Resumen lenguaje C

Eva Lucrecia Gibaja Galindo
Dpto. Informática y Análisis Numérico

Estructura general de un programa C

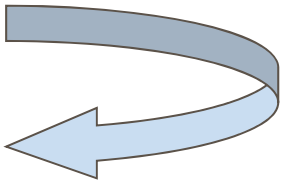
```
Directivas del preprocesador
•    #include
•    #define
```

```
Declaraciones globales:
*    prototipos de funciones
*    variables globales //No se deben utilizar
```

```
main()
{
   Declaración de variables locales

   Sentencias ejecutables
}
```

```
Definición de otras funciones
```

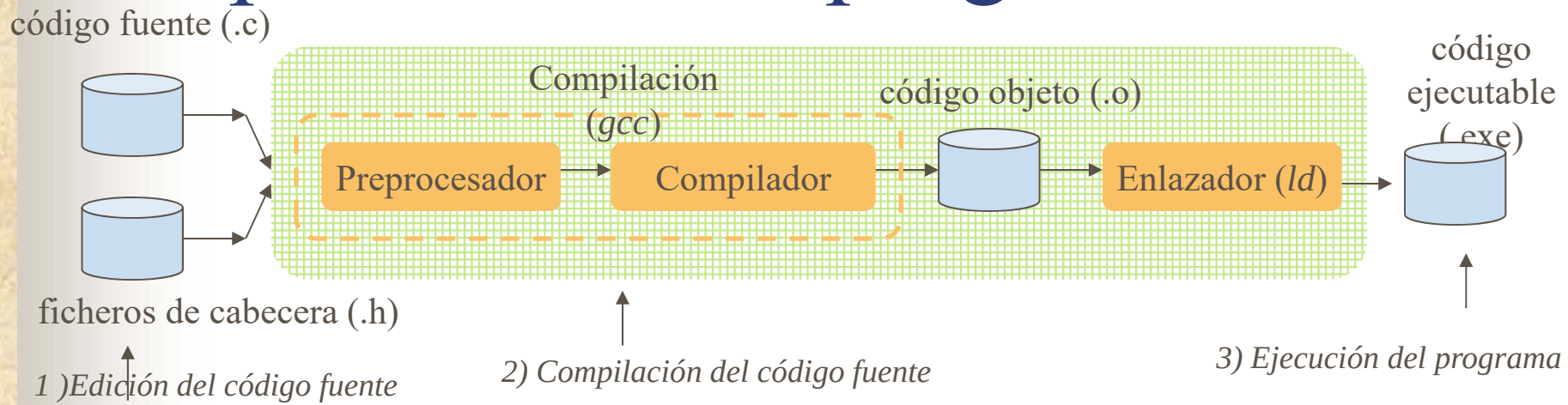


```
# include <stdio.h>
```

```
main()
{
  printf("Hola mundo");
}
```

- **guardar el código fuente con extensión .c**
- **hola.c, hola.o y hola.exe son tres ficheros diferentes**

Compilación de un programa C



- Llamada implícita al enlazador. No poner la opción -c
  - `gcc hola.c` → compila y genera *`a.exe`*
  - `gcc hola.c -o hola.exe` → compila y genera ejecutable *`hola.exe`*
- Genera solo el correspondiente .o
  - `gcc -c hola.c` → compila y genera `hola.o`
  - `gcc -c hola.c -o otroNombre.o` → compila y genera `otroNombre.o`
- Activar todos los warnings
  - `gcc -c -Wall hola.c -o hola.exe`
  - `gcc -Wall hola.c -o hola.exe`

Elementos del lenguaje C

- **Biblioteca estándar de C**. Conjunto de funciones para realizar entrada y salida de datos, administración de memoria, manipulación de cadenas, etc.
  - <***stdio.h***> Funciones de E/S
  - <***string.h***> Funciones para cadenas
  - <***math.h***> Funciones matemáticas. En este caso habrá que compilar con la opción ***-lm***
  - <***stdlib.h***> Reserva de memoria, conversión de datos, números aleatorios
  - <***limits.h***> <***float.h***>. Límites de implementación, INT_MAX, INT_MIN, etc.
  - <***time.h***> Funciones de fecha y hora

Constantes

- **Constantes** : Su valor no se puede modificar durante la ejecución del programa
  - **Constantes definidas :** *#define <nombre constante> <valor>*

```
#define Pi 3.1416
#define iva 16
#define ciudad "Cordoba"
```

  - **Constantes declaradas**: *const <tipo> <nombre constante> = valor ;*

```
const int edad = 67;
const char caracter ='a';
const char cadena[] = "hola";
```

Variables

- Representan un objeto cuyo valor cambia durante la ejecución del programa
- Declaración de variables

  <tipo> <nombre$_1$>, <nombre$_2$>,....,<nombre$_i$>;

  - Al hacer la declaración, se reserva la memoria necesaria para albergar los datos del tipo declarado
  - No sabemos nada sobre el valor de estas variables

- Algunos ejemplos de declaraciones:

```
int nElementos, suma=0; // dos enteros, inicializa suma a 0
float sueldo;                       //tipo real
char opcion = 'S';                  //tipo caracter
char dni[10], nombre[30]= "Plutarquete" ; //cadena de caracteres
```

Tipo de dato entero

Subconjunto del conjunto matemático *Z*. Variantes:

Tipo	Memoria	Rango
`int`	2 *bytes*	[-32768, 32767]
`unsigned int`	2 *bytes*	[0, 65535]
`short int`	2 *bytes*	[-128, 127]
`long`	4 *bytes*	[-2147483648, 2147483647]
`unsigned long`	4 *bytes*	[0, 4294967295]
`char`	*1 byte*	[0, 255]

Cuidado! Esta tabla sólo es orientativa. El rango de valores para cada tipo depende de cada S.O. y compilador !!

