PROGRAMACIÓN EN C PARA INGENIEROS ELECTRÓNICOS

ANA ESTELA RUIZ LINARES

TECNOLÓGICO
NACIONAL DE MÉXICO

CARTA DE RECONOCIMIENTO DEL AUTOR DE LOS DERECHOS A FAVOR DEL TECNM

Ciudad de México, 30/01/2023

TECNOLÓGICO NACIONAL DE MÉXICO.
PRESENTE

Bajo protesta de decir verdad, Ana Estela Ruiz Linares , personal docente adscrito al Instituto Tecnológico de Minatitlán del Tecnológico Nacional de México, manifiesto que en cumplimiento de mis actividades relacionadas con el Año Sabático elaboré la obra titulada " Programación en C para Ingenieros Electrónicos". Con base en lo anterior, y con fundamento en los artículos 83 de la Ley Federal del Derecho de Autor y 46 de su Reglamento, reconozco que el Tecnológico Nacional de México es titular de los derechos patrimoniales sobre la misma y le corresponden las facultades relativas a la divulgación, integridad de la obra y de colección, conservando el derecho a figurar como autor. Asimismo, respondo por la autoría y originalidad de la citada obra; y relevo de toda responsabilidad al Tecnológico Nacional de México de cualquier demanda o reclamación que llegara a formular alguna persona física o moral que considere que con esta obra es afectado en alguno de los derechos protegidos por la Ley en cita, asumiendo todas las consecuencias legales y económicas.

ATENTAMENTE

Ana Estela Ruiz Linares

AUTOR

Programación en C para Ingenieros Electrónicos

Contribución académica

Analizando los objetivos educacionales del Programa Educativo de Ingeniería Electrónica, se determinó que el desarrollo de este libro impactará mayormente en los siguientes:

1. El egresado desarrolla investigación aplicada, realiza un posgrado e innova para la solución de problemas propios de su perfil profesional en el sector productivo y educativo.
2. El egresado diseña, desarrolla, implementa e integra sistemas electrónicos de medición y control de procesos, que satisfagan las necesidades de los diferentes sectores industriales y de servicios, para contribuir en el desarrollo y progreso regional, nacional e internacional.
3. El egresado administra y gestiona en una organización las actividades de instalación, actualización, operación y mantenimiento de equipos y sistemas electrónicos para la optimización de los procesos industriales y de servicios, con ética y responsabilidad social.
4. El egresado, participa activamente en grupos multidisciplinarios nacionales e internacionales, para resolver los problemas de automatización y sistemas electrónicos de los diferentes sectores productivos y de servicios de las organizaciones, buscando la optimización de estos.

Tanto en un ambiente académico, tal como un curso de posgrado, como en un ambiente industrial o de servicio, es deseable tener conocimientos de programación para simular modelos, configurar equipos de automatización y control, diseñar prototipos robóticos, entre otras actividades. Además, en caso de que al egresado se le pida aprender otro lenguaje de programación, los conocimientos que adquiera del lenguaje C le permitirán avanzar en su aprendizaje.

Dedicatoria

Dedico este libro a tres grandes hombres que llenan mi vida:

A mi padre, por enseñarme el amor y la dedicación al trabajo,

a mi esposo, por todo su apoyo mientras escribía este libro,

a mi hijo, por esa sonrisa que ilumina mi corazón,

Los amo.

Agradecimiento

Agradezco al Tecnológico de Minatitlán y al Tecnológico Nacional de México por su apoyo, ya que este libro fue elaborado durante el ejercicio del Periodo Sabático autorizado por el TecNM.

Agradezco a Dios por todo su amor.

Índice

Introducción

¿Por qué debo aprender a programar si me estoy formando para ser ingeniero electrónico?, posiblemente sea la pregunta que te hagas cuando veas en tu plan de estudios la materia de Programación. Sin embargo, sólo tienes que mirar alrededor, escuchar los avances tecnológicos en las noticias y te darás cuenta de que en estos tiempos es cuando más ligada está la electrónica de la programación.

Ya sea para simular algún modelo electrónico, tal como un dron y su control de vuelo; automatizar algún proceso mediante un brazo robótico; diseñar algún equipo que utilice Internet de las cosas o programar PLC's (Controlador Lógico Programable) en el sector industrial, es deseable que sepas programar.

La intención de este libro es apoyarte en el aprendizaje de la programación estructurada utilizando el lenguaje C, a través de un lenguaje fácil de comprender y acompañado de varios ejemplos que ilustren su aplicación en la resolución de sencillos problemas en el área de la electrónica.

Aunque el lenguaje C está clasificado como un lenguaje de alto nivel, trabaja a un nivel más bajo que el resto de los lenguajes, esto significa que su código es mucho más entendible para la computadora, lo que produce una mayor rapidez en su ejecución, característica apreciada en las simulaciones de modelos electrónicos.

El libro se divide en cinco capítulos. En el capítulo uno, se dan las bases de todo lenguaje de programación, aprenderás a declarar variables para almacenar tus datos y a escribir expresiones para leer desde teclado e imprimir en la pantalla de tu computadora. En el capítulo dos, se muestran las estructuras que te permitirán tomar decisiones y repetir acciones. En el

capítulo tres, aprenderás a escribir tus códigos en bloques que te permitirán reutilizar tu código. En el capítulo cuatro, conocerás y utilizarás los arreglos y su utilidad en la programación. Finalmente, en el capítulo cinco, verás una introducción a un tema más avanzado como lo es la entrada/salida a archivos.

Para la ejecución de los códigos escritos en este libro, se utiliza Embarcadero Dev C++ 6.3, que es un IDE con un compilador incluido, gratuito y fácil de utilizar.

A pesar de que los problemas que se plantean en el libro no tienen mucha complejidad, entenderlos será la pauta para que, posteriormente, profundices tus conocimientos de programación y los apliques en la solución de problemas más complejos.

Finalmente, te sugiero que cuando estudies los códigos que verás en el libro, no te concretes a sólo verlos. Codifícalos y compílalos para que los analices y comprendas mejor su funcionamiento.

Capítulo 1

VARIABLES Y EXPRESIONES

OBJETIVO GENERAL

Al finalizar el capítulo, el estudiante será capaz de hacer programas que involucren entrada y salida de datos.

OBJETIVOS PARTICULARES:

- Conocer los conceptos de variables y tipos de datos.
- Reconocer expresiones escritas en C.
- Saber usar las funciones de entrada y salida de datos.

1.1. Introducción

En este capítulo empezarás a escribir tus primeros programas. Aunque hay varias formas de representar el esquema o forma general que tiene un programa escrito en lenguaje C, en la figura 1 se ilustra la forma que adoptaremos en este libro.

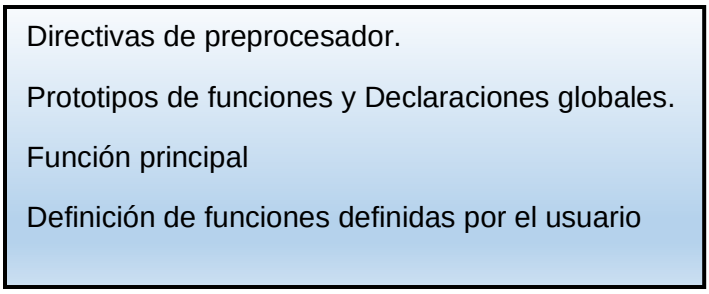


Figura 1. Esquema general de un programa en C.

Para tener tu primera experiencia en programación, escribe en el editor de texto del Dev C++ el código de la figura 2. Compílalo y ejecútalo, verás una salida como la de la figura 3. Si se genera algún error, comprueba que hayas escrito correctamente el código.

```
1  #include <stdio.h>
2  void main ( )
3  {
4      int R1;
5      int R2;
6      int R_Total;
7      printf("\t Introduce el valor de R1, en ohms: ");
8      scanf("%d",&R1);
9      printf("\t Introduce el valor de R2, en ohms: ");
10     scanf("%d",&R2);
11     R_Total=R1+R2;
12     printf("El equivalente serie es %d ohms",R_Total);
13 }
```

Figura 2. Código de tu primer programa en C.

Como acabas de observar, el código permite obtener la resistencia equivalente de dos resistencias conectadas en serie. Se dice que esto es una *aplicación en modo consola* porque se ejecuta en una ventana de MS-DOS a través de líneas de comandos. No te preocupes si no entiendes el código que acabas de escribir, a continuación, se explicarán las partes que lo integran.

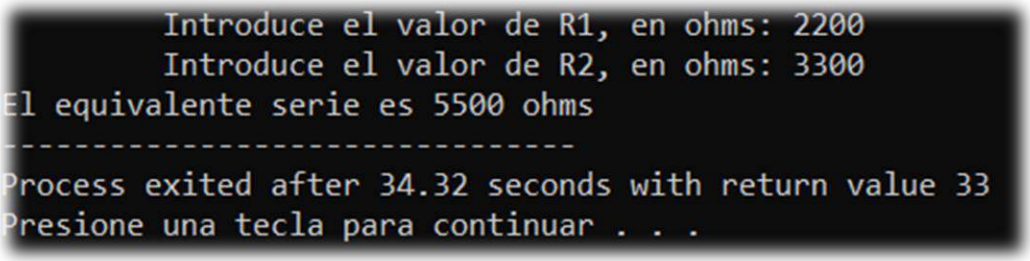


Figura 3. Salida del programa que obtiene el equivalente de dos resistencias conectadas en serie.

1.2. Tipos de datos

Los programas trabajan con datos, estos pueden ser dados por el usuario (persona que utiliza el programa o aplicación que escribiste) o generados por la computadora. Un *tipo de dato* es un atributo o propiedad que define el formato en el que se guardará ese dato, esto es, el tamaño en bytes que se utilizará en la memoria para almacenarlo y las operaciones que soporta.

En C existen cinco tipos de datos que se consideran básicos: ***char*** para guardar caracteres del código ASCII, ***int*** para números enteros, ***float*** para números reales con punto decimal, ***double*** para números con punto decimal de doble precisión, y ***void*** que no guarda nada y cuya razón de existir se explicará en el capítulo 3. Además, existen *modificadores* a esos tipos de datos: ***signed***, ***unsigned***, ***short*** y ***long***. En general, los números enteros se pueden guardar en *int*, *short*, *long* y *long long*; mientras que los números con punto decimal o punto flotante utilizan *float*, *double* y *long double*. Es importante mencionar que sólo los tipos que guardan números soportan operaciones aritméticas, en otras palabras, no podrás hacer una división, por ejemplo, con dos tipos char.

Los modificadores *signed* y *unsigned* pueden preceder a cualquiera de los tipos de datos para números enteros. También se pueden utilizar solas, como

especificadores de tipo, en cuyo caso se entienden como tipo *int*, esto es, *signed int* y *unsigned int*, respectivamente. Cuando *int* se usa sola, se entiende que es *signed*. Cuando *long* y *short* se utilizan solas, se entienden como *long int* y *short int*.

En la tabla 1 se muestran los tipos de datos básicos y algunos modificados, el número de bytes que utilizan y el rango de los datos que puede almacenar. Aunque en realidad, el tamaño en bytes varía dependiendo del compilador y la plataforma en que se programa, esto es, si es 16, 32, ó 64 bits. Así que podría suceder que en alguna otra fuente de información encuentres valores distintos a los que observas en la tabla. Para saber el número de bytes que utiliza tu computadora para cada tipo de dato, ejecuta el código de la figura 4. En la figura 5 se muestra los valores obtenidos usando el sistema operativo Windows 10 de 64 bits.

Tabla 1. Tipos de datos básicos y modificados.

Tipo de dato	Tamaño en bytes	Rango
char	1	-128 a 127
unsigned char	1	0 a 255
signed char	1	-128 a 127
short	2	-32768 a 32767
unsigned short	2	0 65535
int	4	-2147483647 a 2147483647
unsigned int	4	4294967295
signed int	4	-2147483647 a 2147483647
long	4	-2147483647 a 2147483647
long long	8	-9223372036854775807 a 9223372036854775807
unsigned long long	8	18446744073709551615
float	4	3.4E-38 a 3.4E+38
double	8	1.7E-308 a 1.7E+308
long double	10	3.4E-4932 a 1.1E+4932

Es necesario enfatizar la importancia de conocer los tipos de datos para poder determinar aquellos que se utilizarán en el código que vayas a escribir, ya que, si optas por un especificador que no sea el adecuado, podrás producir un *desbordamiento* al querer guardar un valor mayor al que pueda almacenar o, un mal uso del recurso de memoria al reservar más bytes de lo que realmente se requiera.

Para ilustrar el concepto de desbordamiento, vuelve a correr el código de la figura 2, esta vez dale a R1 el valor de 2147483647 y a R2 el valor de 1; teóricamente,

deberías tener una salida de 2147483648. Sin embargo, el código produce un número negativo, tal como se muestra en la figura 6. Lo que observas es debido a que 2147483647 es, como se ve en la tabla 1, el valor máximo que puede almacenarse en un *int*. Así que, como el número de bits no le permite guardar 2147483648, en su lugar muestra el valor mínimo que puede almacenar, que es -2147483647, esto es, como si reiniciara. Vuelve a correr el programa, ahora con R1 igual a 2147483648 ¿qué observas?

```c
#include<stdio.h>
void main()
{
    printf(" Bytes reservados para un tipo char: %i\n",sizeof(char));
    printf(" Bytes reservados para un tipo short: %i\n",sizeof(short));
    printf(" Bytes reservados para un tipo int: %i\n",sizeof(int));
    printf(" Bytes reservados para un tipo float: %i\n",sizeof(float));
    printf(" Bytes reservados para un tipo long: %i\n",sizeof(long));
    printf(" Bytes reservados para un tipo long long: %i\n",sizeof(long long));
    printf(" Bytes reservados para un tipo double: %i\n",sizeof(double));
    printf(" Bytes reservados para un tipo long double: %i\n",sizeof(long double));
}
```

Figura 4. Código para observar el tamaño en bytes en tu computadora de algunos tipos de datos.

```
Bytes reservados para un tipo char: 1
Bytes reservados para un tipo short: 2
Bytes reservados para un tipo int: 4
Bytes reservados para un tipo float: 4
Bytes reservados para un tipo long: 4
Bytes reservados para un tipo long long: 8
Bytes reservados para un tipo double: 8
Bytes reservados para un tipo long double: 16
```

Figura 5. Salida del programa de la Figura 4 mostrando el número de bytes que ocupa cada tipo de dato.

```
        Introduce el valor de R1, en ohms: 2147483647
        Introduce el valor de R2, en ohms: 1
l equivalente serie es -2147483648 ohms
```

Figura 6. Ilustración de desbordamiento en un tipo de dato int.

Ahora observarás por qué se dice que el especificador de tipo *char* no soporta operaciones aritméticas. Modifica el código de la figura 2 de tal manera que se parezca al de la figura 7; aunque de momento no comprendas el código, sólo enfócate en las líneas 4, 5 y 6, observa que se cambiaron los *int* por *char*. Ejecuta el código y dale a R1 el valor de 1 y a R2 el valor de 2 ¿qué obtienes a la salida? Aunque se escribe una instrucción de suma en la línea 12, el resultado en lugar de ser un 3 es una letra ‘c’, tal como se muestra en la figura 8.

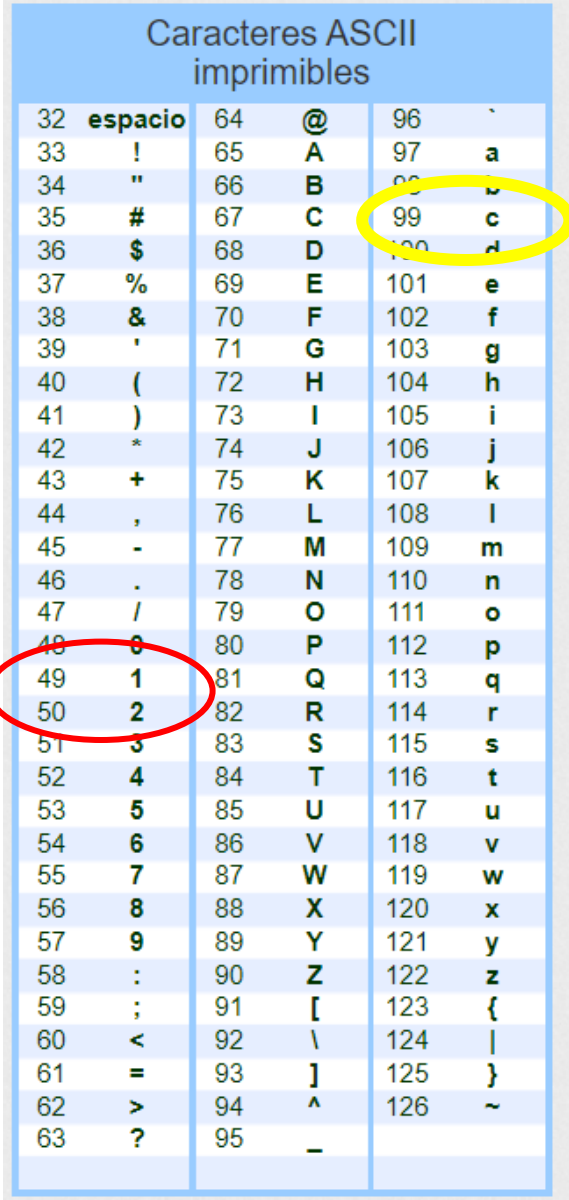
```
1   #include <stdio.h>
2   void main ( )
3   {
4       char R1;
5       char R2;
6       char R_Total;
7       printf("\t Introduce el valor de R1, en ohms: ");
8       R1=getchar();
9       fflush(stdin);
10      printf("\t Introduce el valor de R2, en ohms: ");
11      R2=getchar();
12      R_Total=R1+R2;
13      printf("El equivalente serie es %c ohms",R_Total);
14  }
```

Figura 7. Código mostrando el uso inadecuado de un tipo de dato.

```
        Introduce el valor de R1, en ohms: 1
        Introduce el valor de R2, en ohms: 2
l equivalente serie es c ohms
```

Figura 8. Salida obtenida con el código de la Figura 7.

Pero ¿por qué sucede esto? Resulta que el '1' que se le asignó a R1 realmente se guarda como un número 49 porque R1 está declarada como *char* y 49 es el valor que le corresponde al caracter '1' en el código ASCII, tal como lo puedes observar encerrado en rojo en la figura 9. Algo similar sucede con R2, al darle el '2' realmente almacena el número 50. Cuando se ejecuta la línea 12, se suma 49+50 y el resultado,99, corresponde al caracter 'c', tal como lo puedes ver encerrado en amarillo en la misma figura.



Caracteres ASCII imprimibles

32	espacio	64	@	96	`
33	!	65	A	97	a
34	"	66	B		
35	#	67	C	99	c
36	$	68	D		
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
		80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
		83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Figura 9. Código ASCII. Tomada de https://elcodigoascii.com.ar/

1.3. Declaración de variables

Una *variable* es un espacio de memoria que se reserva para poder guardar información tal como un número entero, flotante o caracter. C no permite utilizar variables antes de ser definidas y declaradas. Una *definición* consiste en decirle al compilador que reserve cierta cantidad de memoria para almacenar algún dato; la cantidad de bytes reservados y las operaciones que soporte dependerá del tipo de dato al que corresponda dicha variable. En una *declaración*, se le da un nombre a ese espacio de memoria reservada. La *sintaxis* o regla para una declaración sencilla de variable es la siguiente:

Sintaxis de declaración simple de una variable:

```
tipo_de_dato identificador ;
```

donde:

tipo_de_dato puede ser *char*, *int*, o cualquier otro de la Tabla 1,
identificador es el nombre asignado por el programador con el cual se reconocerá a la variable,
; indica al compilador la terminación de la instrucción

Para asignar un *identificador* válido a una variable se deben seguir ciertas pautas:

1. Se pueden utilizar letras excepto la ñ porque no pertenece al alfabeto de C; por esta misma razón, las letras no deben ir con acentos.
2. Se pueden usar números o guión bajo (_).
3. No se debe empezar por un número.
4. No debe haber espacio en blanco.
5. C es un lenguaje *sensible a mayúsculas y minúsculas*.
6. No se pueden utilizar *palabras reservadas*.

Ejemplos de identificadores válidos

Promedio valor_capacitor NuevoRegistro prom10

Ejemplos de identificadores no válidos

10prom año_2 división int Nuevo Registro

Que C sea sensible a mayúsculas y minúsculas significa que para el compilador
Total total TOTAL
son identificadores diferentes.

Las *palabras reservadas* son propias del lenguaje, ya que tienen un significado especial para el compilador. Son palabras reservadas las siguientes:

auto	else	long	switch
break	enum	register	typedef
case	extern	restrict	union
char	float	return	unsigned
const	for	short	void
continue	goto	signed	volatile
default	if	sizeof	while
do	inline	static	
double	int	struct	

Aunque a las variables se le pueda dar cualquier nombre o identificador que cumpla los criterios anteriores, se sugiere asignarles un nombre que tenga relación con la información que almacenará. Por ejemplo, si se desea una variable para guardar el resultado de una suma, podría identificarse como Suma, SUMA, o S.

EJERCICIO 1 DE IDENTIFICADORES

Escribe al lado del identificador una V si es válido o una N si es No Válido
P ____ ConTaDor ____ switch ____ 3_altura ___

Solución:
V, **V**, **N** (palabra reservada), **N** (empieza con un número)

Ejemplos de declaración simple de una variable

unsigned char codigo; long double capacitancia3; float base;
int valor45; unsigned int R_equiv; double M_33;

Cuando se tenga que declarar más de una variable del mismo tipo, se puede utilizar una *declaración múltiple*. La declaración múltiple tiene el mismo efecto que la declaración sencilla, reservar espacio en memoria, la ventaja es escribir menos líneas de código. Recuerda, el requisito para poder hacer una declaración múltiple es que las variables tienen que ser del mismo tipo de dato.

Sintaxis de declaración múltiple de variables:

```
tipo_de_dato identificador_1, identificador_2, …, identificador_n ;
```

donde:

tipo_de_dato puede ser *char*, *int*, o cualquier otro de la Tabla 1,
identificador_1, identificador_2, …, identificador_n son nombres asignados por el programador con los cuales se reconocerán a las variables,
; indica al compilador la terminación de la instrucción

Con los conocimientos anteriores, se comenzará a analizar el código de la figura 2 cuyo enunciado de problema es: *"Escribir un programa que solicite al usuario dos valores de resistencias y entregue como salida el valor de la resistencia equivalente de ellas conectadas en serie. Todos los valores de las resistencias deberán estar en ohms".*

Antes de empezar la codificación, hay que identificar tres partes que integrarán al código*: Entrada*, *Salida* y *Procesamiento*. En este subtema se verán los dos primeros. Se considera *Entrada* o *datos de entrada* aquella información que el programa necesite para funcionar, esta información puede ser dada por el usuario o ser generada por código. La *Salida* es el resultado que el usuario espera. Tanto para la Entrada como para la Salida se utilizan variables.

