EJERCICIOS DE INICIACIÓN A LA PROGRAMACIÓN EN LENGUAJE C

ACADEMIA

C/ Cartagena 99 1ºC. 28002 Madrid Tel. 91 51 51 321

e.mail: academia@cartagena99.com

web: www.cartagena99.com

INTRODUCCIÓN. EL PRIMER PROGRAMA EN C

Resumen-Teoría

Para realizar un programa en C necesitamos un **compilador** de lenguaje C instalado en nuestro ordenador. Llamamos compilador al programa capaz de traducir nuestras órdenes en órdenes comprensibles por el PC, es decir, un programa que nos permite crear programas.

Nuestras órdenes estarán escritas en lo que conocemos como **lenguaje de programación** (C, en este caso), las órdenes comprensibles por el PC se llaman código máquina, son ceros y unos, y se guardan en ficheros con extensión ".exe", que es lo que llamamos **ficheros ejecutables o programas**.

Los compiladores suelen incorporar un **entorno de desarrollo** con ventanas, en el que podremos escribir nuestro programa, además de otra serie de herramientas y facilidades para el programador (nosotros). Un compilador que recomendamos a nuestros estudiantes es el Dev-C++, que se puede descargar gratuitamente en: www.bloodshed.net/devcpp.html.

Ejercicios

Ejercicio 1: Primer programa (Hola Mundo)

Para realizar tu primer programa, instala en tu PC este compilador, o cualquier otro que tengas, y crea un proyecto de consola (son los que no tienen interfaz gráfico, se ejecutan en una ventana de fondo negro). Al crear el proyecto normalmente se te creará con un fichero en blanco por defecto, si no es así, tendrás que crear un nuevo fichero: Ponle el nombre que quieras, sálvalo con la extensión ".c", y añádelo al proyecto.

En este punto ya estamos preparados para escribir el primer programa. Teclea en el fichero lo siguiente:

```
#include<stdio.h>
int main()
{
printf("Hola mundo!!");
system("PAUSE");
return 0;
}
```

Compila y ejecuta este programa. Al ejecutarse, se abrirá una ventana negra (la consola de MS-DOS) con el texto: `Hola mundo!!`

PRESENTAR TEXTO POR PANTALLA.

Resumen-Teoría

Si nos fijamos en el primer programa, el programa del ejercicio del punto anterior, veremos que pone: printf(“Hola mundo !! ”);

Esta línea del programa es la que se encarga de presentar la frase por pantalla. De ahí se deduce que la instrucción *printf* sirve para presentar texto por pantalla, y que el texto que se presenta es lo que está en comillas.

Por ejemplo:

```
printf(“Frase de prueba.”);
printf(“ Otra frase...”);
```

Presenta por pantalla: *Frase de prueba. Otra frase...*

Nótese que aunque las órdenes estén en líneas diferentes la presentación de las frases es independiente de esto. Concretamente, para C, los saltos de línea en el código son considerados como separadores igual que los espacios simples, e ignorados si hay más de uno. Por ejemplo:

```
printf(“Frase de prueba.”);printf(“ Otra frase...”);
```

Es idéntico a:

```
printf(“Frase de prueba.”);  printf(“ Otra frase...”);
```

E idéntico a:

```
printf(“Frase de prueba.”);
printf(“ Otra frase...”);
```

Además, hay algunas cosas que conviene saber, como cómo hacer saltos de línea, tabulaciones, etc. Para esto se utilizan unos caracteres especiales, que son caracteres normales precedidos del carácter de diagonal invertida \(generalmente se obtiene pulsando AltGr + tecla de arriba a la izquierda). Estos caracteres especiales se llaman secuencias de escape, se pueden incluir en cualquier parte de la frase, y son los siguientes:

\a	carácter de alarma
\b	retroceso
\n	nueva línea
\r	regreso de carro
\t	tabulador horizontal
\\	diagonal invertida
\'	apóstrofe
\"	comillas

Otro aspecto de C que se puede probar en este apartado son los comentarios. A menudo sucede que nos interesa incluir comentarios en el programa que escribimos. Esto es de gran utilidad para poder entender de un vistazo aquello que hemos escrito nosotros mismos, u otro programador. Para esto disponemos de comentarios de línea y comentarios de bloque.

Comentario de línea: Se marcan con dos barras // y van de las dos barras hasta el final de la línea.

```
printf("hola \a"); //imprime "hola" y hace sonar la alarma.
```

Comentario de bloque: Se marcan con barra y asterisco el inicio del comentario /* y con asterisco barra el fin del comentario */.

Esto no es comentario /*este texto sí es un comentario */ esto ya no es comentario

Ejercicios

Ejercicio 1: Con saltos de línea

Escribe un programa que presente por pantalla:

Hola

Mundo

Ejercicio 2: Presentando por pantalla varios caracteres

Escribe un programa que presente por pantalla:

apostrofe(') *comillas("")* *diagonal invertida(/)*

Nótese que entre las palabras hay dos tabulaciones.

Ejercicio 3: Dando alarma

Escribe un programa que presente por pantalla:

Alarma

Y haga sonar la alarma del PC.

Ejercicio 4: Presentando por pantalla

Escribe un programa que presente tu nombre encuadrado entre asteriscos.

Ejercicio 5: Probando secuencias de escape

¿Qué crees que imprimirá la siguiente sentencia? Comprueba tu respuesta.

```
printf("Frase de prueba.\rotra frase encima\ny otras\b
mas\n");
```

TIPOS DE DATOS DEL LENGUAJE C

Resumen-Teoría

Tipos de Datos:

Los tipos de datos de un lenguaje son, como su propio nombre indica, los tipos de datos con los que se puede trabajar en un lenguaje de programación. El lenguaje C ofrece una colección de tipos de datos bastante limitada, aunque no por ello poco funcional. Dicha colección se compone de los siguientes tipos:

- **char:**

- **Contenido:** Puede contener un carácter. Un carácter se codifica con un Byte, es decir 8 bits. Por lo que el tipo char también puede contener números enteros representables con 8 bits.
- **Tamaño:** 1 byte.
- **Ejemplos:** 'a' (carácter 'a' ó numero 97), ver tabla ASCII a continuación.

- int:
  - **Contenido:** Un número entero
  - **Tamaño:** El determinado por la arquitectura para números enteros. En arquitecturas Intel/x86 es 4 bytes
  - **Ejemplos:** 18 (número 18, expresado en base 10); 0x12 (número 18 expresado en base 16); 022 (número 18 expresado en base 8).

- float:
  - **Contenido:** Un número en coma flotante o número decimal.
  - **Tamaño:** El determinado por la arquitectura para números en coma flotante. En arquitecturas Intel/x86 es 4 bytes
  - **Ejemplos:** 23.4; -45.56.

- double:
  - **Contenido:** Un número en coma flotante de precisión doble (doble número de decimales que un float) .
  - **Tamaño:** El determinado por la arquitectura para números en coma flotante de doble precisión. En arquitecturas Intel/x86 es 8 bytes.
  - **Ejemplos:** Igual que los float pero permiten poner más decimales.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Figura 0.1 Tabla de caracteres ascii

Variables:

Para trabajar con datos de estos tipos, es necesario poder guardar dichos datos en variables. ¿Qué es una variable? Una variable se define como «*Espacio de memoria, referenciado por un identificador, en el que el programador puede almacenar datos de un determinado tipo.* » o lo que es lo mismo: Un sitio donde guardar un dato de un tipo determinado. Al identificador de la variable se le suele llamar “nombre” de dicha variable.

Para usar una variable hay que declararla. Declarar una variable es indicar al compilador que debe reservar espacio para almacenar valores de un tipo determinado, que serán referenciados por un identificador determinado. En C debemos declarar **todas** las variables antes de usarlas, establecer el tipo que tienen y, en los casos que sea necesario, darles un valor inicial.

A la hora de declarar una variable debemos tener en cuenta diversas restricciones :

- Los nombres de variables se componen de letras, dígitos y el caracter de subrayado _.
- El primer caracter del nombre debe ser una letra o el carácter de subrayado.
- Las letras mayúsculas y minúsculas se consideran distintas en el lenguaje C.
- Las palabras reservadas del lenguaje no se pueden usar como nombres de variable.

Algunos ejemplos de variables:

Declaración de una variable entera llamada "mi_variable":

```
int mi_variable;
```

Declaración de una variable entera llamada "mi_variable" e inicialización de dicha variable con el valor 34.

- `int mi_variable=34;`

Declaración de una variable float llamada "mi_float":

```
float mi_float;
```

Declaración de una variable entera llamada "mi_float" e inicialización de dicha variable con el valor 34.223.

```
float mi_float=34.223;
```

Introducción de datos (Función *scanf*):

Esta función se puede utilizar para la introducción de cualquier combinación de valores numéricos o caracteres.

En términos generales la función scanf se escribe:

scanf("cadena de control", arg1,arg2,...,argn);

En la cadena de control se incluyen grupos de caracteres, uno por cada dato de entrada. Cada grupo debe comenzar con el signo de porcentaje, que irá seguido, en su forma más sencilla, de un carácter de conversión que indica el tipo de dato correspondiente.

Carácter de conversión	Significado
%c	El dato es un carácter.
%d	El dato es un entero decimal.
%e	El dato es un valor en coma flotante.

%f	El dato es un valor en coma flotante.
%g	El dato es un valor en coma flotante.
%i	El dato es un entero decimal, octal o hexadecimal.
%s	El dato es una cadena de caracteres.

Cada nombre de variable debe ir precedido por un ampersand (&).

Los datos que se introducen deben corresponderse en tipo y orden con los argumentos de la función scanf.

Ejemplo:

```
scanf(“%d”, &a);
```

lee un número entero introducido por el teclado y guarda el valor leído, en la variable a.

Ejemplo:

- `scanf(“%f”, &a);`

lee un número con decimales introducido por el teclado y guarda el valor leído, en la variable a.

Ejemplo:

```
scanf(“%c”, &a);
```

lee un carácter introducido por el teclado y guarda el valor leído, en la variable a. Esta sentencia, es equivalente a: `a=getchar();`

Ejemplo:

```
int  i;
float j;
scanf("%d %f",&i,&j);
```

lee un entero y un float introducidos por este orden por el teclado.

Presentación de datos (función printf):

Se pueden escribir datos en el dispositivo de salida estándar utilizando la función de biblioteca printf. Es análoga a la función scanf, con la diferencia que su propósito es visualizar datos en vez de introducirlos.