Tipo de dato entero. Operadores

- Operadores

Binarios (Dos operandos) Notación infija: a*b	+	Suma
	-	Resta
	*	Producto
	/	División
	%	Módulo
Unarios (un operando)	++	Incremento
	--	Decremento

- Nota:
  - Notación prefija. El operador va al principio de la expresión: *ab
  - Notación postfija. El operador va al final de la expresión: ab*
  - Notación infija. El operador va entre los dos operandos: a*b

Tipo de dato entero. Operadores

Operador	Sentencia abreviada	Sentencia no abreviada
+=	m+=n;	m=m+n;
-=	m-=n;	m=m-n;
*=	m*=n;	m=m*n;
/=	m/=n;	m=m/n;
%=	m%=n;	m=m%n;

Tipo de dato carácter

- Un carácter es cualquier elemento de un conjunto de caracteres predefinidos o alfabeto
  - La mayoría de las computadoras utilizan el código ASCII
- Código ASCII (256 caracteres): Conjunto finito y ordenado de caracteres. Cada carácter tiene un código numerado de 0 a 255
  - Caracteres de Control no imprimibles
  - ! “ · $ % & ’ ( ) * + , - . /
  - 0 1 2 3 4 5 6 7 8 9 0
  - : ; < = > ? @
  - A B C D E F G H I J K L...
  - a b c d e f g h i j k l....
- En unión con la estructura *array*, se utilizan para almacenar ***cadenas de caracteres*** (las veremos más adelante)

Ejercicio 5. Código ASCII

	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		

Tipo de dato carácter

- **En C no existe el tipo carácter como tal**
- Dos representaciones:
  - Internamente se representan como un número (código ASCII)
    - Un carácter necesita al menos 1 byte para almacenar su número de orden en un tipo numérico
    - Podemos utilizar tanto el tipo entero *char* (es el más adecuado), como el tipo entero *int* (de 0 a 255)
  - Como un literal de carácter. Se representan entre comillas simples:
    - ’!’ , ’A’, ’a’, ’5’
- Se pueden realizar operaciones aritméticas (las aplicables a los enteros)

```
char c;
c = 'A';      /* Almacena la 'A' . Equivale a c=65   */
c = c + 2;    /* Almacena la 'C'. Equivale a c= 65+2 */
```

Tipo de dato carácter

```
char c;
c = 'A';   /* Almacena 'A' (código 65) */
c = 65;   /* Almacena 'A' (código 65) */
c = '7';   /*  Almacena '7' (código 55) */
c = 7;    /* Almacena el caracter cuyo código es 7 */

//Imprime el caracter y su codigo ASCII

printf("caracter:%c codigo:%d\n", c, c);
```

Expresiones de caracteres:

```
'A' + 1 devuelve 66, es decir, 'B'.
65 + 1 devuelve 66.
'C' – 2 devuelve 65, es decir, 'A'
Cuidado!!! 'A' + '1' devuelve 114 (65+49).
```

Tipo de dato real

- Rango

Tipo	Memoria	Precisión
`float`	4 *bytes*	7 dígitos
`double`	8 *bytes*	15 dígitos
`long double`	8 *bytes*	19 dígitos

- Operadores. `+, -, *, /`

```
float r;
r = 5.0 * 7.0;     /* Asigna a r el valor  35.0      */
r = 5.0 / 7.0;     /* Asigna a r el valor 0.7142857 */
r = 5.0 / 7;       /* Asigna a r el valor 0.7142857 */
r = 5 / 7;         /* Asigna a r el valor 0         */
```

El tipo del resultado de una operación es el mayor de los tipos de los operandos

Tipo de dato real. Operadores no predifinidos

- Un compilador de C no los reconoce a no ser que se le especifique dónde encontrarlos. Suelen estar en la biblioteca `math`.

  abs(), pow(), cos(), sin(), sqrt(), tan(), log(), exp(), log10(),...

Ojo!! Para incluir la librería matemática hay que compilar con la opción –lm: gcc –lm casting.c

```
#include<math.h>
main()
{
  float ladoa, ladob, ladoc;

  ladoa = 3;    /* Conversión implícita, convierte int en float   */
  ladob = 4.0;  /* Conversión implícita, convierte double en float */
  ladoc = sqrt(ladoa*ladoa + ladob*ladob);
}
```

Tipo de dato real. Operadores no predifinidos

Tipo que devuelve

Tipo(s) que se le pasa

`int abs(int)`	Regresa el valor absoluto
`double pow(double x,double y)`	Calcula **x** ^{y}. Puede producirse un error de dominio si **x** es negativo e **y** no es un valor entero. También se produce un error de dominio si el resultado no se puede representar cuando **x** es cero e **y** es menor o igual que cero. Un error de recorrido puede producirse.
`double cos(double x)`	Calcula el coseno de **x** (medido en radianes).
`double sin(double x)`	Calcula el seno de **x** (medido en radianes).
`double sqrt(double x)`	Calcula la raíz cuadrada del valor no negativo de **x**. Puede producirse un error de dominio si **x** es negativo.
`double tan(double x)`	Calcula la tangente de **x** (medido en radianes).
`double log(double x)`	Calcula el logaritmo natural (o neperiano). Se produce un error de dominio si el argumento es negativo y un error de recorrido si el argumento es cero.
`double exp(double x)`	Calcula la función exponencial de **x**.
`double log10(double x)`	Calcula el logaritmo en base 10 de **x**. Puede producirse un error de dominio si el argumento es negativo y un error de recorrido si el argumento es cero.
`double ceil(double x)`	La función *ceil* retorna el resultado de la función "techo" de **x**.
`double floor(double x)`	La función *floor* retorna el resultado de la función "suelo" de **x**.