Examinando el enunciado del problema, se determina que la Entrada serán los dos valores de las resistencias y la Salida el valor de la resistencia equivalente. Estas variables deben guardar números enteros y soportar la operación suma. Tomando en cuenta lo anterior y examinando la Tabla 1, se determina que serán variables de tipo *int*. Ahora, ya podrás reconocer que las líneas 4, 5 y 6 (ver figura 10) corresponden a tres declaraciones simples de variables de tipo entero y cuyos identificadores son R1 y R2 para los datos de *Entrada* y R_Total para la *Salida*. Recuerda que el identificador puede ser cualquier nombre válido.

```
4       int R1;
5       int R2;
6       int R_Total;
```

Figura 10. Declaraciones simples de tres variables de tipo entero (int).

EJERCICIO 2 DE IDENTIFICADORES

Comprueba la sensibilidad de C a las mayúsculas y minúsculas cambiando en la línea 4 la letra R por r. Compila y ejecuta el programa. Esto generará un error que podrás corregir al poner nuevamente R.
Prueba cambiando los nombres de los identificadores, por ejemplo, si decides que en lugar de R1 nombrarás a la variable como valor1 sustituye todas las R1 por valor1. Compila y ejecuta el programa.

EJERCICIOS DE DECLARACIÓN DE VARIABLES

Escribe la declaración para una variable de tipo flotante con identificador “voltaje”:
_____

Declara una variable de nombre operador que guarde un caracter: _____

Dado que las tres variables son del mismo tipo, sustituye las tres líneas de declaraciones simples por una sola línea con una declaración múltiple:

Solución:
float voltaje; **char operador;** **int R1,R2,R_Total;**

Aunque no hay una regla que diga que las declaraciones deben estar en las primeras líneas, sí es una buena práctica colocarlas después de la llave de apertura, debajo de main ( ); realmente la única condición es que la declaración se haga antes de utilizar la variable por primera vez. Es importante subrayar que las variables sólo se deben declarar una vez. Si se intenta utilizar una variable antes de declararla, el compilador generará un error; si, por el contrario, se procede a declarar una variable en múltiples ocasiones, también se generará error.

Las variables declaradas dentro de las llaves se llaman variables locales y las que se escriben debajo de las directivas de preprocesador se conocen como variables globales. Sus características se estudiarán en el capítulo 3.

1.4. Declaración de constantes

La información necesaria para la ejecución de un código no siempre proviene de las variables, en ocasiones se utilizan valores determinados por *constantes*, se llaman así porque su valor no cambia durante toda la ejecución del programa. Ejemplos de constantes pueden ser el valor de π (3.1416), el valor de la constante gravitacional

(9.8 ms^2), la constante K de la Ley de Coulomb (9x10^9 Nm^2C^{-2}), o cualquier otro valor que no cambie. Las constantes pueden ser de tipo entero, flotante, caracter o cadena. La sintaxis de declaración de una constante es la siguiente:

Sintaxis de declaración de constante:

```
const tipo_de_dato identificador = valor;
```

donde:

const palabra reservada para declarar constantes,
tipo_de_dato para enteros: int, long; para flotantes: float. double, long double; para caracteres: char
identificador nombre válido de la constante,
= operador de asignación,
valor valor de la constante,
; indica al compilador la terminación de la instrucción

Es importante tener en cuenta las siguientes consideraciones: Las constantes numéricas pueden ser positivas o negativas. Los números reales o de punto flotante pueden llevar punto decimal, estar en forma de notación exponencial o tener ambas características. Regularmente, el identificador va escrito en mayúsculas. El valor de una variable tipo *char* debe ir encerrado entre comillas simples.

Ejemplos de declaración de constantes

const float PI=3.1416;	número con punto decimal
const double K=9e9;	número con notación exponencial
const long double C=0.25e-9	número con punto decimal y notación exponencial
const int n_resistencias=250;	número entero
const char inicial='A'	caracter

Las *secuencias de escape* son constantes tipo *char* que se utilizan para especificar acciones en pantallas e impresoras tales como nueva línea o tabulaciones. También se emplean para representar literales de caracteres no imprimibles. En la tabla 2 se muestran algunas secuencias de escape. Más adelante en este mismo capítulo, verás cómo utilizarlas.

Otra forma de tratar a las constantes es a través de una *directiva de preprocesador,* que se explicará en el subtema de Entradas y Salidas. La directiva *#define* establece una constante por medio de un nombre simbólico, sustituyendo un identificador por una constante en todos los lugares donde se encuentre ese identificador.

Sintaxis de constante simbólica:

```
#define identificador valor
```

donde:

#define directiva de preprocesador,
identificador nombre válido de un identificador,
valor valor de la constante

Es importante resaltar que, a diferencia de las declaraciones anteriores, no se escribe el punto y coma al final de la instrucción. El lugar donde se colocan es antes del *main ( )*, junto con las directivas *#include*, como se vio en el esquema de la figura 1.

Ejemplos de constante simbólica

#define PI 3.1416
#define K 9e9
#define INICIAL 'A'

Estas líneas harán que el procesador sustituya 3.1416 en todos los lugares donde encuentre escrito pi, 9e9 en donde vea una K y escriba una 'A' en donde esté escrito la palabra INICIAL. Recuerda que las constantes de tipo caracter va encerradas entre comillas simples y que los identificadores, aunque no es una regla, se escriben con mayúsculas.

1.5. Operadores aritméticos

C tiene operadores *binarios* y *unarios*; se llaman binarios porque requieren de dos operandos y unarios porque sólo necesitan uno. En este libro sólo se tratarán los operadores binarios que se muestran en la tabla 3. El operador / regresa el cociente

de una división, mientras que % regresa el residuo; entonces 30/10 dará 3 y 30%10 (se lee 30 módulo 10) da 0.

El tipo de dato de las variables que utilices en tu codificación impactará en el resultado obtenido de una operación aritmética. Veamos un caso, supóngase que se necesita dividir cinco entre diez y esperas como resultado 0.5; sin embargo, si escribes 5/10, en lugar de 0.5 el compilador arrojará un 0. Esto ocurre porque entero/entero da un entero y la parte entera de 0.5 es 0. Para solucionar esta situación se podría hacer cualquiera de las siguientes alternativas: entero/flotante, flotante/entero, o flotante/flotante, esto es 5/10.0, 5.0/10 o 5.0/10.0. Todo lo anterior es por la conversión automática de tipos que hace C. Así que, si en un momento dado, deseas dividir dos variables y te interesa también la parte decimal, asegúrate de que al menos una de ellas tenga un tipo de dato de tipo flotante.

Los operadores +, -, *, / pueden aplicarse sobre números enteros y flotantes, mientras que el operador % sólo puede ser usado con enteros; cabe mencionar que el término flotante incluye a los tipos *float*, *double* y *long double*, en tanto que entero incluye a los tipos *short*, *int*, *long*.

Tabla 2. Secuencias de escape.

Secuencia de escape	Significado
\a	Alarma
\b	Retroceso
\f	Avance de página
\n	Nueva línea
\r	Retorno de carro
\t	Tabulación horizontal
\v	Tabulación vertical
\'	Comilla simple
\"	Comillas dobles
\\	Barra diagonal inversa
\?	Signo de interrogación literal

Tabla 3. Operadores aritméticos.

Operación	Operador	Ejemplo de uso
Suma	+	a+b
Resta	-	a-b
Multiplicación	*	a*b
División	/	a/b
Módulo	%	a%b

Los operadores tienen una precedencia o prioridad; + y – tienen la misma prioridad; de igual manera * y / están en el mismo nivel jerárquico. Sin embargo, estos últimos tienen mayor prioridad que los primeros. Todos los operadores aritméticos se asocian de izquierda a derecha.

Cuando los operadores tienen la misma prioridad, se realiza la evaluación tomando en cuenta la asociatividad. Para alterar la asociatividad se utilizan paréntesis ya que éstos tienen mayor prioridad.

Ejemplos de precedencia en los operadores aritméticos

5+4*2 =13 Ya que * tiene mayor precedencia, se realiza primero 4*2 y luego se suma con 5.

5*4/2=10 Debido a que estos operadores tienen la misma prioridad, el 5 se multiplica por 4 y el resultado se divide entre 2.

(5+4)*2= 18 Los paréntesis tienen mayor prioridad, así que primero se suma y luego se multiplica.

EJERCICIOS DE PRIORIDAD DE OPERADORES

Escribe el resultado de la evaluación:

4+9/3-2:_____
2+8+6-4:_____
8/4-2*25/5:_____
4+9/(3-2):_____

Solución:

5, 12, -8, 13

1.6. Operador de asignación

La asignación es una operación donde el valor del operando derecho se asigna a la dirección de memoria del operando izquierdo, este último operando se conoce como expresión L (de left en inglés) modificable y regularmente se trata de un identificador. El nombre del identificador indica una ubicación de almacenamiento, mientras que el valor de la variable es el valor almacenado en esa ubicación. El operador de asignación en el lenguaje C es el signo igual (=). En la figura 11 se ilustra la operación de asignación: *A=10.* Esta operación ocasiona que el número 10 se guarde en el espacio de memoria reservada para A. Para que sea posible alterar el valor almacenado en el operando izquierdo, éste no debe ser declarada como una constante.

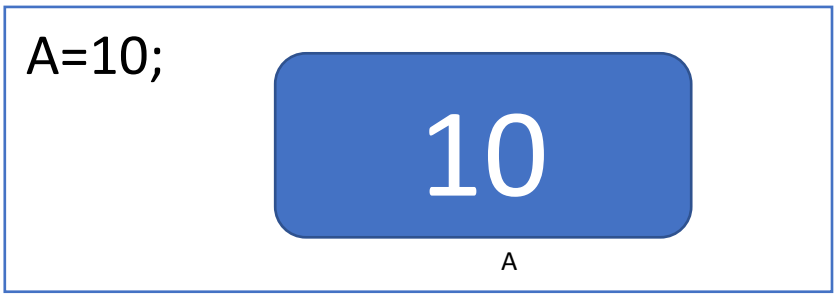


Figura 11. lustración sobre la operación de asignación. En el espacio de memoria reservada para A se almacena el valor de 10.

La sintaxis de asignación es la siguiente:

Sintaxis de asignación:

```
exp1 = exp2
```

donde:

exp1 expresión
= operador de asignación
exp2 expresión

Las expresiones se estudiarán en el siguiente subtema, pero para que entiendas la operación de asignación, basta que pienses que el lado izquierdo es una variable y el lado derecho un valor.

En C existen dos tipos de asignaciones: simple y compuesta. En la asignación simple, que es la hemos estado viendo, el valor del operando de la derecha se almacena en la variable del lado izquierdo. En la asignación compuesta primero se realiza una operación aritmética, de desplazamiento o bit a bit antes de almacenar el resultado. En la tabla 4 se presentan los operadores de asignación, todos estos son operadores binarios. Los operadores de asignación unarios son el operador de incremento y el operador de decremento, cuyos símbolos son ++ y --, respectivamente. Los ejemplos de cómo utilizar los operadores de asignación se verán en el siguiente subtema.

Tabla 4. Operadores de asignación.

Operador	Operación realizada
=	Asignación simple
***=**	Asignación y multiplicación
/=	Asignación y división
%=	Asignación y módulo
+=	Asignación y suma
-=	Asignación y resta

1.7. Expresiones aritméticas

Una *expresión aritmética* en C es una secuencia de operandos, operadores y llamadas a funciones con el fin de realizar un cálculo, hacer una asignación o almacenar un valor modificado. Son las expresiones aritméticas las que se utilizan en la parte de *Procesamiento* de un código. Cuando al final de una expresión se agrega un punto y coma (;) se dice que se tiene una *proposición*.

Ejemplos de expresiones aritméticas usando asignación simple

numero=10

nuevo_valor= x+2*y-valor_anterior

Ejemplos de expresiones aritméticas usando asignación compuesta

Ejemplo 1:

x+=10

Esto es equivalente a escribir *x=x+10*. Hay que hacer hincapié que esto no se trata de una expresión matemática, donde claramente no tiene sentido, sino de una expresión de programación donde tiene el siguiente significado: Toma el valor que se encuentra en la dirección de memoria identificada por *x* y súmale 10, finalmente, almacena ese nuevo valor en la misma variable. En la figura 12 se ilustra la operación compuesta: primero se evalúa y luego se almacena.

Ejemplo 2:

promedio/=3

Esto es equivalente a escribir *promedio= promedio/3*. Significa que el compilador tomará el valor actual de la variable *promedio,* lo dividirá entre 3 y lo guardará nuevamente en esa misma dirección de memoria. Digamos que *promedio* tiene inicialmente guardado un 21, después de la división, su nuevo valor será un 7. Si en lugar de / se tuviera el operador de módulo (%), el nuevo valor de *promedio* sería un 0, ya que ese es el residuo cuando se divide 21/3.

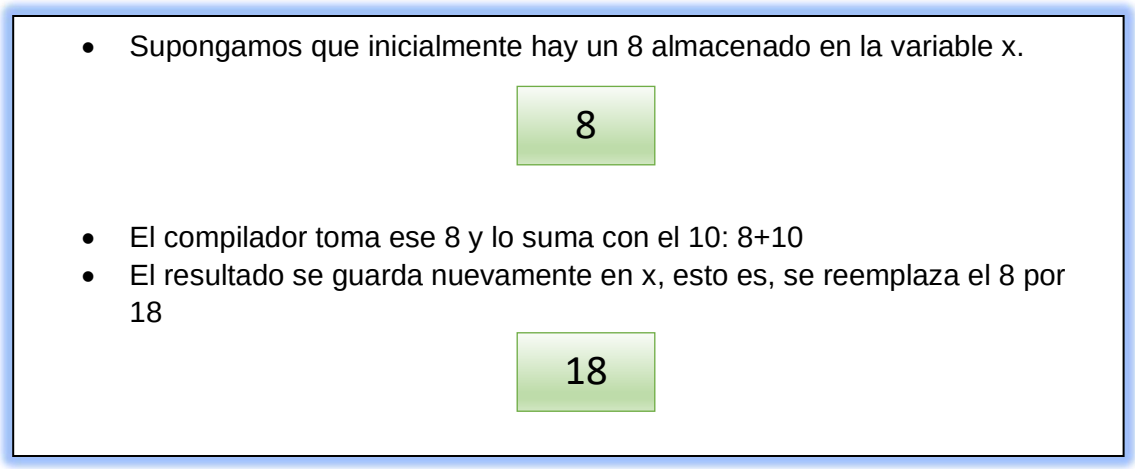


Figura 12. Ilustración del concepto de asignación compuesta.

Ejemplo 3:

x++

Esta expresión de incremento es equivalente a x=x+1. El principio es el mismo que la asignación compuesta usando el operador de suma, se toma el valor actual de *x*, se le suma un 1 y se guarda, nuevamente en *x*, el nuevo valor. Vamos a suponer que *x* inicialmente tiene almacenado el 14, después de la operación de incremento, guardará el 15.

Ejemplo 4:

x--

Esta expresión de decremento es equivalente a x=x-1. Hagamos el valor inicial de *x* igual a 8, después del operador --, el nuevo valor almacenado en *x* será 7. Es necesario mencionar que tanto el operador de incremento como el de decremento sólo se usa cuando se desea alterar el valor inicial en una sola unidad.

EJERCICIOS DE EXPRESIONES ARITMÉTICAS

Calcula el valor que se guardará en la variable *contador*, suponga que x=5, y=10:

valor = 2*x+5*y/10-10 _____

valor= (y+100)*(5+x)/2*y _____

Solución:

5, 5500

En el código de la figura 2, la parte del Procesamiento está presente en la línea 11:

R_Total=R1+R2;

1.8. Entrada y salida

En los programas escritos en lenguaje C, la comunicación entre la computadora y el usuario se realiza a través del teclado y la impresión en pantalla, éstos representan, respectivamente, la entrada y salida estándar. El código necesario para comunicarse con la entrada y salida estándar no necesita escribirse, ya está hecho, sólo hay que incluirlo en nuestros programas. Las líneas de código que permiten la interacción entrada/salida o I/O, por las siglas en inglés input/output, se encuentran en un archivo llamado *stdio* por ***st****andar* ***i****nput*/***o****utput*.

Además de *stdio*, existen otros archivos que tienen código para realizar tareas específicas como, por ejemplo, calcular el seno de un ángulo, determinar la hora del sistema o unir palabras, conocidas como *cadenas*. Tales archivos se conocen como *librerías* o bibliotecas, tienen extensión .h y no son propiamente parte del lenguaje C. En la tabla 5 se enlistan algunas bibliotecas del ANSI C.

Tabla 5. Biblioteca estándar del ANSI C.

Nombre	Tareas
stdio.h	Entrada y salida
ctype.h	Prueba de caracteres
string.h	Operaciones con cadenas
math.h	Funciones matemáticas
stdlib.h	Conversión numérica, manejo de memoria
time.h	Funciones de fecha y hora

Para poder utilizar los archivos de librería en el código que el programador escriba, hay que hacerlo a través de las *directivas de preprocesador de C*, que se distinguen por tener al inicio el símbolo de numeral, #. El preprocesador es un software que forma parte del compilador y que realiza operaciones sobre el código antes de pasar al proceso de compilación propiamente. Entre las operaciones más comunes del preprocesador están *#include* que permite incluir archivos y *#define* que define un símbolo con una secuencia de caracteres, tal como se vio en el subtema 1.4 para definir constantes.

Sintaxis de #include:

```
#include <nombre_archivo>
```

donde:

#include directiva de preprocesador,
< > corchetes angulares,
nombre_archivo nombre del archivo a ser agregado

Los corchetes angulares (< >) indican al preprocesador que busque el archivo mencionado en la dirección establecida durante la instalación del compilador. Si en lugar de usar < y > se usan comillas dobles, “ “, el archivo por incluir se empezará a buscar en la misma carpeta que el código que se esté programando.

Ejemplos de uso de #include

#include<stdio.h>

#include<math.h>

En la figura 2 puedes ver en la primera línea de código que se agrega el archivo *stdio.h*. Si deseas calcular, por ejemplo, una función trigonométrica, debes incluir la librería math.h.

EJERCICIO DE #include

En el código de la figura 2 elimina la primera línea que corresponde a la directiva del preprocesador que agrega el archivo que permite la comunicación con el teclado

y la pantalla (stdio.h). Compila el código. Observa que en la parte inferior aparece un cuadro mostrando errores.
Para quitar los errores, vuelve a escribir la directiva de preprocesador que agrega a stdio.h.

Aunque será en el capítulo tres donde podrás encontrar las funciones que se encuentran en las librerías, aquí veremos dos funciones muy comunes de stdio.h: *printf ( )* y *scanf( )*, las otras funciones se verán en el capítulo dos. En la tabla 6 podrás ver las funciones de stdio.h que permiten comunicarse con la entrada y salida estándar, en inglés *stdin* y *stdout,* respectivamente.

Tabla 6. Funciones de la librería stdio.h para stdin y stdout

Nombre de la función	Objetivo de la función
printf( )	Imprimir una cadena con formato en pantalla.
scanf( )	Leer un dato desde teclado.
getchar( )	Leer un caracter desde teclado.
putchar( )	Escribir un caracter en la pantalla.
gets( )	Leer una cadena de caracteres desde teclado.
puts( )	Escribir una cadena de caracteres en la pantalla.

La función que usualmente se utiliza para enviar mensajes a la pantalla es printf ( ) porque permite darle un formato a lo que el programador desea que el usuario vea. A partir de ahora, llamaremos *cadena* al mensaje que se envía a la pantalla, y se considerará, informalmente, como una secuencia de caracteres.

Sintaxis de printf( ):

```
printf("cadena con formato");
```

donde:

printf( ) función para comunicarse con stdout,
cadena con formato secuencia de caracteres que se imprime en stdout

Para hacer una cadena con formato se tiene que utilizar un *especificador de formato* que está formado por el símbolo % seguido de un caracter de conversión de la tabla

7, en medio de estos pueden existir ciertos modificadores, tal como se mostrarán posteriormente.

Ejemplos de printf( )

Verás algunos ejemplos de cómo utilizar en diferentes situaciones:

- Enviar a pantalla (stdout) un texto sin formato.

Podrás utilizar este tipo de sintaxis si sólo deseas enviar una cadena sin ningún dato:

printf("Bienvenido al Sistema de Inventario del Laboratorio de Electronica");

printf("Deseas salir del Sistema?");

printf("Introduzca los datos requeridos");

Nótese que la cadena debe ir encerrada entre comillas dobles. Notarás también que a la palabra Electronica que aparece en el primer ejemplo le falta la tilde en la o, eso es porque el alfabeto de C está tomado del inglés, que es un lenguaje que no maneja los acentos; más adelante se muestra cómo se puede corregir este inconveniente.

Tabla 7. Caracteres usados en los especificadores de formato

Caracter de conversión	Tipo de dato utilizado
c	Caracter (char)
d, i	Número entero (int)
ld, D	Número entero long (long int)
f	Punto flotante(float)
e	Notación científica con e minúscula
E	Notación científica con E mayúscula
g	Utiliza %f o %e según sea más corto
G	Utiliza %f o %E según sea más corto
s	Cadena de texto
u	Entero sin signo (unsigned int)
U, lu	Entero sin signo long(unsigned long int)
lf	Flotante de doble precisión (double)
LF	Flotante de doble precisión long (long double)

- Enviar a pantalla (stdout) usando secuencias de escape.

Las secuencias de escape, como se mencionó en el subtema de constantes, son constantes de caracter que puedes utilizar para darle un tipo de formato a la cadena

que se manda a la pantalla, no todas las secuencias de escape de la tabla 2 tendrán un efecto en la pantalla, las que podrás utilizar son las de tabulación horizontal, nueva línea y alarma.

printf("\t SISTEMA DE CALCULO DE RESISTENCIAS EQUIVALENTES");

el uso de \t (tabulación horizontal) hará que la cadena se imprima después de un espacio determinado por el tabulador, recuerda que la impresión se lleva a cabo en la salida estándar que es la pantalla de tu computadora.

Puedes usar las secuencias de escape al principio o al final. Tampoco hay un número establecido de secuencias que puedas usar en las cadenas.

- Enviar a pantalla (stdout) una cadena con formato.

Cuando se requiera imprimir resultados calculados por el programa se utilizan los especificadores de formato.

printf("El valor de la resistencia equivalente es %d ohms", Requiv);

En el momento de imprimirse en pantalla, el lugar que ahora ocupa el especificador de formato %d será reemplazado por un número entero; si el valor que se va a imprimir es de punto flotante, entonces ocuparías %f, tal como te lo indica la tabla 7. El valor que se mostrará será el de la variable que se menciona después de la cadena.

No hay un límite en el número de especificadores de formato que utilices y los puedes colocar en cualquier parte de la cadena. Lo importante es que la cantidad de nombres de variables que escribas después de la cadena debe ser la misma que de especificadores de formato. Recuerda que tienes que tomar en cuenta el tipo de dato para elegir el especificador de formato adecuado.

printf("La resistencia equivalente paralela de %d ohms y %d ohms es %f ohms", R1.R2,Requiv);

Aquí se considera que las variables R1 y R2 son de tipo int, mientras que Requiv es de tipo float. Observa que las variables van separadas por comas.

