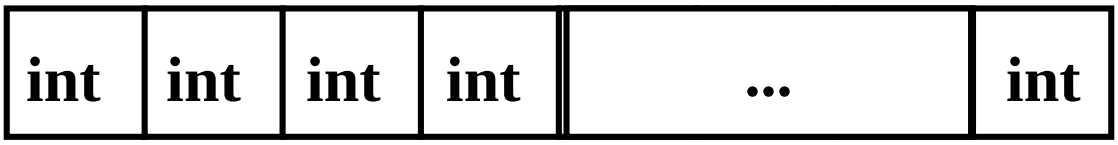
```
#define TALLA   100
#define FALSE   0
#define TRUE                !FALSE
#define max(a,b) ((a)>(b))?(a):b))

i=1;
condicion=TRUE;
mayor=0;
while (condicion==TRUE && i<TALLA)
{
    printf("Introduzca un valor:");
    scanf("%d", &num);
    if (num<=0)
        condicion=FALSE;
    else
        mayor=max(mayor, num);
}

i=1;
condicion=!0;
mayor=0;
while (condicion==!0 && i<100)
{
    printf("Introduzca un valor:");
    scanf("%d", &num);
    if (num<=0)
        condicion=0;
    else
        mayor=((mayor)>(num)) ? (mayor):(num);
}
```

5. Tipos de datos estructurados

---

1. Vectores.
2. Vectores n-dimensionales.
3. Funciones para el manejo de bloques de memoria.
4. Cadenas de caracteres.
5. Vectores y punteros.
6. Estructuras.
7. Uniones.
8. El tipo enumerado.
9. Argumentos de la función main.
10. Memoria dinámica.

- Un **vector** es un conjunto de elementos del mismo tipo que se sitúan en posiciones contiguas de memoria.



- Declaración:

  tipo nombre [tamaño];

  int vector_enteros [20];

  char vector_caracteres[30];

- El primer elemento de un vector es el 0:

  int a[2]; ➔a[0], a[1]

  Los elementos de un vector de tamaño n van desde el 0 hasta n-1.

- Al igual que para cualquier variable, el operador & permite saber la dirección de memoria que ocupa un elemento:

  &a[5]➔ Dirección del 6º elemento del vector a

- El nombre de un vector es la dirección de memoria donde se encuentra:

```
a==&a[0]
```

- La dirección del i-ésimo elemento de un vector es:

```
&a[i] == a+i
```

- El índice que aparece entre corchetes puede ser cualquier expresión que dé como resultado un valor entero.
- Los vectores externos y estáticos pueden ser inicializados:

```
int vi [5]={1,2,3,4,5};
```

- Cuando se pasa un vector a una función se pasa su dirección (referencia):

```
f(a);
```

- Declaración:

```
tipo nombre
  [tamaño1][tamaño2]···[tamaño n];
```

- Cuando en C se declara un vector como el siguiente:

```
float mf[10][30];
```

Se está declarando un vector de 10 elementos, donde cada elemento es un vector de 30 float's. Cada uno de los 10 vectores puede ser tratado individualmente:

```
mf[9][5] ➔    Un float.
mf[5] ➔ Un vector de float's.
mf ➔    Un vector de vectores de
  float's.
```

El posicionamiento de los 10 vectores sigue siendo adyacente.

```
/* Programa que encuentra el máximo en un vector */
#include <stdio.h>

void PedirElementos ( int v[], int n);
int Mayor (int v[], int n);

void main (void)
{
    int vector[10];

    PedirElementos (vector, 10);
    printf("El máximo es: %d\n", Mayor(vector, 10);
}
void PedirElementos ( int v[], int n)
{
    int i;
    for (i=0; i<n; i++)
        scanf (%d", &v[i]);
}
int Mayor (int v[], int n)
{
    int i, maxim=-32000;
    for(i=0; i<n; i++)
        if (maxim<v[i]) maxim=v[i];
    return maxim;
}
```

```
/* Operaciones con vectores bidimensionales */
#include <stdio.h>

void RellenaVector ( long v[], int l, int val);
void RellenaMatriz ( long m[][10], int l, int a, int val);
void VisualizaVector (long v[], int l);
void VisualizaMatriz (long m[][10], int l, int a);

void main (void)
{
    long v1 [10];
    long v2 [5][10];

    RellenaVector (v1, 10, 0);
    RellenaMatriz (v2, 5, 10, 0);
    VisualizaVector ( v1, 10);
    VisualizaMatriz ( v2, 5, 10);
}
void RellenaVector ( long v[], int l, int val)
{
    register int i;
    for (i=0; i<l; i++) v[i]=val;
}
void RellenaMatriz ( long m[][10], int l, int a, int val)
{
    register int i;
    for (i=0; i<l; i++) RellenaVector (m[i], a, val);
}

/* Intentad completar las funciones de visualización */
```