Tipo lógico

- Es un tipo de dato muy común en los lenguajes de programación que se utiliza para representar los valores verdadero y falso que suelen estar asociados a una condición (p.e. ser un número primo)
- El tipo de dato lógico no existe en lenguaje C, pero se puede simular:
  - En C, es verdadero todo aquello distinto de 0. Sólo el 0 es falso.
    - 105: verdadero
    - -1: verdadero
    - 0: falso

Tipo lógico. Operadores lógicos

- Son los operadores clásicos de la lógica NO, Y, O que en C son los operadores ! && || respectivamente

	&& (Y)	|| (O)
≠ 0 ≠ 0	verdadero	verdadero
≠ 0 0	falso	verdadero
0 ≠ 0	falso	verdadero
0 0	falso	falso

	! (NO)
≠ 0	falso
0	verdadero

```
main () {
   int a, b, c, d;
   a = 1;
   b = 2;
   c = 0;
   d = (a && b); //Asigna verdadero a d
   d = (b && c); // Asigna falso a d
   d = (c || c); // Asigna falso a d
   d = !a;       // Asigna falso a d
   d = (c && (a && b)) || (b && c);
}
```

Tipo lógico. Operadores relacionales

- Son los operadores de comparación
  - == (igual)
  - != (distinto)
  - <, >
  - <= (menor o igual)
  - >= (mayor o igual)
- El resultado de evaluarlos es un valor lógico (=0, ≠ 0)
- Pueden aplicarse a operandos tanto enteros como reales, y tienen el mismo sentido que en Matemáticas
  - La expresión `(4<5)` devuelve valor `verdadero` (≠)
  - La expresión `(4>5)` devuelve el valor `falso` (=0)

Entrada y salida de datos

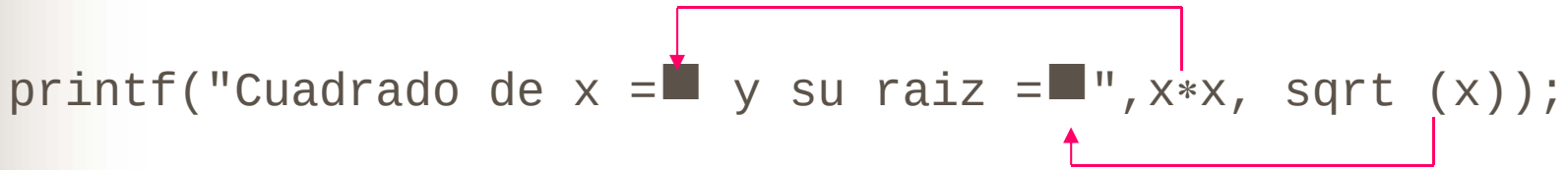
- **Salida de datos**. Operaciones que permiten, por lo general, imprimir información útil en la pantalla del ordenador
  - printf
  - puts
- **Entrada de datos**. Operaciones que permiten, por lo general, introducir datos desde el teclado del ordenador y almacenarlos en variables de un programa
  - scanf
  - gets
  - getchar

Salida de datos. printf

- Es la operación de salida de datos más frecuente y se encuentra en la biblioteca `<stdio.h>`. Su sintaxis más elemental es la siguiente:

```
printf(<formato>, <expresión1>, <expresión2>,...);
```

  - `<formato>` es una plantilla, separada por comillas dobles, que contiene:
    - Letras, dígitos, etc. que son impresos tal cual en la pantalla
    - Una serie de *huecos* cada uno de los cuales se rellena con el valor de cada una de las expresiones que aparecen, de izquierda a derecha, al final de la operación



Salida de datos. printf

- Estos huecos, que se llaman *especificadores de formato*, se marcan con el carácter **%** y van acompañados de una letra que indica el tipo de la expresión correspondiente

Tipo de dato	Especificador de formato
int	%i %d
float	%f %e(notación científica)
char	%c %d(código ASCII)
cadena de caracteres	%s
long	%ld %u
double	%lf
puntero	%lu %p

```
printf("Cuadrado de X = %f y su raiz = %f",x*x, sqrt(x));
```

Salida de datos. printf

- Entre el carácter % y el *especificador de formato* puede haber, por el siguiente orden, uno o varios de los elementos que a continuación se indican:
  - Un signo (-), que indica *alineamiento* por la izda (el defecto es por la dcha)
  - Un número entero positivo, que indica la *anchura* mínima del campo en caracteres
  - Un punto (.), que separa la anchura de la *precisión*
  - Un número entero positivo, la *precisión*, que es:
    - En un *string*: El nº máximo de caracteres a imprimir
    - En un *int* o *long*: El número de cifras mínimas. Si el numero tiene menos de esas cifras rellena con ceros a la izquierda
    - En un *float* o *double*: El nº de decimales
  - Un *cualificador*: una (h) para *short* o una (l) para *long* y *double*

Salida de datos.printf

```
#include <stdio.h>
void main(void)
{  int x=45;
   double y=23.354;
   char z[]="Esto es vida";
   //utilizamos <> para ver claramente la anchura del campo de caracteres

   printf("Voy a escribir <45> utilizando el formato %%d: <%d>\n", x);
   printf("Voy a escribir <45> utilizando el formato %%1d: <%1d>\n", x);
   printf("Voy a escribir <45> utilizando el formato %%10d: <%10d>\n\n", x);
   printf("Voy a escribir <45> utilizando el formato %%-10d: <%-10d>\n\n", x);
   printf("Voy a escribir <45> utilizando el formato %%-10.6d: <%-10.6d>\n\n", x);

   printf("Voy a escribir <23.354> utilizando el formato %%f:<%f>\n", y);
   printf("Voy a escribir <23.354> utilizando el formato %%.3f: <%.3f>\n", y);
   printf("Voy a escribir <23.354> utilizando el formato %%5.1f: <%5.1f>\n", y);
   printf("Voy a escribir <23.354> utilizando el formato %%-10.3f:<%-10.3f>\n", y);
   printf("Voy a escribir <23.354> utilizando el formato %%.0f: <%.0f>\n\n", y);

   printf("Voy a escribir <Esto es vida> utilizando el formato %%s: <%s>\n", z);
   printf("Voy a escribir <Esto es vida> utilizando el formato %%.7s:<%.7s>\n", z);
   printf("Voy a escribir <Esto es vida> utilizando el formato %%-15.10s: <%-15.10s>\n", z);
   printf("Voy a escribir <Esto es vida> utilizando el formato %%15s: <%15s>\n", z);
}
```

Entrada de datos.scanf

- Es la operación de entrada de datos más común y se encuentra en la biblioteca `stdio`

```
scanf("<especif. formato>", &<variable>);
```