A continuación, veremos cómo imprimir con acento:

printf("Bienvenido al Sistema de Inventario del Laboratorio de Electr%cnica", 162);

observa que para que la palabra Electronica se imprima con la tilde en la letra o, en su lugar se coloca el especificador de formato %c porque es un caracter; además, no se escribe el nombre de alguna variable, sino que directamente el valor de su código ASCII. En la tabla 8 puedes observar los códigos ASCII para algunos caracteres que no se encuentran en el alfabeto del lenguaje C.

- **Modificar ancho del campo**

Para especificar un mínimo de ancho de campo, se coloca un número entero entre el símbolo % y el caracter de conversión.

printf(“El valor de la resistencia equivalente es %5d ohms”, Requiv);

Si la cadena a imprimir tiene un número de caracteres mayor a lo especificado, no surte ningún efecto, se imprime toda la cadena; supongamos que Requiv=2563456, el ancho del campo es siete, que es mayor que cinco, así que se imprimiría

El valor de la resistencia equivalente es 2563456 ohms

Tabla 8. Código ASCII extendido

Símbolo	Código ASCII
ñ	164
Ñ	165
á	160
Á	181
é	130
É	144
í	161
Í	214
ó	162
Ó	224
ú	163
Ú	233

Por el contrario, si Requiv=256, se rellenará el campo con espacios en blanco.

El valor de la resistencia equivalente es 256 ohms

Si deseas que el relleno sea con ceros, entonces coloca un 0 antes del número.

printf(“El valor de la resistencia equivalente es %05d ohms”, Requiv);

lo que produciría:

El valor de la resistencia equivalente es 00256 ohms

Ahora, supongamos que Requiv fuera de tipo float y guardara el valor de 1256.258

printf(“El valor de la resistencia equivalente es %010f ohms”, Requiv);

se imprimiría en pantalla:

El valor de la resistencia equivalente es 001256.258 ohms

Nota que el ancho del campo ocupa diez lugares, incluyendo el punto decimal, y que el caracter de conversión d cambió por f por tratarse de un flotante. Modificar el ancho del campo te resultará útil si deseas obtener una impresión más ordenada de tus columnas.

- **Modificar precisión**

Para modificar la precisión, se coloca un punto después del ancho de campo, si existe, y se escribe un número entero. Si el especificador de formato es %f, %e o %E, ese otro número indica cuántos decimales aparecen.

Supongamos que Requiv guarda el valor 256.8951:

printf(“El valor de la resistencia equivalente es %.2f ohms”, Requiv);

En este ejemplo no existe el entero que limita el ancho de campo, lo que significa que no importa la longitud del número a imprimir, sin embargo, sí se está limitando el número de decimales a dos, produciendo la siguiente salida:

El valor de la resistencia equivalente es 256.89 ohms

Recuerda que el número de ancho de campo incluye la parte entera, la parte decimal y el punto.

Cuando el especificador de formato es %g o %G, el número indica los dígitos significativos.

Si el modificador de precisión se aplica a una cadena, %s, el número antes del punto indicará la longitud mínima de la cadena y el número después del punto determinará la longitud máxima.

printf(“%10.21s” , “La importancia de las resistencias equivalentes”);

imprimirá en pantalla:

La importancia de las

Hay que tener en cuenta que, si la cadena tiene una longitud mayor al ancho máximo de campo, ésta se truncará, tal como ocurrió en el ejemplo anterior.

EJERCICIO DE printf( )

Explica qué hacen las líneas 7,9 y 12 del código de la figura 2:

```
printf("\t Introduce el valor de R1, en ohms: ");
printf("\t Introduce el valor de R2, en ohms: ");
printf("El equivalente serie es %d ohms",R_Total);
```

Otra función de stdio.h que es muy utilizada es scanf( ), esta función permite la comunicación con la entrada estándar del sistema o stdin (standar input) representada por el teclado.

Sintaxis de scanf( ):

```
scanf("e_f",&variable);
```

donde:

scanf( ) función para comunicarse con stdin,
e_f especificador de formato

Es muy importante que observes lo siguiente: el especificador de formato, que depende del tipo de dato, debe ir encerrado entre comillas dobles; después sigue una coma para separar los elementos, que en el capítulo tres llamaremos *argumentos*; antes del nombre de la variable en donde se almacenará el dato leído desde teclado tiene que ir el símbolo & para indicar que es una dirección de memoria. Para identificar los identificadores de formato, puedes volver a consultar la tabla 7.

Ejemplos de scanf( )

Si se requiere leer un número entero y almacenarlo en una variable llamada num, se escribe:

```
scanf("%d",&num);
```

si el tipo de dato de num es flotante, se cambia el especificador de formato por %f.

Puedes leer más de un dato dentro de una sola función scanf( ), desde teclado podrás separar los datos con un espacio en blanco:

```
scanf("%d%d",&num1,&num2);
```

También se puede leer un caracter. Sin embargo, debido a la forma en que se implementa la función scanf( ), almacena la entrada en el buffer de línea y eso puede representar problemas cuando el usuario interactúe con el programa. Para solucionar este inconveniente, hay que limpiar el buffer de entrada como se muestra a continuación:

```
scanf("%c",&codigo);
```

```
fflush(stdin);
```

Es muy importante que consideres que la única ocasión en que no debe ir el símbolo & es cuando se lea una cadena:

```
scanf("%s",mensaje);
```

A veces, cuando se está iniciando en el aprendizaje del lenguaje C, es muy usual la confusión entre un caracter y una cadena, por lo que aquí te muestro unos ejemplos:

Caracteres son un símbolo del código ASCII, tal como una letra o un caracter especial: 'e', 'p', '+'

Cadenas son secuencias de caracteres, tal como un nombre de persona o ciudad: "Enrique", "Aguilar", "Veracruz", "alumno"

EJERCICIO DE scanf( )

Explica qué hacen las líneas 8 y 10 del código de la figura 2:

```
scanf("%d",&R1);
scanf("%d",&R2);
```

En este momento ya debes ser capaz de entender el primer código que escribiste: En las primeras líneas se declaran tres variables de tipo entero para almacenar los dos valores de las resistencias de entrada y el valor de la resistencia equivalente. Se envía un mensaje a la pantalla para pedirle al usuario el valor de una de las resistencias, ese valor se lee y se guarda en la variable R1. Se imprime otro mensaje en la pantalla para pedir el valor de la segunda resistencia, se lee el valor y se guarda en R2. Para obtener la equivalente serie, se suman los dos valores dados por el usuario y el resultado se guarda en la variable R_Total. Finalmente, se envía un mensaje a la pantalla junto con el resultado obtenido.

Entrada del programa: R1 y R2.
Procesamiento: Sumar R1+R2 y guardar el resultado en R_Total.
Salida del programa: R_Total

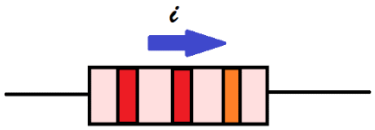
1.9. Ejercicios y Problemas

En este subtema se mostrarán tres problemas que te ayudarán a aprender a plantear tu propio código. Es importante mencionar que los pasos que se muestran sólo son una guía en tu proceso de aprendizaje y que, en el campo de la

codificación, regularmente hay varias propuestas de soluciones a un mismo problema.

Caso de estudio 1: Ley de Ohm

Hacer un programa que calcule el voltaje en las terminales de una resistencia conociendo el valor de dicha resistencia y el valor de la corriente que fluye por ella. El usuario deberá introducir el valor de la resistencia y la corriente dados en ohms y en amperes, respectivamente. La salida será el valor del voltaje en volts.



PROPUESTA DE SOLUCIÓN

a) Analizar el enunciado del problema para identificar Entrada, Salida y Procesamiento del programa.

Entrada: Son los datos que se esperan del usuario.

El valor de la resistencia, en ohms.

El valor de la corriente, en amperes.

Procesamiento: Es el cálculo que se tendrá que hacer para alcanzar la salida.

Usando la Ley de Ohm,

Voltaje=Intensidad x Resistencia

Salida: Es el valor que se espera obtener.

El valor del voltaje, en volts.

b) Definir las variables y constantes que se utilizarán.

Se usarán tres variables de tipo entero para guardar los valores de resistencia, corriente y voltaje, se les llamará R,I y V, respectivamente. Aquí no se usarán constantes.

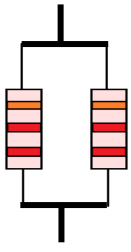
```
#include<stdio.h>

//Programa para calcular el voltaje a través de una resistencia

void main(  )
{
      int V, R, I;
      printf("\t\t===PROGRAMA PARA CALCULAR VOLTAJE USANDO LEY DE OHM===\n");
      printf("  Introduzca el valor de la resistencia en ohms: \n");
      scanf("%d",&R);
      printf("  Introduzca el valor de la corriente en amperes: \n");
      scanf("%d",&I);
      V=I*R;
      printf(" El voltaje a través de la resistencia es: %d volts", V);
}
```

El símbolo // sirve para hacer un *comentario*. Un comentario sólo sirve para hacer algunas anotaciones sobre el código que el compilador ignora en el momento de la traducción.

Recuerda que con printf( ) imprimes en pantalla y con scanf( ) lees un dato desde teclado.

Caso de estudio 2: Resistencia equivalente en paralelo

Hacer un programa que calcule la resistencia equivalente de dos resistencias conectadas en paralelo. Los valores de las resistencias serán en ohms.



PROPUESTA DE SOLUCIÓN

a) Analizar el enunciado del problema para identificar Entrada, Salida y Procesamiento del programa.

Entrada: Son los datos que se esperan del usuario.

El valor de las dos resistencias, en ohms.

Procesamiento: Es el cálculo que se tendrá que hacer para alcanzar la salida.

De acuerdo con la fórmula para obtener resistencia equivalente en paralelo:

$$R_{equiv} = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}$$

Salida: Es el valor que se espera obtener.

El valor de la resistencia equivalente en paralelo.

b) Definir las variables y constantes que se utilizarán.

Se usarán dos variables de tipo entero para guardar los valores de las dos resistencias de entrada y una variable de tipo flotante para la resistencia equivalente; se les llamará R1, R2 y Requiv, respectivamente. Requiv se consideró que debe ser flotante porque será resultado de una división. Aquí no se usarán constantes.

```
#include<stdio.h>
void main ( )
{
  int R1,R2; //declaración múltiple
  float Requiv;
  printf("\n\t APLICACION PARA CALCULAR RESISTENCIA EQUIVALENTE EN PARALELO\n");
  printf("\t Introduce el valor de R1 y R2, en ohms: ");
  scanf("%d%d",&R1,&R2);
  Requiv=1/(1/(float)R1+1/(float)R2);
  printf("El equivalente paralelo de %d y %d es %.2f ohms",R1,R2,Requiv);
}
```

En este código propuesto verás aplicado varios temas expuestos anteriormente, así como también dará oportunidad de explicar otro tema relevante en el lenguaje C.

Comentemos algunos puntos del código:

Ya que hay dos variables del mismo tipo, se hace una declaración múltiple. Si en este punto te preguntas si se pudieron hacer declaraciones sencillas, la respuesta es sí, la declaración múltiple se sugiere para hacer un código más compacto.

Observa que en los printf( ) puedes utilizar las secuencias de escape para darle formato a la cadena que imprimas en pantalla. No es necesario que los utilices en el orden sugerido; prueba colocando doble tabulación "\t\t" para que veas el efecto. También puedes agregar, si lo deseas, algún caracter especial en el mensaje que aparece como encabezado del código:

```
printf("\n\t &&APLICACION PARA CALCULAR RESISTENCIA EQUIVALENTE EN PARALELO&&\n");
```

De igual manera, observa que en el printf( ) que muestra la salida, hay tres especificadores de formato. Aquí intenta escribir tu propia cadena de salida, como por ejemplo:

```
printf("%.2f ohms es el equivalente paralelo de %d y %d es ", Requiv ,R1,R2);
```

sólo recuerda que el número de especificadores de formato debe ser el mismo que el de nombres de variables, toma en cuenta el tipo de dato para escribir el especificador de formato y verifica que la ordenación de los nombres de variables esté de acuerdo con la cadena que mandes a imprimir.

Con la intención de presentar un código más compacto, se leen ambos valores de las resistencias en una sola función scanf( ). Nota la diferencia con el código de la figura 2, donde se pide el valor de cada resistencia de forma individual. En este punto, no es mejor una forma que otra, sólo son diferentes estilos que puedes utilizar.

La parte más interesante de este código está en el procesamiento. Observa muy bien el uso de paréntesis y trata de relacionar lo explicado en el tema de expresiones aritméticas sobre prioridad o jerarquía de los operadores:

```
Requiv=1/(1/(float)R1+1/(float)R2);
```

Intenta quitar algunos paréntesis y ejecuta el código para que compruebes su efecto. Siempre ten en cuenta el orden en que se ejecutan los operadores y agrega todos los paréntesis que sean necesarios para alterar ese orden cuando así se requiera.

Debe llamar tu atención que se escribió dentro de unos paréntesis la palabra float tanto delante de R1 como de R2. Esto se llama hacer un *cast*. Un cast es una conversión explícita de un tipo de dato; se usa cuando deseamos que una variable de cierto tipo se comporte temporalmente como otro tipo que está encerrado entre paréntesis. Será un buen ejercicio averiguar qué sucede si quitas el cast:

```
Requiv=1/(1/R1+1/R2);
```

puede ser que el programa se detenga abruptamente o que te marque un error de división por cero. Lo podrás corregir haciendo el *casting* nuevamente.

Caso de estudio 3: Ley de Coulomb

Hacer un programa que calcule la fuerza entre dos cargas eléctricas usando la ley de Coulomb. Considere la carga en Coulombs y la distancia en metros

PROPUESTA DE SOLUCIÓN

a) Analizar el enunciado del problema para identificar Entrada, Salida y Procesamiento del programa.

Entrada: Son los datos que se esperan del usuario.

Los valores de las cargas, en coulombs.

El valor de la distancia entre las cargas, en metros.

Procesamiento: Es el cálculo que se tendrá que hacer para alcanzar la salida.

Hay que utilizar la fórmula de la Ley de Coulomb:

$$F = K\frac{q_1 * q_2}{r^2}$$

donde $K = 9x10^9 \ \frac{Nm^2}{C^2}$

Salida: Es el valor que se espera obtener.

El valor de la fuerza entre las cargas.

b) Definir las variables y constantes que se utilizarán.

Se usarán dos variables de tipo entero para guardar los valores de las dos cargas: q1 y q2. La variable para la distancia, d, se propone en flotante para darle más rango al usuario. Debido a que los valores de la fuerza son exponenciales, se propone la variable fuerza de tipo double. Además, se usará K como constante tipo double y no como variable, ya que ese valor nunca cambiará.

```
#include <stdio.h>
main()
{
	const double K=9e9;
	int q1,q2;
	float d;
	double fuerza;
	printf("\n\n\t\t ***   LEY DE COULOMB  ***");
	printf("\n La Ley de Coulomb permite calcular la fuerza entre dos cargas\n\n");
	printf("Introduzca los valores de las cargas q1 y q2 en Coulombs y la distancia, d, en
	metros\n");
	printf("\n\tValor de q1: ");
	scanf("%d",&q1);
	printf("\n\tValor de q2: ");
	scanf("%d",&q2);
	printf("\n\tValor de d: ");
	scanf("%f",&d);
	fuerza=K*q1*q2/(d*d);
	printf("\nLa fuerza entre cargas es de %G N",fuerza);
}
```

En este código se pretende que observes dos aspectos: cómo escribir una notación exponencial correctamente:

```
const double K=9e9;
```

y cómo utilizar el especificador de formato %G para que se pueda mostrar correctamente el resultado en notación exponencial:

```
printf("\nLa fuerza entre cargas es de %G N",fuerza);
```

El objetivo es que, al llegar a este punto, ya debes ser capaz de entender estos códigos y escribir tus propios programas. Recuerda que la programación se aprende programando, si es necesario, refuerza tu conocimiento volviendo a leer el subtema que desees.

Referencias y bibliografía:

Kernighan B., Ritchie D. (1978). El lenguaje de programación C. Prentice Hall.

García-Bermejo Giner, J. R. (2008). Programación estructurada en C. Pearson Educación.

Jiménez Castells, M. y Otero Calviño, B. (2015). Fundamentos de ordenadores: programación en C. Universitat Politècnica de Catalunya.

Menchaca García, F. R. (2010). Fundamentos de programación en Lenguaje C. Instituto Politécnico Nacional.

Joyanes Aguilar, L. (2005). C algoritmos, programación y estructuras de datos. McGraw-Hill España.

Gaxiola Pacheco, C. G. y Flores Gutiérrez, D. L. (2008). Metodología de la programación con pseudocódigo enfocado al lenguaje C. Plaza y Valdés, S.A. de C.V.

Moreno Pérez, J. C. (2015). Programación. RA-MA Editorial.

Shildt H (2000) C The complete reference. McGrawHill

Capítulo 2

ESTRUCTURAS DE SELECCIÓN Y REPETICIÓN

OBJETIVO GENERAL

Al finalizar el capítulo, el estudiante será capaz de hacer programas que involucren estructuras de selección y repetición.

OBJETIVOS PARTICULARES:

- Conocer y aplicar estructuras de selección.
- Conocer y aplicar estructuras de repetición.

2.1 Introducción

En el capítulo anterior aprendiste a escribir programas básicos donde el flujo de ejecución de las instrucciones se realizaba en forma secuencial; en este capítulo aprenderás a modificar ese flujo de ejecución a través de las estructuras de selección y de repetición. Las estructuras de selección permiten que el usuario o el mismo programa escoja qué bloque de código se ejecutará, como se ilustra en la figura 13. Las estructuras de repetición permiten hacer ciclos, esto es, hacen que un bloque de código se ejecute más de una vez, la generalización del funcionamiento de estas estructuras se muestra en la figura 14.

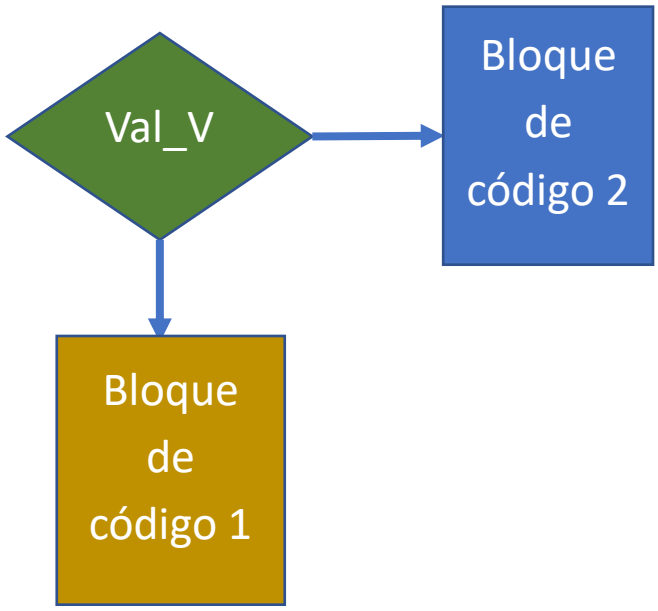


Figura 13. Ilustración sobre el funcionamiento de las estructuras de selección.

Las estructuras de selección que se estudiarán en este capítulo son:

- if
- switch case

mientras que las de repetición son:

- for
- while
- do while

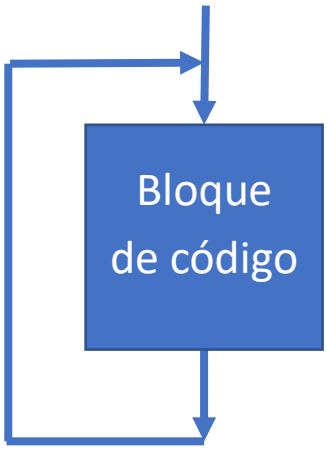


Figura 14. Ilustración sobre el funcionamiento de las estructuras de repetición.

2.2 Estructura if

La estructura sencilla de *if* funciona de la siguiente manera: se evalúa una condición, si la condición es verdadera se ejecuta la instrucción (que a partir de ahora conoceremos como sentencia) que está a continuación, si es falsa no se ejecuta, ver la figura 15.

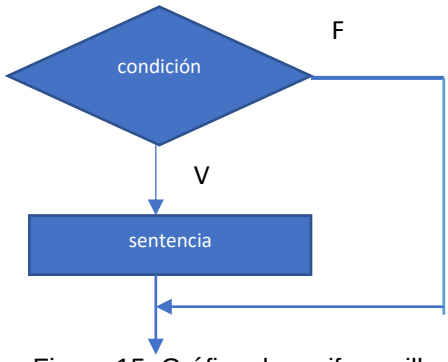


Figura 15. Gráfica de un if sencillo.

Sintaxis de un if sencillo:

```
if ( condición) sentencia;
```

donde:

if palabra reservada,
condición es una expresión por evaluar,
sentencia instrucción a ser ejecutada.

Observa que la condición va entre paréntesis y que la sentencia termina con un punto y coma. Para poder construir las expresiones por evaluar se utilizan operadores relacionales y operadores lógicos. Como no es necesario que vayan ambos operadores en una condición, primero estudiaremos los relacionales (tabla 9).

Tabla 9. Operadores relacionales.

Operador	Significado
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
==	Igual
!=	Diferente

Sintaxis para usar los operadores relacionales:

```
exp op  exp
```

donde:

exp expresión aritmética,
op operador relacional.

Ejemplos de uso de operadores relacionales

promedio < 10

valor1 == valor2

n<=(a+b)

Ejemplo de if sencillo

if (suma>20) printf(“Valor límite rebasado”);

en este ejemplo, se evalúa si el valor que se encuentra almacenado en la variable suma es mayor que 20; si es verdadero, se imprime un mensaje en la pantalla, si es falso, no se imprime.

Un valor verdadero en lenguaje C es cualquier valor diferente de cero, un falso es un cero. Observa que después de la condición no lleva punto y coma.

EJERCICIO RESUELTO DE if SENCILLO

Escribe un programa que pida tres valores de resistencias, si su equivalente serie es mayor que 100, que imprima un mensaje diciendo que intente con otros valores.

Solución

```
#include <stdio.h>
main()
{
int R1,R2,R3,Requiv;
printf("Introduzca tres valores de resistencias\n");
scanf("%d%d%d",&R1,&R2,&R3);
Requiv=R1+R2+R3;
if(Requiv>100) printf("\nIntente con otros valores!!!\a");
printf("Requiv= %d",Requiv);
}
```

Observa que el valor de la resistencia equivalente siempre se imprime, pero el mensaje sólo se imprime si se cumple la condición.

:

Acabas de comprobar que para que se ejecute la sentencia que está al lado de if, se tiene que cumplir una condición. Sin embargo, hay situaciones en las que se requiere que se cumplan más de una condición. Para unir las condiciones se utilizan los operadores lógicos, en la tabla 10 se muestran las que utilizaremos en este libro.

