



Escuela Técnica
Roberto Rocca

Роберто Рокка

LENGUAJES 5° - 6° TEL

ETRR

2024

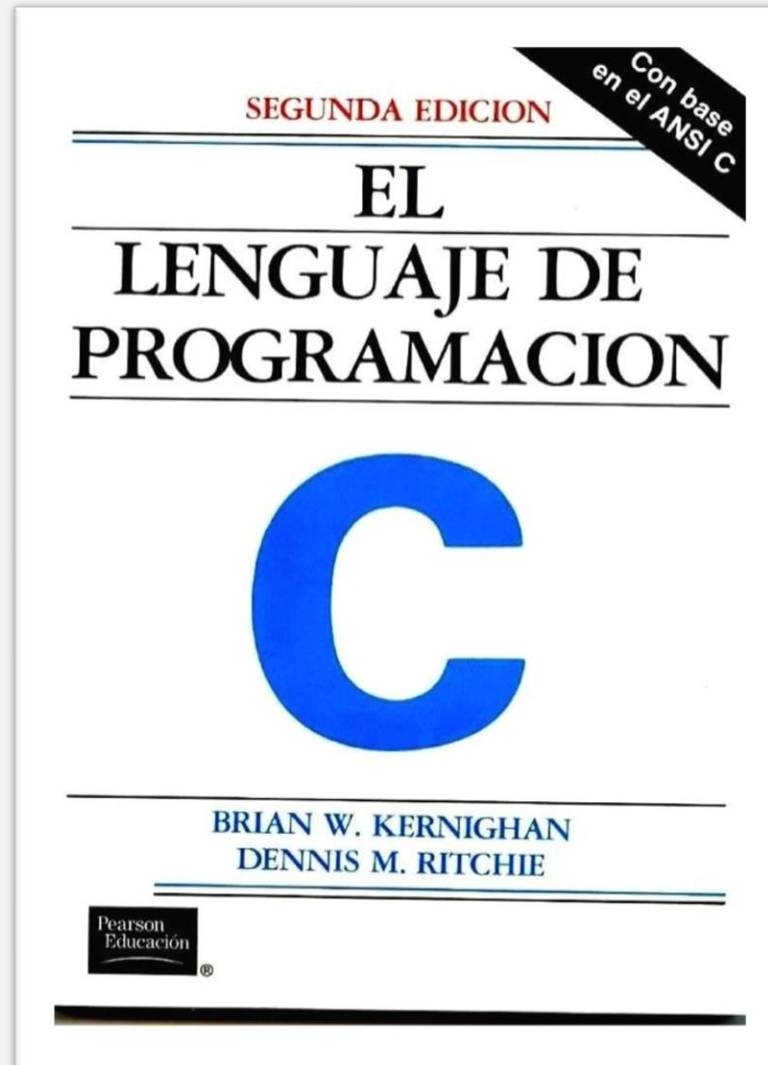
Ing. Pavelek Israel



Codificación en C

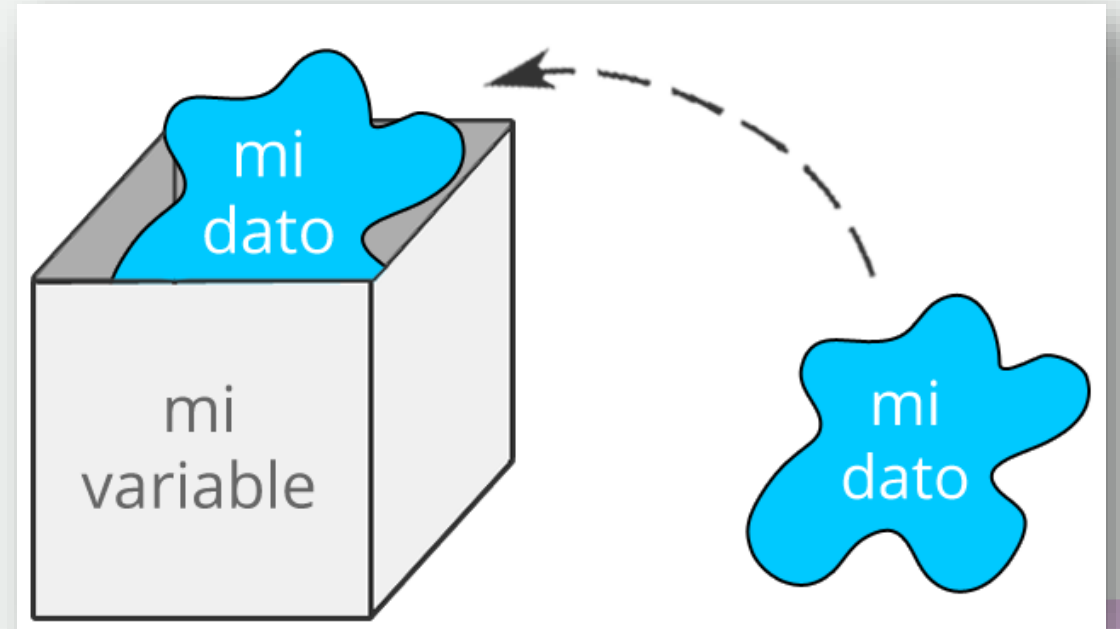
Desarrollado por **Dennis Ritchie** entre 1969 y 1972 en los **Laboratorios Bell**, como evolución del anterior lenguaje **B**. La primera estandarización del lenguaje C fue en **ANSI**, con el estándar X3.159-1989. El lenguaje que define este estándar fue conocido vulgarmente como **ANSI C**. Posteriormente, en 1990, fue ratificado como estándar ISO (ISO/IEC 9899:1990). La adopción de este estándar es muy amplia por lo que, si los programas creados lo siguen, el código es portable entre plataformas y/o arquitecturas.

El libro que lo define todo...



Recordemos que es una variable...

En programación, una **variable** está formada por un espacio en el sistema de almacenamiento y un nombre simbólico que está asociado a dicho espacio. Ese espacio contiene una cantidad de información conocida o desconocida, es decir un valor. El nombre de la variable es la forma usual de referirse al valor almacenado: esta separación entre nombre y contenido permite que el nombre sea usado independientemente de la información exacta que representa



Tipos de variables en C

Tipo de dato	Descripción	Longitud
char	Carácter o entero pequeño (byte)	1byte
int	Entero	dependiendo el compilador puede ser de 2bytes o 4 bytes
float	Punto Flotante simple precisión	4Bytes
double	Punto Flotante doble precisión	8Bytes
void	Sin tipo (uso especial)	-

Modificadores de variables en C

Modificador	Descripción
unsigned	Sin signo
signed	con signo
short	Corto
long	largo
const	Espacio de memoria constante
volatile	Variable cuyo valor es modificado externamente
static	Modificador para variables locales estáticas
extern	Para indicar que la variable se encuentra definida en otro archivo

Combinaciones modificadores y tipos de variables

Tipo	Cantidad de bits	Rango numérico
char	8	-128 a 127
unsigned char	8	0 a 255
signed char	8	-128 a 127
short (int)	16	-32768 a 32767
int OJO!!!!	16	-32768 a 32767
unsigned int	16	0 a 65535
signed int	16	-32768 a 32767
short int	16	-32768 a 32767
unsigned short int	16	0 a 65535

Combinaciones modificadores y tipos de variables

Tipo	Cantidad de bits	Rango numérico
signed short int	16	-32768 a 32767
long int	32	-2147483648 a 2147483647
signed long int	32	-2147483648 a 2147483647
unsigned long int	32	0 a 4294967295
long	32	-2147483648 a 2147483647
unsigned long	32	0 a 4294967295
float	32	3.4E-38 a 3.4E+38
double	64	1.7E-308 a 1.7E+308
long double	64 ó 80 (según versión).	1.7E-308 a 1.7E+308 ó 3.4E-4932 a 1.1E+4932

Nombre de variables

- En 'C' hay palabras reservadas que no pueden utilizarse como nombre de variable.

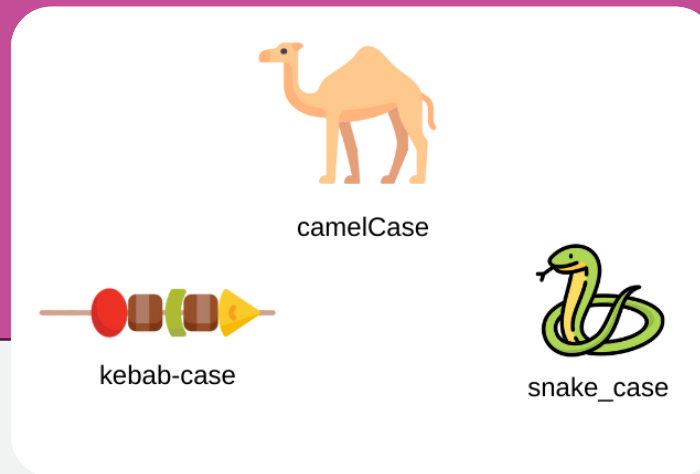
Auto	Double	Int	Struct
Break	Else	Long	Switch
Case	Enum	Register	Typedef
Char	Extern	Return	Union
Const	Float	Short	Unsigned
Continue	For	Signed	Void
Default	Goto	Sizeof	Volatile
Do	If	Static	While

Nombre de variables

- Un nombre válido ha de empezar por una letra o por el carácter de subrayado _, seguido de cualquier cantidad de letras, dígitos o subrayados.
- No se pueden utilizar números como primer caracter
- OJO: Se distinguen mayúsculas de minúsculas. (CASE SENSITIVE)
- No se pueden utilizar palabras las reservadas antes mencionadas
- Muchos compiladores no permiten letras acentuadas o eñes.

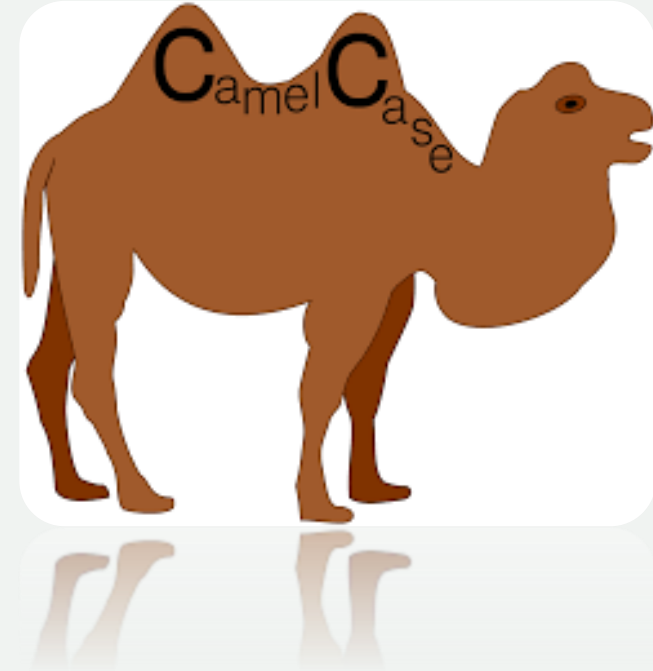
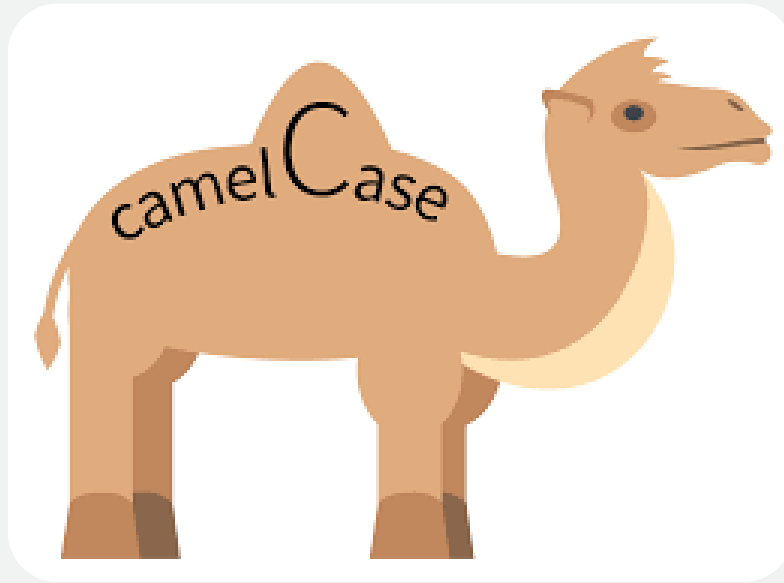
Estilos para los nombres de variables

- Siempre conviene utilizar nombres que ayuden a comprender que contiene la variable.
- Existen algunos formatos adoptados por los programadores:
 - Camel case
 - snake_case
 - kebab-case



Camel Case

- **Upper Camel Case**
 - Ejemplo: **EjemploDeNomenclatura**
- **Lower Camel Case**
 - Ejemplo: **ejemploDeNomenclatura**



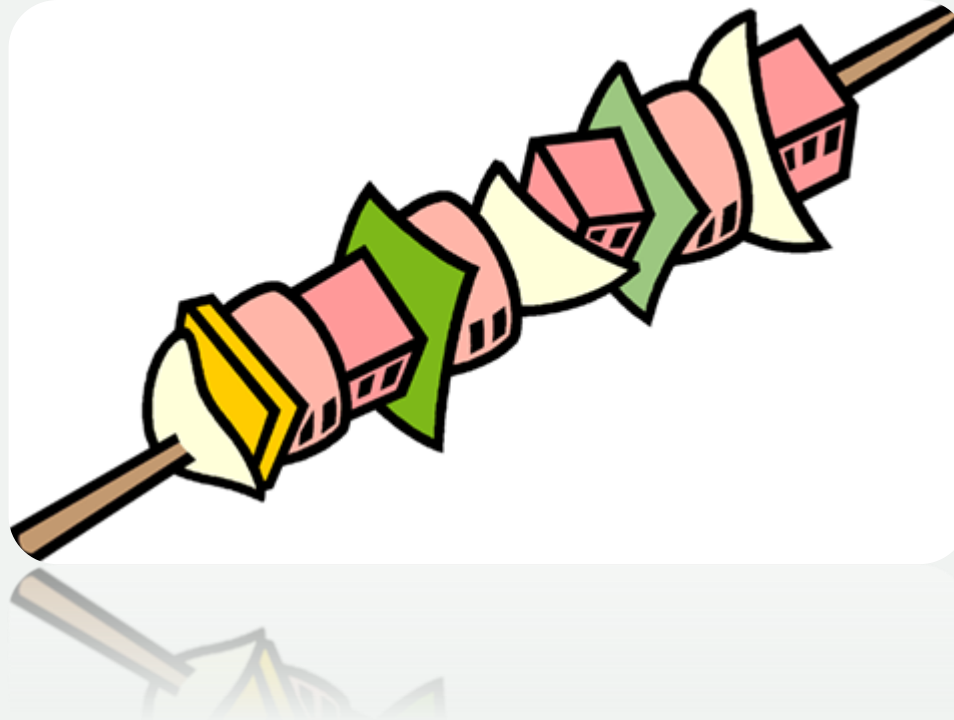
Snake Case

- Cuando cada una de las palabras, se separa por un guión bajo (_). Es común en los nombres de variables y funciones en C, aunque también Ruby.
- Ejemplo: `ejemplo_de_nomenclatura`



Kebab Case (No apto para C)

- Es igual que el Snake Case, esta vez, son guiones medios (-) los que separan las palabras. Su uso más común es de las URLs.
- Ejemplo: [ejemplo-de-nomenclatura](#)



Aclaración

Recordar que los valores de int pueden cambiar cuando trabajamos en sistemas embebidos por lo general int es una variable de 16 bits, pero en las computadoras modernas int es de 32 bits. Mas adelante veremos un **operador** llamado **sizeof()** que nos permite saber el tamaño de las variables en la PC donde se ejecutan.

Biblioteca stdint.h

Como algunos nombres de variables nativas en C prestan confusión existen nuevas definiciones, que ayudan a comprender el rango y tipo de variables. Para poder utilizarlas hay que incorporar la biblioteca stdint.h

```
#include <stdint.h>
```

Nombre	Tipo integrado equivalente
int8_t, uint8_t	signed char, unsigned char
int16_t, uint16_t	short, unsigned short
int32_t, uint32_t	int, unsigned int
int64_t, uint64_t	long long, unsigned long long

Aclaraciones variables bool

- Algunos compiladores habilitan el uso de variable del tipo `_Bool` para poder trabajar con bits. Este tipo de dato es valido desde C99 son variables, que pueden entonces tomar el valor 0 o 1 (False o True).
- También hay que tener en cuenta que, si reservamos una única variable de 1 bit, seguramente el compilador reserve al menos todo un byte para dicha variable.

