

C

Una breve introducción

Bibliografía:

- *El lenguaje de programación C*. B. W. Kernighan y D.M. Ritchie. Prentice-Hall.
- *Numerical Recipes in C. The Art of Scientific Computing*. W. H. Press, S.A. Teukolsky, W.T. Vetterling y B. P. Flannery. Cambridge University Press.

Estructura del curso

- Introducción
- Programas y su estructura
- Variables
- Operaciones aritméticas
- Input y output
- Control de flujo, iteracciones y bucles
- Vectores y punteros
- Funciones

Introducción

- C es un lenguaje de programación de medio nivel creado por Dennis M. Ritchie entre 1969 y 1972 en los laboratorios Bell.
- Su objetivo era la programación de sistemas operativos, fue usado por primera vez para crear Unix.
- Es muy utilizados en aplicaciones de ingeniería (robótica, cibernética, Big Data, aeronáutica) y ciencia (física, matemáticas, estadística, química).
- C++ fue creado en los 80's con el objetivo de extender el lenguaje C a programación orientada a objetos. Permite redefinirlos operadores y crear nuevos tipos.
- Hay multitud de librerías científicas en C/C++ (Numerical Recipes, ARPACK, ...)

Programas

- Un programa es una **lista de instrucciones** que permiten a un ordenador resolver un problema.
- **El problema a resolver se rompe en partes** que definen una estructura: el problema complejo original al principio del programa y las partes sencillas y más pequeñas al final.
- **Divide et impera: divide y vencerás (Julio César)** ¡Funciones!
- Los pasos para realizar un programa son:
 - Examinar el problema y dividirlo en varias partes.
 - Examinar cada parte y dividirla en partes más pequeñas.
 - Establecer un **plan lógico** que una todas las partes.
 - Escribir el programa **utilizando funciones**.
 - Comprobar el programa: esto es un **arte**.

Estructura de un programa

- Un programa se divide en encabezado, función **main** y otras funciones.

```
# include <stdio.h>
# define PI 3.1415926535
int main()
{
    printf("Hola mundo\n");
}
```

→ Biblioteca estándar

→ Definición de constantes

Función principal (main)

- Cada línea debe terminar con ;
- Los comentarios no son traducidos por el compilador y empiezan por // o están entre /* y */.
- ¡¡Hay que usar muchos comentarios!!
- Se pueden definir constantes antes de la función main mediante la orden **#define**. Esas definiciones se aplican a cualquier función.

Estructura de un programa

```
21 main(){
22     double x,y,z;
23     int i,j,k,l,m,n;
24     int cont;
25     int semilla;
26     fcomplex c[SIZE]; //la dimensión es binomial(D,3)
27     fcomplex slater[D][D][D][SIZE]; //la dimensión es binomial(D,3)
28     double rtraza;
29     double *autovalores;
30     double **autovectores;
31
32     FILE *f1,*f2,*f3,*f4,*f5;
33
34     f2=fopen("semilla.dat","r");
35     fscanf(f2,"%i",&semilla);
36     fclose(f2);
37
38     autovalores=dvector(1,SIZE);
39     autovectores=dmatrix(1,SIZE,1,SIZE);
40
41     f1=fopen("autovalores5.dat","w");
42
43     dranini_(&semilla);
44
45     iniciaslater(slater);
46
47     for (cont=0;cont<REP;cont++)
48     {
49         inicia(c);
50         eigen(c,slater,autovalores, autovectores);
51
52         for (i=1;i<=2.*D;i++) fprintf(f1,"%lf\n",autovalores[i]);
53         fprintf(f1,"\n");
54     }
55
56     free_dvector(autovalores,1,D);
57     free_dmatrix(autovectores,1,D,1,D);
58
59     fclose(f1);
60 }
61
62 void iniciaslater(fcomplex slater[D][D][D][SIZE])
63 {
64     int i,j,k;
65     int count;
66
67     for (count=0;count<SIZE;count++)
68         for (i=0;i<D;i++)
69             for (j=0;j<D;j++)
70                 for (k=0;k<D;k++) slater[i][j][k][count]=Complex(0.,0.);
71 }
```

Programador con poca experiencia
(pocos comentarios)

```
32 int main (int argc, char **argv)
33 {
34     int i,j,k,l;
35     double x,y,z;
36
37     // Hamiltonian parameters
38     double omega; // frequency for the uniform chain
39     double v; // coupling strength
40
41     // Bath parameters
42     double gamma1,gamma2; // coupling to the baths
43     double n1,n2; // baths occupation numbers
44
45     // System matrices
46     double **w; // Hamiltonian matrix (dimension N)
47     double **L,**M; // baths matrices (dimension N)
48     double **eq; // equation matrix (dimension N)
49     double *indep; // independent vector of the equation (dimension N)
50
51     // Output: populations vector
52     double *pop;
53
54     // Auxiliary variables for the equation solver
55     extern int dim;
56     int *indx; //auxiliary vector for the subroutine ludcmp
57     double d; //Auxiliary value for ludcmp
58
59     FILE *f1;
60
61     // Populations vector
62     pop=dvector(1,N);
63
64     // Equation matrix
65     eq=dmatrix(1,D,1,D);
66
67     // Equation independent vector
68     indep=dvector(1,D);
69
70     // Auxiliary vector for ludcmp
71     indx=ivector(1,D);
72
73     // Uniform chain parameters
74     omega=1.;
75     v=1;
76
77     // Baths parameters
78     gamma1=.1;
79     gamma2=.1;
80     n1=2;
81     n2=1;
82 }
```

Programador con experiencia
(muchos comentarios)

Estructura típica de una simulación

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

Bibliotecas

```
# define PI 3.1415926535
# define CTE 1.325667
# define TIME_MAX 300
```

Definición de constantes

```
double evolution (int number_planets, double *masses);
double measure (int planet);
```