Tabla 10. Operadores lógicos.

Operador	Significado
&&	Y
‖	O
!	NO

El operador lógico && genera un valor verdadero sólo cuando los dos operandos son verdaderos, si alguno de los operandos es falso, generará un falso. El operador || genera un valor verdadero si al menos uno de sus operandos es verdadero, sólo genera un falso cuando ambos operandos son falsos.

Ejemplos de uso de operadores lógicos

(capacitor1 == capacitor2) && (resistencia1!=resistencia2)

(capacitor1<capacitor2) || (resistencia1<resistencia2)

En el primer ejemplo, el operador && generará un valor verdadero sólo si el valor del capacitor1 es igual al del capacitor2 **Y** el valor de la resistencia1 sea diferente de la resistencia2. En el segundo ejemplo, el operador || generará un verdadero cuando el valor del capacitor1 sea menor que el del capacitor2 **O** el valor de la resistencia1 sea menor que el valor de la resistencia2.

La estructura sencilla del if no altera el flujo de ejecución de un programa cuando la condición por evaluar es falsa, sin embargo, en muchas situaciones es deseable que se ejecute cierto código en caso de que la condición no se cumpla. Para esas situaciones se utiliza la estructura *if else* cuya gráfica se muestra en la figura 16.

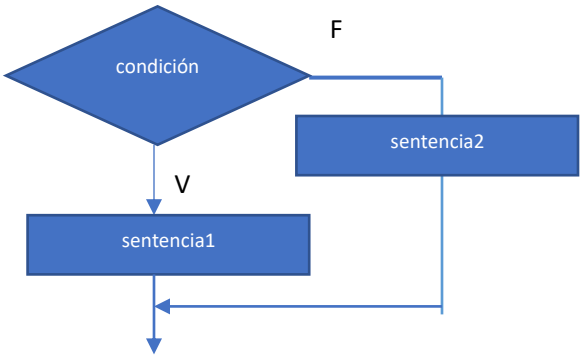


Figura 16. Gráfica de un if else.

Sintaxis de if else:

```
if ( condición) sentencia1;
  else sentencia2;
```

donde:

if , else son palabras reservadas,
condición es una expresión por evaluar,
sentencia1 instrucción a ser ejecutada cuando la condición sea verdadera,
sentencia2 instrucción a ser ejecutada cuando la condición sea falsa.

Ejemplo de if else

```
if (temperatura<30) printf (“Temperatura ambiental agradable :)”);
   else printf(“Temperatura ambiental calurosa”);
```

en este ejemplo, se evalúa si el valor de la variable temperatura es menor que 30; si es verdadero, se imprime *Temperatura ambiental agradable :),* pero si es falso, esto es, si el valor de temperatura es igual o mayor que 30, se imprime *Temperatura ambiental calurosa*.

EJERCICIO RESUELTO DE if else

Escribe un programa que indique el funcionamiento de cierto dispositivo electrónico en base a sus valores de voltaje de salida. Si el valor del voltaje se encuentra en el intervalo [10, 15] V que imprima en pantalla: *El dispositivo funciona correctamente*, en caso contrario, *El dispositivo presenta fallas*.

Solución

```
#include<stdio.h>
#define MAX 15
#define MIN 10
main()
{
float V;
printf("\nAPLICACION DE DIAGNOSTICO DE DISPOSITIVO ELECTRONICO\n");
printf("\n\t Valor de voltaje: ");
scanf("%f",&V);
if((V>=MIN) &&(V<=MAX))
  printf("\n El dispositivo funciona correctamente");
  else
      printf("\n El dispositivo presenta fallas!\a\a\a");
printf("\n DIAGNOSTICO FINALIZADO");
}
```

Observa el uso de las directivas #define para el uso de las constantes, aunque se hubiera podido usar directamente el número 10 y 15 en las condiciones, es una buena práctica hacerlo como aquí se muestra.
Otra buena práctica de programación es la *indentación*, que no es nada más que colocar el inicio de la línea de la sentencia más a la derecha. Observa su utilización en las líneas del if else.

:

Hasta el momento, hemos usado sólo un printf( ) en la parte de las sentencias de if. Sin embargo, debes tener presente que puedes escribir un *bloque de código*. El concepto de *bloque de código* es una serie de sentencias encerradas entre llaves y que se considera como una sola unidad; dentro de él puede ir, si es necesario, otra estructura if o if else, a esto se le conoce como *if anidados*, gráficamente puede tener un aspecto como el de la figura 17.

Analizando la gráfica, vemos que primero se evalúa una condición, si es verdadera, se evalúa la condición de un if más interno, se le dice interno porque va dentro de otro if. Es importante mencionar que pueden existir otras variantes a la gráfica de la figura 17, por ejemplo, el if interno puede ir del lado falso de la condición, if else if, así como haber más de dos estructuras if. Observa la sintaxis para un if else if y después veamos su uso a través del siguiente ejercicio resuelto.

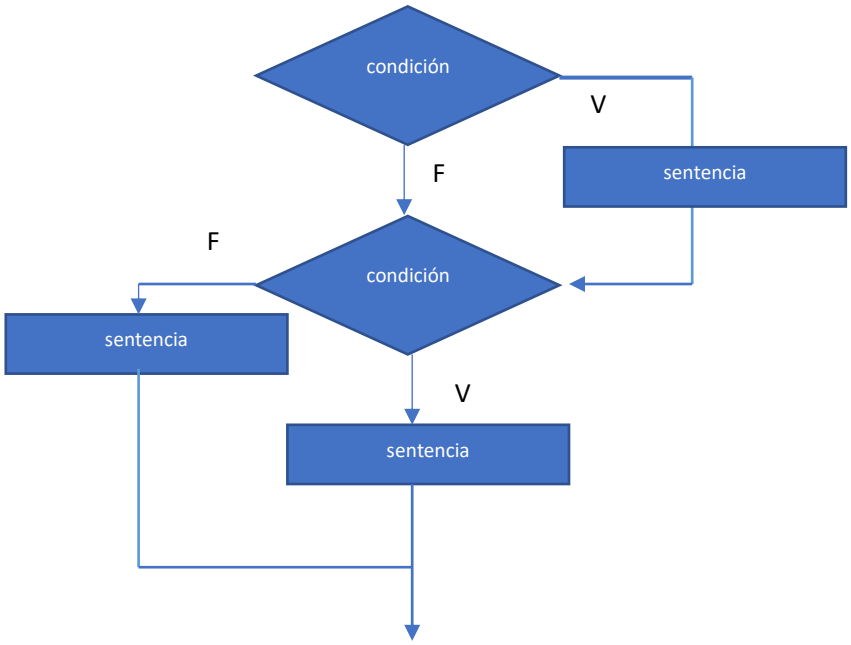


Figura 17. Gráfica de un if anidado

Sintaxis de if else if:

```
if (condición) bloque de código;
    else
        if (condición) bloque de código;
```

donde:

if , else son palabras reservadas,
condición es una expresión por evaluar,
bloque de código serie de sentencias encerradas entre { }

EJERCICIO RESUELTO DE if ANIDADOS

Escribe un programa que le dé al usuario la opción de calcular voltaje, corriente o resistencia, según la Ley de Ohm.

Solución

```
#include<stdio.h>
main()
{
        int opcion,V,I,R;
        printf("\n\n\t\t\t\t LEY DE OHM");
        printf("\n\t\t 1. CALCULO DE CORRIENTE");
        printf("\n\t\t 2. CALCULO DE VOLTAJE");
        printf("\n\t\t 3. CALCULO DE RESISTENCIA");
        printf("\n\t\t Escoge una opcion...");
        scanf("%d",&opcion);
        if (opcion==1)  //opción de corriente
         {
                printf("\n\t\t Valor de voltaje en volts: ");
                scanf("%d",&V);
                printf("\n\t\t Valor de resistencia en ohms: ");
                scanf("%d",&R);
                printf("\n Circula una corriente de %.2f amperes",(float)V/R);
         }
         else
           if (opcion==2)  //opción de voltaje
           {
                  printf("\n\t\t Valor de corriente en amperes: ");
                  scanf("%d",&I);
                  printf("\n\t\t Valor de resistencia en ohms: ");
                  scanf("%d",&R);
                  printf("\n Hay un voltaje de %d volts",I*R);
           }
            else
              if (opcion==3)  //opción de resistencia
              {
                      printf("\n\t\t Valor de voltaje en volts: ");
                      scanf("%d",&V);
                      printf("\n\t\t Valor de corriente en amperes: ");
                      scanf("%d",&I);
                      printf("\n Hay una resistencia de %.2f ohms",(float)V/I);
              }
              else
               printf("\n Opcion no v%clida",160);
}
```

Observa en el ejercicio anterior, el uso del casting, el especificador de precisión en el flotante, la indentación, el uso de bloques de código en los if else y, lo novedoso aquí, el uso de expresiones aritméticas dentro del printf ( ); como no se requiere almacenar los resultados de los cálculos porque no se vuelven a utilizar, y como son operaciones sencillas, se puede escribir como se propone en ese código. Realmente, no es necesario hacerlo así para que funcione el programa, también se pudo hacer primero el cálculo y luego mostrar el resultado, tal como se hizo en los ejercicios anteriores. Recuerda que en la programación no hay una sola solución ante un problema.

2.3 Estructura switch case

A diferencia de un if else que sólo permite escoger entre dos opciones, verdadero o falso, la estructura *switch case* permite una selección entre múltiples opciones. Funciona de la siguiente manera: evalúa una variable, que sólo puede ser de tipo entero o caracter, y dependiendo de su valor, ejecutará cierto bloque de código. En la figura 18 se muestra una generalidad del funcionamiento de esta estructura. Es importante mencionar que al igual que las estructuras if, también es permitido tener switch case anidados, esto es, un switch case dentro de otro switch case externo.

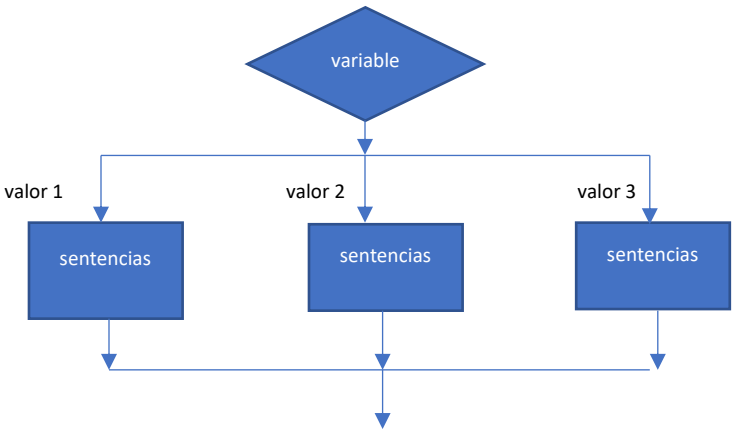


Figura 18. Gráfica de switch case.

A continuación, veamos la forma general de la sintaxis de switch case para posteriormente comentar algunos puntos importantes por considerar cuando utilices esta estructura.

Sintaxis de switch case:

```
switch ( var )
  {
     case valor1:
                    sentencias;
                     break;
      case valor2:
                    sentencias;
                     break;
      …
       case valorn:
                    sentencias;
                     break;
       default:
                sentencias;
   }
```

donde:

switch, case, break, default son palabras reservadas de C
var es el nombre de la variable a ser evaluada
valor1, valor2, valor son los posibles valores que puede tener la variable evaluada.

Analicemos con más detalle el funcionamiento de la selección múltiple. La variable var tiene que ser de tipo entero o caracter y puede tener múltiples valores, valor1, valor2, …, valorn. Primeramente, var es evaluada, supongamos que tiene almacenado valor5; posteriormente, el compilador compara valor5 con el valor del primer case (valor1), como no son iguales se pasa al siguiente case (valor2), y así continúa hasta que encuentre una igualdad, en este caso, sería hasta el quinto case. Cuando el compilador encuentra una igualdad, ejecuta las sentencias que se encuentren debajo de ese case hasta donde encuentre escrito un *break*, por esta razón, es muy importante asegurarse que se haya escrito al final de cada case. El *default* es opcional.

Si var es de tipo caracter, recuerda encerrar entre comillas sencillas los valores que escribas en los case: ‘valor1´. En general, no deben repetirse los valores de los case, pero sí puede suceder que más de un case permita que se ejecuten las mismas sentencias, tal como se muestra en el siguiente ejemplo.

Ejemplo de switch case

```
switch (día)
{
  case 5:
          printf(“Hoy toca el servicio de mantenimiento a caldera”);
          break;
case14:
case 28:
         printf(“Hoy toca el servicio de mantenimiento a las válvulas de seguridad”);
         break;
}
```

EJERCICIO RESUELTO DE switch case

Escribe un código que le de al usuario la opción de calcular el equivalente serie o paralelo de dos capacitancias cuyos valores están en µF.

Solución

```
#include<stdio.h>
main()
{
        int opcion,C1,C2;
        float Cequiv;
        printf("\t\t CALCULO DE CAPACITANCIAS EQUIVALENTES");
        printf("\n\t\t Introduzca el primer valor de capacitancia en %cF: ",230);
        scanf("%d",&C1);
        printf("\n\t\t Introduzca el segundo valor de capacitancia en %cF: ",230);
        scanf("%d",&C2);
        printf("\n\t\t Escoge una opcion...");
        printf("\n\t\t 1. EQUIVALENTE SERIE");
        printf("\n\t\t 2. EQUIVALENTE PARALELO\n");
        scanf("%d",&opcion);
        switch (opcion)
        {
                case 1:
                        Cequiv=C1*C2/(float)(C1+C2);
                        printf("\n El equivalente serie de %d %cF y %d %cF es %f %cF",
C1,230,C2,230,Cequiv,230);
                        break;
                case 2:
                        Cequiv=C1+C2;
                        printf("\n El equivalente paralelo de %d %cF y %d %cF es %f %cF",
C1,230,C2,230,Cequiv,230);
                        break;
                default:
                        printf("\n\t\t Opcion incorrecta\a");
        }
}
```

2.4 Estructura for

Una de las estructuras que el lenguaje C tiene para hacer iteraciones o ciclos es *for*, El concepto de iteraciones es repetir un bloque de código hasta que se alcance cierta condición. En la figura 19 se muestra gráficamente el comportamiento de for, hay un bloque de código que se repite mientras la condición sea verdadera, cuando se vuelve falsa, se rompe el ciclo. La parte del código que se repite se llama *cuerpo del ciclo*.

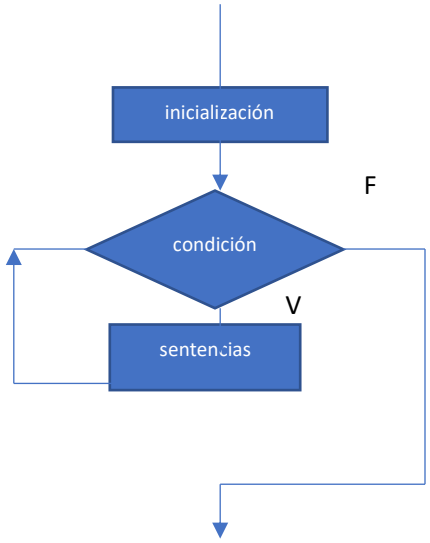


Figura 19. Gráfica del funcionamiento general de un ciclo for.

Sintaxis de for:

```
for(inicialización; condición; incremento) sentencias;
```

donde:

for palabra reservada,
inicialización expresión aritmética donde se inicializa la variable que lleva el conteo,
condición expresión relacional donde va la condición que se evalúa,
incremento expresión aritmética donde se actualiza la variable que lleva el conteo.

Aunque usar *for* con las tres expresiones dentro de los paréntesis, tal como se ve en la sintaxis, es la forma más usual de encontrarlo, puede prescindir de una, dos o las tres expresiones, lo que no puede faltar son los punto y coma. Esto es, escribir for (; ;) ; es válido, pero representa un ciclo infinito, en otras palabras, el sistema se queda ciclado. También es relevante mencionar que a pesar de que la tercera expresión se llame incremento, realmente puede ser un decremento. A continuación, se presentan algunos ejemplos básicos de cómo utilizar el ciclo for y se explicará su funcionamiento:

Ejemplos de for

- for (i=0;i<3;i++) printf(“ Hola :)\n”);

1) Se empieza por la inicialización, aquí hay una expresión aritmética que establece el valor de la variable i igual a cero.
2) Se evalúa la expresión relacional: 0<3, dando un verdadero.
3) Entra al cuerpo del ciclo y se imprime “Hola” en la pantalla.
4) En la parte del incremento hay una expresión aritmética que hace que el nuevo valor de i sea uno.
5) Se evalúa la expresión relacional: 1<3, dando un verdadero.
6) Entra al cuerpo del ciclo y se imprime “Hola” en la pantalla.
7) En la parte del incremento se cambia el valor de i a dos.
8) Se evalúa la expresión relacional: 2<3, dando un verdadero.
9) Entra al cuerpo del ciclo y se imprime “Hola” en la pantalla.
10) En la parte del incremento se cambia el valor de i a tres.
11) Se evalúa la expresión relacional: 3<3, dando un falso y rompiendo el ciclo.

- for (i=3;i>0;i--) printf(“ Hola :)\n”);

1) Se empieza la expresión aritmética que establece el valor de la variable i igual a tres.
2) Se evalúa la expresión relacional: 3>0, dando un verdadero.
3) Entra al cuerpo del ciclo y se imprime “Hola” en la pantalla.
4) En la parte del incremento hay una expresión aritmética que hace que el nuevo valor de i sea dos.
5) Se evalúa la expresión relacional: 2>0, dando un verdadero.
6) Entra al cuerpo del ciclo y se imprime “Hola” en la pantalla.
7) En la parte del incremento se cambia el valor de i a uno.
8) Se evalúa la expresión relacional: 1>0, dando un verdadero.
9) Entra al cuerpo del ciclo y se imprime “Hola” en la pantalla.
10) En la parte del incremento se cambia el valor de i a 0.
11) Se evalúa la expresión relacional: 0>0, dando un falso y rompiendo el ciclo.

- for (i=2;i<=6;i+=2) printf(“%d”,i);

1) Se empieza la expresión aritmética que establece el valor de la variable i igual a dos.
2) Se evalúa la expresión relacional: 2<=6, dando un verdadero.
3) Entra al cuerpo del ciclo y se imprime “2” en la pantalla.
4) En la parte del incremento hay una expresión aritmética que hace que el nuevo valor de i sea cuatro.
5) Se evalúa la expresión relacional:4<=6 , dando un verdadero.
6) Entra al cuerpo del ciclo y se imprime “4” en la pantalla.
7) La expresión aritmética del incremento hace que el nuevo valor de i sea seis.
8) Se evalúa la expresión relacional:6<=6 , dando un verdadero.
9) Entra al cuerpo del ciclo y se imprime “6” en la pantalla.
10) La expresión aritmética del incremento hace que el nuevo valor de i sea ocho.
11) Se evalúa la expresión relacional: 8<=6, dando un falso y rompiendo el ciclo.

EJERCICIO RESUELTO DE for

Escribe un código que obtenga la resistencia equivalente de diez resistencias conectadas en serie.

Solución

```
#include<stdio.h>
void main ( )
{
        int i,R, R_Total=0;
        for(i=1;i<=10;i++)
        {
                printf("\t Introduce el valor de la resistencia %d en ohms: ",i);
                scanf("%d",&R);
                R_Total=R_Total+R;
        }
        printf("El equivalente serie es %d ohms",R_Total);
}
```

En el ejercicio resuelto se puede observar la estrategia de usar una variable como *acumulador.* Analicemos el funcionamiento del ejercicio propuesto, primero se inicializa la variable que funcionará como acumulador. *Inicializar* una variable significa darle un valor antes de usarla, observa en la línea de las declaraciones, las variables i y R se declaran y R_Total se inicializa a cero. En el cuerpo del ciclo vemos la sentencia:

R_Total=R_Total+R;

esto hace que el valor de R que dé el usuario se sume con el valor de R_Total, que es cero y el resultado de la suma será el nuevo valor de R_Total. Conforme se

continúe el ciclo, el valor que se guarda en R_Total irá cambiando al sumarle el nuevo valor de resistencia dado por el usuario.

También se puede usar una variable para ir acumulando valores parciales de productos. La diferencia es que la variable que funcione como acumulador se inicializa con uno; además, la sentencia queda como:
Acumulador=Nuevo_valor*Acumulador

2.5 Estructura while

while es otra estructura que tiene el lenguaje C para hacer iteraciones. Su funcionamiento se puede resumir de la siguiente manera: Evalúa una condición, si es verdadera, entra al cuerpo del ciclo; si es falsa, no. En otras palabras, el bloque de código que sigue después de la condición se repite mientras la condición sea verdadera. Por esta característica, se recomienda su uso cuando antes de ejecutar cierto bloque de código, se tiene que cumplir alguna condición. Gráficamente, este funcionamiento nos lleva nuevamente a la figura 19.

Sintaxis de while:

```
while(condición) sentencias;
```

donde:

while palabra reservada,
condición expresión relacional que se evalúa.

Ejemplo de while

```
while (n<3)
    {
        printf (“Hola\n”);
        n++;
     }
```

Supongamos que n tiene un valor inicial de 1

1) Se evalúa la expresión relacional: 1<3, dando un verdadero.
2) Entra al cuerpo del ciclo y se imprime “Hola” en la pantalla.
3) El valor de n se incrementa a dos.
4) Se evalúa la expresión relacional:2<3, dando un verdadero.
5) Entra al cuerpo del ciclo y se imprime “Hola” en la pantalla.
6) El valor de n se incrementa a tres.
7) Se evalúa la expresión relacional:3<3, dando un falso y rompiendo el ciclo.

Observa que, si el valor inicial de n no cumpliera la condición, tal como n=5 al evaluar la expresión relacional: 5<3, daría un falso y no se entraría al ciclo.

EJERCICIO RESUELTO DE while

Escribe un código que obtenga la resistencia equivalente de diez resistencias conectadas en serie.

Solución

```
#include<stdio.h>
void main ( )
{
        int i=1,R,R_Total=0;
        while (i<11)
        {
                printf("\t Introduce el valor de la resistencia %d en ohms: ",i);
                scanf("%d",&R);
                R_Total=R_Total+R;
                i++;
        }
        printf("El equivalente serie es %d ohms",R_Total);
}
```

2.6 Estructura do while

A diferencia de while, la estructura *do w*hile primero ejecuta el bloque de código escrito en su cuerpo y después evalúa la condición para determinar si lo repite. Debido a esta característica se utiliza ampliamente en los programas donde se imprime un menú al usuario. Gráficamente, también se comporta como se muestra en la figura 19.

Sintaxis de do while:

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

donde:

do while palabras reservadas,
condición expresión relacional que se evalúa.