Biblioteca stdbool.h

La biblioteca para utilizar para trabajar con valores booleanos es stdbool. Esta biblioteca redefine `_Bool` bajo el nombre `bool`, por compatibilidad, como así también define las macros `false` como 0 y `true` como 1.

```
#include <stdbool.h>
```

Comentarios // /-*/*

- En C podemos agregar comentarios en donde queramos a modo de agregar palabras que nos ayuden a entender que hace esa parte del código.
- Podemos hacer comentarios
 - De una única línea, todo lo que esté después de *//* no es tomado en cuenta
 - Comentarios multilínea empezando con */** y cerrando con **/*

Asignando valores en diferentes bases

Cada compilador toma una base por defecto, por lo tanto si no colocamos nada debemos chequear como está configurado el compilador

Modificadores antes del número (prefijos)

Prefijo	Base
0x	Hexa
0B	Binario
0	Octal
	Decimal (aunque puede variar)

Variables char (carácter)

Las variables char, son valores de 8 bits, que pueden ser manejados como números chicos signados o no, pero también usarse como valores equivalentes ASCII recordando que un carácter ASCII se relaciona con un valor de 8 bits por la siguiente tabla

TABLA DE CARACTERES DEL CÓDIGO ASCII

1	25	↓	49	1	73	I	97	a	121	y	145	æ	169	†	193	‡	217	‡	241	†
2	26	•	50	2	74	J	98	b	122	z	146	æ	170	‡	194	‡	218	‡	242	‡
3	27	♥	51	3	75	K	99	c	123	{	147	ô	171	‡	195	‡	219	‡	243	‡
4	28	♦	52	4	76	L	100	d	124		148	ô	172	‡	196	‡	220	‡	244	‡
5	29	♠	53	5	77	M	101	e	125	}	149	ô	173	‡	197	‡	221	‡	245	‡
6	30	♣	54	6	78	N	102	f	126	~	150	û	174	‡	198	‡	222	‡	246	‡
7	31	♥	55	7	79	O	103	g	127	¸	151	û	175	‡	199	‡	223	‡	247	‡
8	32	•	56	8	80	P	104	h	128	¸	152	ÿ	176	‡	200	‡	224	‡	248	‡
9	33	!	57	9	81	Q	105	i	129	¸	153	ÿ	177	‡	201	‡	225	‡	249	‡
10	34	"	58	:	82	R	106	j	130	¸	154	ÿ	178	‡	202	‡	226	‡	250	‡
11	35	#	59	;	83	S	107	k	131	¸	155	ç	179	‡	203	‡	227	‡	251	‡
12	36	\$	60	<	84	T	108	l	132	¸	156	ç	180	‡	204	‡	228	‡	252	‡
13	37	%	61	=	85	U	109	m	133	¸	157	ç	181	‡	205	‡	229	‡	253	‡
14	38	&	62	>	86	V	110	n	134	¸	158	ç	182	‡	206	‡	230	‡	254	‡
15	39	'	63	?	87	W	111	o	135	¸	159	ç	183	‡	207	‡	231	‡	255	‡
16	40	(64	@	88	X	112	p	136	¸	160	á	184	‡	208	‡	232	‡	255	‡
17	41)	65	A	89	Y	113	q	137	¸	161	í	185	‡	209	‡	233	‡	255	‡
18	42	*	66	B	90	Z	114	r	138	¸	162	ó	186	‡	210	‡	234	‡	255	‡
19	43	+	67	C	91	[115	s	139	¸	163	ú	187	‡	211	‡	235	‡	255	‡
20	44	,	68	D	92	\	116	t	140	¸	164	ñ	188	‡	212	‡	236	‡	255	‡
21	45	-	69	E	93]	117	u	141	¸	165	Ñ	189	‡	213	‡	237	‡	255	‡
22	46	.	70	F	94	^	118	v	142	¸	166	•	190	‡	214	‡	238	‡	255	‡
23	47	/	71	G	95	_	119	w	143	¸	167	•	191	‡	215	‡	239	‡	255	‡
24	48	0	72	H	96	`	120	x	144	¸	168	•	192	‡	216	‡	240	‡	255	‡



Ejemplo

Escribir:

- **unsigned char a=0x30;**
- **unsigned char a='0'**
- Son expresiones equivalentes, si colocamos entre simple comillas un símbolo ASCII asignará a la variable el valor de acuerdo a la tabla vista anteriormente.
- No es lo mismo el '1' que el numero 1.
- Si es lo mismo 0x31 que el carácter '1'

Ejemplos

- `a=0x23;` `//hexa`
 - `a=35;` `//decimal`
 - `a=0b00100011;` `//binario`
 - `a=043;` `//octal`
 - `a='#'` `// equivalente ASCII`
- Todas estas expresiones equivalentes y podemos usar la que nos sea más cómoda según el dato que manejemos

Primer código en C

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

Zona de inclusión de bibliotecas

```
int main(void){
```

Zona de declaraciones globales
Función principal (main) que devuelve un tipo int y no recibe nada (void)

Llave de apertura indicando el comienzo de la función main

```
printf("Hola mundo");  
return (0);
```

Zona de declaraciones locales

Función que imprime en pantalla "Hola mundo"

```
}
```

Llave de cierre indicando fin de la función main

Hace que la función main devuelva el valor 0 al sistema operativo

Funciones

Una función está definida por:

- Un nombre
- Un **tipo de valor** que retorna al terminar
- El/los tipos de valores (parámetros que recibe)

En caso de no retornar nada o de no recibir ningún parámetro se coloca **void** en el lugar correspondiente

Prototipo de una función en C

Tipo_parámetro_retorno NombreFuncion (tipo_param1, tipo_param2);

- La cantidad de parámetros de retorno es única, puede retornar **1 solo tipo de parámetro o ninguno si es void**
- No hay límites en cuanto a la cantidad de parámetros de recepción de una función, si no hay ninguno se coloca void

Ejemplos de prototipos

```
//Función de nombre mifunc  
//Parámetros que recibe: Ninguno  
//Parámetro que devuelve: Ninguno  
void mifunc (void);
```

```
//Función de nombre mifunc2  
//Parámetros que recibe: dos variables del tipo int var1 y var2  
//Parámetro que devuelve: Ninguno  
void mifunc2 (int var1, int var2);
```

Ejemplos de prototipos

/*Función de nombre mifunc3

Parámetros que recibe: dos variables una del tipo int var1 y otra del tipo char var2

Parámetro que devuelve: Ninguno*/

```
void mifunc3 (int var1, char var2);
```

/*Función de nombre mifunc4

Parámetros que recibe: Ninguno

Parámetro que devuelve: un valor del tipo int*/

```
int mifunc4 (void);
```

Ejemplos de prototipos

/*Función de nombre mifunc5

Parámetros que recibe: una variable del tipo int var1

Parámetro que devuelve: un valor del tipo int*/

```
int mifunc5 (int var1);
```


Código/cuerpo de una función

- Una función para poder ser utilizada debe tener su código definido previamente o al menos su prototipo en la zona de declaraciones globales.
- En el prototipo **no es necesario** ingresar los nombres de las variables que reciben los parámetros.

El código de la función empieza con la apertura de la llave y termina cuando dicha llave cierra.

- Las llaves en general definen bloques de código, pudiendo tener bloques dentro de bloques. En bloques que contengan una sola línea no es necesario escribir la llave de apertura y cierre.

Ejemplo

```
void mifunc (unsigned int);

int main(void){
    unsigned int var2;
    mifunc(2);
    printf("Hola mundo");
    return 0;
}

void mifunc (unsigned int var){
    /*mi codigo*/
}
```

/*En el prototipo (al declarar la función) puede o no estar el nombre de las variables, si es importante que figure en donde está el código de la función*/

Ejemplo

```
void mifunc (unsigned int var){  
    /*mi codigo*/  
}  
  
int main(void){  
    unsigned int var2;  
    mifunc(2);  
    printf("Hola mundo");  
    return 0;  
}
```

/*Si la función se escribe con su código arriba de la función en donde se la invoqué no es necesario declarar su prototipo, aunque **no es una buena práctica***/
/En los archivos de bibliotecas que incluimos se encuentran los prototipos de muchas funciones que podemos utilizar/

Invocando a una función

- Para invocar a una función basta llamarla por el nombre, colocando entre paréntesis los parámetros que deseamos pasarle (si es que fuera necesario)
- En caso de que la función devuelva algún parámetro es importante colocar una variable a la izquierda e igualar la función.

Retorno de la función

- Si una función retorna un valor, antes de finalizar, colocamos la instrucción return y entre paréntesis el valor que queremos que retorne.

Declaración de una variable

- Para poder utilizar una variable primero debe ser declarada.
- Podemos tener dos tipos de variables.
 - Globales (son válidas dentro de TODO el programa y dentro de cada función)
 - Locales (solamente son válidas en las funciones que fueron declaradas)

Ejemplo

```
#include <stdio.h>

unsigned int var1;

int main(void){
    unsigned int var2;

    printf("Hola mundo");
    return 0;
}
```

/*En este ejemplo tenemos una variable global llamada var1 que puede ser consultada en todas las funciones que agreguemos al programa, y otra variable var2 que solo existe en el ámbito de la función principal (main) */

Ejemplo

```
unsigned int var1;

void mifunc (void);

int main(void){
    unsigned int var2;

    printf("Hola mundo");
    return 0;
}

void mifunc (void){
    unsigned int var2;
    /*mi codigo*/
}
```

/*En este ejemplo hay 3 variables var1 (global) y 2 variables var2, una local en la función principal main y otra en la función mifunc, las variables en main y en mifunc no están relacionadas y sus contenidos pueden o no ser iguales */

Formas de declarar una variable

Para declarar una variable primero debemos colocar (opcionalmente) el **modificador**, luego el **tipo de variable** y seguido el **nombre**.

```
unsigned int mivar;
```

Darle un valor inicial

Al declarar una variable podemos darle un valor inicial igualando la variable al valor

```
unsigned int mivar=123;
```

Declarando más de una variable

Si queremos declarar más de una variable podemos hacerlo una debajo de la otra o separando por comas, siempre y cuando sean variables del mismo tipo.

```
unsigned int mivar1;  
unsigned int mivar2;  
unsigned int mivar3;
```



```
unsigned int mivar1,mivar2,mivar3;
```

Ejemplo

```
unsigned int suma (unsigned int var1, unsigned int var2);
```

```
int main(void){  
    unsigned int suma=0,resultado;  
    resultado=suma(1,2);  
    return 0;  
}
```

```
unsigned int suma (unsigned int var1, unsigned int var2){  
    unsigned int res=0;  
    res=var1+var2;  
    return(res);  
}
```

/*En el ejemplo anterior se llama a la función suma pasándole los valor 1 y 2, y la función retorna el valor de la suma*/

Mejorando el código...

```
unsigned int suma (unsigned int var1, unsigned int var2);

int main(void){
    unsigned int suma=0,resultado;
    resultado=suma(1,2);
    return 0;
}

unsigned int suma (unsigned int var1, unsigned int var2){
    return(var1+var2);
}
```

/*En el ejemplo anterior se llama a la función suma pasándole los valor 1 y 2, y la función retorna el valor de la suma*/

Usando un valor constante

```
float AreaCircunferencia (unsigned int radio);  
const float PI=3.14;
```

```
int main(void){  
    float area=0  
    area= AreaCircunferencia(1);  
    return 0;  
}  
float AreaCircunferencia (unsigned int radio){  
    return(PI*radio*radio);  
}
```

Aunque no es algo del lenguaje por convención de los programadores los nombres de las constantes van en mayúsculas

printf()

`printf` es una función definida en `stdio.h` (estándar input output) para poder visualizar por pantalla datos.

`printf("text");` todo el texto que se escriba entre doble comillas saldrá textual por pantalla.

printf()

Si queremos mostrar un valor intercalaremos en el lugar que queremos que se muestre un carácter de escape (%), y a continuación formateadores para que muestre el dato en cuestión en el formato requerido.

Formateadores printf()

Formateador	Salida
%d ó %i	entero en base 10 con signo (int)printf ("el numero enteronen base 10 es: %d" , -10);
%u	entero en base 10 sin signo (int)
%o	entero en base 8 sin signo (int)
%x	entero en base 16, letras en minúscula (int)
%X	entero en base 16, letras en mayúscula (int)
%f	Coma flotante decimal de precisión simple (float)
%lf	Coma flotante decimal de precisión doble (double)
%ld	Entero de 32 bits (long)
%lu	Entero sin signo de 32 bits (unsigned long)
%e	La notación científica (mantisa / exponente), minúsculas (decimal precisión simple ó doble)
%E	La notación científica (mantisa / exponente), mayúsculas (decimal precisión simple ó doble)
%c	carácter (char)
%s	cadena de caracteres (string)

Caracteres de escape:

Estos caracteres cuando se encuentran dentro de la salida provocan cosas diferentes.

Carácter de escape	Significado
\a	Suena un beep
\b	Espacio atrás
\f	Form Feed
\n	Carácter de nueva línea
\r	Retorno de carro
\t	Tabulación horizontal
\v	Tabulación vertical
\\	Carácter para salida de barra invertida
\'	Carácter para salida de comilla simple
\"	Carácter para salida de comilla doble
\o	Base octal
\x o \X	Base hexadecimal
\0	Terminador Null

Ejemplos

```
int main (void){  
  
    int var=3,var2=5;  
    printf("el valor de la variable var es: %d\n",var);  
    printf("el valor de la variable var2 es: %d\n",var2);  
    return 0;  
}
```

printf

El % con el formateo correspondiente puede estar en cualquier lugar y ahí es donde será reemplazado por el valor o variable presente luego de cerrar las comillas se encuentra separado por una coma.

Se pueden poner tantos valores como uno quiera desee mostrar en un mismo printf

Ejemplos

```
int main (void){  
  
    int var=3,var2=5;  
    printf("el valor de la variable var es: %d \nel valor de la variable var2 es: %d\n",var,var2);  
    return 0;  
}
```

¿Hay alguna diferencia entre este y el código anterior?

scanf()

scanf es una función definida en stdio.h para ingresar valores por teclados y guardarlos en variables.

Utiliza el mismo formato que printf con los mismos formateadores.

Ejemplo

```
int main (void){  
  
    int var;  
    printf("ingrese un valor\n");  
    scanf("%d",&var);  
    printf("Ud. ingreso: %d",var);  
    return 0;  
}
```

Hay que prestar mucha atención que aparece el & antes de la variable en donde guardamos el valor ingresado por el usuario vía teclado, este operador indica que queremos guardarla en ese lugar.

putchar

- En algunos casos nos interesa solamente mostrar un carácter podemos entonces utilizar la función `putchar()`;
- Esta función posee el siguiente prototipo:
- **`int putchar(int)`**

putchar();

```
int main (void){  
    putchar('@');  
    return 0;  
}
```

putchar();

```
int main (void){  
    putchar(64);  
    return 0;  
}
```



getchar()

Idem que putchar pero para recibir un único carácter por teclado



Ejemplo

```
#include <stdio.h>

int main (void) {
    char c;
    printf("Ingrese un caracter\n");
    c = getchar();
    printf("Ud. a ingresado:");
    putchar(c);
    return(0);
}
```

Operadores en C

- Aritméticos
- Relacionales
- Lógicos
- Asignación
- Incremento y decremento

Operadores Aritméticos

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División
%	Modulo

Operador Módulo %

El operador módulo realiza la división entre dos operandos, pero en lugar de devolver el cociente, devuelve el resto de dicha operación.