Definición de funciones

```
main()
{
```

```
    int i,j,k,l; // contadores
    int x,y,z; //variables reales
    int t; //time
```

```
    double masses[9]; //masas de los planetas
    double radios[9]; //radios planetas
```

```
    FILE *f1, *f2; //puntero para los ficheros de entrada y salida
```

```
    f1=fopen("entrada.txt","r"); //fichero de entrada
    f2=fopen("salida.txt","w"); //fichero de salida
```

Declaración de variables

Declaración de ficheros i/o

```
    i=0;
    while(fscanf(f1,"%lf",&x)!=EOF) masses[i]=x;
```

Lectura de datos de entrada

```
    x=0;
    for (t=0;t<TIME_MAX;t+=0.001)
    {
        evolution (7,masses);
        for (i=1;i<=9;i++) x+=measure(i);
    }
```

Cómputo

```
    fprintf(f2,"%lf\n",x);
```

Salida de datos

```
    fclose(f1)
    fclose(f2)
```

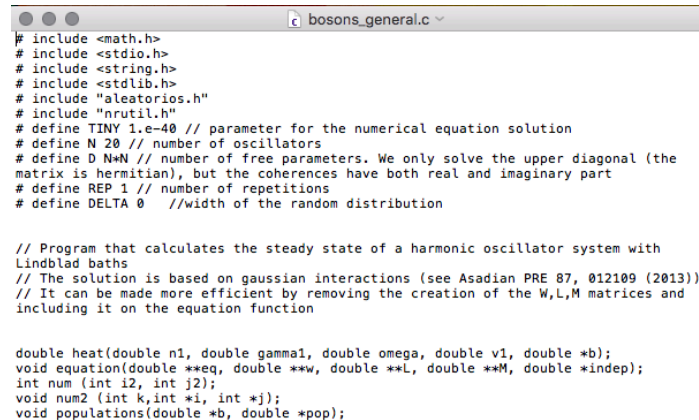
Cierre de ficheros

```
}
```

Función principal (main)

Edición, compilación y ejecución

- Para escribir un programa en C/C++ se puede usar cualquier editor de texto plano (vi, emacs, kate, notepad, TextEdit, geany, Eclipse)



```
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "aleatorios.h"
#include "nrutil.h"

#define TINY 1.e-40 // parameter for the numerical equation solution
#define N 20 // number of oscillators
#define D N*N // number of free parameters. We only solve the upper diagonal (the
matrix is hermitian), but the coherences have both real and imaginary part
#define REP 1 // number of repetitions
#define DELTA 0 //width of the random distribution

// Program that calculates the steady state of a harmonic oscillator system with
Lindblad baths
// The solution is based on gaussian interactions (see Asadian PRE 87, 012109 (2013))
// It can be made more efficient by removing the creation of the W,L,M matrices and
including it on the equation function

double heat(double n1, double gamma1, double omega, double v1, double *b);
void equation(double **eq, double **w, double **L, double **M, double *indep);
int num (int i2, int j2);
void num2 (int k, int *i, int *j);
void populations(double *b, double *pop);
```

- Después lo guardamos con **extensión .c**, por ejemplo [planetas.c](#).
- A continuación **compilamos el programa** con el programa gcc (g++ para C++): [gcc planetas.c -o planetas.exe -lm -O3](#)
- La opción -o nos deja elegir el nombre del ejecutable, y la opción -O3 es una optimización
- Para ejecutar escribimos en el terminal: [./planetas.es &](#)

Edición, compilación y ejecución

- Para hacer más fácil la compilación podemos programar un script.
- El siguiente script ([gcompila](#)) toma un fichero programa.c y devuelve el fichero compilado como .exe

```
if [ -f $1.c ]  
then  
    gcc -c $1.c  
    gcc -o $1.exe $1.o -lm -O3  
else  
    echo "No existe el archivo $1.c"  
fi
```

- Para usar el script hay que darle permiso de ejecución y usar el comando: `./gcompila programa` (debe existir [programa.c](#))

Variables

- ▣ Una variable es una magnitud que puede cambiar de valor.
- ▣ Se deben declarar al principio de cada función. Los nombres no deben contener caracteres 'exóticos' (\$ % / \) y deben reflejar el uso de la variable. Tampoco pueden empezar por un numero
- ▣ Hay distintos tipos de variable, el rango depende de la máquina
 - ▣ `int` -> entero
 - ▣ `long` -> entero largo
 - ▣ `float` -> racional
 - ▣ `double` -> racional, doble precisión
 - ▣ `char` -> carácter (entero de un solo byte)

Variables: enteras y reales

■ Enteros

- Los enteros son números sin decimales. Se suelen usar como contadores.
- ¡¡Operar con enteros puede llevar a errores de redondeo!!
- La forma de declararlos es:

`int variable1, variable2`

■ Reales

- Los números reales en realidad son racionales.
- Tienen una precisión limitada.
- La forma de declararlos es:

`float variable1, variable2`

¡Terminando en punto y

También tenemos un error de representación asociado a

Variables: doble precisión

- ▣ Las variables de tipo `float` tienen precisión simple, ya que se almacenan en una única posición de memoria. Para calculos precisos esta precisión puede ser demasiado pequeña.
- ▣ Para calculos precisos hay que usar la variable `double`. Esta utiliza dos posiciones de memoria.