Ejemplo de do while

```
do
    {
        printf (“Hola\n”);
        n++;
     } while (n<3);
```

Supongamos que n tiene un valor inicial de 1

1) Entra al cuerpo del ciclo y se imprime “Hola” en la pantalla.
2) El valor de n se incrementa a dos.
3) Se evalúa la expresión relacional:2<3, dando un verdadero.
4) Entra al cuerpo del ciclo y se imprime “Hola” en la pantalla.
5) El valor de n se incrementa a tres.
6) Se evalúa la expresión relacional:3<3, dando un falso y rompiendo el ciclo.

Observa que, si el valor inicial de n no cumpliera la condición, tal como n=5, de todas maneras se imprimiría un “Hola” en la pantalla ya que no se evalúa al inicio, sino al finalizar el cuerpo del ciclo para saber si continúa la iteración. Así, al evaluar la expresión relacional: 6<3 (es 6 porque el valor de n se incrementa en el cuerpo del ciclo) daría un falso y se rompería el ciclo.

.

EJERCICIO RESUELTO DE do while

Escribe un código que obtenga la resistencia equivalente de diez resistencias conectadas en serie.

Solución

```
#include<stdio.h>
void main ( )
{
        int i=1,R,R_Total=0;
        do
        {
                printf("\t Introduce el valor de la resistencia %d en ohms: ",i);
                scanf("%d",&R);
                R_Total=R_Total+R;
                i++;
        }while (i<11);
        printf("El equivalente serie es %d ohms",R_Total);
}
```

Como pudiste observar, tanto en while como en do while se usó el mismo ejemplo de imprimir tres veces la palabra “Hola” en la pantalla y se planteó el mismo problema en el ejercicio resuelto. La intención es mostrar que puedes escoger cualquiera de las dos estructuras para resolver un problema de iteración. La estructura for también se pudo usar para resolver el ejercicio, sin embargo, es más común su uso para situaciones donde se lleva un contador del número de iteraciones.

Finalmente, hay que mencionar que tanto las estructuras de selección como las de repetición se pueden anidar. Esto es, puedes escribir un do while dentro de un for y todo esto dentro de un switch case, por ejemplo.

2.7 Ejercicios y Problemas

En este subtema se presentan dos casos de problemas clásicos de electrónica: uso de la Ley de Ohm y cálculo de resistencias equivalentes. Los códigos integran el uso de estructuras de selección y estructuras de repetición. Antes de cada código, se detallan algunos puntos que se tuvieron en consideración en el análisis del problema. La intención es que con esos puntos y observando el código propuesto, seas capaz de hacer tu propia propuesta de solución.

Caso de estudio 1: Ley de Ohm con menú

Hacer un programa que presente un menú donde el usuario pueda escoger calcular voltaje, corriente, resistencia o salir del programa. Los valores de voltaje, de resistencia y de corriente serán dados en volts, ohms y en amperes, respectivamente.

PROPUESTA DE SOLUCIÓN

- Se usarán tres variables de tipo flotante para guardar los valores de resistencia, corriente y voltaje, se les llamará R,I y V, respectivamente; el hecho de que sean flotantes permitirá obtener un valor de I y R con punto decimal sin tener que hacer un cast.

- Para cambiar el flujo de ejecución en cada uno de los casos que escoja el usuario: cálculo de voltaje, corriente, resistencia o salir del programa, se utilizará un switch case.

- La variable que guardará el valor de la opción escogida por el usuario será de tipo char, y sus posibles valores serán 'V', 'C', 'R', 'S'; para hacer más flexible el programa, también podrá valer 'v', 'c', 'r', 's'. Aquí cabe mencionar que la razón de haber escogido este tipo de dato es sólo fin didáctico.

- Ya que se usará el scanf() para leer un caracter, hay que limpiar el buffer de entrada con la función fflush(stdin).

- Para hacer las iteraciones se utilizará un do while para que el menú se muestre al menos una vez. El ciclo debe seguir mientras el usuario no decida salir del sistema, en otras palabras, mientras su opción no sea 'S' ni 's'.

```
#include <stdio.h>
main()
{
      char opcion;
      float I,V,R;
      do{
         printf("\n\t\t LEY DE OHM");
         printf("\n\t\t (V)OLTAJE");
         printf("\n\t\t (C)ORRIENTE");
         printf("\n\t\t (R)ESISTENCIA");
         printf("\n\t\t (S)ALIR");
```

```
        printf("\n\t\t Escoge una de las opciones...");
        fflush(stdin);
        scanf("%c",&opcion);
        switch (opcion)
        {
        case 'V':// cálculo de voltaje
        case 'v':
                printf("\n\t\t Introduzca el valor de corriente en amperes: ");
            scanf("%f",&I);
            printf("\n\t\t Introduzca el valor de resistencia en ohms: ");
            scanf("%f",&R);
            V=I*R;
            printf("\n\t\t Hay un voltaje de %.2f volts",V);
            break;
          case 'C': //cálculo de corriente
                case 'c':
                   printf("\n\t\t Introduzca el valor de voltaje en volts: ");
            scanf("%f",&V);
                printf("\n\t\t Introduzca el valor de resistencia en ohms: ");
                scanf("%f",&R);
                I=V/R;
            printf("\n\t\t Circula una corriente de %.2f amperes",I) ;
            break;
          case 'R': //cálculo de resistencia
          case 'r':
                printf("\n\t\t Introduzca el valor de voltaje en volts: ");
            scanf("%f",&V);
            printf("\n\t\t Valor de corriente en amperes: ");
            scanf("%f",&I);
            R=V/I;
            printf("\n\t\t Hay una resistencia de %.2f ohms",R);
            break;
          case 'S':
                case 's':
                   break;
                default:
                   printf("\n\t Opcion incorrecta...Intente de nuevo");
        }
      }while(opcion!='S'&& opcion!='s');
}
```

Recuerda que el símbolo // sirve para hacer un *comentario* y lo puedes agregar para ayudar a leer tu código.

Caso de estudio 2: Resistencias equivalentes con menú

Hacer un programa que presente un menú al usuario donde pueda escoger calcular resistencias equivalentes en serie o en paralelo. Para el cálculo de las equivalencias, el sistema deberá ir pidiendo el valor de cada resistencia que el usuario quiera reducir, cuando el valor ingresado sea cero, el sistema dejará de pedir valores. Los valores de las resistencias serán en ohms. El programa sólo terminará cuando el usuario decida salir del programa.

PROPUESTA DE SOLUCIÓN

- Se usará una variable de tipo flotante para almacenar el valor de resistencia que vaya dando el usuario; dos variables de tipo entero, una para guardar la opción del usuario y otra para el número de resistencia; aunque este último no lo pida el enunciado del problema, se agrega para facilidad del usuario. Para guardar la resistencia equivalente se utilizará una variable de tipo flotante, esta servirá tanto para la equivalente paralelo como para la equivalente serie. Además, se utilizará otra variable de tipo flotante que servirá como auxiliar en el cálculo de la suma del recíproco de la resistencia.

- Para cambiar el flujo de ejecución en cada uno de los casos que escoja el usuario se utilizará un switch case.

- La variable que guardará el valor de la opción escogida por el usuario será de tipo int, y sus posibles valores serán 1,2,3.

- Dentro de los casos 1 y 2 habrá otra estructura de repetición, se usará el do while para que pida el valor de la resistencia al menos una vez. Este do while se repetirá mientras el valor de la resistencia no sea cero.

- En ese mismo ciclo se deberá inicializar a cero la variable que actúe como acumulador para que no almacene resultados anteriores.

- En el caso de la equivalente paralelo se sumará primero el inverso de cada resistencia; finalmente, se obtendrá el inverso de la suma para calcularla según la fórmula.

- En ese mismo caso, se usará un if para saber si el valor de la resistencia es cero antes de intentar sacar el inverso y así evitar un error de división entre cero. Si es cero, se saldrá del ciclo con un *break*.

- Se utilizará un do while externo para mostrar el menú del programa. El ciclo debe seguir mientras el usuario no decida salir del sistema, en otras palabras, mientras su opción no sea 3.

```
#include <stdio.h>
main()
{
      int opcion,n;
      float R, Requiv, Rinverso;
      do{
         printf("\n\t\t CALCULO DE RESISTENCIAS EQUIVALENTES");
         printf("\n\t\t 1) SERIE");
      printf("\n\t\t 2) PARALELO");
         printf("\n\t\t 3) SALIR");
         printf("\n\t\t Escoge una de las opciones...");
         scanf("%d",&opcion);
         switch (opcion)
         {
         case 1 :// equivalente serie
                  printf("\n\t\t RESISTENCIA EQUIVALENTE SERIE ");
                  Requiv=0;
                  n=1;
                  do{
                           printf("\n\t\t Introduzca el valor de R%d: ",n);
                 scanf("%f",&R);
                           Requiv=Requiv+R;
                           n++;
                           }while(R!=0);
              printf("\n\t\t Resistencia equivalente= %.2f ohms",Requiv);
              break;
            case 2 : //equivalente paralelo
                           printf("\n\t\t RESISTENCIA EQUIVALENTE PARALELO ");
                  Rinverso=0;
                  n=1;
                  do{
                           printf("\n\t\t Introduzca el valor de R%d: ",n);
                 scanf("%f",&R);
                 if(R==0)break;
                           Rinverso=Rinverso+1/R;
```

```
                        n++;
                        }while(R!=0);
                        Requiv=1/Rinverso;
                        printf("\n\t\t Resistencia equivalente= %.2f ohms",Requiv);
            break;
               case  3 :
                  break;
               default:
                  printf("\n\t Opcion incorrecta...Intente de nuevo");
         }
      }while(opcion!=3);
}
```

Recuerda que la programación se aprende programando, si es necesario, refuerza tu conocimiento volviendo a leer el subtema que desees.

Referencias y bibliografía:

Kernighan B., Ritchie D. (1978). El lenguaje de programación C. Prentice Hall.

García-Bermejo Giner, J. R. (2008). Programación estructurada en C. Pearson Educación.

Jiménez Castells, M. y Otero Calviño, B. (2015). Fundamentos de ordenadores: programación en C. Universitat Politècnica de Catalunya.

Menchaca García, F. R. (2010). Fundamentos de programación en Lenguaje C. Instituto Politécnico Nacional.

Joyanes Aguilar, L. (2005). C algoritmos, programación y estructuras de datos. McGraw-Hill España.

Gaxiola Pacheco, C. G. y Flores Gutiérrez, D. L. (2008). Metodología de la programación con pseudocódigo enfocado al lenguaje C. Plaza y Valdés, S.A. de C.V.

Moreno Pérez, J. C. (2015). Programación. RA-MA Editorial.

Shildt H (2000) C The complete reference. McGrawHill

Capítulo 3

PROGRAMACIÓN MODULAR

OBJETIVO GENERAL

Al finalizar el capítulo, el estudiante será capaz de hacer programas usando funciones propias del lenguaje y funciones definidas por el usuario.

OBJETIVOS PARTICULARES:

- Identificar funciones propias del lenguaje C.
- Saber hacer funciones definidas por el usuario.

3.1 Introducción

Ya en estos momentos, debes ser capaz de escribir tu propio código utilizando tanto las estructuras de control de selección como las de ciclos, esto es, ya debes ser capaz de hacer programación estructurada. Sin embargo, hasta el momento, sólo haz codificado en la parte correspondiente a la función principal *main ( ).* En este capítulo aprenderás a escribir tu código en otros espacios conocidos como bloques o módulos, es decir, aprenderás a hacer programación modular.

La programación modular es un paradigma o estilo de programación en donde el código de la solución a un problema se divide en varios módulos. También en la electrónica se puede observar la modularidad. En la figura 20 se observa una placa electrónica a la cual se le puede agregar más funcionalidad conectando otros módulos a través de las ranuras de extensión. La función primordial del circuito electrónico está en la placa principal, pero si se llegara a requerir otras funciones, no sería necesario rehacer la placa principal para agregarle los circuitos requeridos, bastará con conectarle otros módulos. De igual forma, cuando ya no se requieran, se podrán remover de la placa principal.

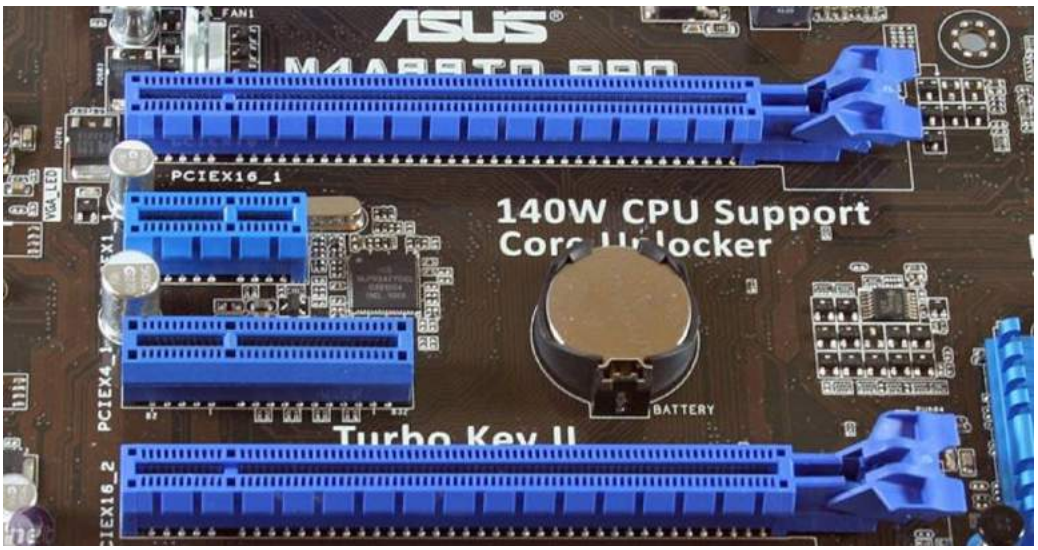


Figura 20. Tarjeta electrónica con ranuras de extensión. Imagen tomada de https://hardzone.es/reportajes/que-es/pci-express-caracteristicas/. Descargada febrero del 2023

Dos ventajas de la programación modular son: Reducir la complejidad de la solución y, permitir la reutilización. En el ejemplo anterior, se observa que hacer uso de la modularidad permitió tener una placa principal menos compleja. Por otro lado, el

hecho de tener otras funciones en diferentes módulos permite agregarlas o quitarlas cuando sea necesario; no sólo en esta placa principal sino en cualquier otra placa que permita su conexión, esta característica se conoce como reutilización o reusabilidad.

Ahora, veamos la modularidad en el área del software. Explicaré el concepto con un ejemplo: Supongamos que se pide un programa que calcule el valor de la potencia disipada por una resistencia. El programa debe tener un menú en donde el usuario pueda escoger alguna de estas fórmulas para hacer el cálculo: $P = V^2/R$ o $P = I^2R$ dependiendo de si conoce el valor del voltaje que hay entre las terminales de la resistencia R o el valor de la corriente que circula por ella.

La solución al problema la podríamos diseñar usando cualquiera de los dos modelos de la figura 21. El lado izquierdo de la figura es una programación no modular, como la que hiciste en el capítulo anterior, en el lado derecho puedes observar una posible solución a través de la programación modular.

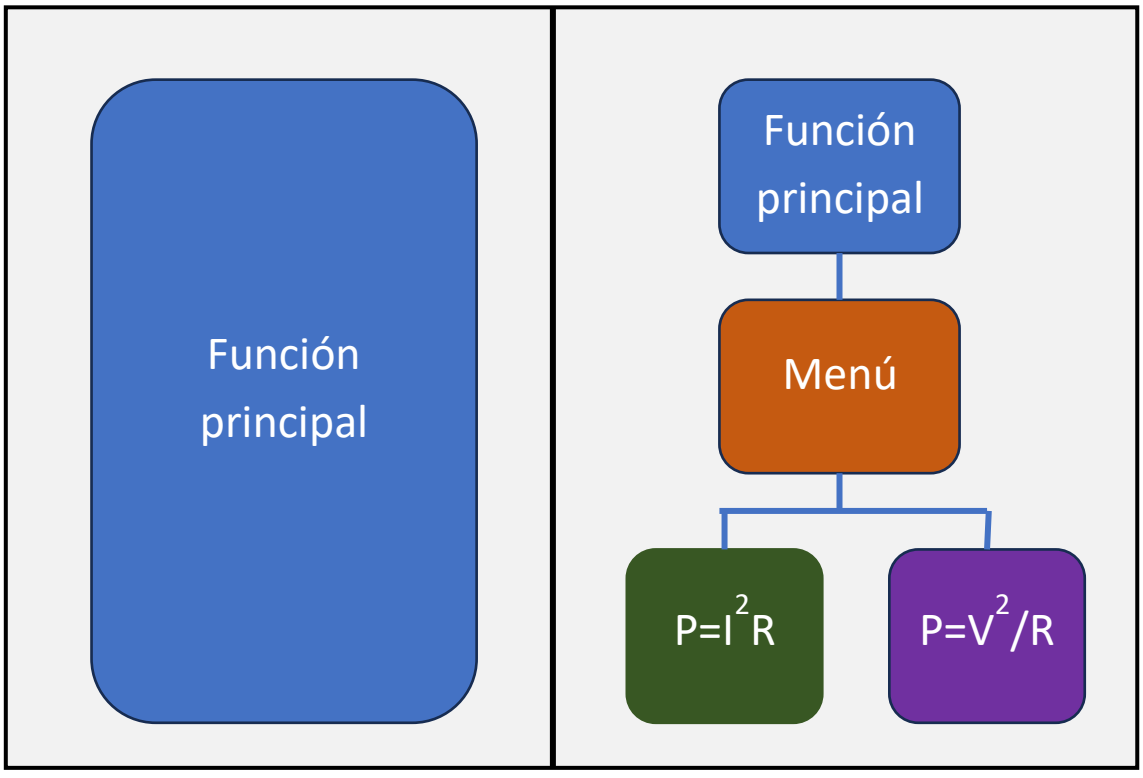


Figura 21. Gráfico en donde se ilustra la diferencia entre un modelo de programación modular (lado derecho) y otro que no lo usa (lado izquierdo).

En la programación modular se busca dividir el problema para disminuir su complejidad, esta técnica es conocida como “divide y vencerás”. Así, en el ejemplo anterior, en vez de ver el problema como un todo, podríamos dividirlo en tres problemas “más pequeños” como:

1. Hacer un programa que ofrezca un menú al usuario para presentarle las fórmulas que puede utilizar.

2. Hacer un programa que calcule la potencia disipada usando el valor de la resistencia y el valor de corriente.

3. Hacer un programa que calcule la potencia disipada usando el valor de la resistencia y el valor de voltaje.

De esta manera, tendríamos tres programas, cada uno de ellos sería un módulo. En el lenguaje C, estos módulos reciben el nombre de *funciones*. Una función es un boque de código que se considera como una unidad que resuelve una tarea específica; tal como mostrar un menú o calcular una potencia; tiene un nombre y es capaz de recibir o regresar algún valor. En C, existe una función que siempre debe existir y es la función principal, main ( ). De hecho, todo el lenguaje C funciona a través de funciones, las cuales se clasifican en dos tipos: aquellas que ya vienen escritas y son propias del lenguaje y, las que son definidas por el programador.

Como recordarás del capítulo 1, antes de utilizar una variable hay que declararla. De igual forma, el primer paso cuando trabajas con funciones es escribir el *prototipo de función*, esto sirve para indicarle al compilador el tipo de dato retornado y los tipos de datos de los valores que reciba. Los valores que recibe una función se llaman *argumentos o parámetros*. Una función puede recibir uno o más parámetros o, incluso, no recibir ninguno. En cambio, sólo puede retornar un valor, aunque no es obligatorio.

Sintaxis de prototipo de función:

tipo de dato del valor retornado **identificador** **(**tipo de dato del parámetro 1 **identificador1**, tipo de dato del parámetro 2 **identificador2**, …,tipo de dato del parámetro n **identificador n)**;

donde:

tipo de dato del valor retornado puede ser cualquiera de la Tabla 1 o ***void***.

identificador es el nombre con el cual se reconocerá a la función.
tipo de dato del parámetro 1, tipo de dato del parámetro 2,…, tipo de dato del parámetro n son los tipos de datos de los parámetros de la función.
identificador1, identificador2, identificador n son los nombres que el programador asignará a los parámetros.
; indica al compilador el fin del prototipo.

Ejemplos de prototipos de funciones

```
int promedio (int num1, int num2, int num3, int num4);
float corriente (float voltaje, int resistencia);
char suma (float a, float b);
double R_serie(  );
```

La intención de este apartado es que aprendas la sintaxis del prototipo, así que un buen ejercicio es identificar en los ejemplos anteriores cuáles son los tipos de datos retornados, los nombres de las funciones y sus argumentos. No te inquietes si no entiendes su significado, a lo largo del capítulo te lo explicaré con más detalle.

3.2 Funciones propias del lenguaje

Como ya sabes, C funciona utilizando funciones, así que no te extrañará saber que ya has estado usando funciones desde el capítulo uno. Las funciones que has estado empleando se conocen como funciones propias del lenguaje. Son bloques de código que realizan una tarea específica y se encuentran codificados en archivos con extensión .h llamados *librerías* de C. printf( ) y scanf ( ) son dos ejemplos de funciones propias del lenguaje que se encuentran en el archivo stdio.h, esto significa que ya no tenemos que codificarlas, pero sí debemos incluir el archivo en donde están su prototipo y su *definición*.

A continuación, se muestran, en forma de tablas, algunas funciones propias de C. En las tablas encontrarás las funciones y la descripción de la tarea específica que realiza cada función. En el título de cada tabla verás la librería en donde se encuentran dichas funciones y que tienes que agregar a través de la directiva de preprocesador #include para que funcionen.

Debes notar que todas estas funciones llevan paréntesis después del nombre, dentro de los paréntesis hay unos valores. Observa también que se escribe un tipo

de dato antes del nombre. Ya debes ser capaz de reconocer que se tratan de los prototipos de las funciones.

Tabla 11. Funciones para operaciones matemáticas. Librería math.h.

Función	Descripción
double sin (double x);	Calcula seno de x en radianes.
double cos (double x);	Calcula coseno de x en radianes.
double tan (double x);	Calcula tangente de x en radianes.
double fabs(double x);	Calcula el valor absoluto de x.
double pow(double x, double y);	Calcula x elevado a la y.
double log(double x);	Calcula In x.
double log10(double x);	Calcula log x.
double exp(double x);	Calcula e^x.
double sqrt(double x);	Calcula la raíz cuadrada no negativo de x.

Tabla 12. Funciones para caracteres. Librería ctype.h

Función	Descripción
int tolower (int c);	Convierte un caracter c de mayúscula a minúscula. Utiliza el valor del código ASCII del caracter.
int toupper(int c);	Convierte un caracter c de minúscula a mayúscula. Utiliza el valor del código ASCII del caracter.