`a=13%10;`

Luego de ejecutar esa línea, el valor de a es 3 (el resto de hacer 13/10)

Operadores Relacionales

Estos operadores se utilizan para evaluar expresiones, y determinar si estas relaciones son falsas o verdaderas

Operador	Descripción
<	Menor
>	Mayor
<=	Menor o igual
>=	Mayor o igual
==	Igual
!=	distinto

Operadores lógicos

Los operadores lógicos se utilizan para unir expresiones o negar una expresión siempre que ellas puedan ser catalogadas como verdaderas o falsas

Operador	Descripción
&&	And (y)
	Or (o)
!	Not (no)

Operadores lógicos

Ejemplo:

$(a > 2) \&\& (a < 4)$

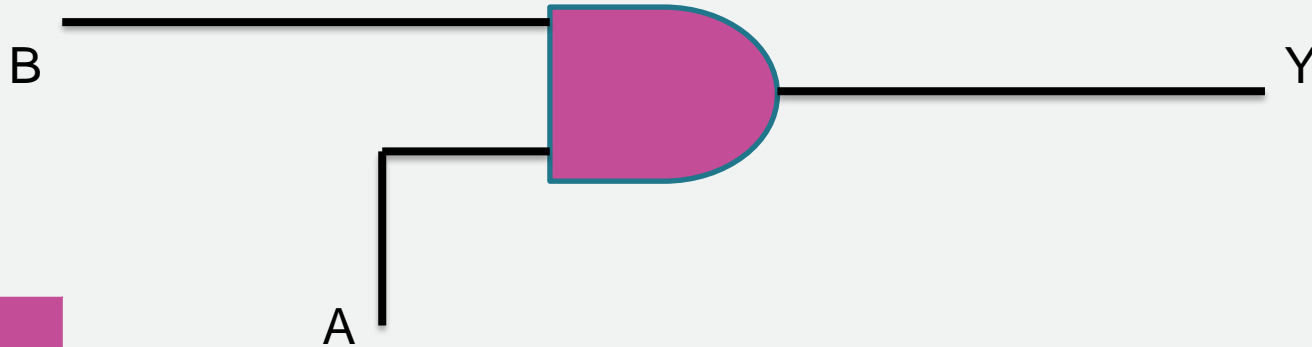
El código evalúa si a es mayor a 2 Y además a es menor a 4, se pueden unir la cantidad de expresiones que se quiera, para crear condiciones, utilizando los 3 operadores lógicos

Operadores a nivel de bit

Estos operadores realizan la operación lógica entre dos operandos bit a bit.

Operador	Descripción
&	And
	Or
^	Xor
<<	Desplazamiento binario a la izquierda
>>	Desplazamiento binario a la derecha
~	Complemento (negación)

Tabla de verdad reducida (and)



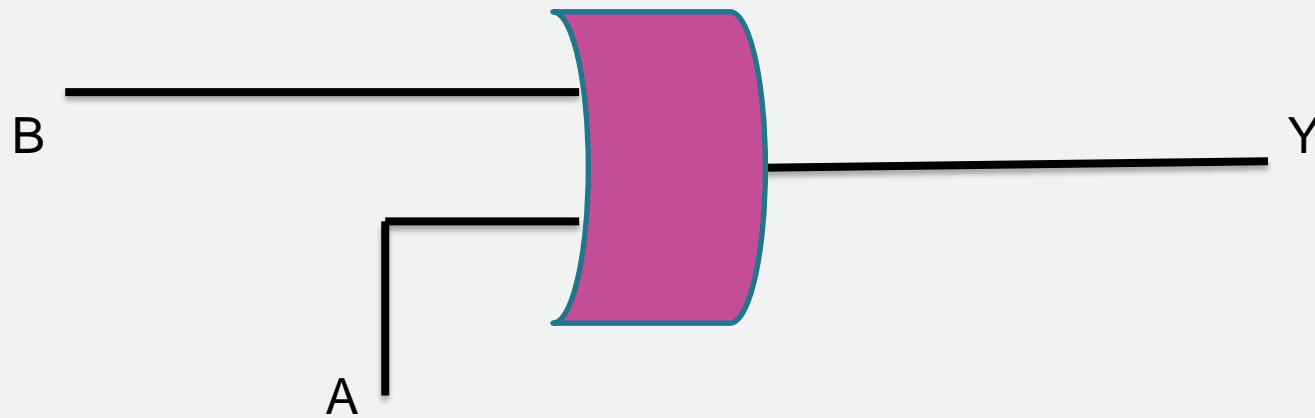
Fuerza ceros

A	Y
0	0
1	B

&

B	0	0	1	1	0	1	0	1
A	0	0	0	0	1	1	1	1
Y	0	0	0	0	0	1	0	1

Tabla de verdad reducida (or)

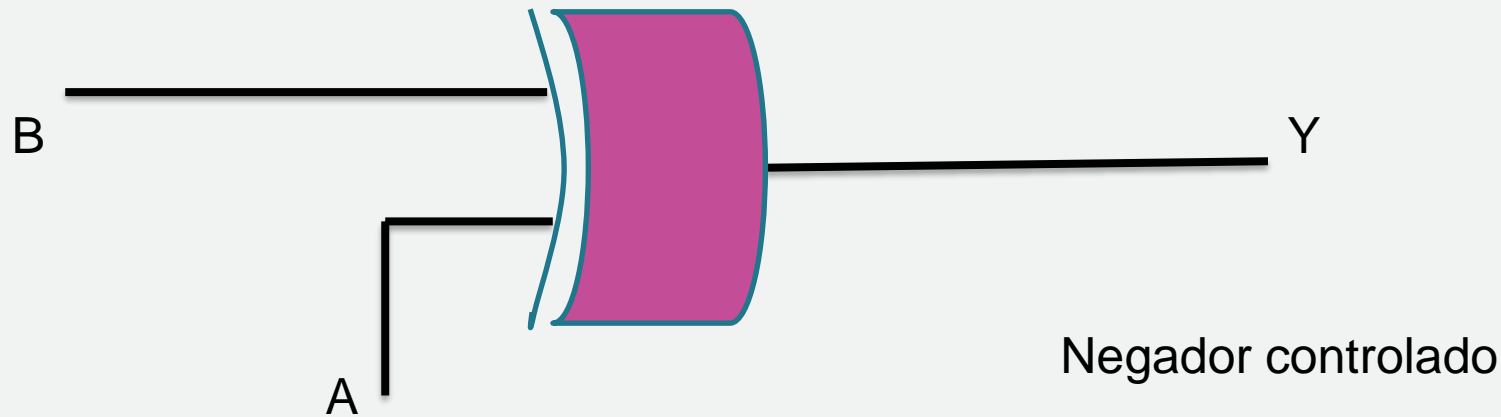


Fuerza unos

A	Y
0	B
1	1

B	0	0	1	1	0	1	0	1
A	0	0	0	0	1	1	1	1
Y	0	0	1	1	1	1	1	1

Tabla de verdad reducida (Xor)



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

A	Y
0	B
1	!B

B	0	0	1	1	0	1	0	1
A	0	0	0	0	1	1	1	1
Y	0	0	0	0	1	0	1	0

Ejemplos

AND

- `a=0b11001100;`
- `b=0b01100111;`
- `c=a&b; //el valor resultado en c es=0b01000100;`

Ejemplos

OR

- ▣ `a=0b11001100;`
- ▣ `b=0b01100111;`
- ▣ `c=a|b; //el valor resultado en c es=0b11101111;`

Ejemplos

XOR

- ▣ `a=0b11001100;`
- ▣ `b=0b01100111;`
- ▣ `c=a^b; //el valor resultado en c es=0b10101011;`

Ejemplos

Desplazamiento, se coloca la variable a desplazar y la cantidad de desplazamientos

- `a=0b00001100;`
- `b=a<<2; //el valor resultado en c es=0b00110000;`
- Se desplazo a dos lugares a la izquierda completando con ceros de la izquierda a la derecha

Ejemplos

COMPLEMENTO

- `a=0b00001100;`
- `b=a>>2; //el valor resultado en c es=0b00000011;`
- Se desplazo a dos lugares a la derecha completando con ceros de la derecha a la izquierda

Ejemplos

Desplazamiento, se coloca la variable a desplazar y la cantidad de desplazamientos

□ `a=0b00001100;`

□ `b= ~ a; //el valor resultado en c es=0b11110011;`

Combinando expresiones

```
a=0b11001100;
```

```
b=0b01100111;
```

```
c=a&(~b); //el valor resultado en c es=0b10001000;
```

Operador de asignación

- El operador de asignación es el símbolo =
- En casos donde la operación es fuente y destino podemos abreviar de la siguiente forma:

Normal	Versión compacta
$a=a+b$	$a+=b$
$a=a-b$	$a-=b$
$a=a*b$	$a*=b$
$a=a/b$	$a/=b$
$a=a>>2$	$a>>=2$
$a=a<<2$	$a<<=2$
$a=a\&b$	$a\&=b$
$a=a b$	$a =b$
$a=a^{\wedge}b$	$a^{\wedge}=b$

Operador de incremento y decremento

Normal	Versión compacta
<code>a=a+1</code>	<code>a++</code>
<code>a=a-1</code>	<code>a--</code>

Operador de incremento y decremento

Cuando asignamos puede ser post o pre incremento o decremento. Es decir, si asigna primero y luego incrementa o decrementa o incrementa o decrementa y luego asigna.

Normal	Pre/Pos	Detalle
<code>a=i++;</code>	Pos incremento	<code>a=i;</code> <code>i=i+1;</code>
<code>a=++i;</code>	Pre incremento	<code>i=i+1;</code> <code>a=i;</code>
<code>a=--i;</code>	Pre decremento	<code>i=i-1;</code> <code>a=i;</code>
<code>a=i--;</code>	Pos decremento	<code>a=i;</code> <code>i=i-1;</code>

Usando el preprocesador

Existen algunas directivas (instrucciones que son propias del programa que utilizamos y no del lenguaje C que se traduce en código de máquina, estas instructivas nos facilitan la escritura del código y no generan código extra, a la hora de compilar el compilador reemplaza por su equivalente.

#define

- #define LED 1

- La línea de código anterior hace que en el código cuando aparezca la palabra LED sea reemplazada por la expresión a la derecha, en este caso el 1.
- Esto nos permite que el código se ajuste sin tener que cambiar en TODOS los lugares uno por uno.
- También permite un código más claro, ya que en lugar de números podemos trabajar con expresiones más simples que nos ayudan a entender y recordar como funciona el código.

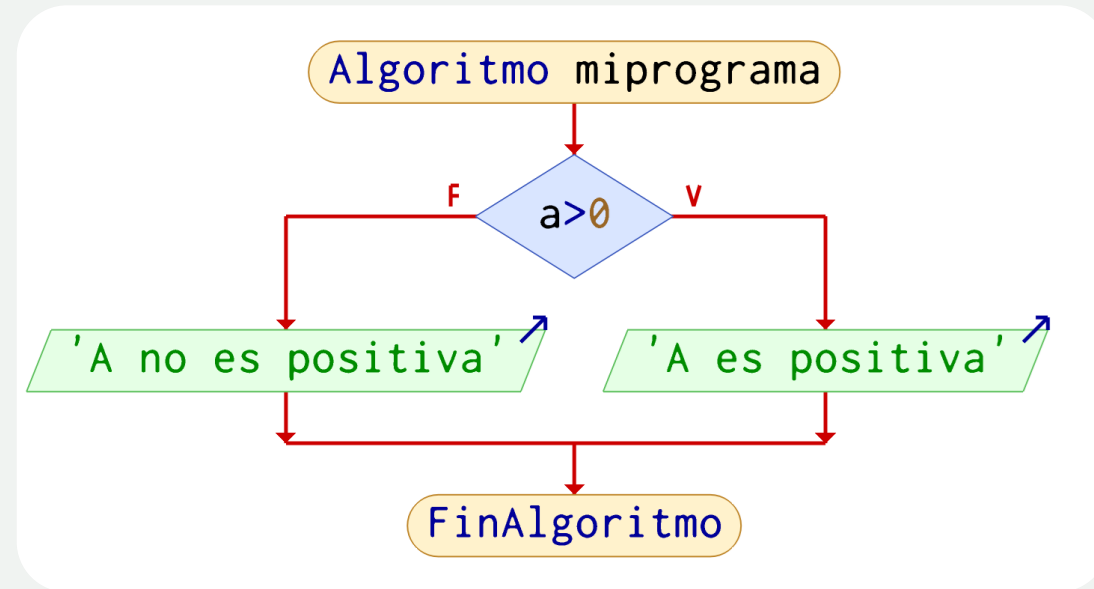
Control de secuencia bloque decisión

```
if(condición){  
    //codigo que se ejecuta si la condición es verdadera  
}else{  
    /*codigo que se ejecuta si la condición es falsa,  
    si no hay puede obviarse el else y las llaves*/  
}
```

Dentro de la condición puede haber una expresión relacional como las que vimos antes, que pueden combinarse con operadores lógicos como los que vimos antes, es importante entender que esa condición tiene que poder determinar su grado de veracidad o no.

Control de secuencia bloque decisión

```
if(a>0){  
    printf("a es positiva");  
}else{  
    printf("a no es positiva");  
}
```



Recordar

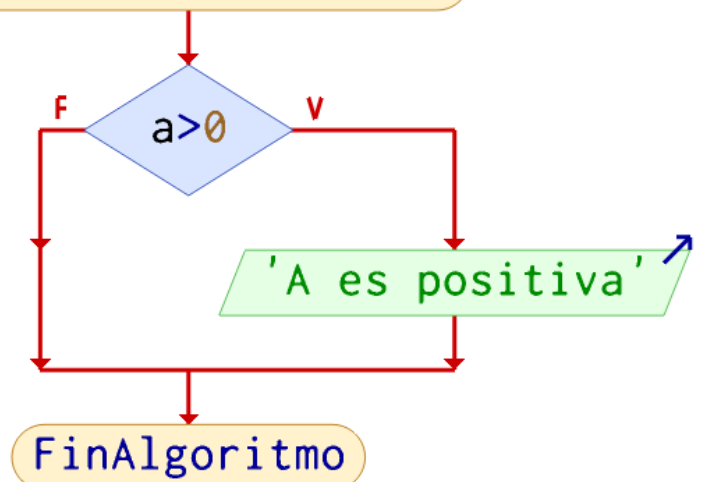
```
if(a>0)printf("a es positiva");  
else printf("a no es positiva");
```

Recordá que si dentro de una llave hay solamente una línea de código puede obviarse la llave, y el software interpreta que la primera línea es la única dentro

Recordar

```
if(a>0)printf("a es positiva");
```

Algoritmo miprograma



Si en la condición no hay código en la rama falsa no es necesario colocar la palabra else ni sus llaves.

Tips

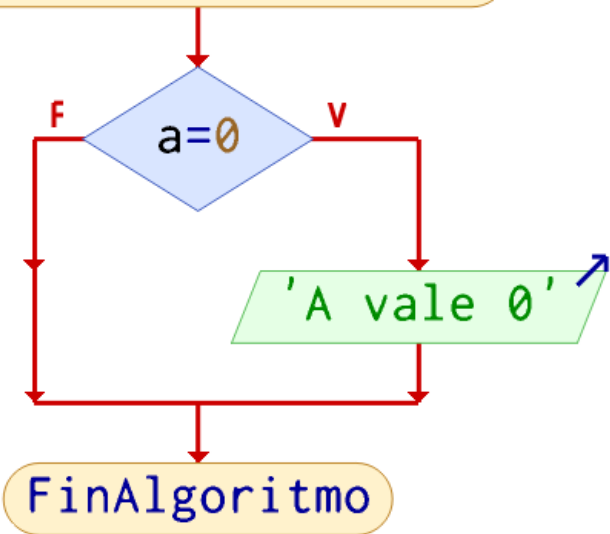
____ Que una expresión sea verdadera significa que dicha expresión vale cualquier valor excepto 0, mientras que si la condición es falsa se asigna con el valor 0.

por lo tanto...

Este código

```
if(a==0)printf("a vale 0");
```

Algoritmo miprograma

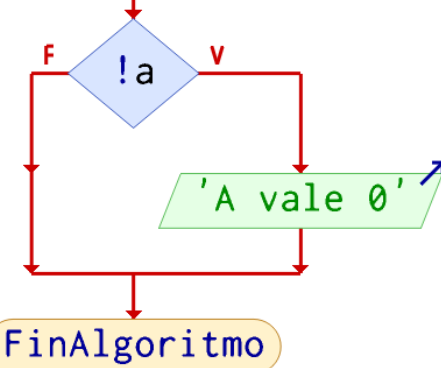


Recordar que en Pseint se utiliza un solo igual, pero en C eso es asignar, mientras que el doble = evalúa el grado de igualdad de dos expresiones, en este caso el valor de una variable y una constante.