Por defecto, siempre double

- ▣ La declaración es:

`double variable1, variable2`

Tambien hay "long doubles"

Variables: Números enteros

- La representación de los números (tanto enteros como reales) en la máquina se hace en sistema binario.

esto debería ser un ce

- El número binario $[b_{31}, b_{30}, b_{29}, \dots, b_0]$ de 4 bytes (o 32 bits, 1 byte = 8 bits), con $b_k=0,1$, representa el entero

$$[b_{31}, b_{30}, \dots, b_0]_2 = (-1)^{b_{31}} \sum_{k=0}^{30} b_k 2^k$$

- El entero más grande de 32 bits es **2147483647** ~ **2.147x10⁹**

Variables: Números en coma flotante

- La representación en coma flotante permite almacenar números racionales extremadamente grandes y pequeños de una manera muy eficiente y compacta.
- Representación en coma flotante de un número racional:

$$x = (-1)^{b_{31}} \left(1 + \sum_{k=1}^{23} b_{23-k} 2^{-k} \right) \times 2^{\text{exp}-\text{bias}}$$

- $s=0,1$ es el bit de **signo**, la **mantisa (b)** contiene los dígitos del número a representar, y **el exponente (exp)** indica donde colocar el punto decimal (una vez sustraído el bias).
- Para un número real de 4 bytes (o 32 bits) se reservan 23 bits para la mantisa y 8 para el exponente: eso nos da 7-8 cifras significativas

$$x = (-1)^{b_{31}} \left(1 + \sum_{k=1}^{23} b_{23-k} 2^{-k} \right) \times 2^{\text{exp}-\text{bias}}$$

Variables: Caracteres

- El tipo de variable `char` es un entero de un solo byte. Permite almacenar un caracter.
- Los caracteres se codifican siguiendo el [código ASCII](#).
- Se pueden asignar caracteres mediante el uso de `'`

`char a='a'`

- Para usar cadenas de caracteres se usan vectores de la categoría `char`.
- Para manipular cadenas de caracteres (copiar, concatenar, comparar) está la librería `<string.h>` de la biblioteca estándar.
- Se pueden introducir caracteres por el teclado con la orden `getchar()` (`<stdio.h>`).

Variables: Caracteres (Código ASCII)

Ctl	Dec	Hex	Char	Code	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
^@	0	00		NUL	32	20	sp	64	40	@	96	60	`	128	80	Ç	160	A0	Á	192	C0	Ļ	224	E0	α
^A	1	01	␣	SOH	33	21	!	65	41	A	97	61	a	129	81	ü	161	A1	í	193	C1	Ľ	225	E1	β
^B	2	02	␢	SIX	34	22	"	66	42	B	98	62	b	130	82	é	162	A2	ó	194	C2	⌈	226	E2	Γ
^C	3	03	␣	EIX	35	23	#	67	43	C	99	63	c	131	83	â	163	A3	ú	195	C3	⌋	227	E3	Π
^D	4	04	␤	EOI	36	24	\$	68	44	D	100	64	d	132	84	ä	164	A4	ñ	196	C4	⌋	228	E4	Σ
^E	5	05	␥	ENQ	37	25	%	69	45	E	101	65	e	133	85	à	165	A5	ë	197	C5	⌋	229	E5	σ
^F	6	06	␦	ACK	38	26	&	70	46	F	102	66	f	134	86	ä	166	A6	ë	198	C6	⌋	230	E6	μ
^G	7	07	␧	BEL	39	27	'	71	47	G	103	67	g	135	87	ç	167	A7	ë	199	C7	⌋	231	E7	τ
^H	8	08	␨	BS	40	28	(72	48	H	104	68	h	136	88	ê	168	A8	ë	200	C8	⌋	232	E8	ξ
^I	9	09	␩	HI	41	29)	73	49	I	105	69	i	137	89	ë	169	A9	ë	201	C9	⌋	233	E9	θ
^J	10	0A	␪	LF	42	2A	*	74	4A	J	106	6A	j	138	8A	è	170	AA	ë	202	CA	⌋	234	EA	Ω
^K	11	0B	␫	VI	43	2B	+	75	4B	K	107	6B	k	139	8B	ï	171	AB	ë	203	CB	⌋	235	EB	δ
^L	12	0C	␬	FF	44	2C	,	76	4C	L	108	6C	l	140	8C	î	172	AC	ë	204	CC	⌋	236	EC	ø
^M	13	0D	␭	CR	45	2D	-	77	4D	M	109	6D	m	141	8D	ï	173	AD	ë	205	CD	⌋	237	ED	g
^N	14	0E	␮	SO	46	2E	.	78	4E	N	110	6E	n	142	8E	ÿ	174	AE	ë	206	CE	⌋	238	EE	€
^O	15	0F	␯	SI	47	2F	/	79	4F	O	111	6F	o	143	8F	ÿ	175	AF	ë	207	CF	⌋	239	EF	ñ
^P	16	10	␰	SLE	48	30	0	80	50	P	112	70	p	144	90	ÿ	176	B0	ÿ	208	D0	⌋	240	F0	≡
^Q	17	11	␱	CS1	49	31	1	81	51	Q	113	71	q	145	91	ÿ	177	B1	ÿ	209	D1	⌋	241	F1	+
^R	18	12	␲	DC2	50	32	2	82	52	R	114	72	r	146	92	ÿ	178	B2	ÿ	210	D2	⌋	242	F2	>
^S	19	13	␳	DC3	51	33	3	83	53	S	115	73	s	147	93	ÿ	179	B3	ÿ	211	D3	⌋	243	F3	<
^T	20	14	␴	DC4	52	34	4	84	54	T	116	74	t	148	94	ÿ	180	B4	ÿ	212	D4	⌋	244	F4	¡
^U	21	15	␵	NAK	53	35	5	85	55	U	117	75	u	149	95	ÿ	181	B5	ÿ	213	D5	⌋	245	F5	÷
^V	22	16	␶	SYN	54	36	6	86	56	V	118	76	v	150	96	ÿ	182	B6	ÿ	214	D6	⌋	246	F6	÷
^W	23	17	␷	ETB	55	37	7	87	57	W	119	77	w	151	97	ÿ	183	B7	ÿ	215	D7	⌋	247	F7	÷
^X	24	18	␸	CAN	56	38	8	88	58	X	120	78	x	152	98	ÿ	184	B8	ÿ	216	D8	⌋	248	F8	÷
^Y	25	19	␹	EM	57	39	9	89	59	Y	121	79	y	153	99	ÿ	185	B9	ÿ	217	D9	⌋	249	F9	÷
^Z	26	1A	␺	STB	58	3A	:	90	5A	Z	122	7A	z	154	9A	ÿ	186	BA	ÿ	218	DA	⌋	250	FA	÷
^[27	1B	␻	ESC	59	3B	;	91	5B	[123	7B	{	155	9B	ÿ	187	BB	ÿ	219	DB	⌋	251	FB	÷
^\	28	1C	␼	FS	60	3C	<	92	5C	\	124	7C		156	9C	ÿ	188	BC	ÿ	220	DC	⌋	252	FC	÷
^]	29	1D	␽	GS	61	3D	=	93	5D]	125	7D	}	157	9D	ÿ	189	BD	ÿ	221	DD	⌋	253	FD	÷
^^	30	1E	␾	RS	62	3E	>	94	5E	^	126	7E	~	158	9E	ÿ	190	BE	ÿ	222	DE	⌋	254	FE	÷
^_	31	1F	␿	US	63	3F	?	95	5F	_	127	7F	Δ†	159	9F	ÿ	191	BF	ÿ	223	DF	⌋	255	FF	÷