Tabla 13. Funciones para manipular datos de entrada y salida. Librería stdio.h

Función	Descripción
int printf(char *cadena, const char *formato, ...);	Imprime en pantalla. Devuelve un valor negativo en caso de error.
char * gets(char *cadena);	Lee una cadena con espacios en blanco.
int getchar(void);	Lee un caracter desde teclado.
int scanf(const char *formato, ...);	Lee datos desde el teclado.
int fflush(FILE *stream);	Limpia el búffer señalado por stream.
FILE * fopen(const char *nombre, const char *modo);	Abre un archivo cuya dirección de memoria está señalada por nombre. El argumento modo se verá en el capítulo 5.
int fputs(const char *cadena, FILE *stream);	Escribe en un archivo la cadena apuntada.
int fseek(FILE *stream, long int desp, int pos);	Mueve el puntero al archivo una distancia de desp bytes desde la posición pos que puede ser el principio del archivo, la posición actual del puntero o el fin del archivo.

int fclose(FILE *stream);	Cierrra el archivo señalado por stream.
int fgetc(FILE *stream);	Lee un caracter desde un archivo.
size_t fread(void *puntero, size_t taman, size_t n, FILE *stream);	Lee n elementos, cada uno de tamaño taman bytes, desde el archivo señalado por puntero hasta la cadena stream.
int feof(FILE *stream);	Comprueba si se ha llegado al final del archivo señalado por stream.
int fprintf(FILE *stream, const char *formato, ...);	Imprime datos en un archivo.
long int ftell(FILE *stream);	Devuelve la posición actual del puntero dentro del archivo.
char *fgets(char *cadena, int n, FILE *stream);	Lee una cadena de n caracteres desde el archivo señalado por stream.
int fscanf(FILE *stream, const char *formato, ...);	Lee datos desde un archivo.
size_t fwrite(const void *puntero, size_t taman, size_t n, FILE *stream);	Escribe n elementos, cada uno de tamaño taman bytes, desde la cadena señalada por puntero hasta el archivo señalado por stream.

Tabla 14. Funciones para uso general. Librería stdlib.h

Función	Descripción
int atoi(const char *numPtr);	Convierte la cadena apuntada por numPtr a un número de tipo entero.
double atof(const char *numPtr);	Convierte la cadena apuntada por numPtr a un número de tipo flotante.
long int atol(const char *numPtr);	Convierte la cadena apuntada por numPtr a un número de tipo long.
void *calloc(size_t n, size_t taman);	Reserva memoria para un arreglo de n elementos, cada uno de taman bytes.
void exit(int estado);	Cierra todos los archivos y buffers, y termina el programa.
void free(void *ptr);	Libera un bloque de memoria cuyo principio está señalado por ptr.
void *malloc(size_t taman);	Reserva taman bytes de memoria, devuelve un puntero al principio del espacio reservado
int rand(void);	Regresa números enteros pseudo-aleatorios.
void srand(unsigned int semilla);	Inicializa el generador de números aleatorios
int system(const char *cadena);	Pasa el mando al sistema operativo. Devuelve cero si la orden se ejecuta correctamente; en otro caso devuelve un valor distinto de cero, regularmente -1.

Tabla 15. Funciones para cadenas. Librería string.h

Función	Descripción
char *strcat(char*s1, const char *s2);	Agrega la cadena s2 al final de s1. El resultado se guarda en s1.
int strcmp(const char *s1, const char *s2);	Compara la cadena apuntada por s1 con la cadena apuntada por s2.
char *strcpy(char *s1, const char *s2);	Copia la cadena apuntada por s2 (incluyendo el carácter nulo) a la cadena apuntada por s1.
size_t strlen(const char *s);	Calcula el número de caracteres de la cadena apuntada por s.

Tabla 16. Funciones para manipular hora y fecha del sistema. Librería time.h

Función	Descripción
time_t time(time_t *tiempoPtr);	Determina el tiempo en formato condensado.
double difftime(time_t tiempo1, time_t tiempo0);	Calcula la diferencia entre dos tiempos en formato condensado: tiempo1 - tiempo0.
struct tm *gmtime(const time_t *tiempoPtr);	Convierte el tiempo en formato condensado apuntado por tiempoPtr en el tiempo en formato separado, expresado como Tiempo Universal Coordenada (UTC).
char *asctime(const struct tm *tiempoPtr);	Convierte el tiempo en formato separado en la estructura apuntada por tiempoPtr en una cadena en la forma: Wed Jan 4 02:03:55 2023\n\0.

Es importante que sepas que existen las funciones que ya vienen en las librerías para que no tengas que escribir más código. Las funciones mencionadas en este apartado son las más comunes, si deseas más información, consulta el link de la sección de Bibliografía y Referencia de este capítulo.

EJERCICIO DE FUNCIONES PROPIAS

Menciona la función que realiza la siguiente tarea específica y la librería en donde están su prototipo y su definición:
a) Elevar una variable al cuadrado.
b) Sacar la raíz cuadrada de un número.
c) Convertir “A” a un número entero.
d) Abrir un archivo.

Solución:
a) **pow( ), math.h;** b) **sqrt( ) o pow( ), ambas en math.h;** c) **atoi( ), stdlib.h;** d) **fopen( ), stdio.h.**
Observa que para mencionar a la función no necesitamos escribir el prototipo, sólo su identificador y los paréntesis vacíos.

3.3 Funciones con parámetros

Como te mencioné anteriormente, un parámetro o también llamado argumento de una función es un valor que recibe dicha función. En C, una función puede recibir uno, dos o más parámetros; los tipos de datos pueden coincidir o no. Ejemplificaré el tema con el módulo para calcular la potencia usando la fórmula $P=I^2R$: Hacer una función que reciba como parámetros el valor de la resistencia, en ohms, y el valor de la corriente que circula por ella, en amperes.

Hay que darle un nombre a la función, en este ejemplo le pondré P_corriente, tú puedes nombrar a una función como desees ya que no es una función propia del lenguaje, sólo tienes que seguir las mismas pautas que sigues para nombrar a las variables. Consideremos que tanto el valor de la resistencia como el de la corriente son números enteros, hay que darle un identificador a cada variable del argumento y propongo R e I, respectivamente. En la figura 22 te muestro el código que deberás escribir en el editor del Dev C++. Compílala y ejecuta. ¿Qué observas a la salida?

```
1  #include<stdio.h>
2  void P_corriente(int R,int I); //PROTOTIPO DE FUNCIÓN
3  void main ( )
4  {
5      int I;
6      int R;
7      printf("\t Introduce el valor de la resistencia, en ohms: ");
8      scanf("%d",&R);
9      printf("\t Introduce el valor de la corriente que fluye por ella, en amperes: ");
10     scanf("%d",&I);
11
12 }
```

Figura 22. Código para obtener los valores del argumento de una función.

La respuesta debe ser ¡nada! Analicemos el código, en él existe un prototipo de función; observa en dónde está colocado. Los prototipos de funciones se escriben antes de la función main(). Regularmente después de las directivas de preprocesador. Además del prototipo de la función hay que escribir su *definición*. *Definir una función* significa escribir el código necesario para que cumpla con su tarea específica, en este ejemplo la tarea es calcular la potencia en una resistencia usando el valor de la corriente.

Modifica tu código para que sea igual al de la figura 23. Compílalo y ejecútalo. Tendrás una salida como la figura 24. Regresemos al código, observa que en la parte inferior está la definición de la función P_corriente( ), empieza con una línea similar al prototipo de función, excepto el punto y coma (línea 17). Las líneas de código que van entre llaves (líneas 18 y 26) y que realizan la tarea buscada, en este caso, calcular la potencia, se llama cuerpo de la función. Lo que observas en la figura 23 es una propuesta, tú puedes hacer tu propio código, es decir, tú haces la definición de la función; por eso este tipo de funciones, a diferencia de los que ya vienen definidas como printf( ) o scanf( ),se llaman *funciones definidas por el usuario*.

Para usar la fórmula $P=I^2R$, en esta propuesta se utiliza una función propia de las librerías de C: la función pow( ) que, como viste en las tablas anteriores, permite elevar una base a un exponente, en este caso se desea elevar el valor de la corriente al cuadrado, observa cómo se utiliza en la línea 21. Observa que para poder utilizar pow( ) se incluyó math.h (línea 2).

```
1  #include<stdio.h>
2  #include<math.h>
3  void P_corriente(int R,int I); //PROTOTIPO DE FUNCIÓN
4  void main ( )
5  {
6      int I;
7      int R;
8      printf("\t Introduce el valor de la resistencia, en ohms: ");
9      scanf("%d",&R);
10     printf("\t Introduce el valor de la corriente que fluye por ella, en amperes: ");
11     scanf("%d",&I);
12     P_corriente(R,I); //llamada a función
13
14 }
15
16 //definición de la función P_corriente( )
17 void P_corriente(int R,int I)
18 {
19     double Potencia;
20     printf("\n\n\t  FUNCION PARA CALCULAR POTENCIA P=I%cR \n ",253);
21     Potencia=pow(I,2)*R;
22     printf("\n\t\t Valor de R:   %d ohms\n", R);
23     printf("\n\t\t Valor de I:   %d amperes\n", I);
24     printf("\n\t\t             ==============\n");
25     printf("\n\t\t Valor de P:   %.2f watts", Potencia);
26 }
```

Figura 23. Cálculo de potencia usando una función definida por el usuario.

Dentro de la función *P_corriente ( )* hay tres variables declaradas: *R* e *I* están declaradas como argumento de la función y son de tipo *int*, la otra variable es *Potencia* y es de tipo double; estas variables se llaman *variables locales* porque sólo

tienen un *alcance* local, esto es, sólo existen en la función en donde fueron declaradas. Para ilustrar esto, copia la expresión que está en la línea 25 a la línea 13 y compílalo nuevamente ¿Qué sucede? El compilador muestra error porque la variable Potencia no está declarada en main( ), está declarada en P_corriente( ), por eso podemos utilizarla en esa función sin ningún problema pero no en la función principal, en otras palabras, es una variable local de P_corriente( ).

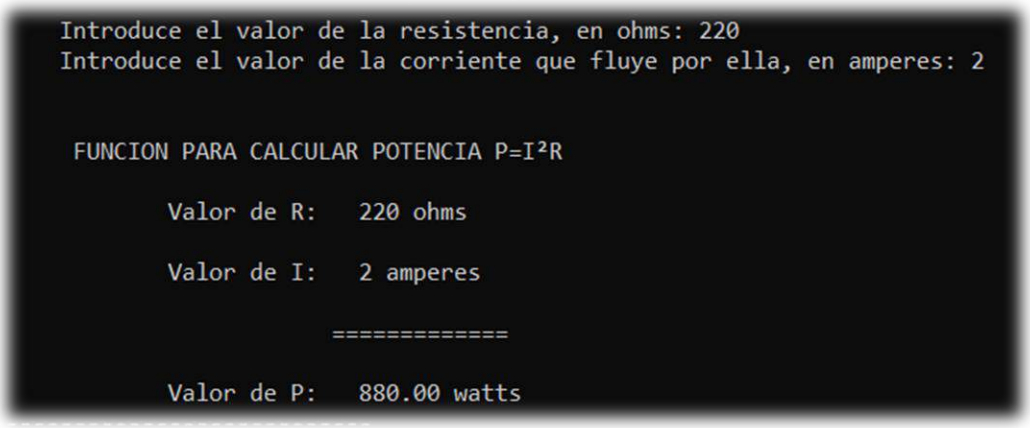


Figura 24. Salida obtenida con el código de la Figura 23.

Otro concepto importante es la *llamada a una función*. Para entender la razón de ser de una llamada a función pon en comentario la línea 12, compílalo y ejecútalo, ¿qué observas? efectivamente, no se realiza el cálculo de la potencia. La llamada a la función P_corriente( ) permite que el compilador deje la línea 11 del main ( ) y busque la línea en donde empieza la definición de la función, en este caso, la línea 17, ejecuta todas las líneas de código hasta alcanzar la llave que cierra el cuerpo de la función y regresa al punto donde fue llamado. Entonces, para que se ejecute el bloque de código que realiza la tarea del cálculo de la potencia, no sólo hay que poner su prototipo y su definición, hay que llamarla.

Para llamar a una función hay que escribir su nombre y mandarle sus parámetros. Observa la línea 12 y date cuenta de que cuando se llama a una función no hay que escribir los tipos de datos correspondientes a los parámetros, sólo sus valores. Esto se conoce como *paso de parámetros por valor* porque lo que realmente se envía es el valor o contenido que está almacenado en esa variable. Existe otro tipo de paso de parámetros llamado *paso de parámetros por referencia*, en donde lo que se envía es la dirección de memoria en donde está la variable. En este capítulo aprenderás cómo pasar parámetros por valor.

También puede suceder que la función no reciba ningún parámetro, en ese caso, todos los valores requeridos son pedidos en la misma función y dentro de los paréntesis no hay nada o se escribe *void*. En la figura 25 encontrarás una solución alterna al mismo problema, pero haciendo que la función P_corriente( ) no reciba ningún parámetro. Compílalo y ejecútalo, dale los mismos valores de I y R ¿la salida es diferente a la propuesta anterior?

```c
#include<stdio.h>
#include<math.h>
void P_corriente( ); //PROTOTIPO DE FUNCIÓN
void main ( )
{

    P_corriente( ); //llamada a función

}

//definición de la función P_corriente( )
void P_corriente( )
{
    double Potencia;
    int I;
    int R;
    printf("\t Introduce el valor de la resistencia, en ohms: ");
    scanf("%d",&R);
    printf("\t Introduce el valor de la corriente que fluye por ella, en amperes: ");
    scanf("%d",&I);
    printf("\n\n\t  FUNCION PARA CALCULAR POTENCIA P=I%cR \n ",253);
    Potencia=pow(I,2)*R;
    printf("\n\t\t Valor de R:   %d ohms\n", R);
    printf("\n\t\t Valor de I:   %d amperes\n", I);
    printf("\n\t\t             ==============\n");
    printf("\n\t\t Valor de P:   %.2f watts", Potencia);
}

```

Figura 25. Propuesta alterna al código de la figura 23.

EJERCICIO DE FUNCIONES CON PARÁMETROS

Se desea tener un programa que sea modular y que calcule el valor de la corriente que pasa por una resistencia conociendo su valor y el valor de su potencia disipada. Para solucionar el problema, se propone obtener el valor de la corriente con la fórmula $I = \sqrt{\frac{P}{R}}$ Se propone también el uso de la función pow( ) para sacar una raíz cuadrada, elevando a una potencia de 0.5. En el código de la figura 26 se muestra una propuesta de solución, sin embargo, al compilarlo se observan errores. ¿Cómo corregirías dicho código?

Solución:
La línea 11 genera un error porque la función corriente ( ) espera dos parámetros y sólo se le está enviando uno. Debe ser **corriente (R, P)**
El uso de pow( ) debe incluir la librería en donde se encuentra su prototipo y su definición. Debe agregarse **#include <math.h>**

```
#include<stdio.h>
void corriente(int R, float P ); //PROTOTIPO DE FUNCIÓN
void main ( )
{
    int R;
    float P;
    printf("\t Introduce el valor de la resistencia, en ohms: ");
    scanf("%d",&R);
    printf("\t Introduce el valor de la potencia disipada, en watss: ");
    scanf("%f",&P);
    corriente(R); //llamada a función
}

void corriente(int R, float P ) //definición de función
{
    double I;
    float razon;
    razon= P/R;
    I=pow(razon,0.5);
    printf("\n\t\t Valor de R:   %d ohms\n", R);
    printf("\n\t\t Valor de P:   %f watts", P);
    printf("\n\t\t              ==============\n");
    printf("\n\t\t Valor de I:   %.2f amperes\n", I);
}
```

Figura 26. Código de ejercicio de Funciones con parámetros.

3.4 Funciones que devuelven valores

Ya vimos cómo enviar valores a una función, ahora veamos cómo recibir valores que regresen desde una función. Al igual que en el apartado anterior, ejemplificaré el tema resolviendo uno de los módulos del problema original presentado en el inicio del capítulo: Hacer una función que se llame menu(), no tenga argumento y regrese un valor entero que corresponda a una de las dos opciones planteadas, calcular la potencia disipada por una resistencia a partir del valor de la corriente que circula por ella o, conociendo el valor del voltaje entre sus terminales. Puedes observar que la tarea específica de la función menu( ) será permitirle al usuario escoger una opción dentro de un menú de opciones. En la figura 27 se observa el código propuesto.

Primeramente, escribamos el prototipo de la función menu( ). Recordemos que la sintaxis establece que, del lado izquierdo del nombre de la función, va el tipo de dato del valor retornado. Si la función va a retornar un valor de punto flotante se escribe *float*; si es de tipo caracter, *char*; si es entero, como en este caso, *int*. Si la función no va a regresar ningún valor, se escribe *void*, que es un tipo de dato que no guarda nada y quedaría como *void función( )*, no debes dejarlo vacío ya que el compilador lo interpreta como que va a regresar un tipo entero.

Ejemplos de prototipos de funciones

float funcion1( ); regresa un valor flotante, no recibe parámetros.

char funcion1(void); regresa un caracter, no recibe parámetros.

int funcion1 (char a); regresa un valor de tipo entero, recibe un caracter.

void funcion1 (int n, float b, char c); no regresa nada, recibe un entero, un flotante y un caracter.

void funcion1(int a, int b, int c, int d); no regresa nada, recibe cuatro enteros.

funcion1( ); regresa un entero, no recibe nada.

```c
#include<stdio.h>
int menu( ); //PROTOTIPO DE FUNCIÓN
void main ( )
{
    int op;
    op=menu();
    printf("\n\t\t Elegiste la opcion: %d\n\n\n",op);
}

int menu( ) //definición de función
{
    int opcion;
    printf("\n\t\t ============================\n");
    printf("\n\t\t     CALCULO DE POTENCIA\n");
    printf("\n\t\t ============================\n\n\n");
    printf("\n\t\t 1) Potencia usando el valor de corriente\n");
    printf("\n\t\t 2) Potencia usando el valor de voltaje\n");
    printf("\n\t\t Escoja una opcion...  ");
    scanf("%d",&opcion);
    return opcion;
}
```

Figura 27. Función que regresa un valor.

De los ejemplos mostrados, debes resaltar que, si dejas vacío el lado izquierdo del nombre de la función, el compilador lo interpreta como un *int*. Pero si dejas vacío los paréntesis, el compilador lo ve como un *void*. Debes ser cuidadoso con esto porque es causa de muchos errores cuando apenas se está aprendiendo a programar con funciones.

El prototipo de función de nuestro ejemplo está en la línea 2 del código de la figura 27. En ese mismo código, observa la línea 6, que nos lleva a la siguiente:

Sintaxis de asignación de valor devuelto por función:

```
identificador = función ( );
```

donde:

identificador es el nombre de la variable en donde se almacenará el valor regresado por la función. Su tipo de dato debe ser el mismo que el retornado por la función.
= es el operador de asignación.
función ( ) es el nombre de la función a la que se llama o invoca, junto con su argumento.

; indica al compilador el fin de la expresión.

Ya que *menu( )* retorna o regresa un valor de tipo entero, la variable local *op* está declarada también de tipo *int*. Pon en comentario la línea 6, compila y ejecuta el programa, ¿qué observas? Exactamente, no se imprime el valor escogido en menu() porque nunca se asignó a op. Otra alternativa a este código sería modificar la línea 7 por:

printf("\n\t\t Elegiste la opcion: %d\n\n\n",menu( ));

observa el cambio, en lugar de poner el nombre de la variable *op*, se hizo directamente la llamada a la función y el valor retornado es el que se imprimirá en el especificador de formato que, obviamente, tiene que ser del mismo tipo. Es muy importante que tengas presente que, aunque una función puede recibir varios parámetros, sólo puede regresar uno. El valor se retorna con la palabra reservada *return*, lo puedes apreciar en la línea 20 que está dentro de la definición de la

función; regularmente es la última línea del cuerpo de la función porque en cuando el compilador la ve, regresa al punto desde donde fue llamada la función.

El código que resuelve todo el problema se observa en la figura 28 que por sus dimensiones se ha dividido en tres partes, A, B y C. En la figura 28A observa:

a) Los prototipos de funciones. Se escriben antes del *main ( )* y terminan con un punto y coma. Líneas 3, 4 y 5.

b) Llamadas a funciones. Cuando esperamos un valor, como en las líneas 11 y 22, colocamos una variable del mismo tipo esperado; cuando no regresa nada, no se escribe ninguna variable (línea 19).

c) Paso de parámetros. En el momento de hacer la llamada a la función, se debe pasar su argumento. Sólo se envía el valor, no el tipo de dato (línea 19).

```
2  #include<stdio.h>
3  #include<math.h>
4  void P_corriente(int I);
5  float P_voltaje();
6  int menu( );
7  int R;
8  void main ( )
9  {
10     int I,op;
11     float Potencia;
12     op=menu(); //llamada a función
13     switch (op)
14     {
15         case 1:
16             printf("\n\t Introduce el valor de la resistencia, en ohms: ");
17             scanf("%d",&R);
18             printf("\t Introduce el valor de la corriente que fluye por ella, en amperes: ");
19             scanf("%d",&I);
20             P_corriente(I); //llamada a función
21             break;
22         case 2:
23             Potencia=P_voltaje(); //llamada a función
24             printf("\n\t\t Valor de P:   %.2f watts", Potencia);
25             break;
26         default:
27             printf("\t\a OPCION NO VALIDA!!!\a\a\n: ");
28     }
29     printf("\n\t\t Gracias por usar la aplicacion! :) ");
30 }
```

Figura 28A. Función principal llamando a las funciones definidas por el usuario.

Analiza también los códigos de las figuras 28B y 28C. Deberás observar lo siguiente: Tanto en la función *P_corriente( )* como *P_voltaje ( )* existe una variable con el mismo nombre, *Potencia*. Esto es posible debido a que son variables locales de distintas funciones. Si intentas darle el mismo nombre a dos variables que se

encuentren en la misma función, causará error, aunque sean de diferente tipo de dato. A diferencia de las variables locales, las *variables globales* existen en todo el programa y se declaran cerca de las directivas de preprocesador. La línea 6 es la declaración de una variable global, R. Debido a que es global no se envía como parámetro a P_corriente( ), su valor, una vez leído, se conoce en todas las funciones.

```
32  int menu( ) //definición de función
33  {
34      int opcion;
35      printf("\n\t\t ============================\n");
36      printf("\n\t\t     CALCULO DE POTENCIA\n");
37      printf("\n\t\t ============================\n\n");
38      printf("\n\t\t 1) Potencia usando el valor de corriente\n");
39      printf("\n\t\t 2) Potencia usando el valor de voltaje\n");
40      printf("\n\t\t Escoja una opcion...  ");
41      scanf("%d",&opcion);
42      return opcion;
43  }
44
45  float P_voltaje()
46  {
47      float V,Potencia;
48      printf("\n\t Introduce el valor de la resistencia, en ohms: ");
49      scanf("%d",&R);
50      printf("\t Introduce el valor del voltaje entre sus terminales, en volts: ");
51      scanf("%f",&V);
52      Potencia=pow(V,2)/R;
53      printf("\n\n\t\t FUNCION PARA CALCULAR POTENCIA P=V%c/R \n ",253);
54      printf("\n\t\t Valor de R:   %d ohms\n", R);
55      printf("\n\t\t Valor de V:   %.2f volts\n", V);
56      printf("\n\t\t             =============\n");
57      return Potencia;
58  }
```

Figura 28B. Definición de funciones.

```
59  void P_corriente(int I)
60  {
61      double Potencia;
62      printf("\n\n\t FUNCION PARA CALCULAR POTENCIA P=I%cR \n ",253);
63      Potencia=pow(I,2)*R;
64      printf("\n\t\t Valor de R:   %d ohms\n", R);
65      printf("\n\t\t Valor de I:   %d amperes\n", I);
66      printf("\n\t\t             =============\n");
67      printf("\n\t\t Valor de P:   %.2f watts", Potencia);
68  }
```

Figura 28C. Uso de variable global.