Es equivalente a este

```
if(!a)printf("a vale 0");
```

Algoritmo miprograma

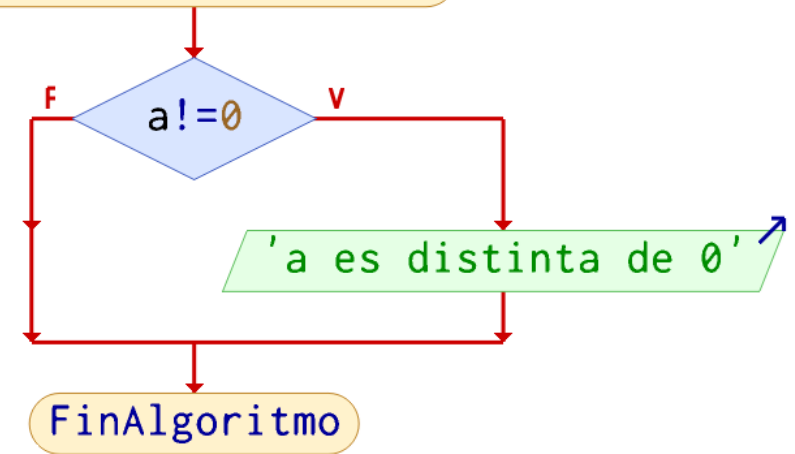


Si a vale 0 al negarla vale 1, por lo tanto la condición se hace verdadera.

Y este código

```
if(a!=0)printf("a es distinto de 0");
```

Algoritmo miprograma

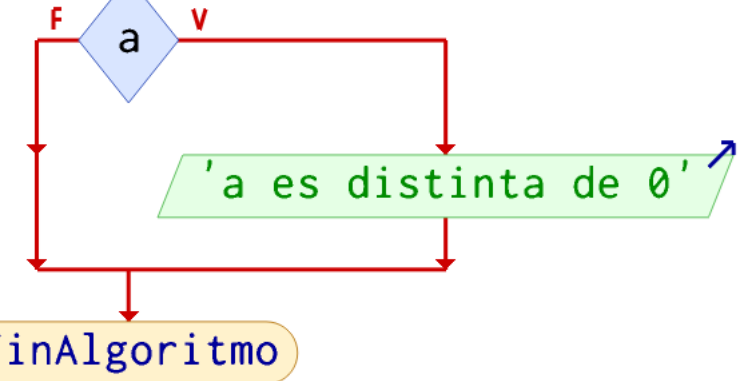


FinAlgoritmo

Es equivalente a este

```
if(a)printf("a es distinta de 0");
```

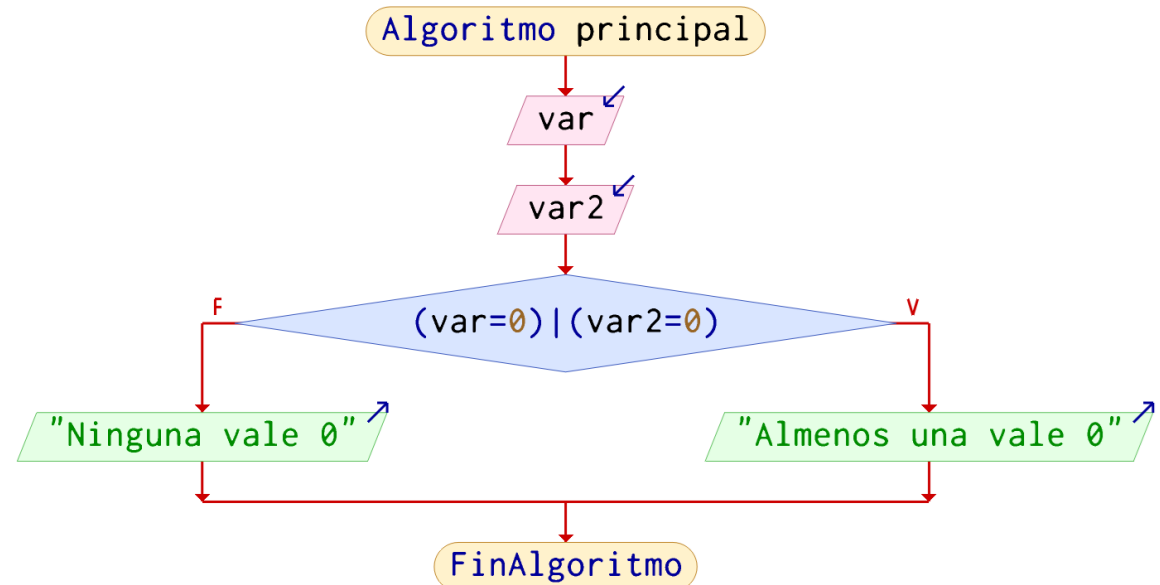
Algoritmo miprograma



Si a es cualquier valor menos cero, la condición es verdadera por lo tanto a es distinta de cero, es falsa solamente cuando a es igual a cero

Combinando expresiones

```
if((var1==0) || (var2==0)){  
    printf("Al menos una vale 0")  
}else{  
    printf("Ninguna vale 0")  
}
```

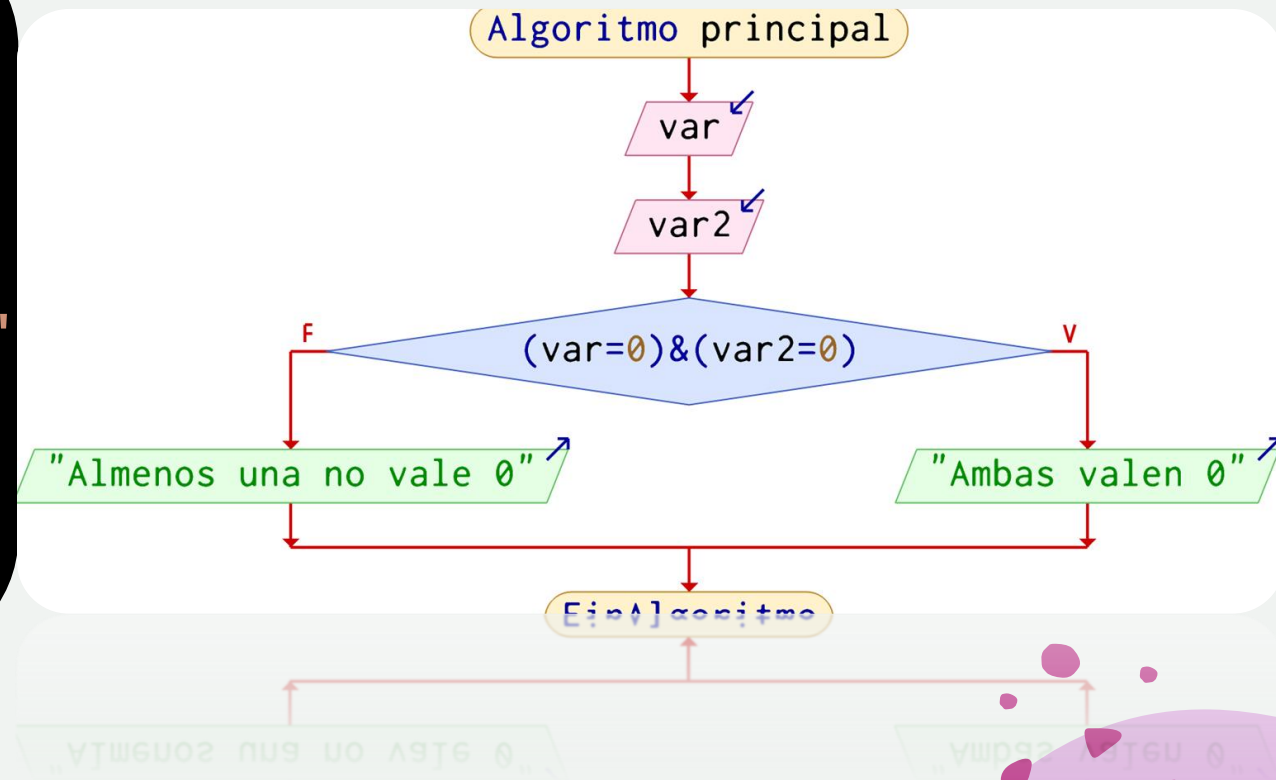


Recordar que, al combinar expresiones con una or, se utilizan dos ||, cuando se utiliza una sola |, se realiza la or bit a bit.

Las llaves pueden obviarse, dado que solo hay una línea de código dentro.

Combinando expresiones

```
if((var1==0)&&(var2==0)){  
    printf("ambas valen 0")  
}else{  
    printf("Al menos una no vale 0")  
}
```



else if

```
if(var==3){  
    //hacer algo...  
}else if (var==4){  
    //hacer algo..  
}else if (var==8){  
    //hacer algo..  
}else{  
    //hacer algo  
}
```

Operador ternario ?

```
if(var>10)var2=var;  
else var2=0;
```

Es equivalente a ...

```
var2=var>10?var:0;
```

Operador ternario ?

variable = condición ? valor si cierto : valor si falso

Control de secuencia bloque while

```
while(condición){  
    /*código que se ejecuta mientras  
    la condición sea verdadera*/  
}
```

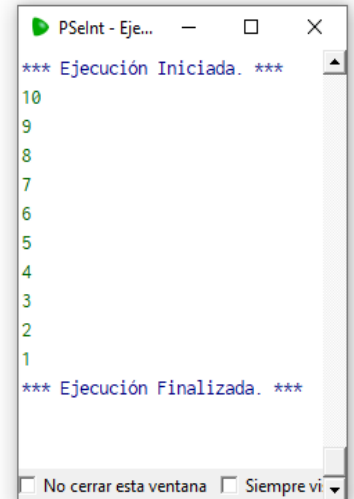
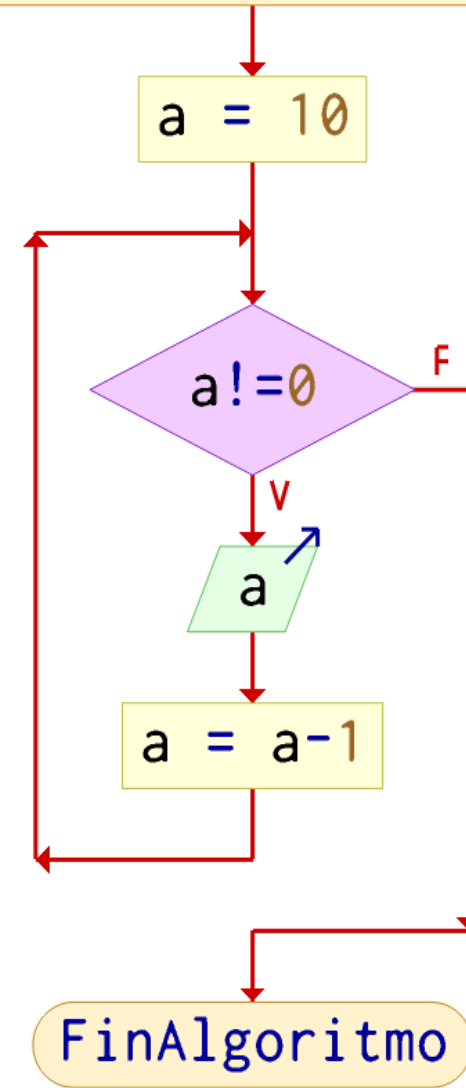
Las condiciones cumplen todo lo visto anteriormente, y mientras sea verdadera el código dentro se ejecuta una y otra vez.

Ciclo while

```
a=10;  
while(a){  
    printf("%d",a);  
    a--;  
}
```

En el siguiente código podríamos haber, colocado directamente a en la condición y funciona igual

Algoritmo principal



Ciclo do while

```
do{  
    /*código a ejecutar mientras  
    la condición sea verdadera*/  
}while(condición);
```

Este ciclo, es igual al ciclo while, pero posee una sutil diferencia, en lugar de evaluar la condición al entrar al ciclo, lo hace al terminar un ciclo, por lo tanto, se asegura siempre que al menos el código entre llave sea ejecutado una vez.

Ciclo for

```
for(inicialización;condición;acción){  
    //código a ejecutar si la condición es verdadera  
}
```

El ciclo **for** contiene 3 campos, separados ;

- El primer campo es lo que se va a ejecutar cuando se entra al ciclo, por lo general es la inicialización de una variable, pero puede ser cualquier cosa, o hasta estar vacío.
- El segundo campo es la condición, esta condición es evaluada al entrar al ciclo y al volver a repetirlo, si la condición es verdadera, el código entre las llaves se vuelve a ejecutar.
- El tercer campo, es un campo que se ejecuta cada vez que se termina un ciclo, antes que se evalúe la condición para saber si se vuelve o no a ejecutar el código entre llaves.

Ciclo for

```
for(i=0;i<10;i++){  
    //código a ejecutar si la condición es verdadera  
}
```

En el ejemplo, se inicializa la variable *i* en 0, si la variable es menor a 10, ejecuta el código, dentro, una vez que termina, ejecuta el campo de acción incrementando a *i* en 1, vuelve a evaluar la condición y si sigue siendo verdadera ejecuta ´de nuevo el código entre llaves, y caso contrario sigue por lo que hay debajo de la llave que cierra.

En este caso el código entre llaves se ejecuta 10 veces, en donde el valor de *i* va de 0 a 10, aunque el décimo primer paso no lo ejecuta la variable *i* termina con el valor 10

Ciclo while y for equivalentes

```
i=0;  
while(i==10){  
    //codigo a ejecutar si la condición es verdadera  
    i++;  
}
```

```
for(i=0;i<10;i++){  
    //código a ejecutar si la condición es verdadera  
}
```

Ciclos infinitos

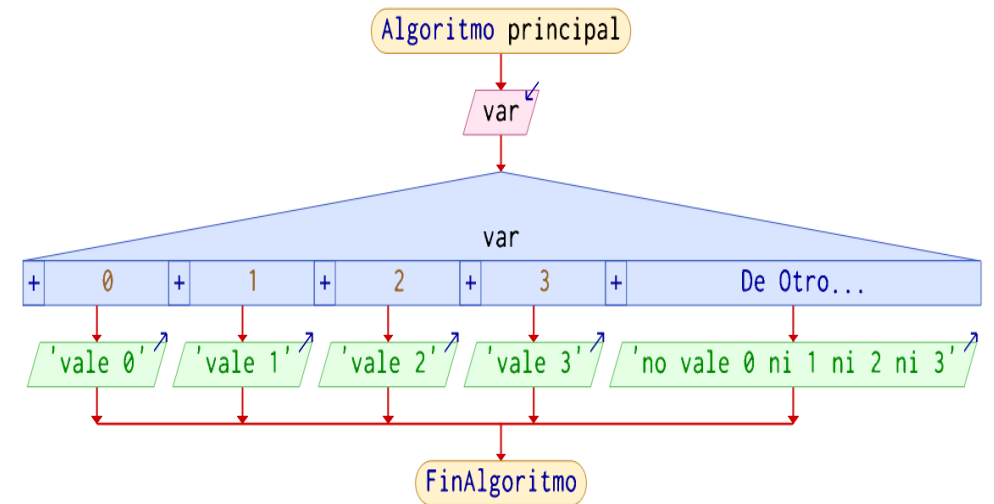
```
while(1){  
    //bucle infinito  
}
```

```
for(;;){  
    //bucle infinito  
}
```

Estos códigos ejecutan infinitas veces el código dentro de las llaves, en los microcontroladores, suele existir este tipo de bucles, dado que el sistema siempre se encuentra funcionando. En Arduino este bucle es llamado loop

Ciclos switch

```
switch(Var){  
    case 0: printf("Vale 0");  
            break;  
    case 1: printf("Vale 1");  
            break;  
    case 2: printf("Vale 2");  
            break;  
    case 3: printf("Vale 3");  
            break;  
    default: printf("No vale 0 ni 1 ni 2 ni 3");  
            break;  
}
```



Ejemplo de calculadora usando Switch

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

```
int var1,var2,res;  
unsigned char op;
```

```
int main (void){
```

```
    printf("Ingrese el primer valor:\n");  
    scanf("%d",&var1);  
    printf("Ingrese el segundo valor:\n");  
    scanf("%d",&var2);  
    printf("Ingrese la operacion\n");  
    //fflush(stdin);  
    scanf(" %c",&op);  
    printf("operacion: %c\n",op);
```

```
    switch(op){  
        case '+':res=var1+var2;  
                break;  
        case '-':res=var1-var2;  
                break;  
        case '*':res=var1*var2;  
                break;  
        case '/':  
            if(var2!=0)res=var1/var2;  
            else Printf("Error no se puede dividir por cero\n");  
            break;  
    }  
    printf("%d %c %d = %d\n",var1,op,var2,res);  
    return 0;  
}
```

Break y continue

- Si me encuentro dentro de un bucle, while, do while, for, switch y el código pasa por la sentencia
 - Break, automáticamente el procesador abandona el bucle.
 - Continue, vuelve al principio del bucle en caso del for ejecuta la acción y evalúa la condición, en caso de los otros bucles solamente vuelve a evaluar la condición.