† ASCII code 127 has the code DEL. Under MS-DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL+BKSP key.

Operaciones aritméticas

- ▣ Las operaciones aritméticas básicas son
 - ▣ Sumar $+$
 - ▣ Restar $-$
 - ▣ Multiplicación $*$
 - ▣ División $/$
 - ▣ Añadir uno a una variable $x++$ ($x=x+1$ ó $x+=1$)
 - ▣ Restar uno a una variable $x--$ ($x=x-1$ ó $x-=1$)
- ▣ **Aritmética real:** Si todas las variables de una expresión son reales la operación se realizará sin truncar números decimales
- ▣ **Aritmética entera:** Si todas las variables y constantes en una expresión son enteros, las sumas, restas, multiplicaciones y exponenciaciones se realizarán sin problema. Sin embargo las divisiones entre enteros ignoran en su resultado la parte decimal que es automáticamente truncada

Operaciones aritméticas:

Aritmética mezcla

- ▣ Las operaciones pueden involucrar números enteros y reales.
- ▣ Los números enteros pueden ser transformados en reales con sólo multiplicar por 1.

$X = 1.*i$

- ▣ Los reales se convierten en enteros con la orden **(int)**. **El resultado trunca los decimales** (no redondea).

$i = (int) x$

Esto convierte lo que tiene inmediatamente a la derecha

▣ Ejemplos:

- ▣ $5/2*3.0=6.0000$ (la división es entera!)
- ▣ $3.0*5/2=7.50000$ (la división es real)
- ▣ $5./2.*3.0=3.0*5./2.=7.500000$
- ▣ ¡¡Para operaciones con salida real hay que convertirlo todo en real!!

Operaciones aritméticas: Orden de prioridad

- Las operaciones matemáticas se realizan siguiendo el siguiente orden de prioridad:
 - Se hacen primero multiplicaciones y divisiones (de izquierda a derecha).
 - Sumas y restas (de izquierda a derecha).
- Para evitar confusiones **se recomienda el uso de paréntesis ()**.
- Ejemplos:
 - $5./2.+3.=5.500000$
 - $5.+3./2.=6.500000$
 - $(5.+3.)/2.=4.000000$

Operaciones aritméticas: <math.h> (biblioteca estándar)

- La librería <math.h> incluye muchas funciones matemáticas

es necesaria la bandera -lm en el

- Las más importantes son (el output es `double`; los input `x` e `y` son `double` y `n` es `int`)

<code>sin(x)</code>	seno de x
<code>cos(x)</code>	coseno de x
<code>tan(x)</code>	tangente de x
<code>exp(x)</code>	exponencial de x
<code>log(x)</code>	logaritmo natural
<code>log10(x)</code>	logaritmo en base 10
<code>pow(x,y)</code>	x^y
<code>sqrt(x)</code>	raíz cuadrada de x
<code>fabs(x)</code>	valor absoluto de x
<code>ldexp(x,n)</code>	$x \cdot 2^n$

Input y output

- Supongamos el siguiente problema: Un cohete es lanzado desde una altura inicial H_0 con velocidad inicial V_0 y una aceleración vertical A . Entonces la altura y velocidad en el instante T después del lanzamiento vienen dadas por las ecuaciones:

$$H = 0.5 A T^2 + V_0 T + H_0$$
$$v = A T + V_0$$

- El siguiente programa asigna el valor $-9.807 \text{ (m/s}^2\text{)}$ a A , 150 (m) a H_0 , 100.00 (m/s) a V_0 y 5.0 (s) a T , y luego calcula los valores de H y T .

Input y output

```
1 # include <stdio.h>
2 # include <math.h>
3
4 /******
5 // This program calculates the velocity and height of a projectile
6 // given its initial height, initial velocity, and constant
7 // acceleration. Variables used are:
8 // H0 : initial height
9 // H  : height at any time
10 // V0 : initial vertical velocity
11 // V  : vertical velocity at any time
12 // A  : vertical acceleration
13 // T  : time elapsed since projectile was launched
14 //
15 // input: none
16 // output: none
17 *****/
18
19
20 int main()
21 {
22     double H0,H,V0,V,A,T;
23
24     A=-9.807;
25     H0=150.0;
26     V0=100.00;
27     T=5.0;
28     H=0.5*A*pow(T,2)+V0*T+H0;
29     V=A*T+V0;
30
31 }
32
```

Input y output: <stdio.h> (Biblioteca estándar)