3.5 Ejercicios y Problemas

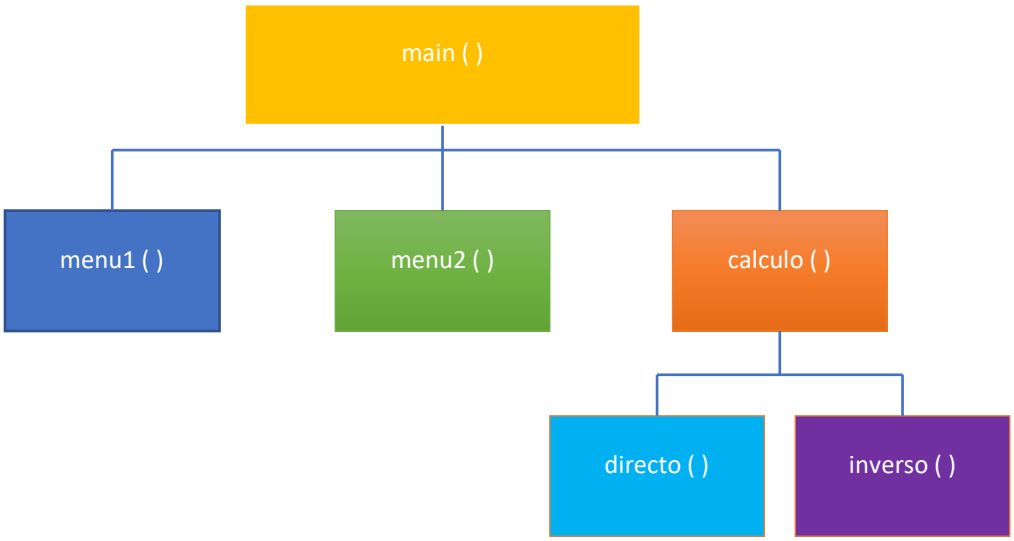
Aquí se presenta un caso en donde verás cómo utilizar funciones con parámetros y funciones que devuelven valores. Además, observarás que una función puede ser llamada o invocada desde la función principal *main ( )* o desde otra función definida por el usuario.

Caso de estudio : Cálculo de resistencias, capacitores e inductores equivalentes

Hacer un programa que muestre al usuario un menú principal en donde pueda escoger el componente con el cual desea trabajar: resistencia, capacitor o inductor. Debe existir un submenú para que el usuario escoja el tipo de conexión que tiene el componente seleccionado: serie o paralelo. El usuario determinará el número de componentes que simplificará. Las resistencias serán dados en ohms, las capacitancias en microfaradios y las inductancias en milihenrios. El programa terminará hasta que el usuario decida salir del programa.

PROPUESTA DE SOLUCIÓN

Se propone un diseño modular como el siguiente:



Tal como se observa en la figura, se proponen cinco funciones definidas por el usuario:

- ***menu1 ( )***. Esta función tendrá la intención de mostrar un menú donde el usuario pueda escoger el componente eléctrico: Resistencia, Capacitor o Inductor, o salir del programa. Es llamada desde la función main(). No va a recibir ningún parámetro y va a regresar un entero que corresponderá a la opción deseada:
    1. Resistencia
    2. Capacitor
    3. Inductor
    4. Salir

- ***menu2( )***. La intención de esta función es mostrar los tipos de conexión y leer la opción escogida por el usuario. Es llamada desde la función *main()*. No va a recibir ningún parámetro y va a regresar un entero que corresponderá a la opción deseada:
    1. Serie
    2. Paralelo
    3. Regresar al menú anterior

- ***calculo( )***. Esta función determinará, en base a las opciones escogidas en *menu1( )* y *menu2( )*, qué operación realizar para obtener el valor equivalente. Recibirá dos parámetros: el tipo de elemento electrónico, regresado por *menu1( )*, y el tipo de conexión, regresado por *menu2( )*, pero no regresará ningún valor.

- ***directo( )***. Esta función será llamada cuando el equivalente se obtenga con la suma directa de los valores de los componentes; tal como las resistencias e inductores en serie y los capacitores en paralelo.

- ***inverso( )***. Esta función será llamada cuando el equivalente se obtenga con el inverso de la suma de los inversos de los valores de los componentes; tal como las resistencias e inductores en paralelo y los capacitores en serie.

Analiza esta propuesta, debes observar las dos ventajas mencionadas de la programación modular. Primero, es menos complejo probar función por función, que todo el código en la función principal y segundo, una vez probada tus funciones, las puedes utilizar en cualquier otro programa. Nuevamente recuerda que este código

no representa la única solución. Repasa los conceptos aprendidos en este capítulo y propón tu propio diseño de solución usando programación modular.

```
#include <stdio.h>
int menu1();
int menu2();
void calculo(int comp,int m);
float directo();
float inverso();
main()
{
	int componente,modo;
	float R, Requiv, Rinverso;
	do{
	   componente=menu1();
	   if (componente==4) break;
	   if(componente!=1&&componente!=2&&componente!=3)
	   {
		printf("\n\t TU OPCION NO CORRESPONDE A NINGUN COMPONENTE");
		continue;
	   }
		do{
			modo=menu2();
			if (modo==3) break;
		if(modo!=1&&modo!=2)
		  {
			printf("\n\t TU OPCION NO CORRESPONDE A NINGUN MODO DE
	CONEXION");
		    continue;
		  }
		}while(modo!=1&&modo!=2);
	   calculo(componente,modo);
	}while(componente!=4);
}

int menu1()
{
	int opcion;
	printf("\n\t\t CALCULO DE EQUIVALENTES ");
	printf("\n\t\t 1) RESISTENCIA");
	printf("\n\t\t 2) CAPACITOR");
	printf("\n\t\t 3) INDUCTOR");
```

```
	printf("\n\t\t 4) SALIR");
	printf("\n\t\t Escoge una de las opciones...");
	scanf("%d",&opcion);
	return opcion;
}
int menu2()
{
	int opcion;
	printf("\n\t\t MODO DE CONEXION ");
	printf("\n\t\t 1) SERIE");
	printf("\n\t\t 2) PARALELO");
	printf("\n\t\t 3) Volver al menu principal");
	printf("\n\t\t Escoge una de las opciones...");
	scanf("%d",&opcion);
	return opcion;
}
void calculo(int comp,int m)
{
	float resultado;
	switch (comp)
	{
	   case 1:
			if(m==1)
			{
				printf("\n\t\t RESISTENCIA EQUIVALENTE SERIE ");
				resultado=directo();
			}
			 else
			 {
				printf("\n\t\t RESISTENCIA EQUIVALENTE PARALELO ");
				resultado=inverso();
			 }
			printf("\n\n\t\t Resistencia equivalente = %.2f ohms\n\n",resultado);
			break;
	   case 2:
			if(m==1)
			{
			   printf("\n\t\t CAPACITANCIA EQUIVALENTE SERIE ");
			    resultado=inverso();
			}
			else
			{
				printf("\n\t\t CAPACITANCIA EQUIVALENTE PARALELO ");
				resultado=directo();
```

```
                }
                printf("\n\n\t\t Capacitancia equiv = %.2f microfaradios\n\n",resultado);
                break;
        case 3:
                if(m==1)
                {
                        printf("\n\t\t INDUCTANCIA EQUIVALENTE SERIE ");
                        resultado=directo();
                }
                else
                {
                        printf("\n\t\t INDUCTANCIA EQUIVALENTE PARALELO ");
                        resultado=inverso();
                }
                printf("\n\n\t\t Inductancia equivalente = %.2f milihenrios\n\n",resultado);
           break;
    }
}
float directo()
{
    float valor,equivalencia=0;
    char opcion;

    do{
       printf("\n\t\t Introduzca el valor del componente: ");
        scanf("%f",&valor);
       equivalencia=equivalencia+valor;
       printf("\n\t Desea agregar otro componente? Teclee 's' para continuar");
       fflush(stdin);
       opcion=getch();
        }while(opcion=='s');
    return equivalencia;
}
float inverso()
{
    float valor,equivalencia=0;
    char opcion;

    do{
       printf("\n\t\t Introduzca el valor del componente: ");
        scanf("%f",&valor);
       equivalencia=equivalencia+1/valor;
       printf("\n\t Desea agregar otro componente? Teclee 's' para continuar");
       fflush(stdin);
```

```
        opcion=getch();
        }while(opcion=='s');
    return(1/equivalencia) ;
}
```

Referencias y bibliografía:

Kernighan B., Ritchie D. (1978). El lenguaje de programación C. Prentice Hall.

García-Bermejo Giner, J. R. (2008). Programación estructurada en C. Pearson Educación.

Jiménez Castells, M. y Otero Calviño, B. (2015). Fundamentos de ordenadores: programación en C. Universitat Politècnica de Catalunya.

Menchaca García, F. R. (2010). Fundamentos de programación en Lenguaje C. Instituto Politécnico Nacional.

Joyanes Aguilar, L. (2005). C algoritmos, programación y estructuras de datos. McGraw-Hill España.

Gaxiola Pacheco, C. G. y Flores Gutiérrez, D. L. (2008). Metodología de la programación con pseudocódigo enfocado al lenguaje C. Plaza y Valdés, S.A. de C.V.

Moreno Pérez, J. C. (2015). Programación. RA-MA Editorial.

Shildt H (2000) C The complete reference. McGrawHill

https://webs.um.es/iverdu/P00LibreriasANSIc.pdf Descargado diciembre 2022

Capítulo 4

VECTORES Y ARREGLOS

OBJETIVO GENERAL

Al finalizar el capítulo, el estudiante será capaz de hacer programas que utilicen vectores y arreglos bidimensionales.

.

OBJETIVOS PARTICULARES:

- Saber declarar vectores y arreglos bidimensionales.
- Utilizar vectores y arreglos bidimensionales en problemas que resuelvan problemas del área de electrónica.

4.1 Introducción

Los arreglos, en general, representan un tema muy importante en programación. Aquí aprenderás cómo declararlos, almacenar datos en él, hacer operaciones con ellos e imprimir sus contenidos. Además de los arreglos unidimensionales y bidimensionales, también aprenderás a utilizar cadenas, que realmente son un tipo especial de arreglos unidimensionales que almacenan caracteres. Finalmente, encontrarás tres casos de estudio en donde podrás observar cómo integrar los conocimientos adquiridos en este capítulo.

4.2 Vectores

Comencemos el tema de vectores con un problema a resolver, supongamos que necesitamos obtener la resistencia equivalente de cinco resistencias colocadas en serie. Con los conocimientos adquiridos hasta el momento, probablemente hayas pensado en declarar cinco variables de tipo entero para guardar los valores de los elementos:

int R1; int R2; int R3; int R4; int R5 o int R1,R2,R3,R4,R5;

Sin embargo, a partir de ahora, pensarás en un *arreglo unidimensional* de tipo int de tamaño 5: int R[5];

Un *arreglo* o *array* en inglés, es un conjunto de variables del mismo tipo que ocupan un lugar contiguo en la memoria y guardan información relacionada bajo un mismo nombre. Puede ser de una, dos, tres o más dimensiones. Cuando se trata de un arreglo unidimensional, algunos autores, le dan el nombre de *vector*. Al igual que, antes de usar una variable tienes que declarla, primero debes declarar al vector si lo vas a utilizar.

Sintaxis de declaración de un vector:

```
tipo_de_dato identificador [ tamaño];
```

donde:

tipo_de_dato puede ser *char*, *int*, o cualquier otro de la Tabla 1,
identificador es el nombre asignado con el cual se reconocerá al vector,
[ ] sirve para encerrar el tamaño del vector,
tamaño es el número de elementos que contendrá el vector,
; indica al compilador la terminación de la instrucción

Ejemplos de declaración de arreglos unidimensionales o vectores

int Resistencias[100]; //se reserva un espacio en memoria para 100 enteros.
float num[20]; //se reserva un espacio en memoria para 20 números flotantes.
char datos[10]; //se reserva un espacio en memoria para 10 caracteres.

Cuando el compilador reconoce la declaración de un vector, tal como

int Resistencias[100],

reserva un espacio de 400 bytes contiguos considerando que, como lo viste en la Tabla 1, un entero ocupa 4 bytes de memoria. En la figura 29 se muestra gráficamente la asignación de memoria. El *tamaño del vector* es el número de elementos que contiene dicho vector; en nuestro ejemplo, el tamaño de Resistencias es 100.

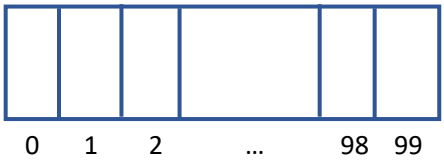


Figura 29. Forma en que se reserva espacio en memoria para un vector.

Todo el espacio reservado se reconoce como Resistencias pero se puede acceder a cada elemento del arreglo a través de un *índice*, que es el número que identifica la posición del elemento. En la figura 29 el valor del índice es el número que se encuentra debajo de cada elemento, observa que el primero es el 0 y el último, 99. En lenguaje C, todos los vectores empiezan con el índice 0 y terminan con el elemento (*tamaño-1*); por eso, en nuestro ejemplo, el último elemento es 99.

Debido a que la asignación de memoria se realiza en el momento de la compilación, es necesario que, en el momento de la declaración, *tamaño* sea una constante, no una variable. Observa los siguientes ejemplos:

NO VÁLIDO

```
main( )
{
int vector1[ x];
int x;
…
}
```

VÁLIDO

```
#define TAMAN 10

main( )
{
const int num=50;
int vector1[ 50];
float vector2[num];
int vector3[TAMAN];
…
}
```

EJERCICIO DE DECLARACIÓN DE VECTORES

1. Escribe la declaración para un vector de tipo flotante con identificador "Fuentes_V" y tamaño 10: _____

2. ¿Es correcta la siguiente declaración de variable? char valores[int x]; ¿Por qué?

Solución:

1. **float Fuentes_V [10];** 2. **No es correcta, dentro de los corchetes no debe ir ningún tipo de dato, además, debe ser un valor constante, no variable.**

Se pueden asignar valores a un vector en el momento de la declaración siguiendo la siguiente sintaxis de inicialización:

Sintaxis de inicialización de un vector:

```
tipo_de_dato identificador [ tamaño] = { valor1, valor2, …valorn};
```

donde:

tipo_de_dato puede ser *char*, *int*, o cualquier otro de la Tabla 1,
identificador es el nombre asignado con el cual se reconocerá al vector,
[ ] sirve para encerrar el tamaño del vector,
tamaño es el número de elementos que contendrá el vector,
= es el operador de asignación,
valor1, valor2, …valorn son los valores que se guardarán en el vector,
{ } son las llaves que encerrarán a los valores,
; indica al compilador la terminación de la instrucción

Ejemplos de inicialización de vectores

float valores [5]= {2.54, 3.15, 1.25, 3.78, 2.71};

int n[ ]= { 8, 57, 32};

char vector1 [ ] = { ‘r’, ‘t’, ‘v’, ‘h’};

En el primer ejemplo se está inicializando un vector de cinco números flotantes, el primer elemento guarda el número 2.54; el segundo, 3.15 y así sucesivamente. Observa que, en el segundo y tercer ejemplo, los corchetes están vacíos, sólo se puede hacer esto en el momento de la inicialización; dado que se están dando los valores del vector, el compilador puede determinar el número de elementos que lo componen. Cuando es un vector de caracteres, los valores deben ir entre comas simples, tal como lo muestra el tercer ejemplo. Cuando se desea asignar el valor a un vector en un momento diferente al de la inicialización, se usa la siguiente sintaxis:

Sintaxis de asignación a un elemento de un vector:

```
identificador [ índice] = valor;
```

donde:

identificador es el nombre del vector,
[ ] sirve para encerrar el valor del índice,
índice es el valor de la posición en donde se almacenará **valor**,
= es el operador de asignación,
valor es el valor que se guardará en el elemento del vector,

; indica al compilador la terminación de la instrucción

Ejemplos de asignación en vectores

valores [0]= 2.54;

n [2]= 32;

vector1 [1] = ‘t’;

El resultado de estos ejemplos se muestra gráficamente en la figura 30.

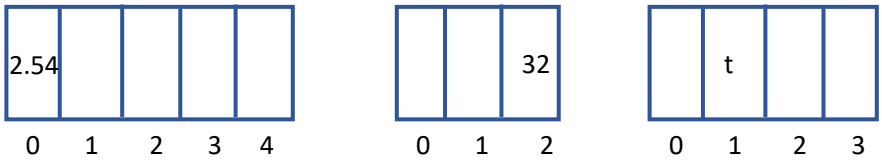


Figura 30. Asignación de valores en un vector.

También puedes hacer operaciones con los valores de los elementos del arreglo; así, la expresión valores[1]=valores[0]+5.13 haría que el segundo elemento del vector valores[ ] almacene el 7.67. Para ejemplificar todo lo anterior, escribe el código de la figura 31, compílalo y ejecútalo. Observa que para hacer un *recorrido* por el vector se utiliza un ciclo *for* que va desde el índice cero hasta el cuatro. El vector está declarado fuera del *main( )* así que, es un vector global; por eso ya no es necesario declararlo en cada función.

```c
#include <stdio.h>
void LeerVector();
void ImprimirVector();
void OperacionVector();
int Vector[5];
int main()
{

    LeerVector();
    printf("\n\t\t Valores originales del Vector\n");
    ImprimirVector();
    OperacionVector();
    printf("\n\t\t Valores finales del Vector\n");
    ImprimirVector();
}
void LeerVector()
{
    int i;
    printf("\n\t\t Introduzca los valores del Vector:\n");
    for (i=0;i<5;i++){
        printf("\n\t\t [ %d ] =    ",i); scanf("%d",&Vector[i]);
    }
}
void ImprimirVector()
{
    int i;
    for (i=0;i<5;i++) printf("\n\t\t [ %d ] = %d  ",i,Vector[i]);
}
void OperacionVector()
{
    int i,k;
    printf("\n\n\t\t Introduzca el valor de un escalar: "); scanf("%d",&k);
    for (i=0;i<5;i++)
     Vector[i]=Vector[i]*k;
}
```

Figura 31. Uso de un vector.

4.3 Cadenas

Una *cadena* es un arreglo unidimensional de caracteres cuyo último elemento es el caracter nulo. La *sintaxis* para declarar una cadena es la misma que para declarar un vector de caracteres. Es importante que para especificar el tamaño de la cadena consideres un espacio más para el caracter nulo. Aunque es posible inicializar una cadena de esta manera:

```
char palabra[7]={‘c’,’o’,’d’,’i’,’g’,’o’,\0’};
```

lo usual es hacerlo de esta otra forma:

```
char palabra[7]=”codigo”;
```

o

```
char palabra[ ]=”codigo”;
```

Observa que, para guardar la palabra “codigo” se tiene que reservar un tamaño de 7 bytes por el caracter nulo. En la inicialización del primer ejemplo se tiene que especificar el caracter nulo. Tanto en el segundo como en el tercer ejemplo el caracter nulo es agregado automáticamente por el compilador. Finalmente, sólo en el caso de una inicialización es posible prescindir del tamaño de la cadena, como se muestra en el tercer ejemplo.

Un aspecto importante en el manejo de cadenas es la asignación de un valor en un momento diferente al de la inicialización. Si intentas escribir:

```
char palabra[13];
palabra=”programacion”;
```

la segunda línea provocará un error. Para copiar un valor a una cadena debes usar una función propia que se encuentra en el archivo string.h y cuyo nombre es *strcpy()*.

Sintaxis para copiar una cadena usando strcpy( ):

```
strcpy ( cadenaDestino, cadenaOrigen) ;
```

donde:

strcpy es el nombre de la función que copia una cadena a otra,
cadenaDestino es la cadena en donde se copiará la **cadenaOrigen** ,
cadenaOrigen es la cadena que será copiada en **cadenaDestino,**
; indica al compilador la terminación de la instrucción

Ejemplos de copias de cadenas

strcpy(cadena1,cadena2); //copia el contenido de cadena2 a la cadena1
strcpy(cadena1,"hola"); //copia la palabra "hola" a la cadena2

Otra operación común con cadenas es la *concatenación*. Supongamos que cadena1 contiene la palabra "1234" y cadena2, "ABC"; si concatenamos la cadena1 con la cadena2 se forma la palabra "1234ABC". La función que permite concatenar dos cadenas es strcat( ), que se encuentra en string.h.

Sintaxis para concatenar dos cadenas usando strcat( ):

```
strcat ( cadena1, cadena2) ;
```

donde:

strcat es el nombre de la función que concatena una cadena con otra,
cadena1 es la cadena en donde se guardará el resultado de su concatenación con la **cadena2** ,
cadena2 es la cadena que se agregará al final de **cadena1,**
; indica al compilador la terminación de la instrucción

Ejemplos de concatenación de cadenas

strcat(nombre_completo,"Lopez");

strcat(nombre_completo, apellido);

Si nombre_completo contiene "Jose" y apellido, "Lopez". Ambas operaciones producirán "JoseLopez". Para dejar un espacio entre el nombre y apellido, primero

se tendrá que concatenar nombre_completo con un espacio en blanco y después concatenar con el apellido.

En ocasiones, durante la codificación, se requiere comparar dos cadenas. Si escribes

```
if (nombre==”Jose”) printf”(“Hola, Jose”);
```

el compilador arroja un error. Para comparar dos cadenas tienes que utilizar la función strcmp ( ). Esta función recibe como parámetros dos cadenas para ser comparadas entre ellas y retorna un valor que puede ser un número positivo, negativo o un cero, ver la tabla 17. Realmente, el compilador no compara los caracteres de la cadena, sino la suma de los códigos ASCII de cada uno de ellos. strcmp( ) también se utiliza para ir ordenando alfabéticamente una lista de palabras.

Sintaxis para comparar dos cadenas usando strcmp( ):

```
strcmp ( cadena1, cadena2)
```

donde:

strcmp es el nombre de la función que compara dos cadenas,
cadena1 y **cadena2** son las cadenas que se comparan.

Ejemplos de comparación de cadenas

strcmp(apellido1,apellido2)

strcmp(apellido1,”Perez”)

strcmp(“Juarez”,apellido2)

Tabla 17. Valores retornados por strcmp( ).