En general la función printf se escribe:

printf(cadena de control, arg1,arg2,...,argn);

Ejemplo:

```
printf("%d", 23);
```

imprime por pantalla: 23

Ejemplo:

```
printf("%f", 27.64);
```

imprime por pantalla: 27.64

Ejemplo:

```
printf("x=%f", 27.64);
```

imprime por pantalla: x=27.64

Ejemplo:

```
printf("h%cla", 'o');
```

imprime por pantalla: hola

Ejemplo:

```
printf("%c=%d", 'x',45);
```

imprime por pantalla: x=45

Ejercicios:

Ejercicio 1: Leyendo números enteros

1.a) Escribe un programa que pida un número entero, y conteste al usuario: "Has introducido el numero (x), gracias".

1.b) Escribe un programa que pregunte al usuario cuántos años tiene, y conteste al usuario: "Ahora se que tienes (x) años, gracias".

1.c) Escribe un programa que pregunte la hora, y conteste con un mensaje: “Hora introducida ok. Son las 18:30:00 (por ejemplo)”.

Ejercicio 2: Leyendo letras

Escribe un programa que pregunte al usuario sus iniciales y conteste diciendo: “Sus iniciales son: A.J.R. (por ejemplo)”.

Ejercicio 2: Leyendo números con decimales

Escribe un programa que pregunte al usuario su altura aproximada, sus iniciales y conteste diciendo: “Sus iniciales son: A.J.R. y su altura 1.34 (por ejemplo)”.

OPERACIONES SIMPLES

Resumen-Teoría:

Sentencias:

Como habréis observado en los ejemplos anteriores, cada línea está terminada con un punto y coma. Esto es así porque cada sentencia en C debe terminar con punto y coma.

Una sentencia o instrucción hace que el ordenador lleve a cabo alguna acción. En C hay tres tipos de sentencias: de expresión, de control y compuestas.

Una sentencia de expresión consiste en una expresión seguida de un punto y coma. Su ejecución hace que se evalúe la expresión.

La sentencia más importante es la de asignación. El operador que se utiliza es el de la igualdad “=”, pero su significado es distinto que en las matemáticas. Este operador en un programa indica que la variable de la izquierda del =, se escribe con el valor de la expresión de su derecha..

Ejemplos:

```
a = 3;
```

Asigna a la variable “a” el valor 3.

```
c = a + b;
```

Asigna a la variable “c” la suma de los valores a y b.

De momento basta con conocer las sentencias de expresión, pero se deja ya apuntado que: Una sentencia compuesta es aquella formada por varias sentencias encerradas entre llaves (las sentencias individuales pueden ser de cualquiera de los tres tipos mencionados); y que una sentencia de control es la que se utiliza para realizar bucles o ramificaciones.

Sentencias de Expresión. Operadores:

En C existen una gran variedad de operadores, que se pueden agrupar de la siguiente manera:

- Operadores aritméticos.
- Operadores relacionales.
- Operadores lógicos.
- Operadores a nivel de bit (bitwise operators).
- Operadores especiales.

Operadores aritméticos:

Los operadores aritméticos nos permiten, básicamente, hacer cualquier operación aritmética, que necesitemos (ejemplo: suma, resta, multiplicación, etc). En la siguiente tabla se muestran los operadores de los que disponemos en C y su función asociada.

<u>Nota:</u> Todos ellos aceptan operandos de cualquier tipo excepto el Módulo,Incremento y Decremento, que sólo acepta operadores enteros.

Tabla: Operadores aritméticos

Operador	*Acción*	*Ejemplo*
-	Resta	x = 5 - 3; // x vale 2
+	Suma	x = 2 +3; // x vale 5
*	Multiplicación	x = 2 *3; // x vale 6
/	División	x = 6 /2; // x vale 3
%	Módulo	x = 5 % 2; // x vale 1
- -	Decremento	x = 1; x--; // x vale 0
++	Incremento	x = 1; x++; // x vale 2

Los incrementos y decrementos se pueden poner de forma prefija y postfija:

a) Preincremento y predecremento (formas prefijas)

Cuando un operador de incremento o decremento precede a su operando, se llevará a cabo la operación de incremento o de decremento antes de utilizar el valor del operando. Veámoslo con un ejemplo:

```
int x,y;
x = 2004;
y = ++x;
/* x e y valen 2005. */
```

b) Postincremento y postdecremento (formas postfijas)

En el caso de los postincrementos y postdecrementos pasa lo contrario: se utilizará el valor actual del operando y luego se efectuará la operación de incremento o decremento.

```
int x,y

x = 2004;
y = x++;
/* y vale 2004 y x vale 2005 */
```

Operadores de Asignación:

Se utilizan para formar expresiones de asignación en las que se asigna el valor de una expresión a un identificador.

El más usado es el operador **=**. Una expresión de asignación recibe el nombre de sentencia de asignación y es de la forma:

identificador = expresión

Donde *identificador* representa generalmente una variable y *expresión* una constante, una variable o una expresión más compleja.

Si los dos operandos de una sentencia de asignación son de diferente tipo, el valor de la expresión de la derecha se convierte automáticamente al tipo del identificador de la izquierda. Hay casos en que esta conversión conlleva una alteración:

Un valor en coma flotante se trunca si se asigna a un identificador entero. Un valor en doble precisión se redondea si se asigna a un identificador de coma flotante.

En C están permitidas asignaciones múltiples de la forma

identificador 1 =...= identificador n = expresión

C posee además los siguientes cinco operadores de asignación: **+=, - =, *=, /= y %=**.

i +=5 equivale a i = i + 5,
I - =7 equivale a i = i - 7, etc.

Operadores relacionales

Al igual que en matemáticas, estos operadores nos permitirán evaluar las relaciones (igualdad, mayor, menor, etc) entre un par de operandos (en principio, pensemos en números). Los operadores relacionales de los que disponemos en C son:

Tabla 3.3: Operadores relacionales.

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

El que devuelven los operadores relacionales, es un valor ``cierto" (true) o ``falso" (false). La mayoría de lenguajes tienen algún tipo predefinido para representar estos valores (boolean, bool, etc); sin embargo en C, se utilizan valores enteros para representar esto:

falso (false)	0
cierto (true)	cualquier valor distinto de 0, aunque normalmente se usará el 1

Para utilizar los operadores relacionales es necesario conocer un operador del tema siguiente, el operador if-else, de control de flujo. Así pues, se adelanta su presentación en este apartado:

La proposición if-else se usa para expresar decisiones. La sintáxis es:

```
if(expresión)
   proposición1;
else
   proposición2;
```

Si la expresión es verdadera, se ejecuta la proposición1, de lo contrario se realiza la proposición2. Por ejemplo:

```
int x=4;
if(x<10)
   printf("es menor que diez");
else
   printf("es mayor que diez");
```

Este fragmento de código imprimiría: es menor que diez. En el siguiente capítulo se verá con más detenimiento el operador if-else.

Operadores lógicos

Como operadores lógicos designamos a aquellos operadores que nos permiten "conectar" un par de propiedades (al igual que en lógica):

Los operadores lógicos de los que disponemos en C son los siguientes:

Tabla: Operadores lógicos.

Operador	Acción
&&	Conjunción (Y)
||	Disyunción (O)
!	Negación

Operadores a nivel de bit (opcional)

Como operadores a nivel de bit entendemos aquellos que toman los operandos como un conjunto de bits y operan con esos bits de forma individual.

Tabla 3.5: Operadores a nivel de bit

Operador	Acción
&	AND a nivel de bit.
|	OR a nivel de bit.
^	XOR a nivel de bit.
~	Complemento.
<<	Desplazamiento a la izquierda.
>>	Desplazamiento a la derecha.

A continuación describiremos cada uno de estos operadores brevemente.

El **operador AND** (&): El operador AND compara dos bits; si los dos son 1 el resultado es 1, en otro caso el resultado será 0. Ejemplo:

```
c1 = 0x45      --> 01000101
c2 = 0x71      --> 01110001
---------------------------
c1 & c2 = 0x41 --> 01000001
```

El **operador OR (|)**: El operador OR compara dos bits; si cualquiera de los dos bits es 1, entonces el resultado es 1; en otro caso será 0. Ejemplo:

```
i1 = 0x47      --> 01000111
i2 = 0x53      --> 01010011
---------------------------
i1 | i2 = 0x57 --> 01010111
```

El **operador XOR (^)**: El operador OR exclusivo o XOR, dará como resultado un 1 si cualquiera de los dos operandos es 1, pero no los dos a la vez. Ejemplo:

```
i1 = 0x47      --> 01000111
i2 = 0x53      --> 01010011
---------------------------
i1 ^ i2 = 0x14 --> 00010100
```

El **operador de complemento (~)**: Este operador devuelve como resultado el complemento a uno del operando:

```
c = 0x45  --> 01000101
----------------------
~c = 0xBA --> 10111010
```

Los **operadores de desplazamiento a nivel de bit (<< y >>)**: Desplazan a la izquierda o a la derecha un número especificado de bits. En un desplazamiento a la izquierda los bits que sobran por el lado izquierdo se descartan y se rellenan los nuevos espacios con ceros. De manera análoga pasa con los desplazamientos a la derecha.

Ejercicios:

Ejercicio 1: Cálculo de precios con descuento

Escribe un programa que pregunte el precio, el tanto por ciento de descuento, y te diga el precio con descuento. Por ejemplo, si el precio que introduce el usuario es 300 y el descuento 20, el programa dirá que el precio final con descuento es de 240.

Ejercicio 2: Cálculo de área y perímetro

Escribe un programa que pregunte al usuario los dos lados de un rectángulo y presente por pantalla el cálculo del perímetro (suma de los lados) y el área (base por altura).

Ejercicio 3: Cambio de dólares a euros.

Suponiendo que 1 euro = 1.33250 dólares. Escribe un programa que pida al usuario un número de dólares y calcule el cambio en euros.

Ejercicio 4: Cálculo de perímetro de circunferencia, área del círculo, y volumen de la esfera.

Suponiendo que pi = 3.1416. Escribe un programa que pida al usuario que introduzca el radio, y presente por pantalla el cálculo del perímetro de la circunferencia (2*pi*r), el área del círculo ($pi*r^2$), y el volumen de la esfera ($V = 4*pi*r^3/3$).

Ejercicio 5: Pasar de días, horas y minutos a segundos.

Escribe un programa que pida al usuario los siguientes datos: días, horas y minutos. Y le conteste con la cantidad de segundos totales que son esos datos.

Ejercicio 6: Pasar de segundos a días, horas y minutos.

Escribe un programa que pida al usuario que introduzca los segundos, y le conteste diciéndole el número de días, horas, minutos y segundos que son.

Ejercicio 7: Solución de la ecuación de segundo grado.

Escribir un programa que pida por teclado los tres coeficientes (a, b y c) de la ecuación $ax^2+bx+c=0$ y calcule las dos soluciones suponiendo que ambas serán reales (es decir que la raíz queda positiva). Nota: $x_{1,2}=(a\pm sqrt(b^2-4ac))/2$, sqrt es una función que devuelve la raíz cuadrada, para poder invocarla es necesario poner en la cabecera del programa: #include <math.h>

Ejercicio 8: Cálculo de la resistencia equivalente.