Estructuras avanzadas de datos

- **Vectores**
- **Strings**
- **Matrices**
- **Estructuras**
 - **Estructuras de campo de bit**
 - **Uniones**

Vectores recordando que son...

Nombre del vector: Temp

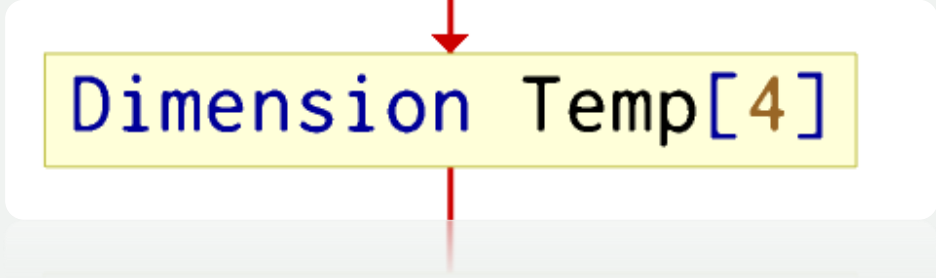
Posee 4 espacios de memoria, el índice que figura entre paréntesis marca a que espacio nos referimos, desde 0 a 3

El espacio del índice puede ser un valor fijo, o utilizar una variable para cambiar el acceso.

Vector: Temp	Contenido
Temp(0)	20
Temp(1)	25
Temp(2)	30
Temp(3)	10

Vectores en Pseint

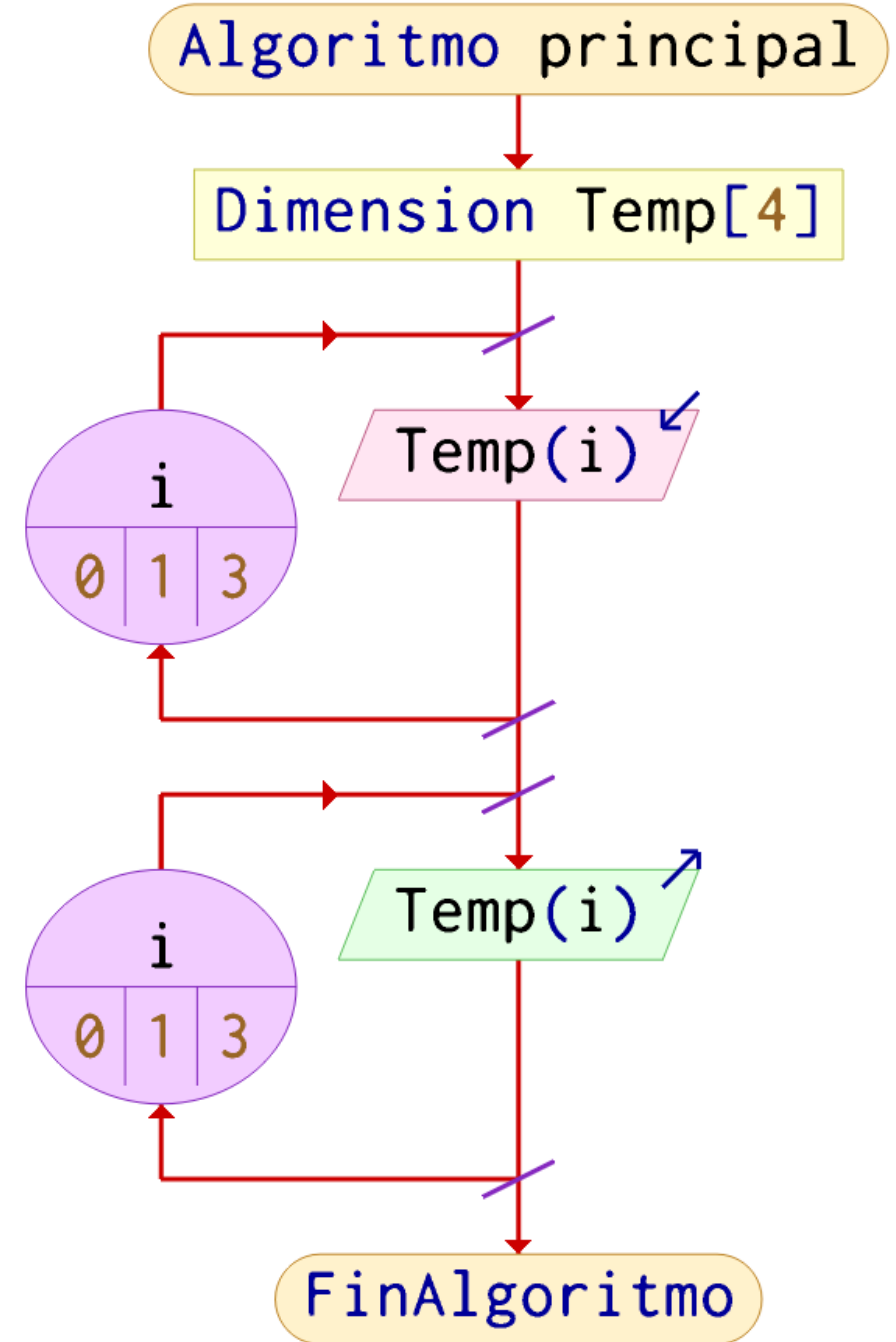
Antes de utilizar los vectores debemos indicar su dimensión, en el ejemplo vemos que se declara un Vector de dimensión 4 llamado Temp que va desde Temp(0) hasta Temp(4)



```
Dimension Temp[4]
```

Vectores en Pseint

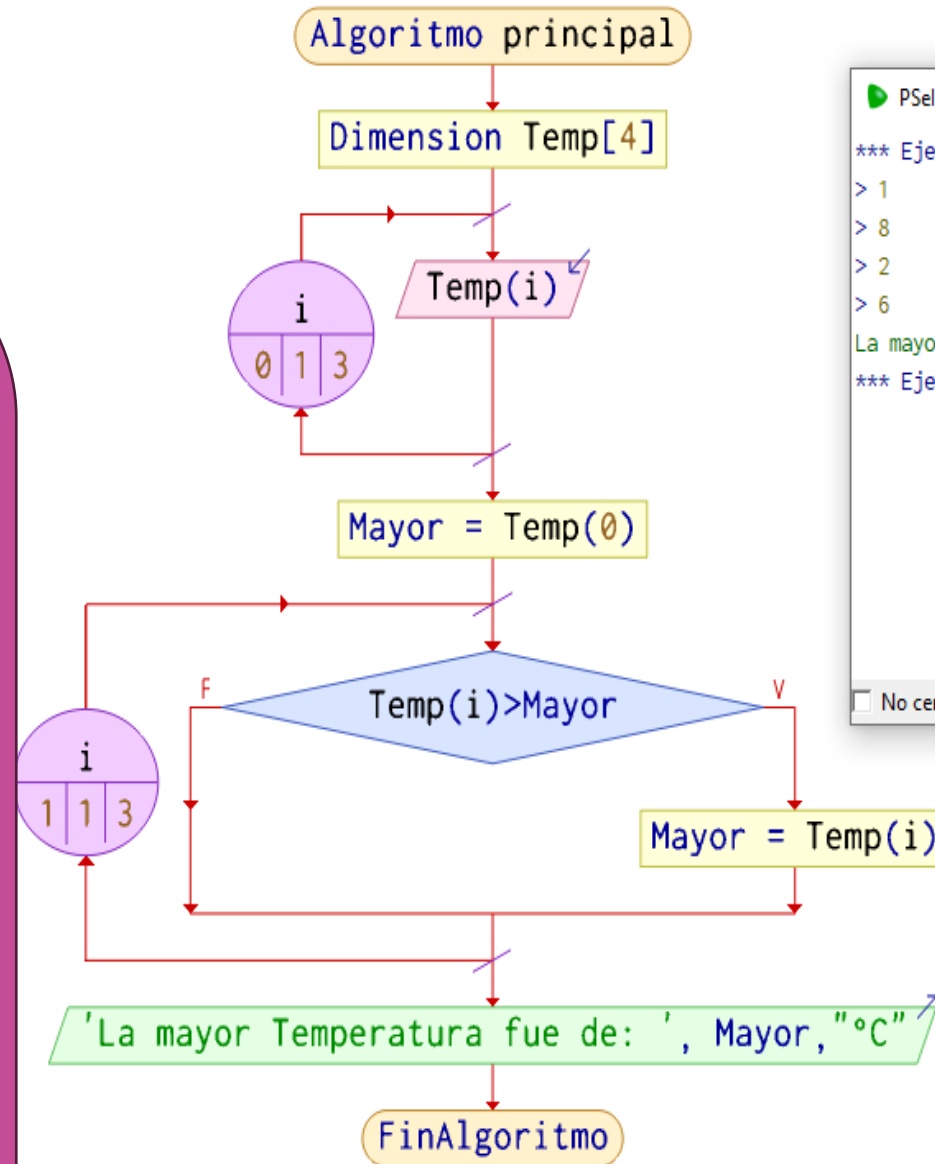
En el siguiente ejemplo se utiliza un ciclo for para ingresar 4 temperaturas y luego se muestran en pantalla con otro ciclo for.



Vectores en Pseint

El siguiente programa muestra en pantalla la mayor temperatura ingresada

- Primero se ingresan las 4 temperaturas
- Luego se busca en el vector cual fue la mayor
- Nota importante, fijarse que el vector se recorre de la 1 a la 3, dado que se asume que cuando hago la primera lectura, al no haber otra temperatura, esa es en primera instancia la mayor



```
PSeint - Ejecutando proceso PRINCIPAL
*** Ejecución Iniciada. ***
> 1
> 8
> 2
> 6
La mayor Temperatura fue de: 8°C
*** Ejecución Finalizada. ***
☐ No cerrar esta ventana ☐ Siempre visible
```

Codificando vectores

- Debemos entender que la estructura de un vector, agrupa muchos tipos de variables juntos, bajo el nombre del vector, que además podemos diferenciar cada variable utilizando un índice.
- En C se trabaja con el índice de 0 en adelante.
- Para declarar un vector primero tenemos que definir el tipo de memoria que será cada variable del vector (todas serán del mismo tipo)
- En lugar de utilizar paréntesis () se utilizan corchetes []
- `unsigned int mivector[10];`
- Con la siguiente línea hemos declarado 10 espacios de memoria int, bajo el nombre mivector, podemos indexar cada espacio con el índice, y tendremos desde `mivector[0]`, hasta `mivector[9]`. 10 espacios de memoria en total.
- El nombre del vector mivector, hace referencia a la dirección donde se encuentra la primer posición de memoria del vector
- `unsigned char mivector2[5];`
- Creamos un vector de 5 posiciones (todas del tipo char), que va desde `mivector2[0]` hasta `mivector2[4]`.

Inicializando el vector

- Como toda variable hay que declararla, y al declararla (como vimos en el slide anterior), podemos además inicializar las variables.
- `unsigned char mivector[4]={0x00,0x01,0x55,0x33};`
- Así damos un valor inicial a cada memoria del vector:

Vector: mivector	Contenido
mivector[0]	0x00
mivector[1]	0x01
mivector[2]	0x55
mivector[3]	0x33

Inicializando el vector

- Como hemos inicializado el vector, no es necesario decirle al vector la cantidad de posiciones que tendrá, es válido también declarar un vector de la siguiente manera:
- **unsigned char mivector[]={0x00,0x01,0x55,0x33};**
- El compilador sabrá que dicho vector es de 4 posiciones.

Vectores de caracteres... strings

- Muchas veces trabajaremos con cadenas de caracteres, strings.
- Las cadenas de caracteres son vectores del tipo char que poseen en su interior un valor que es asociado de acuerdo al formato ASCII a un carácter. Siempre todos los vectores String están terminados en el carácter NULL (0)
- Podríamos escribir:
- `unsigned char mistring[]={ 'H','o','l','a',0 }`
- Por simplicidad la notación anterior es igual a escribir
- `unsigned char mistring[]="Hola"`
- El vector anterior es un vector que posee 5 posiciones, no debemos olvidar el carácter NULL que es agregado automáticamente por el compilador.

Como se ve en memoria el string

Posición de memoria	Mistring	Contenido visto en ASCII	Contenido visto en Hexa
mistring	mistring[0]	'H'	0x48
mistring+1	mistring[1]	'o'	0x6f
mistring+2	mistring[2]	'l'	0x6c
mistring+3	mistring[3]	'a'	0x61
mistring+4	mistring[4]	NULL	0x00

No sabemos cuál es el valor mistring (la posición de memoria) pero sabemos que cada elemento del vector estará una posición de memoria delante de otra y que la palabra mistring es la posición de memoria.

Vectores y String constantes

- En algunas ocasiones, creamos un vector como tabla de datos o un string para mostrar en algún pantalla que no se va a modificar, solo lo usaremos para leerlo.
- Recordemos que el vector en sí es una variable por lo tanto puede ser afectada los modificadores y CONST lo es
- Definiendo el vector o string de la siguiente forma:
- `const unsigned char mistring[]="Hola";`
- `const unsigned int mivector[]={0x01,0x02,0x03};`
- No sabemos donde este vector y string van a estar alojados, pero si al declararlos los constantes sabremos que estarán en memoria de programa y no en memoria de datos. Siempre recordemos que en TODO sistema computacional es menor la cantidad de memoria de datos que la memoria de programa, por lo tanto estamos optimizando recursos

Consiga:

- Realizar una función que compare dos strings (de la misma longitud, definidos como variables globales) y devuelva 0 si son iguales o 1 si son distintos
- Realizar una función para pasar a mayúscula y otra a minúscula un string definido como global. Observar las características de las mayúsculas y minúsculas en la tabla ASCII

Ordenamiento

Hacer un programa que ordene un vector que contiene números de menor a mayor (considerar un vector de 10 elementos)

Algoritmos de ordenamiento

- **Ordenamiento Burbuja (Bubblesort)**
- **Ordenamiento por Selección**
- **Ordenamiento por Inserción**

Ordenamiento Burbuja (Bubblesort)

Este es el algoritmo más sencillo probablemente. Ideal para empezar. Consiste en recorrer el vector repetidamente, comparando elementos adyacentes de dos en dos. Si un elemento es mayor que el que está en la siguiente posición se intercambian.

Ejemplo

```
#include <stdio.h>
#define TAM 10
int lista[TAM]={5,6,1,7,12,8,10,22,2,3};

int main (void){
    int i,j,temp;
    for (i=1; i<TAM; i++){
        for (j=0 ; j<(TAM-1); j++){
            if (lista[j] > lista[j+1]){
                temp = lista[j];
                lista[j] = lista[j+1];
                lista[j+1] = temp;
            }
        }
    }
    for(i=0;i<TAM;i++)printf("%d \t",lista[i]);
    return 0;
}
```

Ordenamiento por Selección.

- Buscar el elemento más pequeño de la lista.
- Intercambiar con el elemento ubicado en la primera posición de la lista.
- Buscar el segundo elemento más pequeño de la lista.
- Intercambiar con el elemento que ocupa la segunda posición en la lista.
- Repetir este proceso hasta que se haya ordenado toda la lista.