- El programa calcula los valores de H y V como se le ha ordenado, pero los almacena internamente de modo que pueda verlos el usuario. Más aún, si queremos realizar el cálculo con otra altura o velocidad iniciales, hemos de reescribir el programa, volverlo a compilar y ejecutar.
- Vamos a explicar cómo ver la información generada por el programa (**output**) y cómo variar la información inicial (**input**) sin tener que modificar el programa cada vez.
- La instrucción para leer datos en el programa desde la consola es **scanf**, mientras que la orden para escribir los resultados del programa en consola es **printf**. Ambas forman parte de la biblioteca **<stdio.h>**. El formato estándar es:

```
scanf(formato,&lista_variables)  
printf(formato,lista_variables)
```

■ Ejemplos:

```
scanf("%i",&i);  
printf("%i\t%f\n",x,y);
```


Input y output: printf (<stdio.h>)

■ Sintaxis: `printf(formato,variable1,variable2,...,variableN);`

■ Conversiones:

<code>%i, %d</code>	Int
<code>%lf, %f</code>	Double
<code>%.8lf</code>	Double con 8 decimales
<code>%e, %E</code>	Double con formato <code>m.dddde±xx</code>
<code>%g,%G</code>	Double; usa %e si el exponente es menor que -4. De otra forma usa %f.
<code>%c</code>	Caracter

Aunque en printf sean

■ Formato:

<code>\t</code>	Inserta un tabulador
<code>\n</code>	Nueva línea

■ El texto se escribe sin más (si no va precedido de `%` ó `\n`)

Input y output: scanf (<stdio.h>)

■ Síntaxis:

```
scanf(formato,&variable1,&variable2,...,&variableN);
```

- Las variables siempre deben llevar **&** delante (son direcciones de memoria, ver el apartado sobre punteros).

■ Conversiones:

%i	Número Entero
%c	Caracter
%s	Cadena de caracteres
%lf, %g	Número Real

¡Ojo! %f para float y %lf para

- ¡¡No se escribe **\n** al final!!

Input y output : printf y scanf. Ejemplo

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

////////////////////////////////////
// This program calculates the velocity and height of a projectile
// given its initial height, initial velocity, and constant
// acceleration. Variables used are:
// H0 : initial height
// H : height at any time
// V0 : initial vertical velocity
// V : vertical velocity at any time
// A : vertical acceleration
// T : time elapsed since projectile was launched
//
// Input: H0, V0, T
// Output: V, H
//
////////////////////////////////////

int main()
{
    int i,j,k;
    double x,y,z;

    double H0; // initial height
    double H; // height at any time
    double V0; // initial vertical velocity
    double V; // vertical velocity at any time
    double A; // vertical acceleration
    double T; // time elapsed since projectile was launched

    A = -9.807;
    printf("Enter the initial height\n");
    scanf("%lf",&H0);

    printf("Enter the initial velocity\n");
    scanf("%lf",&V0);

    printf("Enter time at which to calculate height and velocity\n");
    scanf("%lf",&T);

    H = 0.5*A*pow(T,2.)+V0*T+H0;
    V = A*T + V0;

    printf("At time %lf the vertical velocity is %lf\n",T,V);
    printf("and the height is %lf\n",H);

    return 0;
}
```

Input y output: printf y scanf.

Ejemplo con diferentes formatos

```
# include <stdio.h>

////////////////////////////////////
// Programa para practicar con las funciones scanf y printf
// Probar distintos valores enteros y reales
////////////////////////////////////

int main()
{
    double x,y,z;
    int i,j,k;

    printf("Escribe un número entero\n");

    scanf("%i",&i);

    printf("i=%i\n",i);
    printf("c=%c\n",i);

    printf("Escribe un número real\n");

    scanf("%lf",&x);

    printf("lf=%lf\nlf.3=%.3lf\ne=%e\ng=%g\n",x,x,x,x);

    return 0;
}
```

Input y output: ficheros.

fprintf y fscanf <stdio.h>

- Para acceder a ficheros primero hay que declarar punteros a ficheros

```
FILE *f1,*f2;
```

- Una vez declarados hay que abrir los ficheros con la orden fopen

```
f1=fopen(nombre,modo);
```

- El nombre debe ir entrecomillado y el modo determina las funciones que podemos hacer con el fichero.

"w"	Escritura (destruye el fichero si ya existe)
"r"	Lectura
"a"	Añadido (escritura al final del fichero, si ya existe lo preserva)

- La opción '+' permite leer y escribir en el mismo fichero. (Ej: "r+")

Input y output: ficheros. fprintf y fscanf <stdio.h>

- Para evitar pérdida de datos los ficheros se cierran con la orden `fclose` (y el puntero puede reutilizarse): `fclose(f1);`
- La orden para escribir es `fprintf`, con las mismas conversiones y formato que `printf`.

```
fprintf(f1,formato,variable1,variable2,...,variableN);
```

- Para leer de un fichero la orden es `fscanf`, con las mismas conversiones y formato que `scanf`

```
fscanf(f1,formato,&variable1,&variable2,...,&variableN);
```

Input y output: ficheros. fprintf y fscanf <stdio.h>

■ Ejemplo:

```
# include <stdio.h>

////////////////////////////////////
// Programa para practicar con las funciones fscanf y fprintf
// file1.txt -> fichero de entrada con dos columnas de números reales
// file2.txt -> fichero de salida con dos columnas, con los datos manipulados
////////////////////////////////////

int main()
{
    double x,y;
    double x2,y2;