Escribir un programa que pida por teclado dos resistencias y calcule y presente la resistencia equivalente en paralelo (Req=(R1*R2)/(R1+R2)).

Ejercicio 9: Media de cuatro números.

Escribir un programa que pida por teclado cuatro números y calcule y presente la media de los cuatro.

Ejercicio 10: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int x = 2, y = 6, z = 4;
y = y+4*z;
y +=x;
printf("%d",y);
```

Ejercicio 11: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int x = 2, y = 6, z = 4;
if(x>y || x<z)
                printf("verdadero");
            else
                printf("falso");
```

Ejercicio 12: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int x = 2, y = 6;
if(x<y && x==y)
                printf("verdadero");
            else
                printf("falso");
```

Ejercicio 13: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int x = 2, y = 6;
if( (x<y && x!=y) || !(x==y) )
            printf("verdadero");
        else
            printf("falso");
```


CONTROL DE FLUJO

Resumen-Teoría:

Para el control del programa existen las sentencias de control siguientes: if-else, else-if, switch, while, for y do-while. Estas sentencias sirven para hacer que una varias sentencias se ejecuten repetidamente, así como para decidir si una parte del programa debe o no ejecutarse.

Las llaves '{', '}' se emplean para encapsular sentencias. Todas las sentencias encerradas entre llaves son como una única sentencia.

if-else

La proposición if-else se usa para expresar decisiones. La sintáxis es:

```
if(expresión)
   proposición1;
else
   proposición2;
```

Si la expresión es verdadera, se ejecuta la proposición1, de lo contrario se realiza la proposición2.

No es indispensable el else, es decir, se puede tener una proposición de la forma:

```
if (expresión)
   proposición1;
```

Se debe hacer notar que la proposición1 o la proposición2, pueden ser proposiciones compuestas o bloques. Por ejemplo:

```
if( x >= 5 )
   {
```

```
   total = precio_unidad * x;
   printf("El precio de %d unidades es : %d", x, total);
   }
else
   {
   printf("Pedidos inferiores a 5 unidades no son posibles.");
   }
```


Expresión condicional (__)? __: __;

Otra forma de escribir la proposición if-else es usando una expresión condicional.

```
expresión1 ? expresión2 : expresión 3
```

Que es similar a:

```
if( expresión1 )
   expresión2;
else
   expresión3;
```

Con la diferencia de que, además de permitir poner condiciones en la ejecución de código, como hace el "if", en este caso es la expresión entera la que toma el valor devuelto por expresion2 (si expresión1 es verdadera), o por expresión 3 (si expresión1 es falsa). Por ejemplo:

```
Precio_unitario = (unidades > 20) ? 26:28 ;
```

La variable precio unitario valdrá 26 si unidades es mayor que 20, y 28 en caso contrario.

else-if

Con esta expresión se complementa la expresión vista en el punto anterior. Su sintáxis es:

```
   if( expresión )
proposición;
   else if( expresión )
      proposición;
   else if( expresión )
      proposición;
   else if( expresión )
      proposición;
   else
      proposición;
```

Mediante esta expresión se puede seleccionar una condición muy específica dentro de un programa, es decir, que para llegar a ella se haya tenido la necesidad del cumplimiento de otras condiciones.

switch

La proposición switch, permite la decisión múltiple que prueba si una expresión coincide con uno de los valores constantes enteros que se hayan definido previamente.

Su sintáxis es:

```
switch( expresión ) {
   case exp-const: proposiciones
   break;

case exp-const: proposiciones
   break;

   case exp-const:
   case exp-const: proposiciones
   break;

   default: proposiciones
   }
```

Se compara la "expresión" con cada una de las opciones "exp-const", y en el momento de encontrar una constate idéntica se ejecutan las proposiciones correspondientes a ese caso. Al terminar de realizar las proposiciones del caso, se debe usar la palabra reservada "break" para que vaya al final del switch.

Si ninguno de los casos cumplen con la expresión, se puede definir un caso por omisión, que también puede tener proposiciones.

En todos los casos pueden ser proposiciones simples o compuestas. En las compuestas se usan llaves para definir el bloque.

while

La proposición "while" permite la ejecución de una proposición simple o compuesta, mientras la "expresión" sea verdadera. Su sintáxis es:

```
while( expresión )
   proposición
```

Por ejemplo,

```
while( (c = getchar())  == 's' || c == 'S' )
   printf("\nDesea salir del programa?");

while( (c = getchar())  == 's' || c == 'S' )
   {
   printf("\nDesea salir del programa?");
   ++i;
```

```
printf("\nNúmero de veces que se ha negado a salir: %u",i);
}
```

for

La proposición "for" requiere tres expresiones como argumento. Las expresión1 y la expresión3 comúnmente se emplean para asignaciones, mientras que la expresión2 es la condición que se debe cumplir para que el ciclo "for" se siga ejecutando.

La sintáxis es:

```
for(expresión1; expresión2; expresión3)
   proposición
```

Ejemplo:

```
for(i=0; i <= 500; ++i)
   printf("\nEl número par  # %d, es el doble de: %d", (2*i), i);
```

do-while

La proposición "do-while" es similar a la proposición "while", se ejecuta el ciclo mientras se cumpla la condición dada en "expresión".

La diferencia estriba en que en el "do-while" siempre se evalúa al menos una vez su "proposición", mientras que en el "while" si no se cumple la "expresión" no entra al ciclo.

Sintáxis:

```
do
   proposición
while( expresión );
```

Ejemplo:

```
i=0;
do
   {
   printf("\nEl número par  # %d, es el doble de: %d", (2*i), i);
   ++i;
   }
while( i < 500 );
```

break y continue

Cuando se quiere abandonar un ciclo en forma prematura debido a que ciertas condiciones ya se cumplieron, se puede usar la proposición "break". Ésta sirve par las proposiciones "for", "while" y "do-while".

También se tiene otra proposición relacionada, y esta es el "continue"; su función es la de ocasionar la próxima iteración del ciclo.

Ejemplos:

```
for(h=1; h <= 1000; ++h)
   {
   if( h%5 == 0 )
         continue;
   printf("%i no es múltiplo de 5.\n", h);
   }
```

```
while(1)
   {
   if( getchar() == '$' )
         break;
   printf("\nNo puedo parar, hasta que presione \'$\'");
   }
```

Ejercicios:

Ejercicio 1: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int i;

for(i=0; i<4; i++){
   printf(“>>> %d: %d\n”,i,i*i*2);
   }
```

Ejercicio 2: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int i=4, x=5;

for(i=0; i<4; i++){
   printf(“>>> %d: %d\n”,i,i*x);
   }
```

Ejercicio 3: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int i=4, x=5;

for(i=x; i<10; i++){
   printf("%d, ",i);
   }
```

Ejercicio 4: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int i=4, x=5;

if(x<(2*i))
   printf("verdadero");
else
   printf("falso");
```


Ejercicio 5: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int i=4, x=5;

for(i=0; i<10; i++){
   if(i<x) printf("%d ",i);
   else printf("%d ",i-x);
   }
```

Ejercicio 6: Bucles 1.

Realizar un programa que imprima los números del 1 al 57. Repetir este ejercicio con todos los tipos de bucles (for, while, y do-while).

Ejercicio 7: Bucles 2.

Realizar un programa que pida al usuario un número y presente los números del 1 al número que introdujo el usuario. Repetir este ejercicio con todos los tipos de bucles (for, while, y do-while).

Ejercicio 8: Bucles 3.

Realizar un programa que pida al usuario dos números y presente los números del primero número al segundo que introdujo el usuario. Repetir este ejercicio con todos los tipos de bucles (for, while, y do-while).

Ejercicio 9: Bucles4.

Realizar un programa que imprima por pantalla tántos asteriscos como diga el usuario. Al ejecutarse debe preguntar "Cuantos asteriscos desea imprimir?", leer el número que introduce el usuario e imprimir los asteriscos.

Ejercicio 9: Condición if 1.

Realizar un programa que pida al usuario dos números y diga cuál es el mayor y cuál el menor.

Ejercicio 10: Condición if 2.

Realizar un programa que pida tres números y diga cuáles son pares y cuáles impares.

Ejercicio 11: Condición if 3.

Realizar un programa que pregunte al usuario el momento del día con una letra (m-mañana, t-tarde, n-noche), el sexo con otra letra (m-masculino, f-femenino). El programa dirá: buenos días, tardes, o noches (según el momento) señor o señora según el sexo.

Ejercicio 12: Condición if 4.

Realizar un programa que pida tres números y diga cuál es el mayor, cuál es el segundo mayor, y cuál es el menor.

Ejercicio 13: Bucles con if 1.

Realizar un programa que pida al usuario dos números y presente los números impares que hay desde el primer número al segundo que introdujo el usuario.

Ejercicio 14: Bucles con if 2.

Realizar un programa que pida al usuario dos números y una letra: "i" ó "p".
El programa presentará los números pares (si se pulsó la "p") ó impares (si se pulsó la "i") que hay desde el primer número al segundo que introdujo el usuario. Si se pulsa alguna tecla distinta de "p" ó "i", el programa no imprime ningún número.

Ejercicio 15: Bucles con if 3.

Realizar un programa que pida que se pulse la letra "C" si se pulsa cualquier otra tecla que no sea la "C", dice "letra incorrecta" y vulva a pedir que se pulse la letra "C". Cuando se pulsa la tecla "C" el programa dice "gracias" y termina.

Ejercicio 16: Bucles con if 4.

Realizar un programa que pida que se dos números consecutivos (3 y 4; 9 y 10 etc.). Cuando se introducen tres números consecutivos dice "gracias" y termina. Mientras no se introduzcan tres números consecutivos el programa sigue pidiendo números indefinidamente.

Ejercicio 17: Bucles con if 5.

Realizar un programa que pida que se tres números consecutivos (3, 4 y 5; 9, 10 y 11, etc.). Cuando se introducen tres números consecutivos dice "gracias" y termina. Mientras no se introduzcan tres números consecutivos el programa sigue pidiendo números indefinidamente.

Ejercicio 18: Bucles con if 6. Ecuación de 2º grado.

Realizar un programa que pida los tres coeficientes de una ecuación de 2º grado y calcule las dos soluciones aunque estas sean números imaginarios. Después de calcular las soluciones, el programa dará la opción de seguir resolviendo ecuaciones: Continuar (pulse C)? / Salir (pulse S)?.

Ejercicio 19: Bucles con if7. Ecuación de 2º grado.

Realizar un programa que pida dos números y presente por pantalla la ecuación de segundo grado que tiene por soluciones estos dos números. Ejemplo: 5 y -3, la ecuación sería $(x-5)\ (x+3) = x^2-2x-15$, los coeficientes son 1, -2 y -15. El programa permitirá repetir esta operación tantas veces como el usuario quiera, introduciendo números diferentes, hasta que decida terminar el programa (por ejemplo pulsando 'S').

Ejercicio 20: Factorial.

Realizar un programa que pida un número y calcule su factorial.

Ejercicio 21: Potencias.

Realizar un programa que pida la base y el exponente y calcule la potencia.

Ejercicio 22: Divisores de un número.

Realizar un programa que pida un número y diga todos sus divisores.

Ejercicio 23: Divisores comunes de dos números.

Realizar un programa que pida dos números y diga sus divisores comunes.

Ejercicio 24: Máximo común divisor.

Realizar un programa que pida dos números y diga su máximo común divisor.

Ejercicio 25: Simplificar fracciones.

Realizar un programa que pida el numerador y denominador de una fracción y devuelva la fracción simplificada.

ARRAYS

Resumen-Teoría:

Un array es un conjunto ordenado de datos del mismo tipo. Si queremos un programa que almacene y haga operaciones con un conjunto de mil números (por ejemplo las alturas de mil personas, etc.), no creamos mil variables distintas sino que creamos un array de mil elementos.

Los arrays también se llaman: arreglos, vectores, matrices de una dimensión... y más incorrectamente listas y tablas.

Declaración e inicialización