  - `scanf("%i",&salario),` espera a que el usuario introduzca un valor entero `(int)`en el teclado y, cuando se pulsa la tecla `Intro`, lo almacena en la variable `salario`
  - `scanf("%f",&retencion),` espera a que el usuario introduzca un valor real en el teclado y lo almacena en `retencion`
  - `scanf("%c",&opcion),` espera a que el usuario introduzca un carácter en el teclado y lo almacena en `opcion`
  - `scanf("%s", nombre),` espera a que el usuario introduzca una cadena de caracteres por el teclado y lo almacena en `nombre`

Ojo!! con %s no se pone &

Entrada de datos. scanf

- **OJO!!!** Es muy común olvidar el *ampersand*
  - `scanf("%f",valor);  //ERROR!!!`
- Excepción: **%s** es el único caso en que `scanf` no lleva *ampersand*
  - `scanf("%s",&ciudad);  //ERROR!!!`
- No es conveniente leer varios datos en un mismo `scanf`, es mejor utilizar una operación de entrada para cada uno

```
printf("Introducir base y altura: ");
scanf("%f  %f ", &x, &y);
```



```
printf("\nIntroducir base: ");
scanf("%f", &x);
printf("\nIntroducir altura: ");
scanf("%f", &y);
```

La lectura debe ir precedida de un printf que indique qué dato se quiere leer!

Estructura condicional simple

Algoritmica

```
Si condición
Entonces
    acciones
Finsi
```

```
if (<condición>)

   {<bloque if>};
```

Lenguaje C

Aquí NO hay ";"

Aquí SI hay ";" (opcional)

- *<condición>* es una expresión lógica
- *<bloque if>*
  - Si hay varias sentencias, es necesario encerrarlas entre llaves.
  - Si sólo hay una sentencia, no son necesarias las llaves.

Consejo: Aunque el <bloque if> conste de una única sentencia es recomendable poner las llaves

Estructura condicional simple

Consejo: Es MUY recomendable tabular el texto. Utilizar espacios, no tabuladores.

```
#include <stdio.h>
#include <math.h>
main()
{
    float a, b, c, x1, x2;
    printf("Introduce los coeficientes\n");
    scanf("%f",&a);
    scanf("%f",&b);
    scanf("%f",&c);
    if (a!=0)
    {
        x1 = (-b + sqrt( b*b-4*a*c ) ) / (2*a);
        x2 = (-b - sqrt( b*b-4*a*c ) ) / (2*a);
        printf("Las raices son %f y %f\n",x1,x2);
    }
}
```

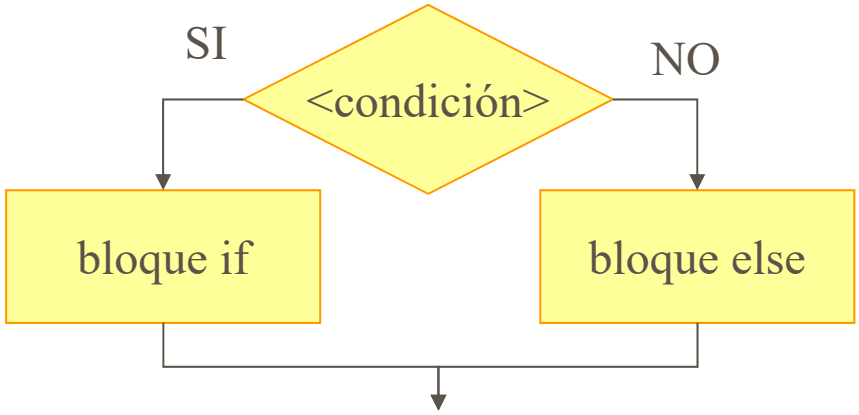
Estructura condicional doble

- *Problema*: Se evalúan dos condiciones equivalentes

```
Si condición
Entonces
    acciones
Sino
    acciones
Finsi
```

Algoritmica

SOLUCIÓN: CONDICIONAL DOBLE



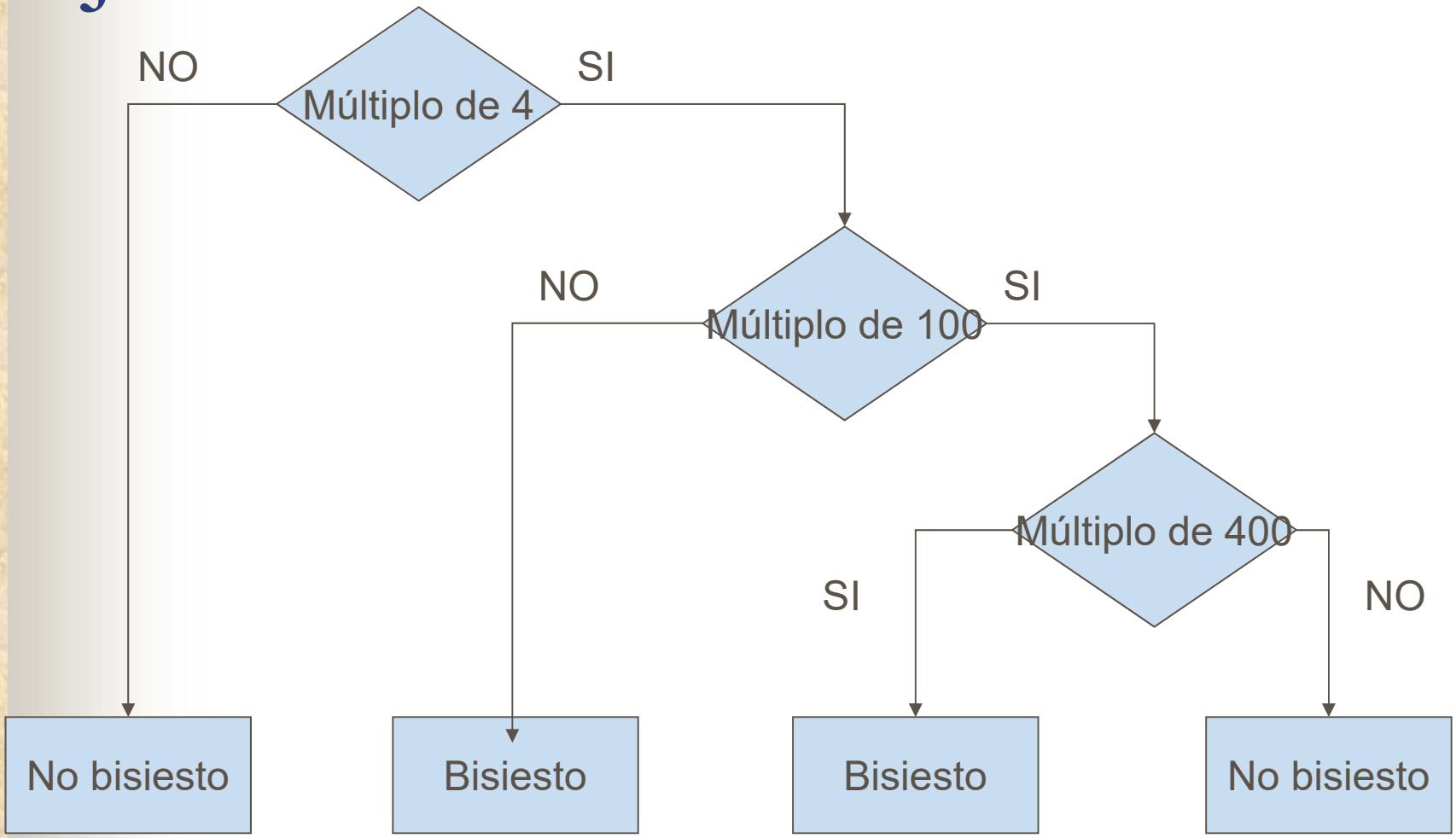

```
if (<condición>)
   {<bloque if>};
else
   {<bloque else>};
```