Funciones para el manejo de buffers

- **memcpy** Copia n bytes de s a d

  void *memcpy(void *d,void *s,size_t n);
- **memchr** Busca el carácter c en n bytes de s

  void *memchr(void *s,int c,size_t n);
- **memcmp** Compara n bytes de s1 y s2

  int memcmp(void *s1, void *s2,size_t n);
- **memset** Pone c en n caracteres de s

  void *memset(void *s,int c,size_t n);

- Las cadenas de caracteres son agrupaciones de caracteres posicionados sobre un vector de tipo char.
- Las cadenas de caracteres tienen siempre un carácter final ASCII 0 que delimita los caracteres que pertenecen a la cadena.
- Inicialización:

  char cadena [8]="Hola";

H	**o**	**l**	**a**	**'\0'**			

Funciones para el manejo de cadenas

- **strcpy** Copia un string en otro

  char *strcpy(char *destino, char *origen);
- **strcat** Añade un string a otro

  char *strcat(char *destino,char *origen);
- **strchr** Busca un carácter

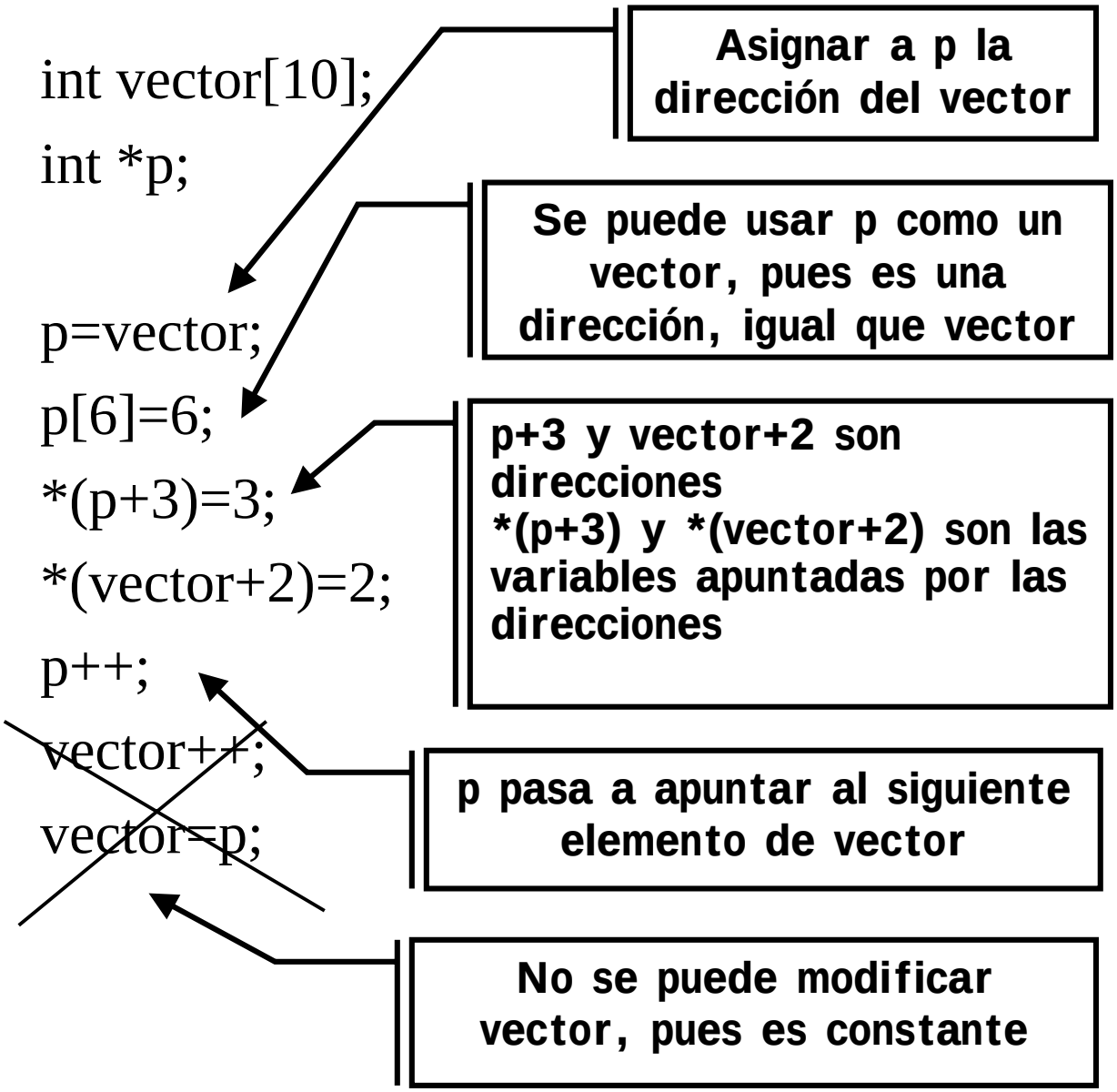
  char *strchr(char *s, int c);
- **strcmp** Compara dos strings