```
int edades[200];
```
(declaración de un array de 200 números enteros)

```
float alturas[40];
```
(declaración de un array de 40 números decimales)

```
int mis_tres_numeros[3]={3,5,7};
```
(declaración de un array de 3 números enteros inicializados con los valores 3, 5 y 7)

Acceso a una posición concreta

```
mis_tres_mumeros[0] = 9; // escribe 9 en la primera posición del array
mis_tres_mumeros[1] = 10; // escribe 10 en la segunda posición del array
mis_tres_mumeros[2] = 11; // escribe 9 en la tercera y última posición del array.
```

Bucles para recorrer arrays

```
for(i=0; i<N;i++) scanf("%d", &miarray[i]);
```

Lee del teclado N números y los asigna al array "miarray".

```
for(i=0; i<3;i++) printf("%d", mis_tres_numeros[i]);
```

Presenta por pantalla un array de tres números.

Cadenas de caracteres. Un array especial

Una cadena de caracteres es un array de caracteres que contiene el '\0' (el carácter nulo, o carácter fin de cadena). Como se verá posteriormente, existen numerosas funciones preparadas para trabajar con cadenas de caracteres, esto es, funciones que procesan un array de caracteres, hasta que se encuentran con el carácter nulo o carácter fin de cadena. Algunos ejemplos son: "puts(cadena)" (imprime cadenas), "gets(cadena)" lee cadena del teclado, "strcat(cadena1, cadena2)" (concatena cadenas), "strcpy" (copia cadenas), "strcmp" (compara cadenas), etc.

Ejemplos:

```
char micadena1[10]={'h','o','l','a','\0'};
char micadena2[50]="que tal";

puts(micadena1); //imprime : hola
puts(micadena2); //imprime : que tal

puts("Introduzca su nombre por favor:");
gets(micadena1);

srcat(micadena2,"estás? ");
srcat(micadena2, micadena1);

puts(micadena2);
```


Matrices, arrays multidimensionales.

En ocasiones es interesante tener los datos de un array ordenados en distintas dimensiones, para esto existen los arrays multidimensionales o matrices. Una matríz es básicamente un array de arrays.

`int matriz1[5][3];//`matriz de cinco filas y tres columnas de números enteros.

`int arraydematrices[7][5][3];//`array de 7 matrices de las anteriores, que también puede pensarse como una matriz de tres dimensiones.

Ejercicios:

Ejercicio 1: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int array[10]={1,3,5,7,9,2,3,4,6,8,10},i;

for(i=0; i<10; i++){
```

```
   printf(“>>> %d\n”, array[i]);
   }
```

Ejercicio 2: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int array[10]={1,3,5,7,9,2,3,4,6,8,10},i;

for(i=1; i<10; i++){
   printf(“>>> %d\n”, array[i]);
   }
```

Ejercicio 3: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int array[10],i;

for(i=0; i<10; i++){
   array[i]=i*i;
   }
for(i=0; i<10; i++){
   printf(“%d”,array[i]);
   }
```

Ejercicio 4: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int array[10],i=0;

while(i<10){

   array[i]=i*i;
   i++;

   }
do{
printf(“%d”,array[--i]);
} while(i>=0);
```

Ejercicio 5: Media, mayor y menor de un conjunto fijo.

Realizar un programa que pida las notas de 40 alumnos por pantalla y muestre un menú de opciones: 1. Listar notas, 2.Calcular la media, 3.Calcular el menor, 4.Calcular el mayor.

Ejercicio 6: Media, mayor y menor de un conjunto prefijado.

Igual que el apartado anterior pero en vez de 20 alumnos ahora el número de alumnos se le preguntará al usuario al iniciar el programa, este número no podrá superar los 100 alumnos (controlar que el usuario introduzca un número menor que 100).

Ejercicio 7: Media, mayor y menor. De un conjunto variable.

Igual que el apartado anterior pero ahora el número de alumnos empieza desde cero y se añade al menú la **opción 5.Agregar nota**, con la que el usuario puede ir agregando las notas de una en una.

Ejercicio 8: Arrays multidimensionales. Edificio1.

Se quiere controlar el número de habitantes de un eidificio con 6 pisos y 4 puertas (A, B, C, y D) en cada piso.
Realizar un programa que pida al usuario que introduzca el número de habitantes de cada puerta del edificio. El programa debe decir la vivienda (piso y puerta) que más habitantes tiene del edificio.

Ejercicio 9: Arrays multidimensionales. Edificio2.

Se quiere controlar el número de habitantes de un eidificio con 6 pisos y 4 puertas (A, B, C, y D) en cada piso.
Realizar un programa que pida al usuario que introduzca el número de habitantes de cada puerta del edificio. El programa debe decir el piso que más habitantes tiene de todo el edificio.

Ejercicio 10: Arrays multidimensionales. Edificio3.

Se quiere controlar el número de habitantes de un eidificio con 6 pisos y 4 puertas (A, B, C, y D) en cada piso.
Realizar un programa que pida al usuario que introduzca el número de habitantes de cada puerta del edificio. El programa debe decir la puerta que más habitantes tiene de todo el edificio.

Ejercicio 11: Arrays multidimensionales. Edificio3.

Se quiere controlar el número de habitantes de un eidificio con 6 pisos y 4 puertas (A, B, C, y D) en cada piso.
Realizar un programa que pida al usuario que introduzca el número de habitantes de cada puerta del edificio. El programa debe mostrar la media de habitantes de cada piso.

Ejercicio 12: Arrays multidimensionales. Rectas en el plano.

Se quiere representar rectas en un plano de números enteros. El plano será un array de 100x100 caracteres. El programa pedirá que se introduzcan los tres coeficientes (números enteros) de una recta de la forma ax+by+c=0, representará los puntos de la recta como asteriscos en la matriz, y presentará por pantalla dicha matriz.

FUNCIONES

Resumen-Teoría:

Los programas sencillos, como los ejemplo planteados hasta ahora, normalmente no necesitan un nivel de estructuración elevado, es decir, pueden escribirse todo seguido como una lista de órdenes sencilla (a pesar de los bucles). Pero cuando los programas crecen un poco, es necesario estructurarlos adecuadamente para mantenerlos legibles, facilitar su mantenimiento y para poder reutilizar ciertas porciones de código. El mecanismo de C que nos permite esto son las funciones. Con los compiladores, los fabricantes nos proporcionan un conjunto importante de funciones de librería.

Declaración, definición y llamada

En la **declaración** declaramos una función, es decir, le decimos al compilador qué tipo de valor devuelve la función, cuántos datos hay que proporcionarle y de qué tipos. Por ejemplo:

```
float funcionprueba (int exponente, float base);
```

En este ejemplo, la function “funcionprueba” devuelve un valor float, y recibe un int y un float.
Cuando una función no devuelve ningún valor hay que poner “void” como tipo retornado. Del mismo modo, si no recibe ningún parámetro, hay que poner también “void” entre los paréntesis:
void funcion (void); // declaración de una función que no recibe ni devuelve nada.

En la **definición** se establece qué hace una función, es decir, se define la función. Por ejemplo:

```
float funcionprueba (int exponente, float base {
int i;
float resultado;

if(exponente=0) {
   resultado=1;
   }
else{
   resultado = base;
   for(i=0;i++;i<exponente)
         {
         resultado = base * resultado;
         }//fin de for
   }//fin de else
return resultado;
}
```

En este ejemplo, la function “funcionprueba” multiplica por sí misma la base tantas veces como indica el exponente, es decir, calcula la potencia.

La definición de la función puede estar antes o después que el main. Cuando se pone antes que el main hace las veces de declaración.

Ámbito de las variables

Al igual que al comienzo del main, en las funciones también se pueden declarar variables (como el caso de "i" y de "resultado" del ejemplo anterior), estas variables se pueden usar únicamente dentro de la función en la que están declaradas. A esto se le llama el ámbito de una función. En C los ámbitos se crear abriendo llave "{" y se cierran cerrando llave "}". Al cerrar un ámbito se destruyen todas las variables en él declaradas.
Si se declara una variable fuera de todos los ámbitos (fuera del main), se dice que es una "variable global", ya que se podrá utilizar dentro de cualquier función.
Los parámetros que recibe una función son variables locales de dicha función que se inicializan en la llamada a la función y, al igual que todas las variables locales, se destruyen al terminar la función. La única excepción a esta regla son los arrays: cuando el argumento que se pasa a la función es un array, este dato no es una variable local, sino que sigue perteneciendo al entorno desde el que se hizo la llamada y los valores modificados se conservarán al terminar la función. Veamos un ejemplo:

```
void datos(int array[])
{
x[0]=7; x[1]=8; x[2]=9;
}

void main(void){
int x[3]={1,2,3};
for(i=0;i<3;i++)printf("%d,",x[i]);// qué imprimirá??
}
```

Imprimirá: 7,8,9.

Ejercicios:

Ejercicio 1: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int mi_funcion(void)
{
return 3+2;
}

void main(void){
printf("La function devuelde %d",mi_funcion());
}
```

Ejercicio 2: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int mi_funcion(int x)
{
return x*x;
```

```
}

void main(void){
printf("La function devuelde %d",mi_funcion(5));
}
```

Ejercicio 3: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int mi_funcion(int x)
{
int y;
y=2+x*3;
return y;
}

void main(void){
printf("La function devuelde %d",mi_funcion(5+2));
}
```


Ejercicio 4: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int mi_funcion(int x)
{
return x*x;
}

void main(void){
int x=3;
mi_funcion(x);
printf("La function devuelde %d", mi_funcion(x));
printf("La variable vale %d", x);
}
```

Ejercicio 5: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int mi_funcion(int x)
{
x=x*5;
return x;
}

void main(void){
int x=3;
mi_funcion(x);
printf("La function devuelde %d", mi_funcion(x));
printf("La variable vale %d", x);
}
```

Ejercicio 6: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int mi_funcion(int x)
{
x=x*5;
return x;
}

void main(void){
int x=3;
x=mi_funcion(x);
printf("La function devuelde %d", mi_funcion(x));
printf("La variable vale %d", x);
}
```


Ejercicio 7: ¿Qué imprime?.

¿Qué imprime el siguiente fragmento de código? Compruébalo.