Lenguaje C

Estructura condicional doble

```
float a, b, c, x1, x2, discriminante;
//Leer a, b, c
if (a!=0)
{    discriminante = (b*b)-(4*a*c);
     if (discriminante < 0)
       printf("\nSoluciones Imaginarias");
     else
       if (discriminante==0)
       {  x1=-b/(2*a);
          printf("\nUna unica raiz: %f", x1);
       }
       else
       {   x1 = (-b + sqrt( b*b-4*a*c ) ) / (2*a);
           x2 = (-b - sqrt( b*b-4*a*c ) ) / (2*a);
           printf("Las raices son %f y %f\n",x1,x2);
       }
}
else
{    x1 = -c/b;
     printf("\nEcuacion de una recta");
     printf("\nUnica raiz es %f\n",x1);
}
```

Ejercicio 4


NO
Múltiplo de 4
SI
NO
Múltiplo de 100
SI
Múltiplo de 400
SI
NO
No bisiesto
Bisiesto
Bisiesto
No bisiesto

Estructura condicional múltiple

Algoritmica

```
Segun variableNumerica Hacer
    valor_1:
          Grupo de Acciones
     valor_2:
          Grupo de Acciones
     ............
          Grupo de Acciones
     De Otro Modo
          Grupo de Acciones
FinSegun
```

Lenguaje C

```
switch(<expresión>)
{
   case<constante1>:
        <sentencias1>
        break;
   case<constante2>:
        <sentencias2>
        break;
  [default:
        <sentencias>]
}
```

- *<expresión>* es un expresión entera.
- *<constante1>* ó *<constante2>* es un literal **tipo entero** o **tipo carácter**
- `switch` sólo comprueba la igualdad
- No debe haber 2 constantes `case` en el mismo `switch` que tengan el mismo valor
- En las estructuras condicionales múltiples también se permite el anidamiento
- Se usa frecuentemente:
  - En la construcción de menús
  - Si se desean realizar las mismas operaciones para un número determinado de constantes

Estructura condicional múltiple

- Si el valor de *expresion* es igual al se una etiqueta case, la ejecución comienza con la primera sentencia de ese case y continua hasta que:
  - Se encuentra el final de la sentencia *switch*
  - O hasta encontrar la sentencia *break*
- Es habitual terminar la ejecución después de ejecutarse un único case, para ello, situar la sentencia *break* como última sentencia del bloque

```
switch(<expresión>)
{
    case<constante1>:
        <sentencias1>
        break;
    case<constante2>:
        <sentencias2>
        break;
   [default:
        <sentencias>]
}
```

Ejercicios 5 y 6

Construcción de menús

```
#include <stdio.h>
main()
{
   int operacion, operando1, operando2, resultado;
   printf("\nSeleccione operacion: 1(suma) 2(resta) 3(multiplicacion): ");
   scanf("%d", &operacion);
   printf("\nOperando1: ");
   scanf("%d", &operando1);
   printf("\nOperando2: ");
   scanf("%d", &operando2);
   switch(operacion)
   {  case 1:  printf("\n%d+%d=%d", operando1, operando2, operando1+operando2);
               break;
      case 2: resultado=operando1-operando2;
               printf("\n%d-%d=%d", operando1, operando2, resultado);
               break;
      case 3: printf("\n%d*%d=%d", operando1, operando2, operando1*operando2);
               break;
      default: printf("\nSe ha equivocado");
   }
}
```

Bucles controlados por contador: Bucle `for`

```
for (<asig inicial>; <condicion>; <incremento>)
        {<bloque for>}
```

```
Para contador<-valor_inicial hasta n [Con Paso paso] Hacer
    cuerpo bucle
Finpara
```

- *<asig. Inicial>* asignación del valor inicial a la variable controladora del ciclo

  <incremento> determina el modo en que la variable controladora cambia su valor para la siguiente iteración
  - **Incrementos positivos**: `VC = VC + incremento`
  - **Incrementos negativos**: `VC = VC - incremento`
- *<condicion>* de permanencia dentro del bucle
- Cuando termina un bucle `for`, la variable contadora se queda con el primer valor que hace que la condición del bucle sea falsa

mediaFor.c

Bucles controlados por contador: Bucle `for`

- La sentencia `for` permite la construcción de una forma compacta de los ciclos controlados por contador, aumentando la legibilidad del código

```
#include <stdio.h>
main(){
   int i, valor, suma;
   float media;
   suma = 0;
   for (i=1 ; i <= 5 ; i++)
   {
     printf("Introduce el %i-esimo numero: ",i);
     scanf("%i",&valor);
     suma = suma + valor;
   }
   media =suma / 5.0;
   printf("La media es %f\n",media);
}
```