Ejemplo

```
#include <stdio.h>
#define TAM 10
int lista[TAM]={5,6,1,7,12,8,10,22,2,3};

int main (void){
    int i,j,temp,pos_men,menor;
    for (i=0 ; i<(TAM); i++){
        menor=lista[i];
        pos_men=i;
        for(j=i;j<TAM;j++){
            if(lista[j]<menor){
                menor=lista[j];
                pos_men=j;
            }
        }
        temp=lista[i];
        lista[i]=lista[pos_men];
        lista[pos_men]=temp;
    }
    for(i=0;i<TAM;i++)printf("%d \t",lista[i]);
    return 0;
}
```

Ordenamiento por Inserción

Este algoritmo también es bastante sencillo. ¿Has jugado cartas?. ¿Cómo las vas ordenando cuando las recibís? Yo lo hago de esta forma: tomo la primera y la pongo en mi mano. Luego tomo la segunda y la comparo con la que tengo: si es mayor, la pongo a la derecha,). Después tomo la tercera y la comparo con las que tengo en la mano, desplazándola hasta que quede en su posición final. Continúo haciendo esto, insertando cada carta en la posición que le corresponde, hasta que las tengo todas en orden.

Para simular esto en un programa necesitamos tener en cuenta algo: no podemos desplazar los elementos, así como así o se perderá un elemento. Lo que hacemos es guardar una copia del elemento actual (que sería como la carta que tomamos) y desplazar todos los elementos mayores hacia la derecha. Luego copiamos el elemento guardado en la posición del último elemento que se desplazó.

Ejemplo

```
#include <stdio.h>
#define TAM 10
int lista[TAM]={5,6,1,7,12,8,10,22,2,3};

int main (void){
    int i,j,temp;
    for (i=0 ; i<TAM; i++){
        temp=lista[i];
        j=i-1;
        while((lista[j]>temp)&&(j>=0)){
            lista[j+1]=lista[j];
            j--;
        }
        lista[j+1]=temp;
    }
    for(i=0;i<TAM;i++)printf("%d \t",lista[i]);
    return 0;
}
```

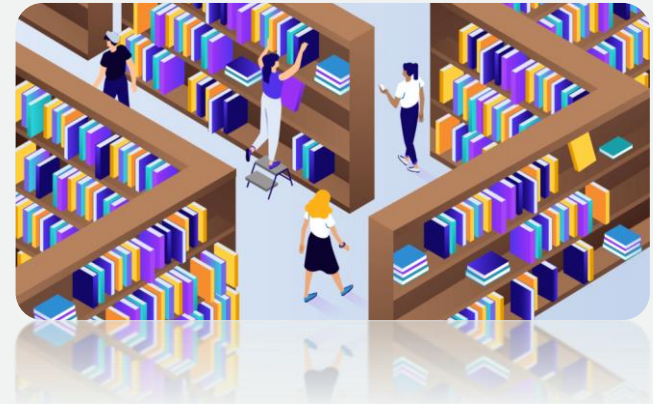
Bibliotecas útiles

- Ver el link y las funciones de ctype.h, string.h y math.h

<https://www.ibm.com/docs/es/i/7.5?topic=files-ctypeh>

<https://www.ibm.com/docs/es/i/7.5?topic=files-stringh>

<https://www.ibm.com/docs/es/i/7.5?topic=files-mathh>



Matrices

- En algunas ocasiones necesitamos trabajar con datos multidimensionales, si pensamos al vector como una columna de información, podemos agregarle otra columna más y un índice para indicar a que columna nos referimos, entonces.

<i>H</i>	<i>O</i>	<i>L</i>	<i>A</i>	<i>0</i>	
<i>H</i>	<i>E</i>	<i>L</i>	<i>L</i>	<i>O</i>	<i>0</i>

Matrices

- En algunas ocasiones necesitamos trabajar con datos multidimensionales, si pensamos al vector como una columna de información, podemos agregarle otra columna más y un índice para indicar a que columna nos referimos, entonces...
- `unsigned char mimatriz[2][2]={0x01,0x02,0x03,0x04};`
- Es necesario incluir el segundo subíndice para que el compilador sepa como trabajamos la matriz, el primero indica la fila y el segundo la columna (aunque en realidad la concepción de fila y columna es nuestra, puede estar al revés)
- `unsigned char mimatriz[1][4]={0x01,0x02,0x03,0x04};` es la declaración de un vector de una sola fila

Matrices

unsigned char mimatriz[filas][columnas]={0x01,0x02,0x03,0x04}

fila\column	0	1
a		
0	0x01	0x02
1	0x03	0x04

Matrices

Las matrices son muy útiles a la hora de trabajar con secuencias, dado que puedo tener en las columnas la secuencia constante de trabajo y cambiando el índice de columna cambio entre una y otra secuencia cambiando solamente una secuencia.

Secuencias

fila\columna	0	1
0	0x01	0x02
1	0x03	0x04
2	0x55	0x66
3	0xAA	0x3a

Código

```
#include <stdio.h>
unsigned char mimatriz[2][4]={
    0x01,0x03,0x55,0xaa,
    0x02,0x04,0x66,0x3a
};

unsigned char i,secuencia=0;

int main (void){

    for(i=0;i<4;i++)printf("0x%x\n",mimatriz[secuencia][i]);
    return 0;
}
```

Quando la variable secuencia valga 0 veremos en pantalla 1,3,85 y 170
Si secuencia vale 1 veremos 2,4,102 y 58

Código

```
#include <stdio.h>
unsigned char mimatriz[2][4]={
    {0x01,0x03,0x55,0xaa},
    {0x02,0x04,0x66,0x3a}
};

unsigned char i,secuencia=0;

int main (void){

    for(i=0;i<4;i++)printf("0x%x\n",mimatriz[secuencia][i]);
    return 0;
}
```

No es necesario, pero algunos agregan llaves para separar cada secuencia y que sea más claro donde comienza uno y donde termina la otra.

Estructuras

En algunas ocasiones nos interesa agrupar datos bajo una estructura determinada.
#ejemplo, si vamos a trabajar con puntos, sabemos que en un sistema cartesiano posee una coordenada en x y otra en y.

Deberíamos entonces declarar por cada punto una coordenada x y otra y.

Estructuras

- unsigned char Punto1coordx;
- unsigned char Punto1coordy;
- unsigned char Punto2coordx;
- unsigned char Punto2coordy;

Muy incómodo de trabajar!, imaginen si tuviéramos muchos puntos!

Estructuras

Podemos entonces crear la estructura de un punto que en su interior contenga dos variables del tipo unsigned char coordX y unsigned char coordY

Código

```
struct puntos {  
    unsigned char coordX;  
    unsigned char coordY;  
  
};
```

Esta declaración no genera código, todavía esta estructura no fue asignada a ninguna variable.

Generando variables del formato de nuestra estructura

```
struct puntos {  
    unsigned char coordX;  
    unsigned char coordY;
```



```
}punto1,punto2;
```

```
struct {  
    unsigned char coordX;  
    unsigned char coordY;
```

```
}punto1,punto2;
```

De esta manera creamos dos variables punto1 y punto2 de forma que cada una contiene dos variables coordX y coordY. Cuando se declaran así las variables no es necesario el nombre de la estructura punta, como vemos a la derecha.

Generando variables del formato de nuestra estructura


```
struct puntos {  
    unsigned char coordX;  
    unsigned char coordY;  
  
};  
  
int main (void){  
    struct puntos punto1,punto2;  
  
    /*  
     micodigo  
    */  
  
}
```

Cuando queremos declarar variables dentro de una función necesitamos que la estructura tenga nombre y declaramos las variables punto1 y punto2 de esta manera.

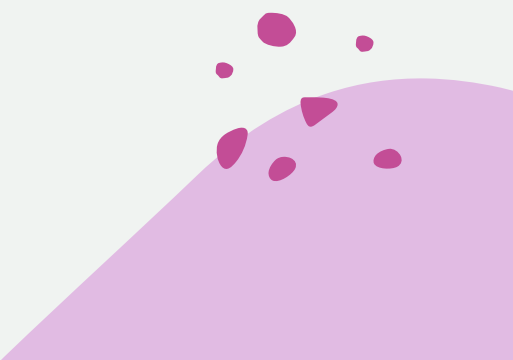
Trabajando con los campos...

```
struct puntos {  
    unsigned char coordX;  
    unsigned char coordY;  
  
};  
  
int main (void){  
  
    struct puntos punto1,punto2;  
  
    punto1.coordX=0;  
    punto1.coordY=0;  
    punto2.coordX=2;  
    punto2.coordY=3;
```

Con el operador `.` Trabajamos con los campos de cada estructura.



Diseñar una estructura de datos para un sistema de alumnos de colegios con al menos 5 campos



Typedef

- En C podemos definir nuevos tipos de variables con la palabra Typedef de modo que ahora nuestra estructura sea un nuevo tipo de dato

Typedef

```
#include <stdio.h>

typedef struct puntos {
    unsigned char coordX;
    unsigned char coordY;
}punto;

int main (void){

    punto punto1,punto2;
    punto1.coordX=3;
    punto1.coordY=2;
    printf("La coordenada X del punto 1 vale: %d\n",punto1.coordX);
    printf("La coordenada Y del punto 1 vale: %d\n",punto1.coordY);
    return 0;
}
```

Definimos un nuevo tipo de dato llamado Punto e instanciamos 2 variables con los nombres punto1 y punto2

Vectores y estructuras

```
struct puntos {
    unsigned char coordX;
    unsigned char coordY;
}punto[10];

int main (void){

    unsigned char i;

    for(i=0;i<10;i++){
        punto[i].coordX=0;
        punto[i].coordY=0;
    }
    /*
    micodigo
    */
}
```

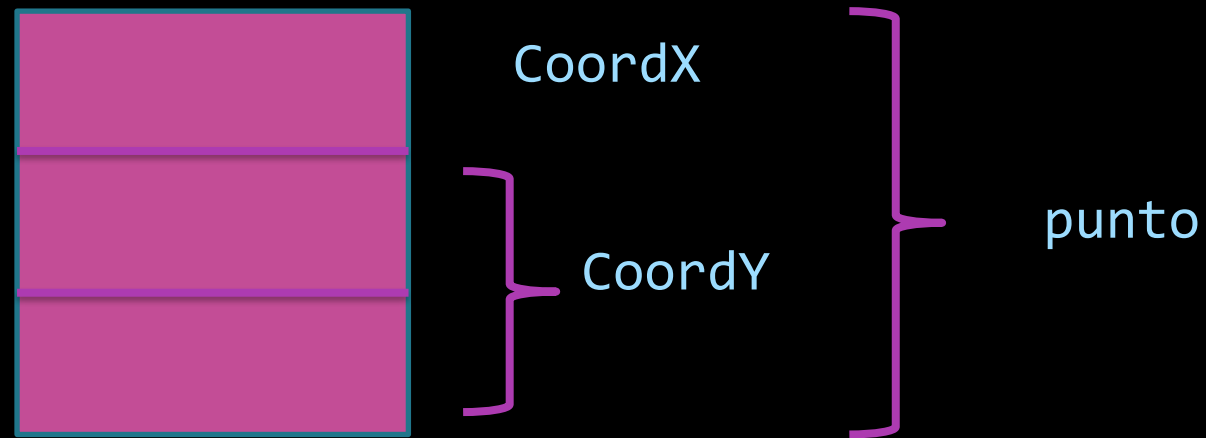
Con el siguiente código generamos un vector punto con 10 elementos, cada elemento tiene su campo coordX y coordY. El el ciclo for inicializamos todos las corrdenadas (x e y) en 0,0

Uniones

- Las estructuras vistas hasta ahora poseen cada un espacio de memorias diferentes de acuerdo al tipo de variable.
- Podemos definir un tipo de estructura llamada **unión** en donde todos los elementos (espacios de memoria) comienzan en la misma posición.

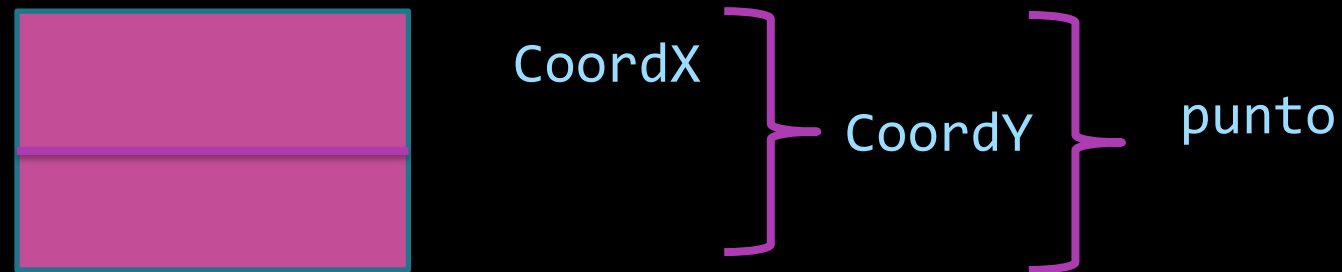
Uniones

```
struct puntos {  
    unsigned char coordX;    //variable de 1 byte  
    unsigned short int coordY; //variable de 2 bytes  
}punto;    //variable de 3 bytes
```



Uniones

```
union puntos {  
    unsigned char coordX;    //variable de 1 byte  
    unsigned short int coordY; //variable de 2 bytes  
  
}punto;    //variable de 3 bytes
```



Coord X y CoordY empiezan en la misma dirección de memoria
(compartida por ambos miembros de la estructura)

sizeof()

- sizeof pese a parecer una función es un operador de C y C++ para determinar el tamaño (en bytes) de una variable. Podemos entonces utilizar:

```
int main (void){  
    printf("La variables char ocupan: %d bytes\n",sizeof  
(char));  
    printf("La variables int ocupan: %d bytes\n",sizeof(  
int));  
    return 0;  
}
```

Sizeof()


- Con Sizeof verificar el tamaño de todas las variables con sus modificadores
- Unsigned short int
- Unsigned long int etc...

sizeof()

- Con las estructuras y las uniones podemos verificar el tamaño de cada variable.

Ejercicio razonar la salida del programa

```
typedef struct puntos {  
    unsigned char coordX;        //variable de 1 byte  
    unsigned short int coordY;   //variable de 2 bytes  
  
}punto;        //variable de 3 bytes  
  
int main (void){  
  
    punto punto1,punto2;  
    punto1.coordX=3;  
    punto1.coordY=2;  
    printf("La coordenada X del punto 1 vale: %d\n",punto1.coordX);  
    printf("La coordenada Y del punto 1 vale: %d\n",punto1.coordY);  
    printf("La variable punto1 ocupa %d bytes\n",sizeof(punto));  
    return 0;  
}
```

¿Imprimió el programa anterior lo esperado de `sizeof()` en caso negativo investigar por qué?

Estructura de campo de bits

- Los campos de bits solo pueden existir en estructuras y nos permite utilizar variables de bit que nosotros definamos

Ejercicio razonar la salida del programa

```
struct encoder {  
    unsigned int encoderCuenta : 23;  
    unsigned int encodervueltas : 4;  
};
```

En este caso definimos una estructura encoder que posee dos campos, el primero encoderCuenta posee 23 bits y el segundo encodervueltas 4. Prestar atención que siempre se coloca unsigned int a la izquierda pese a que trabajamos con tamaños establecidos por los bits colocados luego de los :

Pensar para que podría usar una unión y una estructura de campo de bits juntos

Uniones anónimas

```
union {  
    struct {  
        unsigned int bit0 : 1;  
        unsigned int bit1 : 1;  
        unsigned int bit2 : 1;  
        unsigned int bit3 : 1;  
        unsigned int bit4 : 1;  
        unsigned int bit5 : 1;  
        unsigned int bit6 : 1;  
        unsigned int bit7 : 1;  
    };  
    unsigned char byte;  
}variable;
```

En este caso no es necesario darle una variable ni nombre a la estructura de campo de bits ni a la unión y para acceder así al bit0 escribimos

variable.bit0

Utilizando macros

- `#define bandera1 variable.bit0`
- `#define bandera2 variable.bit1`
- `#define bandera3 variable.bit2`
- `#define bandera4 variable.bit3`
- `#define bandera5 variable.bit4`
- `#define bandera6 variable.bit5`
- `#define bandera7 variable.bit6`
- `#define bandera8 variable.bit7`

Ejercicio

*Utilizar una unión para poder ver el contenido de los **4 bytes** dentro de una variable **float** de a uno.*

Resolución

```
union {  
    float valor;  
    unsigned char bytes[4];  
}var;
```

```
var.valor=-15;  
printf("%x\n",var.bytes[0]);  
printf("%x\n",var.bytes[1]);  
printf("%x\n",var.bytes[2]);  
printf("%x\n",var.bytes[3]);
```


Punteros

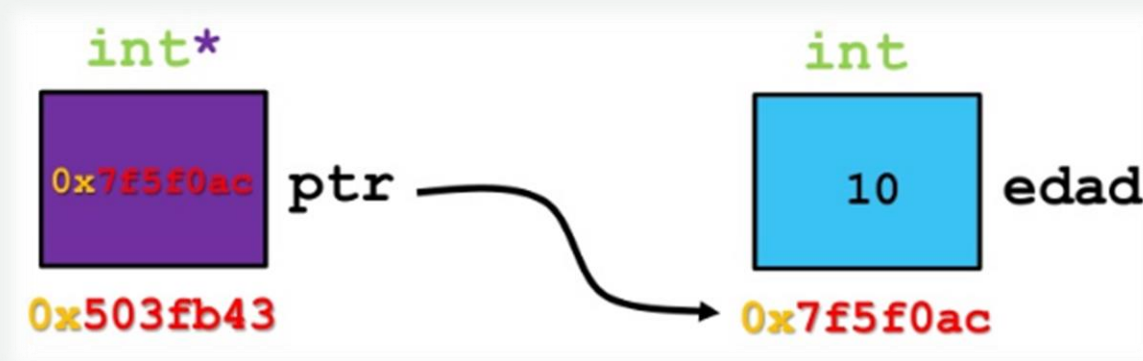
- Algunos dicen que existen dos tipos de lenguajes
 - Los que tienen puntero
 - Los que no

iiiiC Tiene punteros!!!!