```
int mi_funcion(int x)
{
x=x*5;
return x;
}

void main(void){
int y=3, x=4;
printf("La function devuelde %d", mi_funcion(y));
printf("La variable vale %d", x);
}
```

Ejercicio 8: Función cuadrado.

Escribir el código de la función que devuelve el cuadrado de un número que recibe como argumento. Comprobar que funciona utilizando un programa de prueba como el siguiente:

```
int cuadrado(int x)
{
/*código a escribir*/
}

void main(void){
int y=4;
printf("Introduzca un numero:");
scanf("%d",&x);
printf("Su factorial es %d", cuadrado(x));
}
```

Ejercicio 9: Función factorial.

Escribir el código de la función que devuelve el factorial de un número que recibe como argumento. Comprobar que funciona utilizando un programa de prueba como el siguiente:

```
int factorial(int x)
{
/*código a escribir*/
}

void main(void){
int y=4;
printf("Introduzca un numero:");
scanf("%d",&x);
printf("Su factorial es %d", factorial(x));
}
```

Ejercicio 10: Función elmayor.

Escribir el código de la función que devuelve el factorial de un número que recibe como argumento. Comprobar que funciona utilizando un programa de prueba como el siguiente:

```
int elmayor(int x, int y)
{
/*código a escribir*/
}

void main(void){
int y,x;
printf("Introduzca un numero:");
scanf("%d",&x);
printf("Introduzca otro numero:");
scanf("%d",&y);
printf("El mayor es %d", elmayor(x,y));
}
```

Ejercicio 11: Función escribe_asteriscos.

Escribir el código de la función que imprima por pantalla tantos asteriscos como indique el número que recibe como argumento. Comprobar que funciona utilizando un programa de prueba como el siguiente:

```
void escribe_asteriscos(int x)
{
/*código a escribir*/
}

void main(void){
int y,x;
printf("Introduzca un numero:");
scanf("%d",&x);
escribe_asteriscos (x);
}
```

Ejercicio 12: Función divisores.

Escribir el código de la función que imprima por pantalla todos los divisores del número que recibe como argumento. Comprobar que funciona utilizando un programa de prueba como el siguiente:

```
void divisores(int x)
{
/*código a escribir*/
}

void main(void){
int x;
printf("Introduzca un numero:");
scanf("%d",&x);
divisores(x);
}
```


Ejercicio 13: Función divisores comunes.

Escribir el código de la función que imprima por pantalla todos los divisores comunes de dos números que recibe como argumento.

Ejercicio 14: Función máximo divisor comunes.

Escribir el código de la función que **retorne** el máximo común divisor de dos números que recibe como argumento.

PUNTEROS

Resumen-Teoría:

Concepto de puntero

Un puntero es una variable que contiene la dirección de otra variable. Por ejemplo, un puntero a entero es una variable que se utiliza para guardar la dirección de una variable tipo int.

Declaración:

int *punt; // declaración de un puntero a entero llamado "punt"
char *punt1; // declaración de un puntero a char llamado "punt1"
float *punt2; // declaración de un puntero a float llamado "punt2"
...

Estas declaraciones reservan espacio para un puntero, pero no inicializan el puntero, es decir, están apuntando a cualquier parte.

Operadores implicados:

Operador de indirección: '*' Situado a la izquierda de una variable (puntero) devuelve la variable a la que apunta, o lo que es lo mismo, el contenido de la dirección que contiene.

Operador de dirección-de: '&' Situado a la izquierda de una variable devuelve su dirección.

Ejemplos:

```
int *punt; //puntero a entero sin inicializar llamado punt
int x; // variable entera llamada x
int y; // variable entera llamada y
punt=&x; // Escribimos en punt la dirección de x, es decir, punt "apunta" a x.
*punt=4; //Escribimos un 4 en donde apunta punt, es decir, escribimos un 4 en x.
punt=&y;// Escribimos en punt la dirección de y, es decir, punt "apunta" a y.
*punt=8; //Escribimos un 8 en donde apunta punt, es decir, escribimos un 8 en y.

printf("%d, %d",x,y); // ¿qué imprime este printf?
```


La relación entre punteros y arrays

El nombre de un array es la dirección del primer elemento del array. Esto se expresa mediante las siguiente fórmula:

$$X[i] \equiv *(X + i)$$

Esta fórmula es válida tanto si X es un array como si X es un puntero al primer elemento del array. La única diferencia de un lado de la ecuación al otro es diferencia de notación, es decir, en C podemos usar para arrays la notación de array (en el lado de la izquierda de la ecuación) ó la notación puntero (en el lado de la derecha), de forma totalmente indistinta.

Ejemplo:

```
int miarray[7];
int *punt;
punt=&miarray[0]; // idem que punt=miarray;
*punt=5;//idem que punt[0]=5;
(*punt+2)=5;//idem que punt[2]=5;
```

Operaciones con punteros

Los elementos de un array están en posiciones contiguas de memoria, por este motivo, se permiten hacer operaciones con punteros, es decir, sumarle a un puntero un número, equivale a adelantarlo ese número de posiciones de memoria, lo mismo sucede con la resta. Veamos algunos ejemplos:

```
int miarray[7];
int *punt;
```

```
punt=miarray;
*(punt +1)=3; // idem que punt[1]=3 ;
punt=punt+3;//Ahora punt apunta al cuarto elemento del array.
*punt=5; // idem que miarray[3]=5;
miarray[6]=*(punt-2);// idem que miarray[6]= miarray[1];
```

Paso de parámetros por referencia a funciones

Hasta ahora las funciones solamente podían devolver un único valor, es decir, una función sólo podía modificar una única variable del ámbito desde el que se la llamaba (a la función). Los punteros nos permiten un nuevo uso de las funciones, podemos decirles dónde se encuentran nuestras variables (las variables del ámbito de la llamada) para que la función pueda modificarlas. A esto se le llama comúnmente "paso de parámetros por referencia", y consiste en dar como parámetro a una función, en vez de la variable, su dirección, es decir, un puntero a dicha variable.

Ejemplo:

```
void suma_dos(int *x, int *y, int *z)
{
*x=*x+2;
*y=*y+2;
*z=*z+2;
}

void main(void){
int x;
printf("Introduzca tres numeros:");
scanf("%d %d %d %d",&x, &y, &z);
suma_dos (&x, &y, &z);
printf("%d %d %d %d",x, y, z);// qué imprimirá??
}
```

Ejercicios:

Ejercicio 1: Punteros ¿Qué imprime?.

```
int *punt;
int x=7;
int y=5;
punt=&x;
*punt=4;
printf("%d, %d",x,y); // ¿qué imprime este printf?
```

Ejercicio 2: Punteros ¿Qué imprime?.

```
int *punt;
int x=7;
int y=5;
punt=&x;
x=4;
printf("%d, %d",*punt,y); // ¿qué imprime este printf?
```

Ejercicio 3: Punteros ¿Qué imprime?.

```
int *punt;
int x=7;
int y=5;
punt=&x;
x=4;
punt=&y;
printf("%d, %d",*punt,x); // ¿qué imprime este printf?
```

Ejercicio 4: Punteros ¿Qué imprime?.

```
int *punt;
int x=7;
int y=5;
punt=&x;
*punt=3;
punt=&y;
*punt=x;
x=9;
printf("%d, %d",*punt,x); // ¿qué imprime este printf?
```

Ejercicio 5: Punteros ¿Qué imprime?.

```
int *punta, *puntb;
int x=7;
int y=5;
punta=&x;
*punta=3;
puntb=&y;
*puntb=x;
x=9;
printf("%d, %d",*puntb,x); // ¿qué imprime este printf?
```

Ejercicio 6: Punteros ¿Qué imprime?.

```
int *punta, *puntb;
int x=7;
int y=5;
punta=&x;
*punta=3;
puntb=&y;
*puntb=x;
x=9;
printf("%d, %d",*puntb, *punta); // ¿qué imprime?
```

Ejercicio 7: Punteros ¿Qué imprime?.

```
int *punta, *puntb;
int x=7;
int y=5;
punta=&x;
*punta=3;
puntb=&y;
*puntb=x;
x=9;
puntb=punta;
printf("%d, %d",*puntb, y); // ¿qué imprime?
```

Ejercicio 8: Punteros y arrays ¿Qué imprime?.

```
int *punt,i;
int x[5]={1,2,3,4,5};
punt=x;
*punt=9;
for(i=0;i<5;i++)
printf("%d,",x[i]); // ¿qué imprime?
```

Ejercicio 9: Punteros y arrays ¿Qué imprime?.

```
int *punt,i;
int x[5]={1,2,3,4,5};
punt=&x[0];
*punt=9;
punt[3]=7;
for(i=0;i<5;i++)
printf("%d,",x[i]); // ¿qué imprime?
```

Ejercicio 10: Punteros y arrays ¿Qué imprime?.

```
int *punt,i;
int x[5]={1,2,3,4,5};
punt=x;
*x=11;
*(punt+3)=9 ;
for(i=0;i<5;i++)
printf("%d,",x[i]); // ¿qué imprime?
```

Ejercicio 11: Punteros y arrays ¿Qué imprime?.

```
int *punt,i;
int x[5]={1,2,3,4,5};
punt=x;
*(punt+2)=9;
*(x+3)=7 ;
punt[1]=11 ;
for(i=0;i<5;i++)
printf("%d,",*(punt+i)); // ¿qué imprime?
```

Ejercicio 12: Punteros y arrays ¿Qué imprime?.

```
int *punt,i;
int x[5]={1,2,3,4,5};
punt=x+4;
*(punt-2)=9;
punt--;
*(punt)=7 ;
punt[1]=11 ;
for(i=0;i<5;i++)
printf("%d,",*(x+i)); // ¿qué imprime?
```

Ejercicio 13: Punteros y arrays ¿Qué imprime?.