Ojo! aquí NO hay “;” y os equivocáis mucho en esto

Bucles controlados por condición

```
while (<condición>)
{
    <cuerpo bucle>
}
```

```
Mientras condición
    cuerpo bucle
Fin mientras
```

← *Test previo*

```
do
{
   <cuerpo bucle>
}while (<condicion>);
```

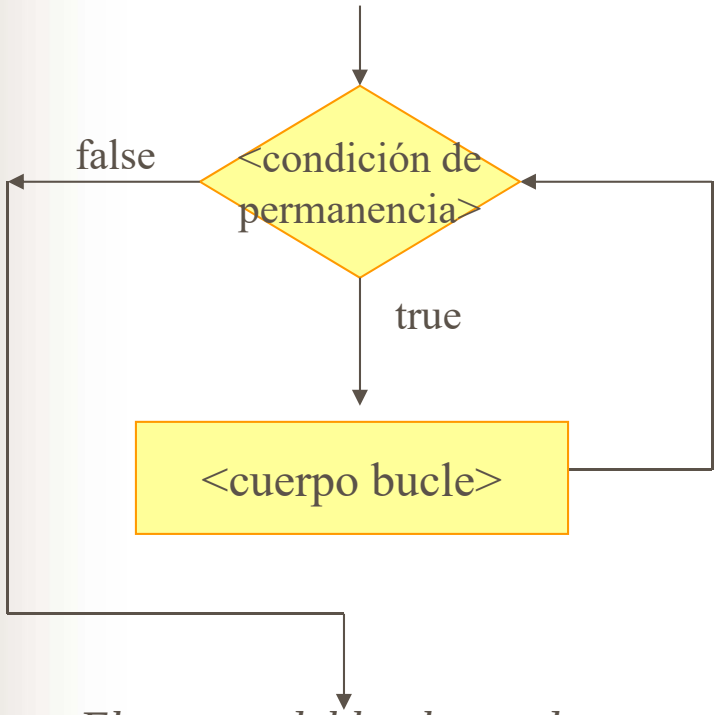
```
repetir
     cuerpo bucle
mientras condición
```

← *Test posterior*

- *Funcionamiento*: En ambos, se va ejecutando el cuerpo del bucle **mientras** la condición sea verdad
  - En un bucle `while`, primero se pregunta y luego (en su caso) se ejecuta
  - En un bucle `do while` , primero se ejecuta y luego se pregunta

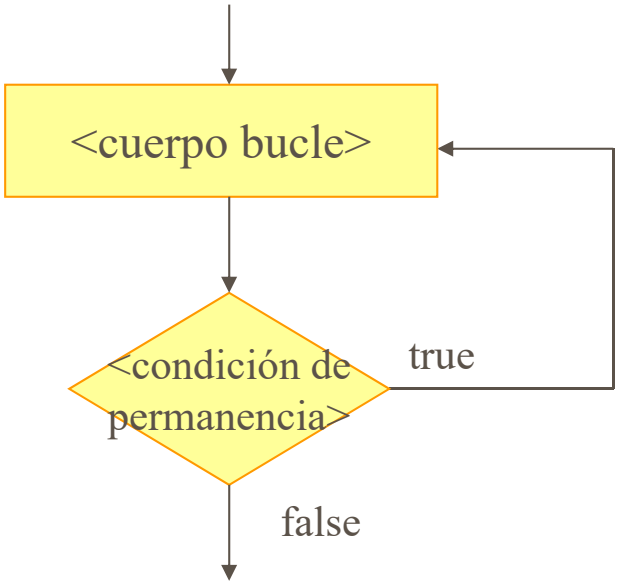
Bucles controlados por condición

Test previo



El cuerpo del bucle puede no llegar a ejecutarse

Test posterior



Siempre se ejecuta al menos una vez el cuero del bucle

Bucles controlados por condición

```
for (i=1, suma=0 ; i <n ; i++)
{
    suma = suma+i;
}
```

```
i =1;
suma =0;
while( i <n )
{
    suma = suma+i;
     i++;
}
```

```
i =1;
suma =0;
do
{
    suma = suma+i;
    i++;
} while( i <n );
```

media.c

Funciones en C

Tipo devuelto

```
<tipo devuelto> <nombre> (<parám. formales>)
{
    [<ctes Locales>]
    [<variables Locales>]
    [<Sentencia>]
return <expresión>;
}
```

```
float media2(float x1, float x2)
{
    return (x1+x2)/2;
}
long factorial(long n)
{   long i, aux=1;
    aux = 1
    for (i=2; i<=n; i++)
        {aux = aux * i;}
    return aux;
}
```

return *termina la ejecución y devuelve el valor*

Funciones en C

- **Tipo de la función**
  - Si no se indica el tipo devuelto, por defecto C asume que es de tipo *int*
  - Si la función no devuelve nada será de tipo *void* (procedimiento)
    - Típico de las funciones para imprimir datos
    - Ej. `void imprimirDato(int n);`
  - El tipo devuelto puede ser cualquier tipo salvo una función o un *array*
    - Se puede devolver un `struct`

Funciones en C

- **Resultado de una función**
  - El cuerpo de la función debe contener ***return expresion;***
  - *expresión* ha de ser del mismo tipo que devuelve la función
  - Tan pronto como el programa encuentra un *return expresión*, la función termina y devuelve *expresión* al módulo llamador
  - *return* solo devuelve un valor
  - Si la función es de tipo *void* podemos omitir el *return* o terminar con ***return;***