Puntero definición

- Un puntero es un tipo de variable que contiene una dirección y que nos brinda la posibilidad de trabajar con la dirección tanto como con el contenido de dicha posición (apuntada)



Declaración de un puntero

□ `TIPO * nombre_puntero; //TIPO es el tipo de dato al que apunta el puntero`

#ej

`char *pchar; //creamos una variable que apunta a una variable de tipo char.`

Los punteros no puede no inicializarse (mala práctica) por tanto debemos hacerlo apuntar a alguna variable.

`char var;`

`char *pchar=&var;`

Creamos una variable var y un puntero pchar que apunta a dicha variable

#Pregunta

¿Cómo podemos ver la cantidad de bytes que la variable pchar ocupa en memoria?

Hacer el código.

*Operadores *y &*

El operador **&** obtiene la dirección de la variable que está al lado. En el ejemplo anterior le asignamos al puntero la dirección de la variable var. Como cuando hacemos un scanf y le pasamos como parámetro la dirección donde se encuentra la variable que va a guardar lo ingresado

*Operadores *y &*

El operador * utilizado al lado de una variable del tipo puntero devuelve el contenido de la variable apuntada por dicho puntero.

```
char var=3;
```

```
char *pchar=&var;
```

```
printf("%d\n",*pchar); //imprime el contenido de a donde apunta pchar, en  
este caso var3
```

Aritmética de punteros

- Los punteros son variables y como tales, pueden incrementarse, decrementarse, sumarse etc.
- Cuando incrementamos un puntero como
- `p++` este se incrementa de acuerdo al tipo de puntero que es, es decir un `(char *p)` se incrementa 1 byte, mientras que un `(short int *p)` se incrementa 2 bytes

Aritmética de punteros

- **`a=*(p++);`**
- **En este caso recordemos que se hace un post incremento, por tanto primero asigna a la variable a el valor del contenido de la dirección apuntada por p y luego se incrementa en de acuerdo al tipo de dato al que apunte.**

Ejercicio

Utilizar un puntero para poder ver el contenido de los 4 bytes dentro de una variable float de a uno.

Resolución

```
float mivar=-15;  
unsigned char *p=(unsigned char *)&mivar;  
printf("0x%x\n",*p);  
printf("0x%x\n",*(p+1));  
printf("0x%x\n",*(p+2));  
printf("0x%x\n",*(p+3));
```

¿Qué es el (unsinged char *) que aparece al lado del puntero? Que pasa si no se coloca?

Resolución v2

```
float mivar=-15;  
unsigned char *p=(unsigned char *)&mivar;  
printf("0x%x\n",p[0]);  
printf("0x%x\n",p[1]);  
printf("0x%x\n",p[2]);  
printf("0x%x\n",p[3]);
```

Los punteros se pueden indexar como vectores

Punteros indexados

Esto se puede hacer, porque si recordamos en los vectores el nombre del vector es la dirección de comienzo de todo el vector en memoria.

Cast (molde)

- Cuando convertimos entre diferentes tipos de datos, por ejemplo

```
int var1=0xffff;  
char var2;  
var2=var1;
```

Cast (molde)

```
int var1=0xffff;  
char var2;  
var2=(char)var1;
```

En el caso del puntero asignábamos un variable float (su dirección) a un puntero a char, por eso colocamos el cast indicando la operación que queremos hacer

Cast (molde)

- En estos casos el compilador puede dar un warning, avisando que podemos perder datos, estos casos le indicamos al compilador que la operación es intencional indicándole el tipo de dato destino.

Ejercicio

Hacer un código que posea una función que reciba una variable, la incremente, la imprima en pantalla y luego al volver a la función principal vuelva a imprimir dicha variable.

¿Este código modifica la variable original de la función principal cuando la incrementamos en la función?

Código para probar:

```
#include <stdio.h>

void mifunc(char mivar);

int main (void){
    char mivarlocal=3;
    mifunc(mivarlocal);
    printf("mivarlocal en main vale: %d\n",mivarlocal);
    return 0;
}

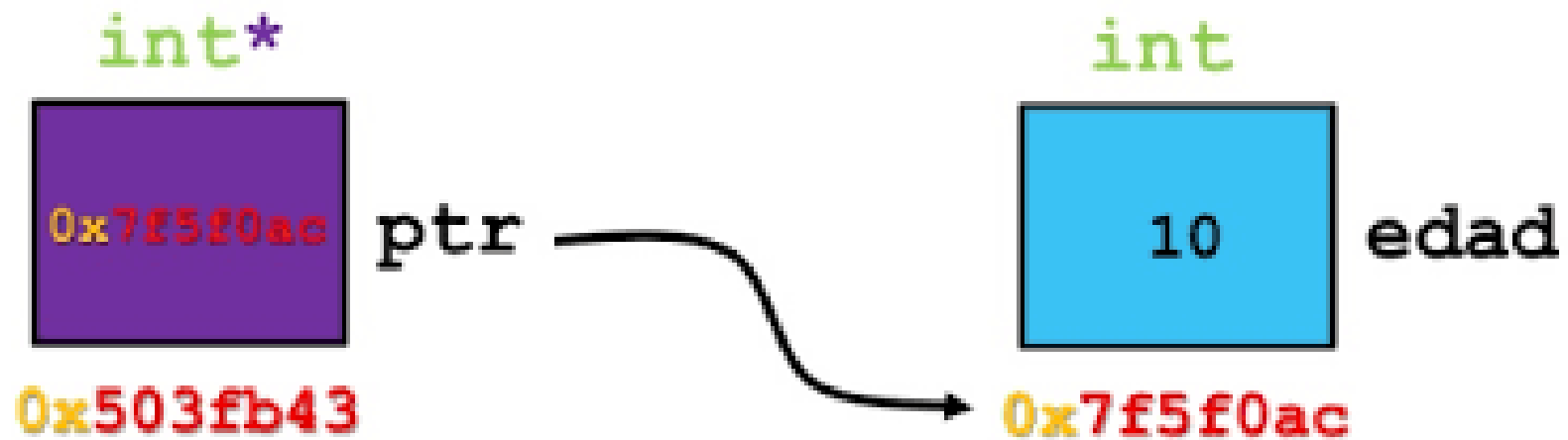
void mifunc(char mivar){
    mivar++;
    printf("mi var en mifunc vale: %d\n",mivar);
}
```

Pasaje de parámetros por valor o por referencia

- En el código anterior vemos que cuando pasamos un parámetro a una función, pasamos un valor, una copia de dicho valor, pero no la variable en si misma.
- Para solucionar esto debemos pasarle la referencia de donde está ubicada la variable para que la pueda modificar

¿Qué debemos utilizar si queremos pasarle a la función la ubicación de nuestra variable?

!!!Pasarle un puntero!!!



Código

```
#include <stdio.h>

void mifunc(char* mivar);

int main (void){
    char mivarlocal=3;
    mifunc(&mivarlocal);
    printf("mivarlocal en main vale: %d\n",mivarlocal);
    return 0;
}

void mifunc(char* mivar){
    (*mivar)++;
    printf("mi var en mifunc vale: %d\n",*mivar);
}
```

¿Qué pasa con mivar cada vez que entro mifunc()?

```
#include <stdio.h>

void mifunc(void);

int main (void){

    mifunc();
    mifunc();
    mifunc();
    mifunc();
    return 0;
}

void mifunc(void){
    int mivar=0;
    printf("%d\n",mivar);
    mivar++;
}
```

¿Como puedo hacer para que mi var conserve el valor al entrar y salir de la función?

Variables static

Cuando entramos a una función y dicha función posee variables locales, estas son creadas nuevamente y eliminadas del programa cuando nos vamos. Si queremos usar una variable local y que mantenga su valor cuando entramos y salimos de la función le agregamos la palabra reservada `static`

Consiga:

Realizar una función que compare dos strings (pasados como parámetros) y devuelva 0 si son iguales o 1 si son distintos.

Realizar una función para pasar a mayúscula y otra a minúscula un string pasado como parámetro. Observar las características de las mayúsculas y minúsculas en la tabla ASCII

Probar el siguiente código con o sin static

```
#include <stdio.h>

void mifunc(void);

int main (void){

    mifunc();
    mifunc();
    mifunc();
    mifunc();
    return 0;
}

void mifunc(void){
    static int mivar=0;
    printf("%d\n",mivar);
    mivar++;
}
```

Variables volatile

Cuando hay variables que son modificadas externamente (por ejemplo por un pulsador, un sensor etc) debemos declararlas volatile, de esta manera el compilador no optimizará el código, dejando las lecturas donde nosotros las ubicamos y sabrá que dichas lecturas vendrán de agentes externos

Punteros a estructuras

```
#include <stdio.h>
struct datos{
    int a;
    int b;
}estructura;

int main(){
    struct datos *p=&estructura; //creamos un puntero (p) que apunta a estructura
    estructura.a=10; //modificamos el valor de a
    p->b=10; //modificamos el valor de b a traves del puntero
    return 0;
}
```

Punteros a estructuras

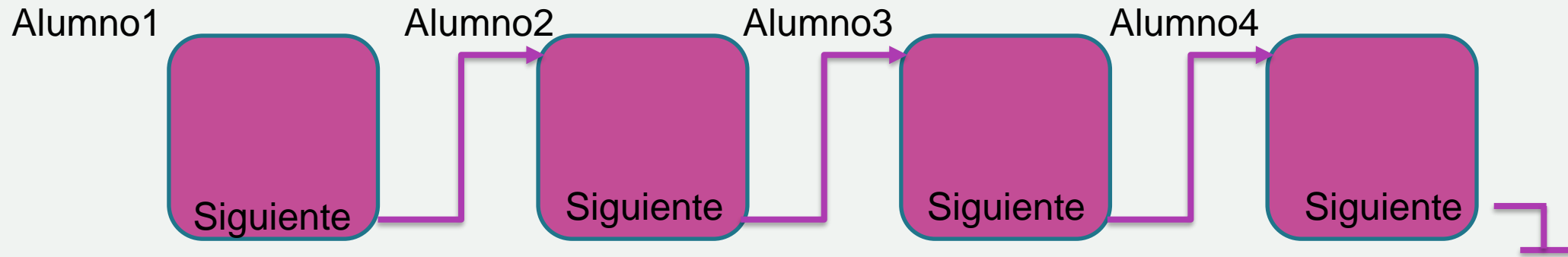
`(*p).b=10;`



`p->b=10`

Punteros a estructuras dentro de estructuras

(Lista simplemente enlazada)



Lista simple enlazada

```
#include <stdio.h>
struct alumnos{
    unsigned char* nombre;
    unsigned char* apellido;
    unsigned int edad;
    unsigned int legajo;
    unsigned int promedio;
    struct alumnos *siguiente;
}alumno1,alumno2,alumno3,alumno4;
```

```
int main(){
    struct alumnos *p=&alumno1;
    alumno1.siguiente=&alumno2;
    alumno2.siguiente=&alumno3;
    alumno3.siguiente=&alumno4;
    alumno4.siguiente=NULL;
    alumno1.nombre="Israel";
    alumno1.apellido="Pavelek";
    alumno2.nombre="Eduardo";
    alumno2.apellido="Perez";
    alumno3.nombre="Juan";
    alumno3.apellido="Lopez";
    alumno4.nombre="Agustina";
    alumno4.apellido="Sanchez";
    while(p->siguiente!=NULL){
        printf("Nombre del alumno: %s \n",p->nombre);
        printf("Apellido del alumno: %s \n ",p->apellido);
        p=p->siguiente;
    }
    printf("Nombre del alumno: %s \n",p->nombre);
    printf("Apellido del alumno: %s \n ",p->apellido);
    return 0;
}
```

Actividad:

Pensar una función que le pasemos un legajo y traiga e imprima los datos de dicho alumno.

Asignación dinámica de la memoria

- La creación de las memorias, hasta el momento, queda limitada al momento de la creación del programa. ¿Qué pasa si queremos crear un nuevo alumno?
- Debemos entonces manejar dinámicamente la creación de nuevas variables.

malloc

- En la librería stdlib.h existen una serie de funciones para el trabajo de reserva de memoria de forma dinámica

Función	Descripción
malloc	Asigna el número especificado de bytes
realloc	Aumenta o disminuye el tamaño del bloque de memoria especificada. Reasigna si es necesario.
calloc	Asigna el número especificado de bytes y los inicializa en cero
free	Libera el bloque de memoria especificada de nuevo al sistema

malloc

- Ejemplo

- ```
int * array = malloc(10 * sizeof(int));
```

-

## *Ejemplo*

*En lugar de tener 4 alumnos fijos o 100 esperando ser asignados (por las dudas) hagamos el ejemplo anterior en donde a medida que es necesario se reserva la memoria.*

```

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

typedef struct alumnos{
 char *nombre;
 char *apellido;
 unsigned int edad;
 unsigned int legajo;
 float promedio;
 struct alumnos *siguiente;
}alumno;

typedef alumno *puntero_lista;

void imprimir_todo(puntero_lista comienzo);
void borrar_lista(puntero_lista *comienzo);
void crear_alumno(puntero_lista *comienzo);

int main(){
 char caract;
 puntero_lista comienzo=NULL;

 do{
 printf("presione \'c\' para crear un alumno \'l\' para listar todos y \'s\' para salir\n");
 do{
 caract=getchar();
 }while((caract!='c') && (caract!='l') && (caract!='s'));
 if(caract=='c')crear_alumno(&comienzo);
 if(caract=='l')imprimir_todo(comienzo);
 }while(caract!='s');
 borrar_lista(&comienzo);
 return 0;
}