```
int *punt,i;
int x[5]={1,2,3,4,5};
punt=&x[0]+3;
*(punt-2)=9;
```

```
punt--;
*(punt)=7 ;
punt[1]=11 ;
punt=x;
for(i=0;i<5;i++)
printf("%d,",punt[i])); // ¿qué imprime?
```

Ejercicio 14: Punteros y funciones ¿Qué imprime?.

```
void suma_dos(int *x, int *y, int *z)
{
*x=*x+2;
*y=*y+2;
*z=*z+2;
}

void main(void){
int x,y,z;
x=3;
y=10;
z=15;
suma_dos (&x, &y, &z);
printf("%d %d %d %d",x, y, z);// qué imprimirá??
}
```

Ejercicio 15: Punteros y funciones ¿Qué imprime?.

```
void datos(int *x, float *y, char *c)
{
*x=8;
*y=4.2;
*c='g';
}

void main(void){
int x=9;
float y=44.6;
char c='a';
datos (&x, &y, &c);
printf("%d %f %c",x, y, c);// qué imprimirá??
}
```

Ejercicio 16: Punteros y funciones ¿Qué imprime?.

```
void datos(int *x, float *y, char *c)
{
printf("%d %f %c",x, y, c);
*x=8;
*y=4.2;
*c='g';
}

void main(void){
int x=9;
float y=44.6;
char c='a';
datos (&x, &y, &c);
printf("%d %d %f %c",x, y, c);// qué imprimirá??
}
```

Ejercicio 17: Punteros y funciones ¿Qué imprime?.

```
void datos(int x, float y, char c)
{
printf("%d %f %c",x, y, c);
x=8;
y=4.2;
c='g';
}

void main(void){
int x=9;
float y=44.6;
char c='a';
datos (x, y, c);
printf("%d %d %f %c",x, y, c);// qué imprimirá??
}
```

Ejercicio 18: Punteros y funciones ¿Qué imprime?.

```
int datos(int x, float y, char c)
{
printf("%d %f %c",x, y, c);
x=8;
y=4.2;
c='g';
return x;
}

void main(void){
int x=9;
float y=44.6;
char c='a';
x=datos (x, y, c);
printf("%d %d %f %c",x, y, c);// qué imprimirá??
}
```

Ejercicio 19: Punteros y funciones ¿Qué imprime?.

```
char datos(int *x, float *y, char *c)
{
printf("%d %f %c",x, y, c);
*x=8;
*y=4.2;
*c='g';
return 'h' ;
}

void main(void){
int x=9;
float y=44.6;
char c='a';
c=datos (&x, &y, &c);
printf("%d %d %f %c",x, y, c);// qué imprimirá??
}
```

REGISTROS O ESTRUCTURAS

Resumen-Teoría:

Un registro o estructura es una variable que sirve para guardar un grupo de datos de distintos tipos. Los arrays sirven para guardar grupos de datos del mismo tipo, por ejemplo 5 números reales, pero si queremos guardar en una misma variable, un número real, un número entero, y un carácter, entonces tendremos que crear una estructura, es decir, un nuevo tipo de variable capaz de guardar estos tres datos.
Ejemplo: Estructura para guardar un número entero (edad), un float (altura) y una cadena de caracteres (nombre):
Definición de la estructura:

```
struct jugador{
char nombre[50]; //campo “nombre”, estructura “jugador”.
int edad; //campo “edad”, estructura “jugador”.
float altura; //campo “altura”, estructura “jugador”.
            };
```

Declaración de una variable de este tipo:

```
struct jugador mijugador;
```

Declaración de un array de este tipo:

```
struct jugador equipo[10];
```

Acceso a un campo de una estructura aislada: (operador “.”)

```
mijugador.edad=4; //ejemplo de acceso al campo edad
printf(“La altura del jugador es: %f”,mijugador.altura);
                    //ejemplo de acceso al campo altura
```

Acceso a un campo de una estructura situada en la posición N de un array de estructuras: (operador “.”)

```
equipo[N].edad=4;
printf(“La altura del jugador es: %f”,equipo[N].altura);
```

Acceso a un campo de una estructura apuntada por un puntero: (operador “->”)

```
struct jugador *punt_a_jugador;//declaración del puntero
punt_a_jugador=&mijugador; //apunta a una estructura
punt_a_jugador->edad=4; //ejemplo de acceso al campo edad
printf(“La altura del jugador es: %f”, punt_a_jugador->altura)
                          //ejemplo de acceso al campo altura
```

Ejercicios:

Ejercicio 1: Estructuras ¿Qué imprime?.

```
struct jugador{
char nombre[50]; //campo “nombre”, estructura “jugador”.
```

```
int edad; //campo “edad”, estructura “jugador”.
float altura; //campo “altura”, estructura “jugador”.
          };

void main(void){
struct jugador mijugador;
strcpy(mijugador.nombre,”Petrovick”);
mijugador.edad=45;
mijugador.altura=1.8;
printf(“Nombre: %s\n Altura:%f\n Edad:%d\n ” , mijugador.nombre,
mijugador.altura, mijugador.edad);
}
```

Ejercicio 2: Arrays de estructuras ¿Qué imprime?.

```
struct medidas{
int alto,ancho,largo;
          };

void main(void){
int i;
struct medidas cubiletes[5];
for(i=0;i<5;i++)
   {
   cubiletes[i].alto=4;
   cubilietes[i].ancho=2*i;
   cubilietes[i].largo=i+1;
   }
for(i=0;i<5;i++)
   {
   printf(“Medidas de cubilete nº%d: %d alto, %d, ancho, %d
largo”,cubiletes[i].alto,cubilietes[i].ancho,cubilietes[i].largo);
   }

}
```

Ejercicio 3: Estructuras.

Crear un programa que pida al usuario los nombres, edades, y alturas, de 10 jugadores. Posteriormente le presentará un menú que le permita: 1. Listar los nombres de los jugadores; 2. Listar las alturas de los jugadores; 3. Listar las edades de los jugadores.

Ejercicio 4: Estructuras.

Crear un programa que pida al usuario los nombres, edades, y alturas, de 10 jugadores. Posteriormente le presentará un menú que le permita: 1. Listar los nombres de los jugadores; 2. Listar las alturas de los jugadores; 3. Listar las edades de los jugadores; 4. Buscar un jugador por su nombre y presentar su altura y su edad; 5. Indicar la edad y el nombre del jugador más alto de la lista.

Ejercicio 5: Estructuras.

Crear una estructura que contenga las coordenadas de un punto del plano (dos coordenadas (x,y), números reales). Realizar un programa que pida las coordenadas de tres puntos del plano, y calcule el perímetro del triángulo que forman (Nota: la distancia entre dos puntos de coordenadas (a,b) y (c,d) se calcula por el teorema de pitágoras como sqrt((c-a)*(c-a) + (d-b)*(d-b)), para usar la funcion sqrt debe incluir la librería math.h.).

RESERVA DINÁMICA DE MEMORIA

Resumen-Teoría:

Todos los programas realizados hasta el momento presentan una misma limitación: Hay que reservar la memoria al escribir el programa, es decir, al declarar las variables y arrays. Con las herramientas vistas hasta ahora sería imposible realizar un programa que preguntara al usuario ¿de cuantos elementos quiere el array? Y reserve el espacio exacto para esos elementos.

Ejemplo:

```
int n,*mi_array;
printf("De cuantos elementos quiere el array?");
scanf("%d",&n);
int mi_array[n]; // ERROR!! no se puede hacer
int *mi_array=(int *)malloc(sizeof(int)*n); // OK, es lo correcto
```

La funcióon malloc reserve espacio para n enteros y devuelve un puntero a int que se asigna al puntero mi_array.

Una vez terminado de utilizar el espacio reservado debemos liberarlo utilizando la función free(); En el ejemplo anterior haríamos:

```
free(mi_array);

Por último, es también importante mencionar la función realloc()
encargada de ampliar y reubicar cuando sea necesario el array.
Ejemplo:
numPtr = (int *)realloc( numPtr, 256 );
//El nuevo tamaño del array apuntato por numPtr es 256
```

Ejercicios:

Ejercicio 1: Reserva Dinámica (malloc, free).

Crear un programa que pregunte al usuario cuántos caracteres desea introducir, reserve espacio para dicho número de caracteres, le permita al usuario introducirlos desde teclado, y por último se los muestre todos seguidos.

Ejercicio 2: Reserva Dinámica (malloc, free).

Crear un programa que pregunte al usuario cuántos caracteres desea introducir, reserve espacio para dicho número de caracteres, le permita al usuario introducirlos desde teclado, y por último se los muestre todos seguidos en el orden inverso al que fueron introducidos.

Ejercicio 3: Reserva Dinámica (malloc, free).

Repetir el ejercicio 3 pero en vez de caracteres con números reales.

Ejercicio 4: Reserva Dinámica (malloc, free).

Repetir el ejercicio 3 pero en vez de caracteres con números enteros.

Ejercicio 5: Reserva Dinámica (malloc, free).

Repetir el ejercicio 3 pero en vez de caracteres con cadenas de caracteres de hasta 50 caracteres (reservando 50 posiciones para cada cadena).

Ejercicio 6: Reserva Dinámica (malloc, free).

Repetir el ejercicio 3 pero en vez de caracteres con cadenas de caracteres de hasta 50 caracteres, reservando las posiciones estrictamente necesarias para cada cadena.

Ejercicio 7: Reserva Dinámica (realloc, free).

Crear un programa que permita al usuario introducir, de uno en uno, cuantos caracteres quiera. Es decir, después de introducir un carácter el programa preguntará: otro(Y/N)? y mientras el usuario elija “Y” permitirle que siga introduciendo nuevos caracteres de forma ilimitada.

Ejercicio 8: Reserva Dinámica (realloc, free).

Repetir el ejercicio 7 pero en vez de caracteres con números reales.

Ejercicio 9: Reserva Dinámica (realloc, free).

Repetir el ejercicio 7 pero en vez de caracteres con números enteros.

Ejercicio 10: Reserva Dinámica (realloc, free).

Repetir el ejercicio 7 pero en vez de caracteres con cadenas de caracteres de hasta 50 caracteres (reservando 50 posiciones para cada cadena).

Ejercicio 11: Reserva Dinámica (realloc, free).

Repetir el ejercicio 7 pero en vez de caracteres con cadenas de caracteres de hasta 50 caracteres, reservando las posiciones estrictamente necesarias para cada cadena.

FICHEROS (ACCEDER AL DISCO DURO)

Resumen-Teoría:

Todos los programas realizados hasta ahora presentan una gran limitación: La información que manejan y generan sólo está disponible mientras el programa esta ejecutándose. Es decir, que cada vez que se ejecuta el programa el usuario debe proporcionar toda la información, y al salir del programa dicha información, más toda la que haya generado el programa se pierde irremediablemente.
El concepto que necesitamos incorporar es el de acceso a ficheros, es decir, acceso al disco duro del ordenador donde se ejecuta el programa. Para acceder al disco duro tenemos que declarar un manejador de fichero y abrir el fichero que queramos asociándole dicho manejador:

```
FILE *mifich; //declaración de un manejador de fichero
mifich=fopen();//apertura del fichero prueba.txt asociándolo al
                                  //manejador (handler) “mifich”
```

El uso de ficheros se lleva a cabo siempre mediante funciones predefinidas en el fichero stdio. El manejador o handler del fichero es lo único que necesitan estas funciones para hacer operaciones con el fichero que queramos. A continuación se enumeran y explican brevemente las funciones más importantes para trabajar con ficheros.

Función fopen:

Sintaxis:
FILE *fopen(char *nombre, char *modo);
ésta función sirve para abrir y crear ficheros en disco. El valor de retorno es un puntero a una estructura FILE. Los parámetros de entrada son:
nombre: una cadena que contiene un nombre de fichero válido, esto depende del sistema operativo que estemos usando. El nombre puede incluir el camino completo.
modo: especifica en tipo de fichero que se abrirá o se creará y el tipo de datos que puede contener, de texto o binarios:
r: sólo lectura. El fichero debe existir.
w: se abre para escritura, se crea un fichero nuevo o se sobreescribe si ya existe.
a: añadir, se abre para escritura, el cursor se situa al final del fichero. Si el fichero no existe, se crea.
r+: lectura y escritura. El fichero debe existir.
w+: lectura y escritura, se crea un fichero nuevo o se sobreescribe si ya existe.
a+: añadir, lectura y escritura, el cursor se situa al final del fichero. Si el fichero no existe, se crea.
t: tipo texto, si no se especifica "t" ni "b", se asume por defecto que es "t"
b: tipo binario.

Función fclose:

Sintaxis:
int fclose(FILE *fichero);
Es importante cerrar los ficheros abiertos antes de abandonar la aplicación. Esta función sirve para eso. Cerrar un fichero almacena los datos que aún están en el buffer de memoria, y actualiza algunos datos de la cabecera del fichero que mantiene el sistema operativo. Además permite que otros programas puedan abrir el fichero para su uso. Muy a menudo, los ficheros no pueden ser compartidos por varios programas.
Un valor de retorno cero indica que el fichero ha sido correctamente cerrado, si ha habido algún error, el valor de retorno es la constante EOF. El parámetro es un puntero a la estructura FILE del fichero que queremos cerrar.

Función fgetc:

Sintaxis:
int fgetc(FILE *fichero);
Esta función lee un carácter desde un fichero.
El valor de retorno es el carácter leído como un unsigned char convertido a int. Si no hay ningún carácter disponible, el valor de retorno es EOF. El parámetro es un puntero a una estructura FILE del fichero del que se hará la lectura.

Función fputc:

Sintaxis:
int fputc(int caracter, FILE *fichero);
Esta función escribe un carácter a un fichero.
El valor de retorno es el carácter escrito, si la operación fue completada con éxito, en caso contrario será EOF. Los parámetros de entrada son el carácter a escribir, convertido a int y un puntero a una estructura FILE del fichero en el que se hará la escritura.

Función feof:

Sintaxis:
int feof(FILE *fichero);
Esta función sirve para comprobar si se ha alcanzado el final del fichero. Muy frecuentemente deberemos trabajar con todos los valores almacenados en un archivo de forma secuencial, la forma que suelen tener los bucles para leer todos los datos de un archivo es permanecer leyendo mientras no se detecte el fin de fichero. Esta función suele usarse como prueba para verificar si se ha alcanzado o no ese punto.
El valor de retorno es distinto de cero sólo si no se ha alcanzado el fin de fichero. El parámetro es un puntero a la estructura FILE del fichero que queremos verificar.

Función rewind:

Sintaxis:
void rewind(FILE *fichero)
Es una función heredada de los tiempos de las cintas magnéticas. Literalmente significa "rebobinar", y hace referencia a que para volver al principio de un archivo almacenado en cinta, había que rebobinarla. Eso es lo que hace ésta función, sitúa el cursor de lectura/escritura al principio del archivo.
El parámetro es un puntero a la estructura FILE del fichero que queremos rebobinar.
Ejemplos:

```
// ejemplo1.c: Muestra un fichero dos veces.
#include <stdio.h>

int main()
{
  FILE *fichero;

  fichero = fopen("ejemplo1.c", "r");
  while(!feof(fichero)) fputc(fgetc(fichero), stdout);
  rewind(fichero);
  while(!feof(fichero)) fputc(fgetc(fichero), stdout);
  fclose(fichero);
  getchar();
  return 0;
}
```

Función fgets:

Sintaxis:
char *fgets(char *cadena, int n, FILE *fichero);
Esta función está diseñada para leer cadenas de caracteres. Leerá hasta n-1 caracteres o hasta que lea un retorno de línea. En este último caso, el carácter de retorno de línea también es leído.
El parámetro n nos permite limitar la lectura para evitar derbordar el espacio disponible en la cadena.
El valor de retorno es un puntero a la cadena leída, si se leyó con éxito, y es NULL si se detecta el final del fichero o si hay un error. Los parámetros son: la cadena a leer, el número de caracteres máximo a leer y un puntero a una estructura FILE del fichero del que se leerá.

Función fputs:

Sintaxis:
int fputs(const char *cadena, FILE *stream);
La función fputs escribe una cadena en un fichero. No se añade el carácter de retorno de línea ni el carácter nulo final.

El valor de retorno es un número no negativo o EOF en caso de error. Los parámetros de entrada son la cadena a escribir y un puntero a la estructura FILE del fichero donde se realizará la escritura.

Función fread:

Sintaxis:
size_t fread(void *puntero, size_t tamaño, size_t nregistros, FILE *fichero);
Esta función está pensada para trabajar con registros de longitud constante. Es capaz de leer desde un fichero uno o varios registros de la misma longitud y a partir de una dirección de memoria determinada. El usuario es responsable de asegurarse de que hay espacio suficiente para contener la información leída.
El valor de retorno es el número de registros leídos, no el número de bytes. Los parámetros son: un puntero a la zona de memoria donde se almacenarán los datos leídos, el tamaño de cada registro, el número de registros a leer y un puntero a la estructura FILE del fichero del que se hará la lectura.

Función fwrite:

Sintaxis:
size_t fwrite(void *puntero, size_t tamaño, size_t nregistros, FILE *fichero);
Esta función también está pensada para trabajar con registros de longitud constante y forma pareja con fread. Es capaz de escribir hacia un fichero uno o varios registros de la misma longitud almacenados a partir de una dirección de memoria determinada.
El valor de retorno es el número de registros escritos, no el número de bytes. Los parámetros son: un puntero a la zona de memoria donde se almacenarán los datos leídos, el tamaño de cada registro, el número de registros a leer y un puntero a la estructura FILE del fichero del que se hará la lectura.
Ejemplo:

```
// copia.c: Copia de ficheros
// Uso: copia <fichero_origen> <fichero_destino>

#include <stdio.h>

int main(int argc, char **argv) {
   FILE *fe, *fs;
   unsigned char buffer[2048]; // Buffer de 2 Kbytes
   int bytesLeidos;

   if(argc != 3) {
      printf("Usar: copia <fichero_origen> <fichero_destino>\n");
      return 1;
   }

   // Abrir el fichero de entrada en lectura y binario
   fe = fopen(argv[1], "rb");
   if(!fe) {
```