Funciones en C

- **La llamada a una función**
  - Las funciones para poder ser ejecutadas, han de ser invocadas o llamadas
  - La llamada debe estar asignada a una variable o formar parte de otra expresión asignada a una variable, en caso contrario el valor devuelto se pierde
    - resultado=factorial(3);
    - valor = maximo(a,b)+7;
    - printf("El maximo es %d", maximo(a,b));
  - Si la función es de tipo *void*, su llamada no se asigna a una variable
    - ImprimeNumero(valor);

media.c

Funciones en C

- Ejemplos de llamada:

```
main()
{
   int x;
   float suma;

   scanf("%i", &x);
   printf("Factorial de %i = %i \n",x,factorial(x));

   suma = media2(factorial(x), factorial (x+1));

   printf("La media entre %i! y %i! es %f \n",x,x+1,suma);
   .....
}
```

llamadas a función

Hay que asignar el valor devuelto por la función a una variable. En otro caso este valor se pierde

Ámbito de una variable

- **Reglas por las que se rigen las variables locales:**
  - Los datos de un módulo (parámetros y variables locales) son invisibles para el resto de módulos.
    - No puede usarse en otros módulos ni en el programa principal
  - Los nombres de las variables locales NO son únicos
    - Dos, tres o más funciones pueden definir variables con el mismo nombre. Cada variable es distinta y es local a la función en que está declarada
  - Las variables locales de las funciones no existen en memoria hasta que se ejecuta la función y solo existen mientras la función está activa
    - Permite que varias funciones compartan la misma memoria para variables locales (pero no a la vez) → ahorro de memoria

Ámbito de una variable

```
#include <stdio.h>
void foo(int a);
int main()
{  int a=6;
   printf("\n1.a: %d", a);
   foo(a);
   printf("\n4.a: %d", a);
   return 0;
}
void foo(int a)
{
   printf("\n2.a: %d", a);
   a = 7;
   printf("\n3.a: %d", a);
   return ;
}
```

cuadrados.c

Ámbito de una variable


```
#include <stdio.h>
int cuadrado(int x);
int restaCuadrados(int n1, int n2);
int main(){
   int n2, n1;
   printf("%i \n", restaCuadrados(7,8));
   n1 = 4;
   n2 = 5;
   aux = 0; //ERROR
   x = 13;  //ERROR
   n1=restaCuadrados(n1);//ERROR
   return 0;
}
int cuadrado(int x){
   int aux ;
   aux = x*x;
   return aux;
}
int restaCuadrados(int n1, int n2){
   int aux;
   aux=cuadrado(n1)-cuadrado(n2);
   return aux;
}
```

Módulos y Prototipos

- En lenguaje C, cada vez que se utiliza un módulo (ya sea de una biblioteca como definida por el programador) es necesario conocer previamente su cabecera o prototipo
  - Esto permite al compilador realizar la comprobación de tipos y número de argumentos
- **Declaración** de una función o prototipo:

```
<tipo devuelto> <nombreFuncion>(<parametros formales>);
```

El prototipo de una función termina en ;

- **Definición** de una función:

```
<tipo devuelto> <nombreFuncion>(<parametros formales>)
{
  Cuerpo de la función
}
```

factorial1.c

Módulos y Prototipos

Primera forma:

- Un fichero.
- Prototipos + main + definición de las funciones.
- Compilación:
  - *gcc factoria12.c –o factorial.exe*

#include
prototipos
main
definición de las funciones

cabecera.h

factorial3.c

Módulos y Prototipos

Segunda forma:

- Dos ficheros.
  - Fichero1: Prototipos
  - Fichero 2: main + definición de las funciones.
- Compilación:
  - *gcc factoria13.c –o factorial.exe*

#include
prototipos

#include
main
definición de las funciones

Nuestros .h los incluímos con “ ” para indicar que se encuentran en el directorio donde se realizan la compilación. < > indica que el fichero se encuentra en alguno de los directorios de ficheros de cabecera del sistema o entre los especificados con la opción -I del compilador

- **Tercera forma**: prototipos en un fichero y *main* y módulos en otro

```
/* Declaracion de prototipos */
  int Factorial (int n);
```

```
#include <stdio.h>
#include <math.h>
#include "cabecera.h"

int main()  {
   int n, x;

   x = 3;
   /* Aqui se usa la funcion */
   n = Factorial(x);
   printf("%f",sqrt(n));
   return 0;
}
```

```
/* Se define la funcion mas adelante */
int Factorial (int n)  {
    int i, n;


    aux = 1;
    for (i=2; i<=n; i++)
        aux = aux * i;
    return aux;
}
```

Vectores

```
#include <stdio.h>
#define TAM 100

void leerVector(float V[], int nEle);
void escribirVector(float V[], int nEle);

int main()
{
  float V[TAM];
  int nEleV=7;

  //Leer nEleV

  leerVector(V, nEleV);
  escribirVector(V, nEleV);

   return 0;
}
```

Tamaño máximo

Declaración

Lllamada

Parámetro

```
void leerVector(float V[], int nEle)
{
   int i;
   for(i=0; i<nEle; i++)
   {
     printf("\nV[%d]: ", i);
     scanf("%f", &V[i]);
   }
}

void escribirVector(float V[], int nEle)
{
   int i;
   for(i=0; i<nEle; i++)
   {
     printf("\nV[%d]: %f", i, V[i] );
   }

}
```

Recorrer

Acceso

Matrices

```
#include <stdio.h>
#define TAM 100

void leerMatriz(float M[][TAM], int nFil, int nCol);
void escribirMatriz(float M[][TAM], int nFil, int
nCol);

int main()
{
  float M[TAM][TAM];
  int nFil=3, nCol=5;

  //Leer nFil, nCol

  leerMatriz(M, nFil, nCol);
  escribirMatriz(M, nFil, nCol);

   return 0;
}
```

Tamaño máximo

Declaración

Lllamada

Parámetro
primer corchete en blanco

```
void leerMatriz(float M[][TAM], int nFil, int nCol)
{
   int i,j;
   for(i=0; i<nFil; i++){
        for(j=0; j<nCol; j++){
           printf("\nM[%d][%d]: ", i, j);
           scanf("%f", &M[i][j]);
        }
   }
}

void escribirMatriz(float M[][TAM], int nFil, int
nCol)
{
   int i,j;
   for(i=0; i<nFil; i++){
        for(j=0; j<nCol; j++){
           printf("\nM[%d][%d]: %f", i, j, M[i][j]);
        }
   }

}
```