  int strcmp(char *s1, char*s2);
- **strlen** Calcula la longitud

  size_t strlen(char *s);
- **strstr** Busca un string en otro

  char *strstr(char *s1, char *s2);

- El nombre de un vector es la dirección donde empieza el vector.
- Por lo tanto, el nombre de un vector es un puntero constante.



```
char * strcpy (char s[], char t[])
{
    int i=0;
    while ((s[i]=t[i])!='\0')      i++;
    return s;
}

char *strcpy (char *s, char *t)
{
    while ((*s++=*t++)!='\0');
    return s;
}

int capicua (char *s)
{
    char *p, *q;

    for (p=s, q=s+strlen(s)-1; *p==*q && q>p;
          p++, q--);
    return p>q;
}
```

- Una **estructura** (struct) es un conjunto que comprende una o más variables que pueden ser de distinto tipo, y que se agrupan bajo el mismo nombre.
- Declaración:

```
struct dir
{
        int numero;
        char calle[16];
        long codigoPostal;
        char ciudad[15];
};
struct cuenta
{
        long numero;
        char nombre[18];
        struct dir direccion;

} variableCuenta;
```

- Inicialización

```
struct familia
{
    char apellido[10];
    char nombrePadre[10];
    char nombreMadre[10];
    int numeroHijos;
} fam1={"García", "Juan", "Carmen", 3};

struct familia fam2={"Pitarch", "Ramón",
                     "Amparo", 7};
```

- Operaciones sobre estructuras:
  - Para acceder a un campo de una estructura se emplea el operador punto .

    ```
    fam2.numeroHijos=8;
    ```

  - La dirección de una estructura se obtiene mediante el operador &.

    ```
    struct familia *pf;
    pf=&fam2;
    ```

  - Si pf es un puntero a una estructura, se puede acceder a los campos de dos formas:

    ```
    (*pf).numeroHijos=9;
    pf->numeroHijos = 9;
    ```

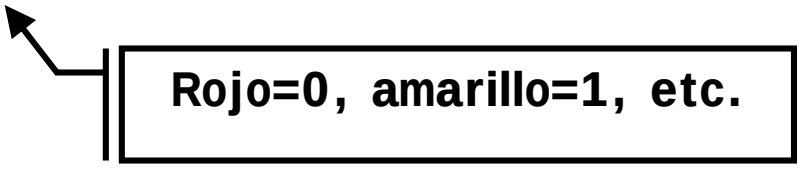
- Una **unión** es un conjunto de elementos de los que sólo puede contener uno de ellos en un momento dado.
- Permite manipular datos de distintos tipos y diferentes tallas usando una única zona de memoria.

```
union datos
{
    int i;
    char c[5];
} unionEjemplo;
```

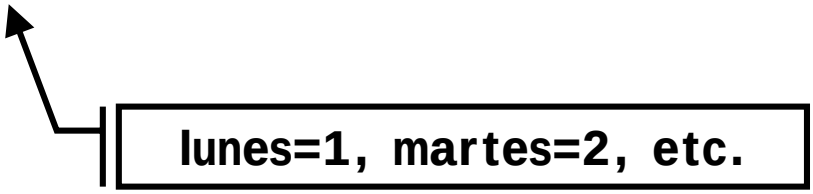
- Se accede a los campos de igual manera que en los struct.