    FILE *f1,*f2; //punteros a ficheros

    f1=fopen("file1.txt","r"); //abrimos el primer fichero para lectura
    f2=fopen("file2.txt","w"); //abrimos el segundo fichero para escribir

    while(fscanf(f1,"%lf\t%lf",&x,&y)!=EOF) // EOF='end of File' lee hasta que se termina el fichero
    {
        // fscanf no tiene '\n' al final del formato!!
        x2=10.*x;
        y2=y/10.;

        fprintf(f2,"%lf\t%lf\n",x2,y2);
    }

    // cerramos los ficheros antes de salir
    fclose(f1);
    fclose(f2);

    return 0;
}
```

Ejercicios: Input/output y aritmética

■ **Ejercicio:** Escriba el programa anterior, compílelo y ejecútelo.

■ **Ejercicio:** Escriba un programa que convierta Celsius a Farenheit y muestre el resultado por pantalla. Recuerde que la relación entre los dos es:

$$F = \frac{9}{5}C + 32$$

■ **Ejercicio:** Escriba un programa que muestre en la pantalla la raíz cuadrada de cada número entero del 1 al 20.

■ **Ejercicio:** Escriba un programa que tome tres números desde el teclado, a, b y c, y muestre en pantalla las raíces de la ecuación

$$ax^2 + bx + c = 0$$

Note que el programa fallará cuando intente calcular raíces de números negativos. Este problema lo solventamos más adelante.

Control de flujo: if - else

- La proposición if-else se utiliza para expresar decisiones. La síntesis formal es:

```
If (condición)
    orden_1;
else
    orden_2;
```

- Múltiples órdenes pueden incluirse con el uso de {}.

```
If (condición)
{
    orden_1a;
    orden_1b;
}
else
{
    orden_2a;
    orden_2b;
}
```

Control de flujo: if - else

- La proposición **else** es opcional, se puede y suele omitir.
- La proposición if se puede anidar. Ejemplo:

```
if (n>0)
    if ( i<0)
        orden_1;
    else
        orden_2;
```

- El uso de tabuladores y llaves **{ }** evita confusiones.
- También se puede usar más de una orden con los operadores **&&** (AND) y **||** (OR).

Control de flujo: else if - switch

- Se puede usar la orden **else if** para ejecutar órdenes si no se ha cumplido la condición anterior y se cumple una nueva.

```
if (condición_1)
else if (condición_2) orden_1;
else if (condición_3) orden_2;
else orden_3;
```

- La última orden sólo se ejecuta si no se cumplen ninguna de las anteriores.
- Con la orden **switch** se pueden hacer decisiones múltiples basadas en el valor entero de una variable o constante.

```
switch (expresión)
case c1: orden_1;
case c2: orden_2;
default: orden_3
```

- **default** se ejecuta si la expresión no coincide con ningún caso.

Control de flujo: operadores

- La proposición a evaluar se define mediante el uso de operadores.

<code>==</code>	Igual a
<code>!=</code>	Distinto a
<code><, <=</code>	Menor a, menor o igual a
<code>>, >=</code>	Mayor a, mayor o igual a
<code>&&</code>	AND (lógico)
<code> </code>	OR (lógico)
<code>!</code>	Negación (lógica)

- Ejemplo** (equivalente al anterior):

```
if (n>0 && i<0) orden_1;  
else orden_2;
```

Control de flujo: if - else

- **Ejemplo:** El siguiente programa determina si un número es menor que -10, está entre -10 y 10, o es mayor que 10.

```
# include <stdio.h>

////////////////////////////////////
// Programa para probar las órdenes if-else
////////////////////////////////////

int main()
{
    double x,y,z;
    int i,j,k;

    printf("Escribe un número entero\n");
    scanf("%i",&i);

    if (i>10)
        printf("Tu número es mayor que 10\n");
    else if (i<=-10)
        printf("Tu número es menor que -10\n");
    else printf("Tu número está entre -10 y 10\n");

    return 0;
}
```

- **Ejercicio:** Usando **if-else**, realizar un programa que determine si un número es par o impar, y si es impar que determine si es múltiplo de 3.

Control de flujo: for

- La orden **for** sirve para crear iteracciones y ciclos (también existe **while**, pero es equivalente).
- La sintaxis es: **for (expr1; expr2; expr3) proposición;**
- La expresión **expr1** se ejecuta al principio del bucle, y **expr3** al final de cada ciclo, junto con la proposición. El bucle continúa mientras se cumpla la expresión **expr2**.
- **Ejemplo:** Este bucle calcula la suma de los números del 1 al 100.

```
x=0.;
```

```
for (i=1; i<=100; i++) x+=i;
```

- Si la segunda expresión **expr2** está en blanco se considera verdadera (creando un bucle infinito).
- La orden **break** sale de un bucle, la orden **continue** salta al siguiente ciclo.

Control de flujo: Ejercicios

- **Conversión de Temperatura:** Escriba un programa que imprima en pantalla una tabla de conversión de Celsius a Fahrenheit de 0C a 100C de grado en grado
- **Rango de las variables reales en C:** Escriba un programa que encuentre el valor máximo que se puede almacenar en una variable real. Por ejemplo, puede multiplicar repetidamente una variable por 2 dentro de un bucle, mostrando en pantalla el resultado hasta que el ordenador dé un error. También puede escribir el valor máximo en un fichero.
- **Rango de las variables enteras en C:** Reescriba el programa anterior para enteros.

Control de flujo: Ejercicios

- **Ejercicio:** Escriba un programa que tome un número del teclado y compruebe su tamaño. Si el número es menor que 100 el programa debe escribir en pantalla “Es pequeño”, y si es mayor o igual a 100 debe escribir “Es grande”. Coloque todo en un bucle de forma que se puedan entrar varios números uno tras otro.
- **Ejercicio:** Reescriba el programa que resolvía la ecuación de segundo grado. Ahora, antes de calcular la raíz cuadrada, compruebe si el discriminante es negativo y, si lo es, escribir en pantalla “No hay raíces reales”.
- **Ejercicio:** Escriba un programa que lea desde el teclado 10 números y los memorice en un vector. Ordénelos de forma que el más pequeño sea el primero de la lista. Un algoritmo sencillo (aunque no eficiente) de ordenación consiste en comparar cada número con todos los demás y ver cuántos son más pequeños: si hay n más pequeños el orden del número es $n+1$.

Arrays y punteros

- Un array es un conjunto de variables ordenadas que comparten el mismo nombre y son del mismo tipo.
- El acceso a cada una de las variables se realiza variando su índice. Por ejemplo, un grupo de 10 estudiantes han recibido notas por un trabajo empleado. Si la nota de cada estudiante se almacena en una variable: `mark`, `mark2`, ..., `mark10`, la nota media será

`sum=(mark1+mark2+...+mark10)/10.0`

- Es más conveniente utilizar un vector `mark[i]`, donde `i` puede tomar valores entre 1 y 10, y utilizar un bucle `for`.