Recorrer

Acceso

Tipo de dato cadena de caracteres

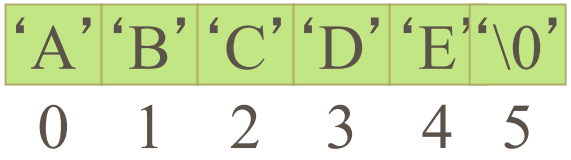
- Conjunto de caracteres, tales como “ABCDEFG”
- Las cadenas se señalan incluyendo un carácter ‘\0’ al final de
- Siempre hay que definir las cadenas con un espacio más del previsto como máxima longitud para el carácter fin de cadena
- Declaración:

  char <nombre> [<longitud>];

  ```
  char cadena [6];
  ```

- Declaración con inicialización:

  ```
  char cadena[6] = "ABCDE";
  ```



  - Con las comillas dobles “” el compilador inserta automáticamente un carácter ‘\0’ al final de la cadena

Entrada de Cadenas de Caracteres

- La forma más común podría ser utilizando `scanf`

```
char cadena[80];
printf("\nIntroduzca una cadena: ");
scanf("%s", cadena);
printf("La cadena es %s", cadena);
```

NO necesita el carácter &,

- Sin embargo, `scanf` deja de leer cuando aparece un carácter en blanco, con lo que el intento de leer la cadena "`hola amigos`" desde el teclado haría que sólo se leyese la cadena "`hola`"

Entrada de Cadenas de Caracteres

- Para poder leer cadenas de caracteres que contengan espacios, se puede usar la función `gets`, especializada en leer cadenas

```
char cadena[80];
printf("\nIntroduzca una cadena: ");
gets(cadena);
printf("La cadena es %s", cadena);
```

Manejo de Cadenas de Caracteres

- Las funciones más útiles de C para manejar cadenas se encuentran declaradas en el archivo de cabecera **`string.h`**. Algunas de estas funciones son las siguientes
- **`int strlen(char *).`** Devuelve la longitud de la cadena de caracteres (sin incluir el carácter nulo)

```
#include <stdio.h>
#include <string.h>
void main()
{
   char cad[15]="Hola";
   printf("%d", strlen(cad)); //Imprime 4
}
```

Manejo de Cadenas de Caracteres

- **`int strcmp(char *cad1, char *cad2)`**. Compara ambas cadenas y devuelve:
  - `0` si las dos son idénticas
  - `<0` si la primera cadena precede *alfabéticamente* a la segunda
  - `>0` si la segunda cadena precede *alfabéticamente* a la primera

```
#include <stdio.h>
#include <string.h>
void main()
{  char cad[15]="Hola", cad2[15]="Mola";
   if(strcmp(cad, cad2)==0)
       printf("Las cadenas son iguales");
}
```

Manejo de Cadenas de Caracteres

- **`char *strcpy(char *dest, char *orig)`**
  - Copia `orig` en `dest` Devuelve `dest`. Se suele ignorar el valor devuelto y se usa como si fuera una función `void`

```
#include <stdio.h>
#include <string.h>
void main()
{
   char cad[15]="Hola", cad2[15];
   cad2=cad; //Error
   strcpy(cad2, cad); //Bien
}
```

pasoParametros.c

Estructuras

```
#include <stdio.h>
struct punto
{
  float x;
  float y;
};
void escribirPunto(struct punto p);
struct punto leerPunto();
int compara(struct punto p1, struct punto p2);
struct punto suma(struct punto p1, struct punto p2);
main()
{
  struct punto p1, p2;
  p1 = leerPunto();
  p2 = leerPunto();
  if (compara(p1, p2))
    printf("\nLos puntos son iguales");
  else
    printf("\nLos puntos son diferentes");
  p1=suma(p1, p2);
  printf("\nLa suma vale:);
  escribirPunto(p1);
}
```

pasoParametros.c

Paso de estructuras a funciones

```
struct punto leerPunto()
{
    struct punto p;
    printf("Introducir coordenada x: ");
    scanf("%f", &p.x);
    printf("Introducir coordenada y: ");
    scanf("%f", &p.y);
    return p;
}
void escribirPunto(struct punto p)
{
   printf("\np.x: %f\t p.y:%f", p.x, p.y);
}
int compara(struct punto p1, struct punto p2)
{
  if((p1.x==p2.x) && (p1.y==p2.y))
    return 1;
  else
    return 0;
}
struct punto suma(struct punto p1, struct punto p2)
{
   struct punto aux;

   aux.x=p1.x+p2.x;
   aux.y=p1.y+p2.y;

   return(aux);
}
```

En C podemos devolver una estructura

vectorStruct.c

Arrays de estructuras

```
#include <stdio.h>
#define MAX_ELE 100
struct punto
{
   float x;
   float y;
};
void leeVector(struct punto vector[], int utiles);
void escribeVector(struct punto vector[],
     int utiles);

void main()
{
  int i;
  struct punto vectorPuntos[MAX_ELE];
  struct punto vector2[2]={{-2,-2}, {-3, -3}};

  leeVector(vectorPuntos, MAX_ELE);
  escribeVector(vectorPuntos, MAX_ELE);
  escribeVector(vector2, 2);

}
```

vectorStruct.c

Arrays de estructuras

```
void leeVector(struct punto vector[], int utiles)
{
  int i;
  for(i=0; i<utiles; i++)
  { printf("\nvector[%d].x: ", i);
    scanf("%f", &(vector[i].x));
    printf("\nvector[%d].y: ", i);
    scanf("%f", &(vector[i].y));
  }
}
void escribeVector(struct punto vector[],
     int utiles)
{ int i;
  for(i=0; i<utiles; i++)
  {
     printf("\nvector[%d].x: %f vector[%d].y: %f", i, vector[i].x ,i,
              vector[i].y);
  }
}
```