- Permite definir una lista de constantes bajo un nombre de tipo.
- Definición:

```
enum arcoiris { rojo, amarillo, verde, azul};
```




```
enum semana {lunes=1, martes, miercoles};
```



Argumentos de la función **main**

- La función main puede recibir parámetros que vienen de la llamada al programa.
- Un programa puede recibir tantos argumentos como se desee, pero la función main sólo recibe 2:
  - El primer argumento es un entero que indica el número de parámetros que ha recibido el programa.

    Siempre el primer argumento es el nombre del programa, por lo que todo programa tiene al menos un argumento.
  - El segundo y último parámetro es un vector de cadenas de caracteres, donde cada cadena es uno de los parámetros recibidos por el programa.

```
int main (int argc, char *argv[])
{
        int i;
        for (i=0; i<argc; i++)
        printf("Argumento %d=%s\n", i, argv[i]);
        return 0;
}
```

- La reserva de memoria en C se hace mediante la función **malloc**.
- La reserva de memoria se realiza en los siguientes pasos:
  - Se pide un bloque de memoria de n bytes.
  - El gestor de memoria marca como reservado un fragmento de memoria que esté libre y lo apunta en la tabla de asignación de memoria.
  - El gestor de memoria devuelve la dirección de memoria donde a localizado el bloque reservado.
- Que un bloque esté reservado significa que nadie, que no sea el proceso que lo ha reservado, puede escribir en él. Se garantiza la integridad de los datos allí almacenados.
- Si metemos datos en una dirección no reservada:
  - No hay garantía de que no sean sobrescritos.
  - Si no se sabe donde se está escribiendo, se puede causar un error de ejecución.

- La liberación de memoria en C se hace con la función **free**.
- Liberar un bloque de memoria significa que ya no se garantiza la integridad de los datos que allí estén almacenados.
- Sólo se debe liberar un bloque de memoria cuando ya no se vaya a hacer uso de los datos allí almacenados.

```
struct datos {
    int v1:
    char vect[30];
};
...
struct datos *p;
p=(struct datos *)malloc(sizeof(struct datos));
...
free (p);
```

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

typedef struct
{
   int tam;
   int ult;
   int *mem;
} pila;

void Insertar ( pila *pp);
void Extraer ( pila *pp);
void main ( void )
{
    pila p;
   char num;

   printf ("Tamaño de la pila:\n");
   scanf ("%d", &p.tam);
   p.mem=(int *)   malloc(sizeof(int)*p.tam);
   p.ult=0;
   num='1';
   while ( num!='0')
   {
     num=getch();
     switch(num)
     {
      case '0': break;
      case '1': Insertar (&p);
                break;
      case '2': Extraer (&p);
                  break;
      }
   }
   free (p.mem);
}
```

```
void Insertar(pila *pp)
{
  if (pp->ult==pp->tam)
  {
          printf("P. llena\n");
    return;
   }
   printf("Valor: ");
   scanf ("%d", &(pp->mem[pp->ult]));
   pp->ult++;
  }

void Extraer(pila *pp)
{
    if (pp->ult==0)
  {
          printf("P.vacia\n");
          return;
   }
   pp->ult--;
   printf("El valor es: %d\n",
                pp->mem[pp->ult]);
  }
```

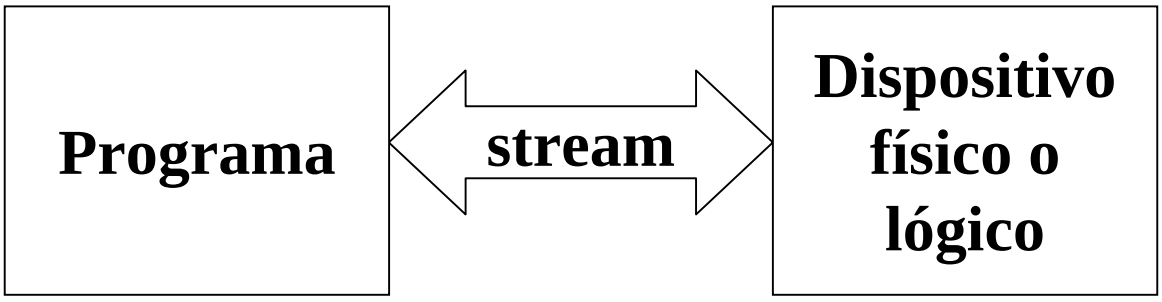
```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