```
sum=0.;  
for (i=1;i<=10;i++) sum+=mark[i];  
sum/=10.;
```

- La ventaja es más importante si tenemos que sumar 1000 ó 10000 datos.

Arrays y punteros: Declaración

- Un array puede tener múltiples dimensiones, definidas por distintos índices.
- Un array se declara al principio del programa como el resto de variables. Hay que indicar el tamaño.

```
int mark[10];
```

```
double table[10][15], var[2][10][5];
```

- El índice inicial es siempre 0.
- Se puede asignar la memoria de manera dinámica con la orden `malloc`. Esto significa que no hay que decidir el tamaño del array al principio y puede variar durante la ejecución del programa (ver [librería "nrutil.h" de Numerical Recipes](#)).

Arrays y punteros: Declaración

- Un puntero es una variable que apunta directamente a una dirección de memoria. Los punteros se utilizan para manipular arrays con una mejora en velocidad muy alta y para pasar argumentos a funciones.
- El operador `*` define un puntero, y el operador `&` da la dirección de memoria de una variable. **Ejemplo:**

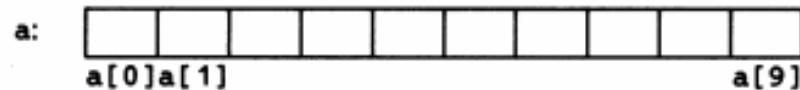
```
int x=1, y=2, z[10];  
int *ip;           // ip es un puntero a entero  
ip=&x;              // ip ahora apunta a x  
y=*ip;             // y es ahora 1  
*ip=0;             // x es ahora 0  
ip=&z[0];           //ip ahora apunta a z[0]]
```

Arrays y punteros: relación.

Los punteros y los arrays están tan relacionados en C que habitualmente se explican juntos. Vamos a analizarlo con un ejemplo (sacado del [libro de K&R](#)).

La orden: `int a[10];`

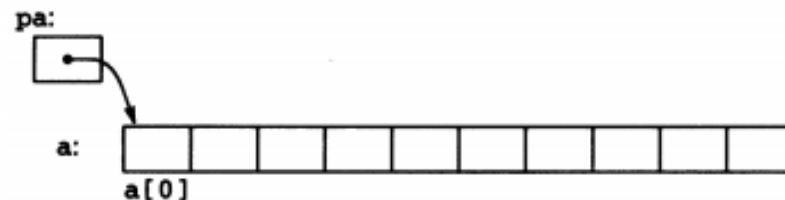
define un array de 10 elementos.



Si definimos un puntero a entero y hacemos la siguiente asignación.

```
int *pa;  
pa=&a[0];
```

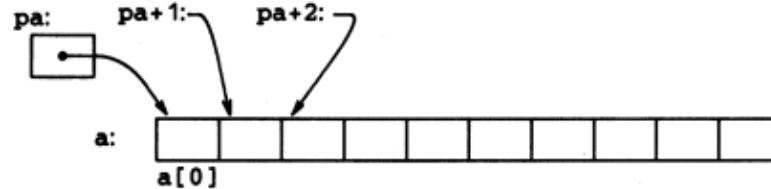
`pa` pasa a apuntar al primer elemento de `a`.



Arrays y punteros:

Si hacemos ahora $x = *pa$ el contenido de $p[0]$ se copiará a x .

Si pa apunta a un punto particular del array, por definición $pa+1$ apuntará al siguiente elemento. $pa+i$ apuntará a i elementos después de pa .



Igualmente $*(pa+i)$ dará el mismo resultado que $p[i]$.

Nota: Se pueden definir también arrays de punteros, punteros a punteros y punteros a funciones.

Funciones

- Las funciones son programas de C. La mayoría de problemas son tan complejos que resulta conveniente dividirlos en pequeños subproblemas. Además, hay tareas que es normal realizar varias veces en el mismo programa, y las funciones evitan tener que reescribir las mismas órdenes varias veces.
- Una función toma una serie de parámetros como argumentos, realiza una serie de operaciones y devuelve un único valor (o ningún valor) como resultado. Hay muchas funciones predefinidas en las librerías estándar de C, incluyendo funciones de entrada y salida (`stdio.h`), funciones matemáticas (`math.h`) o de procesamiento de cadenas (`string.h`).
- En C no hay diferencia entre funciones y subrutinas (como sí la hay en FORTRAN). Una función puede no devolver ningún valor, si está definida como `void`.

Funciones

- Una función se define de la siguiente manera:

```
tipo nombre_función (var1, var2, ..., varN)
{
    declaraciones
    proposiciones
}
```

- El tipo determina la salida de la función, y puede corresponder con cualquier tipo de variable (`int`, `double`, `char`, ...). Si la función no devuelve ningún valor su tipo es `void`. La salida se devuelve con el comando `return`.

¡Note que una función no puede devolver un array!

- Las variables que recibe la función no pueden ser modificados.

Funciones

- **Ejemplo:** El siguiente programa crea y prueba una función exponenciadora.