```

```

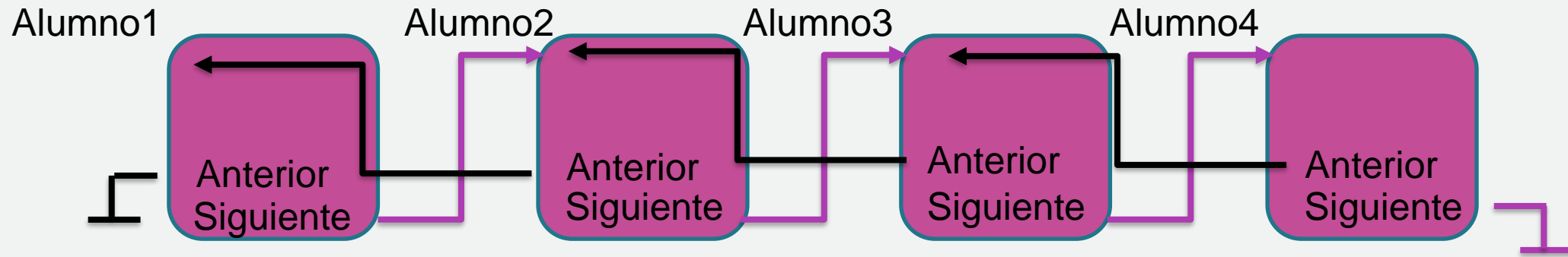
void crear_alumno(puntero_lista *comienzo){
 puntero_lista p= malloc (sizeof (alumno));
 p->siguiente=NULL;
 printf("Ingrese el nombre del nuevo alumno\n");
 char *nombre=(char *)malloc(10);
 scanf("%s",nombre);
 p->nombre=nombre;
 printf("Ingrese el Apellido del nuevo alumno\n");
 char *apellido=(char *)malloc(10);
 scanf("%s",apellido);
 p->apellido=apellido;
 printf("Ingrese la edad del nuevo alumno\n");
 scanf("%d",&(p->edad));
 printf("Ingrese el legajo del nuevo alumno\n");
 scanf("%d",&(p->legajo));
 printf("Ingrese el promedio del nuevo alumno\n");
 scanf("%f",&(p->promedio));
 p->siguiente=*comienzo;
 *comienzo=p;
}

void imprimir_todo(puntero_lista comienzo){
 while(comienzo!=NULL){
 printf("Nombre del alumno: %s \n",comienzo->nombre);
 printf("Apellido del alumno: %s \n",comienzo->apellido);
 printf("Edad del alumno: %d \n",comienzo->edad);
 printf("Legajo del alumno: %d \n",comienzo->legajo);
 printf("promedio del alumno: %f \n\n",comienzo->promedio);
 comienzo=comienzo->siguiente;
 }
}

void borrar_lista(puntero_lista *comienzo){
 puntero_lista actual;
 while(*comienzo!=NULL){
 actual=*comienzo;
 *comienzo=(*comienzo)->siguiente;
 free(actual->nombre);
 free(actual->apellido);
 free(actual);
 }
}

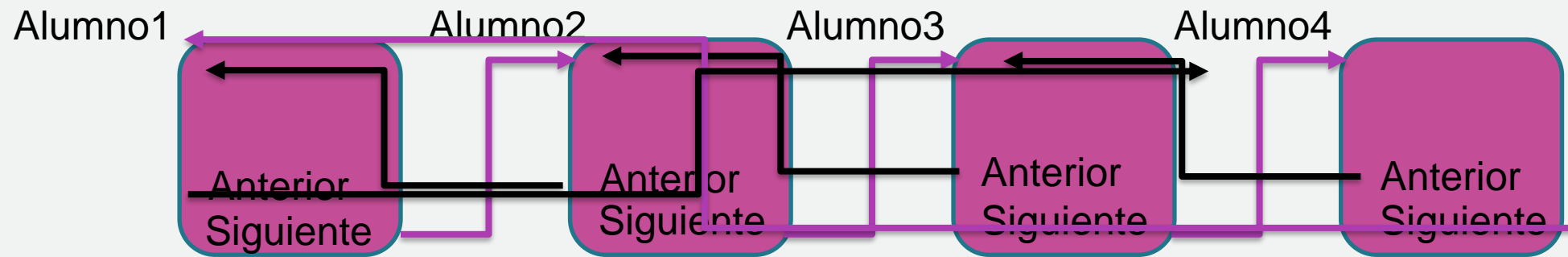
```

# *Listas doblemente enlazadas*



```
typedef struct _nodo {
 int dato;
 struct _nodo *siguiete;
 struct _nodo *anterior;
} tipoNodo;
```

# *Listas doblemente enlazadas circular*



*Probar el siguiente código:*

```
#include <stdio.h>

enum semana {Domingo, Lunes, Martes, Miercoles, Jueves, Viernes, Sabado};

int main(void){
 enum semana hoy;
 hoy = Miercoles;
 printf("Dia %d", hoy+1);
 return 0;
}
```



*¿Qué hace enum?*

# *Recursividad*

Supongamos que debemos realizar la siguiente función matemática:

$n!$  //recordemos que en matemática se llama factorial y  
 $n! = n * (n-1) * (n-2) * \dots$  y que  $1! = 1$  y  $0! = 1$

# *factorial*

```
int main (void) {
 int cont=0,numero=0,factorial=1;

 for (cont=numero;cont >= 1;cont--)factorial*=cont;
 printf("%d\n",factorial);

 return(0);
}
```

# *Recursividad*

El mismo ejemplo puede también resolverse de la siguiente forma:

```
#include <stdio.h>

long factorial(int numero);

int main (void) {

 printf("%d\n",factorial(1));
 return(0);
}

long factorial(int numero){
 if (numero<=1)return 1;
 else return (numero * factorial(numero-1));
}
```

*¿Qué pasa en la función  
factorial?*

## *¿Qué pasa en la función factorial?*

El llamado de una función a sí misma dentro, se llama recursividad, no todos los lenguajes lo admiten, pero en muchos casos soluciona de forma simple problemas más complejos.

# *Archivos*

Los datos que hemos tratado hasta el momento han residido en la memoria principal. Sin embargo, las grandes cantidades de datos se almacenan normalmente en un dispositivo de memoria secundaria. Estas colecciones de datos se conocen como archivos (antiguamente ficheros).

# *Tipos de Archivos*



Archivo de texto



Archivo binario

Archivo de texto

Archivo binario



## *Archivos de texto*

Un archivo de texto es una secuencia de caracteres organizadas en líneas terminadas por un carácter de nueva línea. En estos archivos se pueden almacenar canciones, fuentes de programas, base de datos simples, etc. Los archivos de texto se caracterizan por ser planos, es decir, todas las letras tienen el mismo formato y no hay palabras subrayadas, en negrita, o letras de distinto tamaño o ancho.

## *Archivos binario*

Un archivo binario es una secuencia de bytes. No tendrá lugar ninguna traducción de caracteres.

Ejemplos de estos archivos son Fotografías, imágenes, texto con formatos, archivos ejecutables (aplicaciones), etc.

# *Archivos*

En C, un archivo es un concepto lógico que puede aplicarse a muchas cosas desde archivos de disco hasta terminales o una impresora. Se asocia una secuencia con un archivo específico realizando una operación de apertura. Una vez que el archivo está abierto, la información puede ser intercambiada entre este y el programa.

# *Funciones en Stdio.h para archivos*

| <b>Nombre</b> | <b>Función</b>                                                               |
|---------------|------------------------------------------------------------------------------|
| fopen()       | Abre un archivo                                                              |
| fclose()      | Cierra un archivo                                                            |
| fgets()       | Lee una cadena de un archivo                                                 |
| fputs()       | Escribe una cadena en un archivo                                             |
| fseek()       | Sitúa el puntero de lectura/escritura de un archivo en la posición indicada. |
| fprintf()     | Escribe una salida con formato en el archivo                                 |
| fscanf()      | Lee una entrada con formato desde el archivo                                 |
| feof()        | Devuelve verdadero si se llega al final del archivo                          |
| ferror()      | Devuelve verdadero si se produce un error                                    |
| rewind()      | Coloca el localizador de posición del archivo al principio del mismo         |
| remove()      | Borra un archivo                                                             |
| fflush()      | Vacía el buffer                                                              |

## *Puntero a un archivo*

El puntero a un archivo es el hilo común que unifica el sistema de E/S con buffer. Un puntero a un archivo es un puntero a una información que define varias cosas sobre él, incluyendo el nombre, el estado y la posición actual del archivo. En esencia identifica un archivo específico y utiliza la secuencia asociada para dirigir el funcionamiento de las funciones de E/S con buffer. Un puntero a un archivo es una variable de tipo puntero al tipo FILE que se define en STDIO.H. Un programa necesita utilizar punteros a archivos para leer o escribir en los mismos. Para obtener una variable de este tipo se utiliza una secuencia como esta:

```
FILE *f;
```

## *Abriendo el archivo*

La función `fopen()` abre una secuencia para que pueda ser utilizada y la asocia a un archivo.

Su prototipo es:

```
FILE *fopen(const char nombre_archivo, const char modo);
```

# *Modo de archivo*

| Modo | Significado                                                |
|------|------------------------------------------------------------|
| r    | Abre un archivo de texto para lectura.                     |
| w    | Crea un archivo de texto para escritura.                   |
| a    | Abre un archivo de texto para añadir.                      |
| rb   | Abre un archivo binario para lectura.                      |
| wb   | Crea un archivo binario para escritura.                    |
| ab   | Abre un archivo binario para añadir.                       |
| r+   | Abre un archivo de texto para lectura / escritura.         |
| w+   | Crea un archivo de texto para lectura / escritura.         |
| a+   | Añade o crea un archivo de texto para lectura / escritura. |
| r+b  | Abre un archivo binario para lectura / escritura.          |
| w+b  | Crea un archivo binario para lectura / escritura.          |
| a+b  | Añade o crea un archivo binario para lectura / escritura.  |

## *Cerrando el archivo*

La función `fclose()` cierra una secuencia que fue abierta mediante una llamada a `fopen()`. Escribe toda la información que todavía se encuentre en el buffer en el disco y realiza un cierre formal del archivo a nivel del sistema operativo. Un error en el cierre de una secuencia puede generar todo tipo de problemas, incluyendo la pérdida de datos, destrucción de archivos y posibles errores intermitentes en el programa.

El prototipo de esta función es:

```
int fclose(FILE *F);
```



# *Moviéndose dentro del archivo*

```
fseek(x, 0L, SEEK_SET);
```

- 0 =>SEEK\_SET : Cuenta desde el inicio
- 1 =>SEEK\_CUR: Cuenta desde la posición actual del puntero de archivo
- 2 =>SEEK\_END: Cuenta desde el final del archivo

## *Ejemplos archivos de texto:*

```
#include <stdio.h>

int main(void){
 FILE* fichero;
 fichero = fopen("Lenguajes.txt", "wt");
 fputs("Aprender a programar (linea 1)\n", fichero);
 fputs("requiere esfuerzo (linea 2)\n", fichero);
 fputs("y dedicacion (linea 3)", fichero);
 fclose(fichero);
 printf("Proceso completado");
 return 0;
}
```

## *Ejemplo #2*

```
#include<stdio.h>
// Ingresar caracteres en un archivo hasta [Enter]
//verificar el archivo con el bloc de notas
int main(){
 FILE *archivo;
 char ch;
 char* nombre="texto.txt";// nombre fisico del archivo
 if((archivo=fopen(nombre,"w"))==NULL)
 printf("\n\n***El archivo %s no pudo abrirse ***\n",nombre);
 else{
 printf("\n\n Ingrese caracteres hasta [Enter]");
 while((ch=getchar())!='\n')
 putc(ch,archivo); // almacena en el archivo hasta [ENTER]
 fclose(archivo) ;
 }
}
```

## *Ejemplo #3*

```
#include<stdio.h>

int main(){
 FILE *archivo;
 char ch;
 char* nombre="texto99.txt";// nombre físico del archivo
 if((archivo=fopen(nombre,"r"))==NULL)
 printf("\n\n***El archivo %s no pudo abrirse ***\n",nombre);
 else{
 printf("\n\n Este es el contenido del archivo %s caracter a caracter\n\n",nombre);
 while((ch=getc(archivo))!=EOF)
 printf("%c",ch);
 fclose(archivo);
 }
}
```

## *Ejemplos archivos binarios:*

```
FILE *cli;
char *nomarchcli = "F:\\ARCHIVOS_PRUEBA\\clientes.dat";
system("cls");
if ((cli = fopen(nomarchcli, "rb+")) == NULL){
 cli = fopen(nomarchcli, "wb+");
 printf("\nEl archivo %s ha sido Abierto nuevo!!", nomarchcli);
}else{
 printf("\nEl archivo %s ha sido Abierto para lectura escritura!!",
nomarchcli);
}
```

## *Ejemplos #2*

```
void IngresoClientes(FILE *x){
 regcli c;
 int cod;
 // Inicializar los archivos de Registros
 printf("\nIngrese codigo del empleado[0-para salir:");
 scanf("%d", &cod);
 while (cod){
 fflush(stdin);
 fseek(x, 0L, SEEK_END);
 c.cod_cli = cod;
 printf("\nIngrese nombre:");
 gets(c.nom_cli);
 c.cuenta = 0;
 fwrite(&c, sizeof(c), 1, x);
 fflush(stdin);
 printf("\nIngresar codigo del empleado[0-para salir:");
 scanf("%d", &cod);
 }
}
```

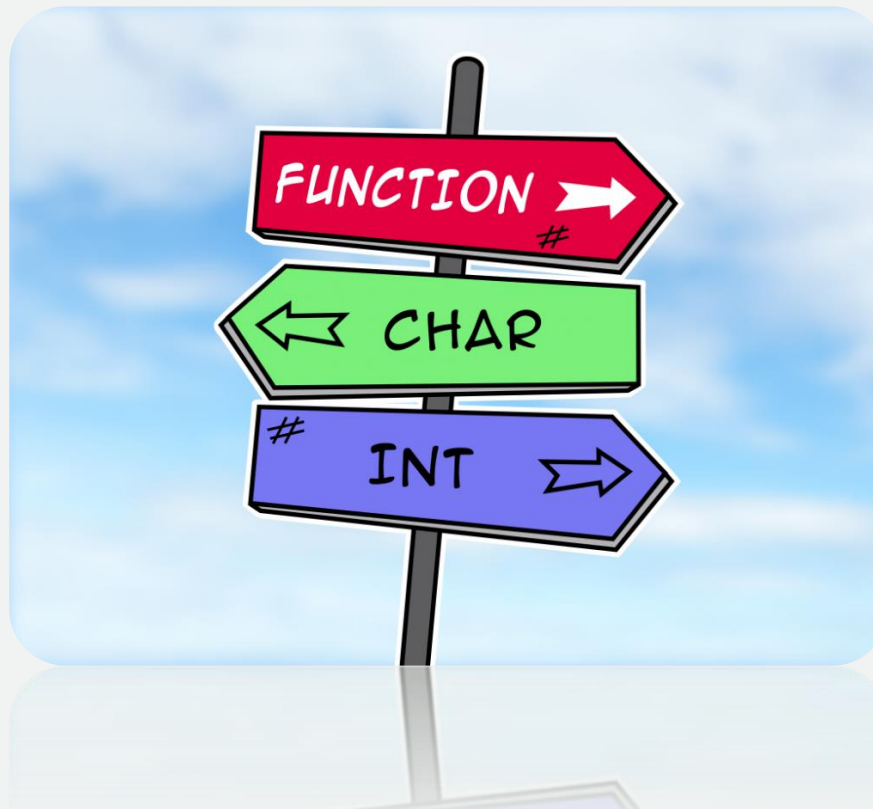
## *Ejemplos #3*

```
void mostrar(FILE *x){
 regcli c;

 fseek(x, 0L, SEEK_SET);
 printf("\n\n\t\t\t Nomina de Clientes\n");
 printf("\nCodigo\tNombre\t\t\t\t\tSaldo\n");
 fread(&c, sizeof(c), 1, x);
 while (!feof(x))
 {
 printf("\n%3d\t%-20s\t\t\t\t\t%6.2f", c.cod_cli, c.nom_cli, c.cuenta);
 fread(&c, sizeof(c), 1, x);
 }
 printf("\n");
}
```

# *Punteros a funciones*

- Los punteros no solamente puede ser usados para señalar variables, sino también funciones.





# *Ejemplo*

```
void funcion1(){
 printf("Se ha entrado en la funcion1\n");
}
void funcion2(){
 printf("Se ha entrado en la funcion2\n");
}
int main(void){
 //Creamos el puntero a la funcion
 void (*puntero_funcion)() = &funcion1;
 //Llamamos la funcion a traves del puntero
 puntero_funcion();
 //Hacemos a puntar al puntero a otra función
 puntero_funcion=&funcion2;
 //Llamamos la funcion a traves del puntero
 puntero_funcion();
 return 0;
}
```

## *Ejemplo con parámetros*

```
int sumar(int a, int b){
 return a+b;
}

int restar(int a, int b){
 return a-b;
}

void funcion_principal(int a, int b, int (*funcion)(int, int)){
 int resultado = funcion(a,b);
 printf("El resultado es %d\n", resultado);
}

int main(){
 //Se definen dos valores enteros cualesquiera
 int num1 = 5;
 int num2 = 4;
 //Se invoca la funcion principal, pasandole la funcion de SUMA
 printf("\nSuma:\n");
 funcion_principal(num1, num2, sumar);
 //Se invoca la funcion principal, pasandole la funcion de RESTA
 printf("Resta:\n");
 funcion_principal(num1, num2, restar);
 return 0;
}
```