```
    printf("El fichero %s no existe o no puede ser abierto.\n", argv[1]);
    return 1;
  }
  // Crear o sobreescribir el fichero de salida en binario
  fs = fopen(argv[2], "wb");
  if(!fs) {
    printf("El fichero %s no puede ser creado.\n", argv[2]);
    fclose(fe);
    return 1;
  }
  // Bucle de copia:
  while((bytesLeidos = fread(buffer, 1, 2048, fe)))
    fwrite(buffer, 1, bytesLeidos, fs);
  // Cerrar ficheros:
  fclose(fe);
  fclose(fs);
  return 0;
}
```


Función fprintf:

Sintaxis:
int fprintf(FILE *fichero, const char *formato, ...);
La función fprintf funciona igual que printf en cuanto a parámetros, pero la salida se dirige a un fichero en lugar de a la pantalla.
Función fscanf:
Sintaxis:
int fscanf(FILE *fichero, const char *formato, ...);
La función fscanf funciona igual que scanf en cuanto a parámetros, pero la entrada se toma de un fichero en lugar del teclado.

Función fflush:

Sintaxis:
int fflush(FILE *fichero);
Esta función fuerza la salida de los datos acumulados en el buffer de salida del fichero. Para mejorar las prestaciones del manejo de ficheros se utilizan buffers, almacenes temporales de datos en memoria, las operaciones de salida se hacen a través del buffer, y sólo cuando el buffer se llena se realiza la escritura en el disco y se vacía el buffer. En ocasiones nos hace falta vaciar ese buffer de un modo manual, para eso sirve ésta función.
El valor de retorno es cero si la función se ejecutó con éxito, y EOF si hubo algún error.
El parámetro de entrada es un puntero a la estructura FILE del fichero del que se quiere vaciar el buffer. Si es NULL se hará el vaciado de todos los ficheros abiertos.

Funciones C específicas para ficheros de acceso aleatorio

Función fseek:

Sintaxis:
int fseek(FILE *fichero, long int desplazamiento, int origen);
Esta función sirve para situar el cursor del fichero para leer o escribir en el lugar deseado.
El valor de retorno es cero si la función tuvo éxito, y un valor distinto de cero si hubo algún error.
Los parámetros de entrada son: un puntero a una estructura FILE del fichero en el que queremos cambiar el cursor de lectura/escritura, el valor del desplazamiento y el punto de origen desde el que se calculará el desplazamiento.
El parámetro origen puede tener tres posibles valores:
SEEK_SET el desplazamiento se cuenta desde el principio del fichero. El primer byte del fichero tiene un desplazamiento cero.
SEEK_CUR el desplazamiento se cuenta desde la posición actual del cursor.
SEEK_END el desplazamiento se cuenta desde el final del fichero.

Función ftell:

Sintaxis:
long int ftell(FILE *fichero);
La función ftell sirve para averiguar la posición actual del cursor de lectura/excritura de un fichero.
El valor de retorno será esa posición, o -1 si hay algún error.
El parámetro de entrada es un puntero a una estructura FILE del fichero del que queremos leer la posición del cursor de lectura/escritura

Ejercicios:

Ejercicio 1: Lector de ficheros de texto.

Crear un programa que abra un fichero llamado “prueba.txt” (previamente creado con el block de notas y guardado en la misma carpeta donde este el programa) y que muestre el contenido del mismo por pantalla carácter a carácter.

Ejercicio 2: Lector de ficheros de texto con guiones.

Crear un programa que abra un fichero llamado “prueba.txt” (previamente creado con el block de notas y guardado en la misma carpeta donde este el programa) y que muestre el contenido del mismo con los caracteres separados por guiones “-“.

Ejercicio 3: Lector de ficheros de texto a mayúsculas.

Crear un programa que abra un fichero llamado "prueba.txt" (previamente creado con el block de notas y guardado en la misma carpeta donde este el programa) y que muestre el contenido del mismo por pantalla pasando a mayúscula los caracteres que estén en minúscula.

Ejercicio 4: Editor de ficheros de texto.

Crear un programa que cree un fichero llamado salidatexto.txt y permita al usuario escribir en él todo el texto que desee. Dejará de introducir texto cuando introduzca un asterisco "*".

Ejercicio 5: Lector de ficheros con números de línea.

Crear un programa que abra un fichero llamado "prueba.txt" (previamente creado con el block de notas y guardado en la misma carpeta donde este el programa) y que muestre el contenido del mismo por pantalla por líneas y cada línea con su número de línea seguido de ">>".

Ejercicio 6: Lector de ficheros con números de línea.

Crear un programa que abra un fichero llamado "prueba.txt" (previamente creado con el block de notas y guardado en la misma carpeta donde este el programa) y que muestre el contenido del mismo por pantalla por líneas y cada línea con su número de línea.

Ejercicio 7: Copiar ficheros.

Crear un programa capaz de copiar un fichero (word por ejemplo, o cualquier otro tipo) en otro. El fichero creado se llamará "copia_de" y el nombre del archivo a copiar. El fichero a copiar lo puede elegir el usuario, y si no existe debe tener la opción de escribir otro nombre hasta que escriba el nombre de un fichero que exista.
Nota: *Para trabajar con ficheros es interesante configurar el explorador de windows para que* ***no*** *oculte las extensiones de los archivos para tipos de archivos conocidos. Esta opción suele encontrarse en Herramientas -> opciones de carpeta -> Ver. Las extensiones de los archivos son usadas por el sistema operativo para poder informar al usuario acerca del tipo de archivo de que se trata sin necesidad de conocer lo que hay dentro. Si creamos un archivo con la extensión .doc por ejemplo, lo etiquetará como un archivo de word, si le ponemos .pdf creerá que es un pdf e intentará abrirlo con acrobat cuando hagamos doble clic, etc. Para nuestras prácticas podemos inventar una extensión que esté sin utilizar, por ejemplo .mio, así reconoceremos nuestros ficheros y los diferenciaremos del resto.*

Ejercicio 8: Copiar ficheros cambiando caracteres.

Crear un programa igual que el de el ejercicio anterior pero esta vez el fichero de copia tiene que crearse con todas las letras en mayúsculas.

Ejercicio 9: Crear base de datos.

Crear un programa para gestionar una agenda de clientes, para una empresa, con los campos: “nombre (50 caracteres)”, “direccion (100 caraceres)”, “telefono (entero sin signo)”, “estado_pagos (float con signo)”. El estado de pagos será un número negativo indicando la cantidad de dinero que debe el cliente.
El programa debe presentar al usuario un menú con las siguientes opciones: 1) Agregar cliente. 2) ver datos de todos los clientes.
El programa trabajará con un archivo (clientes.mio por ejemplo) que debe crear la primera vez que se ejecute el programa.

Ejercicio 10: Continuación 1.

Añadir al programa anterior la opción de buscar clientes por teléfono. Esta opción pedirá al usuario el teléfono del cliente deseado y mostrará los demás datos de dicho cliente por pantalla. Si no lo encuentra, avisará de ello y volverá al menú principal.

Ejercicio 11: Continuación 2.

Añadir al programa anterior la opción de modificar los datos de un cliente y guardarlos de nuevo en el fichero.

Ejercicio 12: Continuación 3.

Añadir al programa anterior la opción de listar solamente los clientes con deudas (es decir, aquellos en los que “estado_pagos” es menor que cero).
.

Ejercicio 13: Continuación 4.

Añadir al programa anterior la opción de borrar un cliente de la agenda.

Ejercicio 15: Continuación 5.

Añadir al programa anterior la opción de trabajar con varias agendas. Al iniciar el programa el usuario podrá elegir el nombre del fichero que quiere cargar. En el menú del programa aparecerá la opción de cambiar de fichero de trabajo. Esta opción serviría si el usuario tiene varias empresas o si quiere guardar los datos de los clientes de distintas secciones en varias bases de datos, por ejemplo.

ESTRUCTURAS DINÁMICAS DE DATOS

Resumen-Teoría:

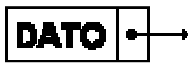
En el tema de memoria dinámica se vio cómo hacer crecer un array tanto como sea necesario para poder guardar datos durante la ejecución del programa. Sin embargo este sistema sólo sirve para pequeños tamaños de memoria, ya que para grandes cantidades, el uso de arrays dinámicos presenta el problema de que ocupan mucha más memoria de la necesaria, dejando espacios de memoria sin utilizar. Al tener que estar todos los elementos de array en almacenados forzosamente en posiciones de memoria contiguas, si tenemos un array de N elementos y queremos hacerlo crecer hasta N+1, el sistema tendrá probablemente que reubicar los N elementos, trasportándolos hasta otra zona de memoria donde sí pueda guardar N+1 elementos contiguos.

Para evitar esto se utilizan las estructuras dinámicas de datos. El objetivo es poder guardar datos ordenados pero que no tengan que estar necesariamente en posiciones contiguas de memoria. Para esto se guardará, junto con cada dato, uno o varios enlaces que lo relacionan con el resto de los datos dela estructura. Según cómo sean las relaciones entre los datos de la estructura, hay varios tipos de estructuras. Los tipos de estructuras fundamentales son: listas, pilas, colas y árboles.

Cada elemento de una lista, pila, cola o árbol, consistirá en una estructura con al menos dos campos: El dato a guardar, un enlace a otro elemento:

```
struct nodo {
int dato; // clave del nodo
struct nodo *sigiente_nodo;
}
```

Y se representa como una variable con dos partes, una de las cuales es un puntero destinado a apuntar a una variable del mismo tipo.

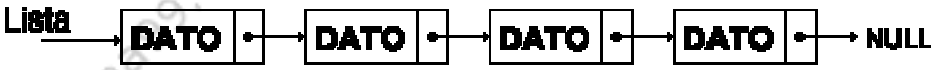


Listas simplemente enlazadas

Es el tipo más sencillo de estructuras dinámicas de datos. La estructura básica de un nodo para crear una lista de datos seria:

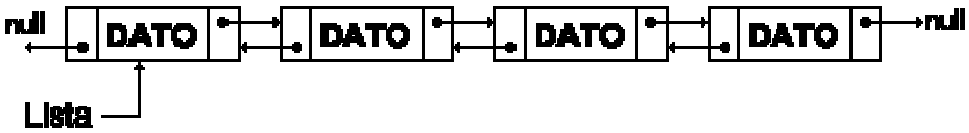
```
struct nodo {
int dato; // clave del nodo
struct nodo *sigiente_nodo;
}
```

La idea consiste básicamente en ir guardando en cada nodo la dirección del último nodo que se guardó, y un puntero llamado "lista" apuntando al último elemento guardado. La representación sería la siguiente:



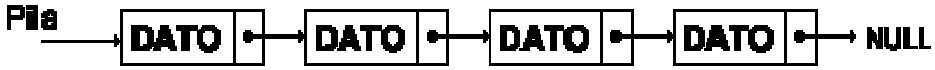
Listas doblemente enlazadas

Esta estructura añade a la anterior, la característica de que cada elemento también apunte al anterior, es decir, al introducido inmediatamente después (en el tiempo, no en localización en memoria). La representación sería la siguiente:



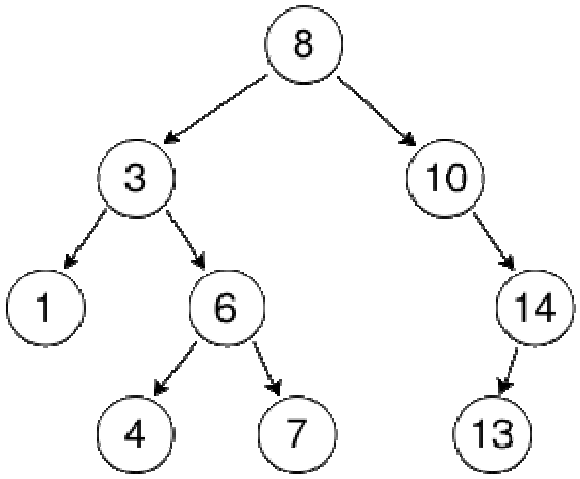
Pilas y colas

La estructura de pilas y colas es la misma que una lista simplemente enlazada. La diferencia está en la forma de recorrer la estructura.
Le llamamos Pila (como una pila de platos) cuando el último elemento que "apilamos" (guardamos) es el primero que extraemos. LIFO (Last In First Out)
Le llamamos Cola (como una cola para entrar al cine) cuando el primer elemento en llegar a la cola es el primero en salir. FIFO (First In First Out).



Árboles binarios

Los árboles binarios son una estructura pensada para poder encontrar fácilmente la información guardada sin tener necesariamente que buscar en todos los nodos. Los árboles binarios consisten en estructuras donde la información se guarda siguiendo un cierto orden elemental.
Este orden es siempre el mismo. Cada estructura del árbol tiene dos punteros (igual que la lista doblemente enlazada, pero en vez de llamarles siguiente y anterior, les llamamos, izquierda y derecha, que es más intuitivo). El orden consiste en que, según van llegando los elementos se van almacenando así: El primero en el único sitio que hay (será la raíz del árbol); el siguiente, si es mayor a la derecha, si es menor a la izquierda, y así sucesivamente.
Ejemplo: Guardar los siguientes nodos en un árbol: 8, 3, 6, 1, 10, 14, 4 , 7, 13.
El resultado es el siguiente:



Ejercicios:

Ejercicio 1: Lista simple 1.

Realizar un programa que permita guardar números en una lista. Debe tener un menu con las opciones, añadir elemento e imprimir todos.

Ejercicio 2: Lista simple 2.

Añadir al programa del ejercicio anterior la posibilidad de buscar un elemento de la lista.

Ejercicio 3: Lista simple 3.

Añadir al programa del ejercicio anterior la posibilidad de eliminar un elemento de la lista.

Ejercicio 4: Árbol binario 4.

Repetir los tres ejercicios anteriores utilizando un árbol binario.