#define FALSE 0
#define TRUE !0

typedef struct lista
{
  int info;
  struct lista *next;
} LISTA;

void Insertar ( LISTA **, LISTA **);
void Extraer (LISTA **, LISTA **);
void LiberarLista ( LISTA **);
char menu (void);

void main ( void )
{
  LISTA *Cabeza;
  LISTA *Cola;
  int salir=FALSE;

  Cabeza=NULL;
  Cola=NULL;
  while (!salir)
  {
    switch(menu())
    {
    case '0': salir=TRUE; break;
    case '1': Insertar (&Cabeza, &Cola); break;
    case '2': Extraer (&Cabeza, &Cola); break;
    }
  }
  LiberarLista (&Cabeza);
}
```

```
void Insertar (LISTA **PCap, LISTA **PCola)
{
  LISTA *aux;
  aux=(LISTA *)malloc(sizeof(LISTA));
  printf("Introduzca un valor:");
  fflush(stdin);
  scanf("%d", &aux->info);
  aux->next=NULL;
  if(*PCola)
  {
    (*PCola)->next=aux;
    *PCola=aux;
  }
  else
    *PCap=*PCola=aux;
}
void Extraer (LISTA **PCap, LISTA **PCola)
{
  LISTA *aux;
  if (*PCap==NULL)
  {
    printf ("La lista está vacía\n");
    return;
  }
  printf("El valor es %d\n", (*PCap)->info);
  aux=*PCap;
  *PCap=(*PCap)->next;
  if (*PCap==NULL)
    *PCola=NULL;
  free (aux);
}
```

```
void LiberarLista (LISTA **PCap)
{
  LISTA *aux;

  while (*PCap!=NULL)
  {
    aux=*PCap;
    *PCap=(*PCap)->next;
    free(aux);
  }
}

char menu (void)
{
  char c;
  do
  {
    printf("0...Salir\n");
    printf("1...Insertar\n");
    printf("2...Extraer\n");
    c=getch();
  }
  while (c<'0' || c>'2');
  return c;
}
```

Entrada/salida en C

1. Streams.
2. Algunas funciones de E/S.
3. Archivos.



- Se distinguen dos tipos de streams:
  - Texto
  - Binarios
- Existen varios streams que siempre están definidos en C:
  - stdin
  - stdout
  - stderr

- La función de salida más significativa es **printf**:

  printf (cadena de control, lista de variables);

- La cadena de control puede contener caracteres de formato:

Código de formato	*Significado*
d	número entero
l	número entero largo
o	número octal
u	número entero sin signo
x	número hexadecimal
e	float o double notación exponencial
f	float o double notación decimal
g	el más corto de e o f
s	cadena de caracteres
c	carácter individual
-	ajustar a izquierda
número	amplitud del campo
. (punto) *número*	amplitud del campo detrás del punto decimal

- Dadas las siguientes declaraciones:

```
int i = 1001;
float x = 543.456;
char a = 'A';
char c [ 11 ] = "1234567890";
```

Se obtendrán las siguientes salidas para cada llamada a `printf`:

- printf ("i =%d, x =%f, a =%c, c =%s|\n", i, x, a, c);
  i =1001, x =543.455994, a =A, c =1234567890|
- printf ("i =%3d, x =%3f, a =%3c, c =%3s|\n", i, x, a, c);
  i =1001, x =543.455994, a = A, c =1234567890|
- printf ("i =%15d, x =%15f, a =%15c, c =%15s|\n",i,x,a,c);
  i = 1001, x = 543.455994, a = A, c = 1234567890|
- printf ("i =%.2d, x =%.2f, a =%.2c, c =%.2s|\n", i, x, a, c);
  i =1001, x =543.46, a =A, c =12|
- printf ("i =%.12d, x =%.12f, a =%.12c, c =%.12s|\n", i,x,a,c);
  i =000000001001, x =543.455993652344, a =A, c =1234567890|

- Otras funciones de salida son:

Nombre	*Acción*	*Prototipo*
`putchar()`	Escribe un carácter	`int putchar(int ch);`
`puts()`	Escribe una cadena	`int puts(const char *s);`

```
int i;
char mensaje[60]="Los caracteres del 32 al 128";

puts(mensaje);
for (i=32; i<=128; i++)
    putchar(i);
```

- La contrapartida a printf para entrada de datos es scanf:

  ```
  scanf (cadena de control, lista de variables);
  ```

- Su formato es similar al de printf.
- Se ha de pasar la dirección de las variables.
  - Una cadena de caracteres ya es una dirección.

    scanf("%s", cadena);
- A menudo es necesario utilizar la función **fflush** para limpiar el buffer de entrada, de manera que scanf no recoja basura.

  fflush(stdin);

  scanf("%c", &carácter);
- Otras funciones de entrada son:

Nombre	*Acción*	*Prototipo*
`getchar()`	Lee un carácter	`int getchar();`
`gets()`	Lee una cadena	`char *gets(char *s);`

- Para manejar un archivo se debe declarar un **FILE ***.
- **fopen** abre un fichero:

```
FILE *fp;
fp=fopen("FICHERO.TXT", "rb");
if(!fp)
        printf("ERROR\n");
```

- Existen 6 modos básicos de apertura:
  - Lectura ➔ r
  - Escritura ➔ w
  - Añadir ➔ a
  - Lect./escrit. ➔ r+ (Debe existir)
  - Lect./escrit. ➔ w+ (Crea o trunca)
  - Lect./añadir ➔ a+ (Crea el fichero)
- Estos modos se multiplican por 2, pues podemos abrir en texto o binario.
  - texto: rt, wt, at, r+t, w+t, a+t
  - binario: rb, wb, ab, r+b, w+b, a+b

- Cerrar un fichero: fclose(fp);
- Escribir un carácter: putc(c, fp);
- Leer un carácter: c=getc(fp);
- Escribir una cadena con formato:
  fprintf(fp, "%d", i);
- Leer una cadena con formato:
  scanf(fp, "%d", &i);
- Escribir un bloque de bytes:
  int v[30];
  fwrite (v, sizeof(int), 30, fp);
- Leer un bloque de bytes:
  int v[30];
  fread (v, sizeof(int), 30, fp);
- Saber si hubo error de final de fichero:
  i=feof(fp);
- Desplazamiento en el fichero:
  fseek(fp, offset, donde);
  - **offset** es un long y **donde** puede valer: 0,1,2.

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

void main(void)
{
  FILE *stream;
  char msg[] = "Prueba de un ejemplo con fread/fwrite";
  char buf[20];

  if ((stream = fopen("NADA.FIL", "w+")) == NULL)
  {
    fprintf(stderr, "No se puede abrir el fichero.\n");
    return 1;
  }

  /* escribimos datos en el fichero */
  fwrite(msg, strlen(msg)+1, 1, stream);

  /* localizamos el principio del fichero */
  fseek(stream, SEEK_SET, 0);

  /* leemos los datos y los mostramos en pantalla */
  fread(buf, strlen(msg)+1, 1, stream);
  printf("%s\n", buf);

  fclose(stream);
}
```

```
#include <stdio.h>

long filesize(FILE *stream);

void main(void) {
   FILE *stream;

   stream = fopen("FICHERO.TXT", "w+");
   fprintf(stream, "Prueba del fichero");
   printf("Tamaño del fichero FICHERO.TXT:  %ld
    bytes\n", filesize(stream));
   fclose(stream);
}
long filesize(FILE *stream)
{
  long curpos, length;
    /* recuperamos y guardamos la posición actual
     del puntero del fichero */
   curpos = ftell(stream);
   printf("El puntero del fichero esta en byte %ld\n",
    curpos);
   /*situamos el puntero en la situación final 'SEEK_END' */
   fseek(stream, 0L, SEEK_END);
   /* obtenemos el byte de la posición final y
     devolvemos el puntero a la posición que tenía
     antes de la llamada a esta función */
   length = ftell(stream);
   fseek(stream, curpos, SEEK_SET);
   return length;
}
```

```
#include <stdio.h>

void main () {
    FILE *fp;

    int i, j;
    fp = fopen ("Fichero.bin", "wb");
    for ( i = 0; i < 10; i++)
        fwrite (&i,sizeof(int),1, fp);
    fclose ( fp );

    fp = fopen ("Fichero.bin", "rb");
    fread (&j, sizeof ( int ), 1, fp);
    while (!feof(fp)) {
        printf ("%d\n", j);
        fread (&j, sizeof(int),1,fp);
    }
    fclose ( fp );

    fp = fopen ("Fichero.txt", "w");
    for ( i = 0; i < 10; i++)
        fprintf (fp,"%d\n",i);
    fclose ( fp );

    fp = fopen ("Fichero.txt", "r");
    fscanf (fp,"%d",&j);
    while (!feof(fp)) {
        printf ("%d\n", j);
        fscanf (fp,"%d",&j);
    }
    fclose ( fp );
}
```