```
# include <stdio.h>

int power (int base, int n);

int main()
//Prueba la función power
{
    int i;

    for (i=0;i<10;i++)
        printf("%i\t%i\t%i\n", i, power(2,i),power(-3,i));

    return 0;
}

int power (int base, int n)
//eleva la base a la n-esima potencia; n>=0
{
    int i,p;

    p=1;
    for (i=1;i<=n;++i)
        p=p*base;

    return p;
}
```


Funciones

- Para evitar problemas al llamar las funciones en otras funciones es recomendable definir todas las funciones antes del **main**.
- Las variables simples que reciben las funciones son llamadas “**por valor**” (en contra de otros lenguajes como FORTRAN, donde pueden ser llamadas “**por referencia**”). Esto significa que la función hace una copia del valor de la variable, y si la modifica no se ve modificada la original. Esto no ocurre con los arrays, que sí pueden ser modificados ya que la función recibe un puntero.
- Para modificar variables en una función hay dos posibilidades. **Pasar un puntero** a la variable, o utilizar **variables externas**.

Funciones: Variables externas

- ▣ Las variables externas se definen antes del main y luego en cada función que se utilicen usando el comando **extern**.
- ▣ **Ejemplo:** La función **power** se puede reescribir pasando una de las variables como externa.

```
# include <stdio.h>

int n;
int power (int base);

int main()
//Prueba la función power
{
    extern int n;
    int i;

    for (i=0;i<10;i++)
    {
        n=i;
        printf("%i\t%i\t%i\n", i, power(2),power(-3));
    }

    return 0;
}

int power (int base)
// power: eleva la base a la n-esima potencia; n>=0
{
    int i,p;
    extern int n;

    p=1;
    for (i=1;i<=n;++i)
        p=p*base;

    return p;
}
```

Primera definición de la variable externa n

Declaración de la variable n como externa en la función main

Declaración de la variable n como externa en la función power

Funciones: Punteros

- También se pueden modificar las variables que recibe la función mediante el uso de punteros.
- **Ejemplo:** Reescribimos la función `power` dando la salida por un puntero.

```
# include <stdio.h>

void power (int *p, int base, int n);

//Prueba la función power
int main()
{
    int i,p;

    for (i=0;i<10;i++)
    {
        power(&p,2,i);
        printf("%i\t%i\t", i, p);
        power(&p,-3,i);
        printf("%i\n", p);
    }

    return 0;
}

// power: eleva la base a la n-esima potencia; n>=0
void power (int *p, int base, int n)
{
    int i;

    *p=1;
    for (i=1;i<=n;++i)
        *p=(*p)*base;

    return;
}
```

Funciones: Arrays

- ▣ Los arrays sí que entran en las funciones como punteros y se pueden modificar.
- ▣ En el caso de vectores (1-dim) hay que indicar sólo el nombre del array y tener cuidado de no escribir fuera de su rango.
- ▣ **Ejemplo:** El siguiente programa calcula los cuadrados de los números enteros entre 1 y 10 y los almacena en el vector `v[]`.

```
# include <stdio.h>

void vec(int v[], int size);

int main()
{
    int i;
    int v[10];
    int size=10;

    vec(v,size);
    for (i=0;i<size;i++)
        printf("%i\t%i\n", i, v[i]);

    return 0;
}

void vec(int v[], int size)
//almacena en el vector v: v[i]=i*i
{
    int i;

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

    return;
}
```

¿Qué pasaría si la función intentase escribir más allá del tamaño del fichero?

Funciones: Arrays

- Si pasamos un array multidimensional como parámetro de una función debemos indicar el tamaño de al menos todas las dimensiones menos la última, para que el compilador sepa donde se encuentra cada dirección de memoria. También se puede (y es recomendable) indicar el tamaño en todas las dimensiones.

- **Ejemplo:**

```
void prod(int v[5][5], int size);

int main()
{
    int i,j;
    int v[5][5];
    int size=5;

    prod(v,size);
    for (i=0;i<size;i++)
        for (j=0;j<5;++j) printf("%i\t%i\t%i\n", i, j, v[i][j]);

    return 0;
}

void prod(int v[5][5], int size)
// Function that calculates the product of two numbers
// v[i][j]=i*j
{
    int i,j;

    for (i=0;i<size;++i)
        for (j=0;j<5;++j)
            v[i][j]=i*j;

    return;
}
```

Funciones: Ejercicio

■ **Ejercicio:** Calcule la función de distribución radial (FDR) para los orbitales 1s, 2s y 2p y normalice el resultado usando el método de Simpson. La FDR es el valor del módulo al cuadrado de la función de ondas multiplicado por $4\pi r^2$. Las funciones de onda son

$$1s \rightarrow \exp(-r/2)$$

$$2s \rightarrow \frac{(2-r)}{\sqrt{32}} \exp(-r/2)$$

$$3s \rightarrow \frac{(6-6r+r^2)}{\sqrt{972}} \exp(-r/2)$$

Problemas

- **Ejercicio, ecuación de van der Waals:** La ecuación de estado de van der Waals es una extensión de la ecuación para los gases ideales que tiene en cuenta algunos aspectos de la desviación de los gases reales con respecto a su comportamiento ideal. La ecuación es:

$$\left(p + \frac{an^2}{V^2}\right) (V - nb) = nRT$$

- en la que n es el número de moles del gas, p es la presión, T es la temperatura, V es el volumen, R la constante de los gases (8.414) y a y b son parámetros fenomenológicos que dependen de cada gas particular. Escriba un programa que imprima una tabla con varias p 's para al menos 10 volúmenes entre $V1$ y $V2$. Las variables a , b , T , n , $V1$ y $V2$ deben de ser entradas por el teclado y la salida debe de ir a un archivo. Use para el hidrógeno $a=0.0247$, $b=26.6 \cdot 10^{-6}$.
- **Ejercicio:** Estime el número π con métodos Montecarlo (ver las notas y este ejemplo interactivo <http://tprc.blogspot.com.es/2013/01/tecnicas-montecarlo-la-estadistica.html>).
- **Ejercicio:** Calcule el mayor autovalor de una matriz simétrica usando el método de las potencias (ver las notas).