Valor retornado	Ejemplos
0	si cadena1=”casa” y cadena2=”casa”. Ambas cadenas deben estar escritas de igual forma, respetando mayúsculas y minúsculas.
>0	si cadena1=”casa” y cadena2=”Casa”. cadena1 > cadena2 porque el código ASCII de ‘C’ es 67 y el de ‘c’,99.
<0	si cadena1=”casa” y cadena2=”CASA”. cadena1 < cadena2 porque el código ASCII de las mayúsculas es menor que el de las minúsculas.

Para leer una cadena desde teclado puedes utilizar scanf( ) o gets( ).

Sintaxis para leer una cadena con scanf( ):

```
scanf ( "%s", cadena) ;
```

donde:

scanf función para leer desde teclado,
"%s" especificador de formato para leer cadena,
cadena nombre de la cadena en donde se guardará lo leído
;

Sintaxis para leer una cadena con gets( ):

```
gets ( cadena) ;
```

donde:

gets( ) es el nombre de la función que lee una cadena desde teclado,
cadena es la cadena en donde se guardará lo leído,
; indica al compilador la terminación de la instrucción

Ejemplos para leer una cadena

```
scanf("%s", nombre);
gets(nombre);
```

Es muy importante que notes dos características de scanf () al leer una cadena; primero, el especificador de formato es %s, no %c; segundo, el nombre de la cadena no lleva &. Aunque ambas funciones leen una cadena desde teclado, scanf( ) detiene su lectura cuando encuentra un espacio en blanco, a diferencia de gets( ). Así, si el usuario teclea "equivalente serie" y se lee con scanf( ), sólo se guardará "equivalente"; para guardar toda la cadena, tendrás que leer con gets( ).

Finalmente, para imprimir también deberás usar el especificador de formato %s dentro de printf( ). En la tabla 15 del capítulo tres podrás ver más funciones de cadenas, tal como strlen( ) que te permite saber la longitud de una cadena; sólo recuerda que tienes que incluir la librería string.h. En la figura 32 se muestra un código para que lo copies al editor de texto de tu compilador, compílalo, ejecútalo y observa lo que hace. Es una aplicación muy sencilla para determinar el Registro Federal de Contribuyente de una persona física, según el Servicio de Administración Tributaria (SAT).

```
#include<stdio.h>
#include<string.h>
void main()
{
    char RFC[11], dato[4];
    char password_usuario[10];
    char password_sistema[]="lenguajeC";
    do{
        printf("\n\t Introduzca su password: ");
        scanf("%s", password_usuario);
        if(strcmp(password_sistema,password_usuario)==0)
       {
           printf("\n\t Introduzca la letra inicial de su apellido paterno ( en mayuscula) : ");
           scanf("%s",dato);strcpy(RFC,dato);
           printf("\n\t Introduzca la primera vocal de su apellido paterno ( en mayuscula) : ");
           scanf("%s",dato); strcat(RFC,dato);
           printf("\n\t Introduzca la letra inicial de su apellido materno ( en mayuscula) : ");
           scanf("%s",dato); strcat(RFC,dato);
           printf("\n\t Introduzca la letra inicial de su nombre ( en mayuscula): ");
           scanf("%s",dato); strcat(RFC,dato);
           printf("\n\t Introduzca el numero de su mes de nacimiento ( use dos digitos) : ");
           scanf("%s",dato); strcat(RFC,dato);
           printf("\n\t Introduzca el numero de su dia de nacimiento ( use dos digitos) : ");
           scanf("%s",dato); strcat(RFC,dato);
           printf("\n\t Su RFC es %s: \n",RFC);
       }
        else printf("\n\t password incorrecto. Intente de nuevo\a\a ");
    }while(strcmp(password_sistema,password_usuario));
}
```

Figura 32. Uso de cadenas.

EJERCICIO DE CADENAS

Revisa el código de la figura 32 y contesta:
1. ¿Para qué sirve la función strcpy() en la línea 14?: _____

2. ¿Qué pasaría si en lugar de strcat( ) se ocupara strcpy( ) en la línea 18?_____

Solución:
1. **Para copiar la letra inicial del apellido en RFC** 2. **Se borraría el contenido anterior de RFC y se sustituiría por la letra inicial del apellido materno.**

4.4 Arreglos bidimensionales

Un *arreglo bidimensional* o *matriz* es un conjunto de variables del mismo tipo, bajo un mismo nombre y guardado en un espacio contiguo de la memoria en forma de filas y columnas, tal como se observa en la figura 33.

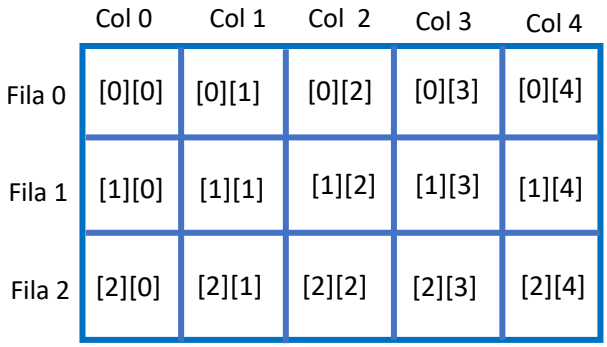


Figura 33. Arreglo bidimensional.

El tamaño de un arreglo bidimensional está determinado por el número de filas y el número de columnas. De esta manera, el tamaño de la matriz de la figura 33 es 3x4. Para tener acceso a un elemento de la matriz se utilizan dos índices: uno para especificar el número de la fila y otro para el número de columna; ambos índices empiezan en 0.

Sintaxis de declaración de un arreglo bidimensional:

```
tipo_de_dato identificador [ n_filas] [ n_columnas];
```

donde:

tipo_de_dato puede ser *char*, *int*, o cualquier otro de la Tabla 1,
identificador es el nombre asignado al arreglo,
n_filas es el número de filas que contendrá el arreglo,
n_columnas es el número de columnas que contendrá el arreglo,
; indica al compilador la terminación de la instrucción.

Ejemplos de declaración de arreglos bidimensionales o matrices

int R[100][10];

float num[5][20];

En el primer ejemplo, R es de tamaño 100x10; guardará valores enteros; el índice de la fila va desde 0 hasta 99 y el índice de la columna, de 0 hasta 9. En el segundo ejemplo, num es de tamaño 5x20; guardará valores flotantes; el índice de la fila va desde 0 hasta 4 y el índice de la columna, de 0 hasta 19.

La inicialización de un arreglo puede escribirse de varias formas, supongamos que tenemos un arreglo llamado matriz de tipo entero y tamaño 3x2:

```
matriz [3][2]={2,5,77,1,98,34};

matriz[3][2]={{2,5},{77,1},{98,34}};

matriz[3][2]={{2,5},
              {77,1},
              {98,34}};
```

son inicializaciones válidas del arreglo.

Para asignar un valor a un elemento del arreglo se utiliza la siguiente sintaxis:

Sintaxis de asignación a un elemento de un arreglo:

```
identificador [ índice_fila] [ índice_columna] = valor;
```

donde:

identificador es el nombre del arreglo,
[ ] sirve para encerrar los valores de los índices,
índice_fila es el valor de la fila en donde está el elemento deseado,
índice_columna es el valor de la columna en donde está el elemento deseado,
= es el operador de asignación,
valor es el valor que se guardará en el elemento del arreglo,

; indica al compilador la terminación de la instrucción

Ejemplos de asignación en arreglos

matriz[0][0]=5;

matriz[2][2]=18;

matriz[2][4]=10;

Estas asignaciones se verían gráficamente como se indica en la figura 34. Algunos compiladores asignan automáticamente valores de cero a los arreglos en el momento de su declaración, así que, dependiendo del compilador, el resto de los elementos del arreglo de la figura 34 serán ceros o tendrán cualquier valor considerado como "basura".

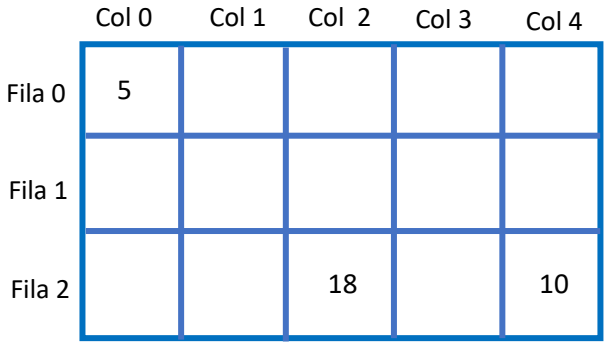


Figura 34. Asignación de valores en un arreglo.

Generalmente, para hacer un recorrido por todos los elementos de un arreglo se utilizan dos ciclos for, uno para el número de filas y el otro para el número de columnas. Es muy importante que cuides que los valores de los índices no sean mayores que los números de fila y columna que tenga el arreglo, ya que podría provocar un error o comportamiento anómalo del compilador por *desbordamiento*. El código de la figura 35 muestra cómo hacer una suma de matrices de 3x3. Codifícalo, compílalo y ejecútalo. Observa que los valores de la primera matriz están inicializadas y los de la segunda, son leídas desde teclado. Fíjate cómo se utilizan los ciclos for anidados para hacer un recorrido por la matriz.

EJERCICIO DE ARREGLOS

Modifica el código de la figura 35 para que los valores de la primera matriz también sean leídas desde teclado.

Solución:

```
for (i=0;i<3;i++)
        for(j=0;j<3;j++){
          printf ("\n\tIntroduzca el elemento [%d][%d] de la primera matriz: ",i,j);
              scanf("%d",&A[i][j]); }
```

```c
#include <stdio.h>
main(){
    int A[3][3]={{1,2,3},{2,4,6},{1,3,7}}; //inicialización de un arreglo
    int B[3][3], C[3][3];
    int i,j;
    printf ("\t ======SUMA DE DOS MATRICES===");
    //llenado de un arreglo
    for (i=0;i<3;i++)
      for(j=0;j<3;j++)
        {
            printf ("\n\tIntroduzca el elemento [%d][%d] de la segunda matriz: ",i,j);
            scanf("%d",&B[i][j]);
        }
    //suma de dos arreglos
    for (i=0;i<3;i++)
      for(j=0;j<3;j++)
        C[i][j]=A[i][j]+B[i][j];
    //impresión de arreglos
    printf ("\n\tMATRIZ A\n");
    for (i=0;i<3;i++)
    {
       for(j=0;j<3;j++) printf ("\t%d",A[i][j]);
       printf ("\n" );
    }

    printf ("\n\tMATRIZ B\n");
    for (i=0;i<3;i++)
    {
       for(j=0;j<3;j++) printf ("\t%d",B[i][j]);
       printf ("\n" );
    }
    printf ("\n\tMATRIZ C\n");
    for (i=0;i<3;i++)
    {
       for(j=0;j<3;j++) printf ("\t%d",C[i][j]);
       printf ("\n" );
    }
}
```

Figura 35. Suma de dos arreglos o matrices.

4.5 Ejercicios y Problemas

Ahora verás cómo aplicar los temas vistos en el capítulo en tres casos concretos:

a) En la determinación del valor de una resistencia a partir de su código de colores.

b) Encontrando las corrientes de mallas en un circuito resistivo.

c) Determinando los voltajes de nodo en un circuito resistivo.

El primer caso de estudio está orientado al uso de cadenas mientras que los otros dos ilustran la utilización de los arreglos. Los tres casos usan programación modular y los casos de arreglos muestran el uso de variables globales. Te recomiendo que, si tienes dudas sobre cómo usar el código de colores en las resistencias o sobre las técnicas de mallas o nodos, revises cualquier libro de circuitos eléctricos. En la sección de Referencias y Bibliografía de este capítulo te muestro mi recomendación tanto para el tema de resolución de circuitos eléctricos mediante mallas o nodos como para el método de Gauss Jordan y Determinantes usados en los casos dos y tres.

Caso de estudio 1: Código de colores en las resistencias

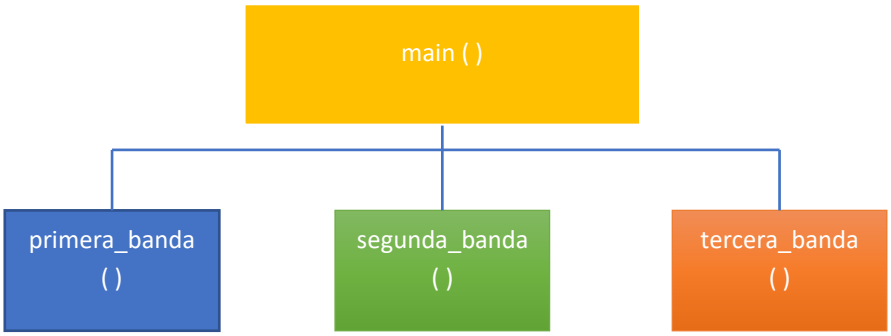
Hacer una aplicación que, a partir de los nombres de los colores de las tres primeras bandas de una resistencia de cuatro bandas, muestre al usuario su valor en ohms. La última banda, correspondiente a la tolerancia, no se pedirá en esta aplicación ya que se considerará que todas tienen una tolerancia de ±10%.

Utilizar la siguiente tabla de código de colores para generar la aplicación.

Color	1ra Banda	2da Banda	3ra Banda (Multiplicadora)
Negro	0	0	1
Café	1	1	10
Rojo	2	2	100
Naranja	3	3	1000
Amarillo	4	4	10000
Verde	5	5	100000
Azul	6	6	1000000
Violeta	7	7	10000000
Gris	8	8	100000000
Blanco	9	9	1000000000

PROPUESTA DE SOLUCIÓN

Aquí se propone un diseño que utiliza tres funciones definidas por el usuario:



- ***primera_banda ( )***. Esta función recibirá dos cadenas como parámetros:
  char color [20];
  char valor [20];
  En color [ ] se almacenará el color dado por el usuario, y en valor [ ] se irá guardando el valor de la resistencia de acuerdo con la primera columna de la tabla del código de colores. Así, si el usuario escribe “ROJO”, en valor [ ] se escribirá un “2”.
  Para mantener la simplicidad del código, la aplicación sólo reconocerá los colores escritos en mayúsculas. Además, se consideró que la primera banda nunca será “NEGRO”.

- ***segunda_banda( )***. Esta función también recibirá las dos cadenas anteriores, sólo que ahora color[ ] contiene el color de la segunda banda de la resistencia. Si el usuario escribe nuevamente “ROJO”, en valor[ ] se agregará un “2” al valor que tenía almacenado, de esta forma, la nueva cadena escrita en valor[ ] sería “22”.

- ***tercera_banda( )***. Al igual que las otras dos funciones, ésta también recibirá las dos cadenas. color [ ] tendrá ahora el color de la tercera banda y agregará a valor [ ] el número de ceros indicado en la tercera columna de la tabla. Siguiendo con el ejemplo, supongamos que ahora el usuario escribe “NARANJA”, lo que provocaría que ahora valor [ ] contenga “22000”.

- ***main( )***. En la función principal se leerá el nombre del color, se llamarán las funciones anteriores y se imprimirá el resultado.

El código propuesto queda de la siguiente forma:

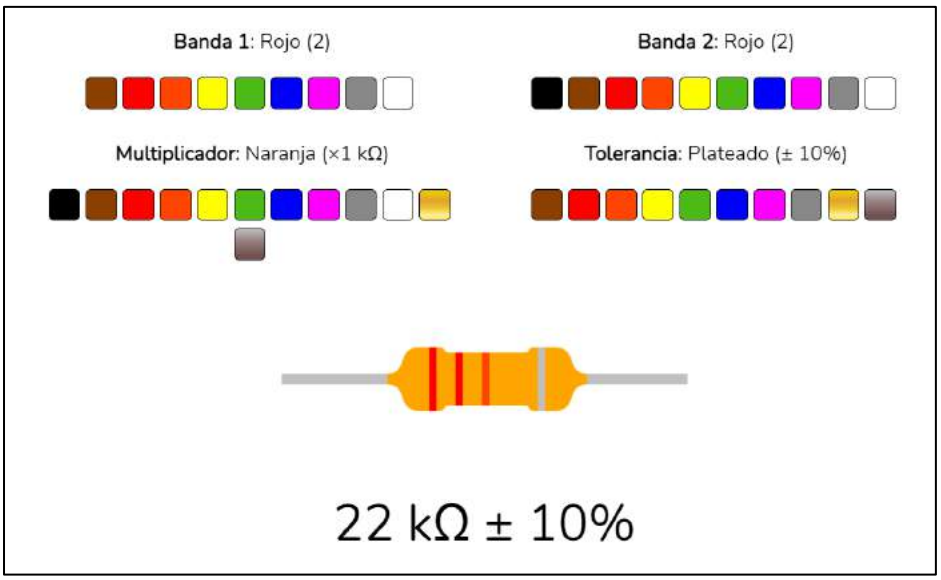
```
#include <stdio.h>
#include <string.h>
void primera_banda(char color[],char valor[]);
void segunda_banda(char color[],char valor[]);
void tercera_banda(char color[],char valor[]);

int main()
{
char color[20], valor[20];

printf("\n\t=== VALOR DE UNA RESISTENCIA A PARTIR DE SU CODIGO DE COLORES ===\n");
printf("\t   Para usar la aplicacion, escriba usando solo MAYUSCULAS\n");
printf("\n\t\t Color de la Primera Banda: ");
scanf("%s",color);
primera_banda(color,valor);
  if (strcmp(valor,"NO VALIDO")!=0)
  {
    printf("\n\t\t Color de la Segunda Banda: ");
    scanf("%s",color);
    segunda_banda(color,valor);
    if (strcmp(valor,"NO VALIDO")!=0)
        {
      printf("\n\t\t Color de la Tercera Banda: ");
      scanf("%s",color);
      tercera_banda(color,valor);
      if (strcmp(valor,"NO VALIDO")!=0)
      printf("\t\t Valor de la resistencia es: %s ohms\n",valor);
      else printf("\n\t\a\aOlvidaste escribir en mayusculas o %s no es un color valido ",color);
     }
    else printf("\n\t\a\aOlvidaste escribir en mayusculas o %s no es un color valido ",color);
    }
   else printf("\n\t\a\aOlvidaste escribir en mayusculas o %s no es un color valido ",color);

printf("\n\n GRACIAS POR USAR ESTA APLICACION...");
}

void primera_banda(char color[],char valor[])
{
       if(strcmp("CAFE",color)==0 ) strcpy(valor,"1");
        else if(strcmp("ROJO",color)==0 ) strcpy(valor,"2");
         else if(strcmp("NARANJA",color)==0 ) strcpy(valor,"3");
          else if(strcmp("AMARILLO",color)==0 ) strcpy(valor,"4");
```

```
          else if(strcmp("VERDE",color)==0 ) strcpy(valor,"5");
           else if(strcmp("AZUL",color)==0 ) strcpy(valor,"6");
            else if(strcmp("VIOLETA",color)==0 ) strcpy(valor,"7");
             else if(strcmp("GRIS",color)==0 ) strcpy(valor,"8");
              else if(strcmp("BLANCO",color)==0 ) strcpy(valor,"9");
               else strcpy(valor,"NO VALIDO");
}

void segunda_banda(char color[],char valor[])
{
  if(strcmp("NEGRO",color)==0 ) strcat(valor,"0");
      else if(strcmp("CAFE",color)==0 ) strcat(valor,"1");
       else if(strcmp("ROJO",color)==0 ) strcat(valor,"2");
        else if(strcmp("NARANJA",color)==0 ) strcat(valor,"3");
         else if(strcmp("AMARILLO",color)==0 ) strcat(valor,"4");
          else if(strcmp("VERDE",color)==0 ) strcat(valor,"5");
           else if(strcmp("AZUL",color)==0 ) strcat(valor,"6");
            else if(strcmp("VIOLETA",color)==0 ) strcat(valor,"7");
             else if(strcmp("GRIS",color)==0 ) strcat(valor,"8");
              else if(strcmp("BLANCO",color)==0 ) strcat(valor,"9");
               else strcpy(valor,"NO VALIDO");
}

void tercera_banda(char color[],char valor[])
{
  if(strcmp("NEGRO",color)==0 ) strcat(valor,"");
      else if(strcmp("CAFE",color)==0 ) strcat(valor,"0");
       else if(strcmp("ROJO",color)==0 ) strcat(valor,"00");
        else if(strcmp("NARANJA",color)==0 ) strcat(valor,"000");
         else if(strcmp("AMARILLO",color)==0 ) strcat(valor,"0000");
          else if(strcmp("VERDE",color)==0 ) strcat(valor,"00000");
           else if(strcmp("AZUL",color)==0 ) strcat(valor,"000000");
            else if(strcmp("VIOLETA",color)==0 ) strcat(valor,"0000000");
             else if(strcmp("GRIS",color)==0 ) strcat(valor,"00000000");
              else if(strcmp("BLANCO",color)==0 ) strcat(valor,"000000000");
               else strcpy(valor,"NO VALIDO");
}
```

Probando la aplicación con el ejemplo anterior:

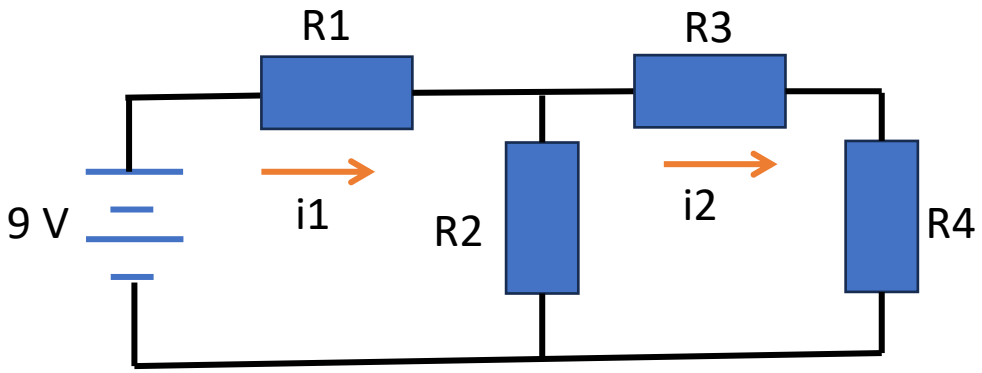
```
=== VALOR DE UNA RESISTENCIA A PARTIR DE SU CODIGO DE COLORES ===
   Para usar la aplicacion, escriba usando solo MAYUSCULAS

         Color de la Primera Banda: ROJO

         Color de la Segunda Banda: ROJO

         Color de la Tercera Banda: NARANJA
         Valor de la resistencia es: 22000 ohms


         GRACIAS POR USAR ESTA APLICACION...
```

Podemos comprobar la salida de forma teórica, usando la tabla de código de colores, o comparando el resultado con la salida de alguna otra aplicación de calculadora de código de colores. Aquí comparo el resultado del código propuesto con el de una calculadora en línea cuyo link te dejo en Referencias y Bibliografía. Sería una buena práctica agregar una función más para determinar la tolerancia de la resistencia.



Caso de estudio 2: Mallas

A partir de la siguiente configuración de circuito resistivo, hacer una aplicación que calcule las dos corrientes de mallas para diferentes valores de resistencias y fuente de alimentación. Los valores de las resistencias estarán dados en ohms y la fuente de alimentación en volts.



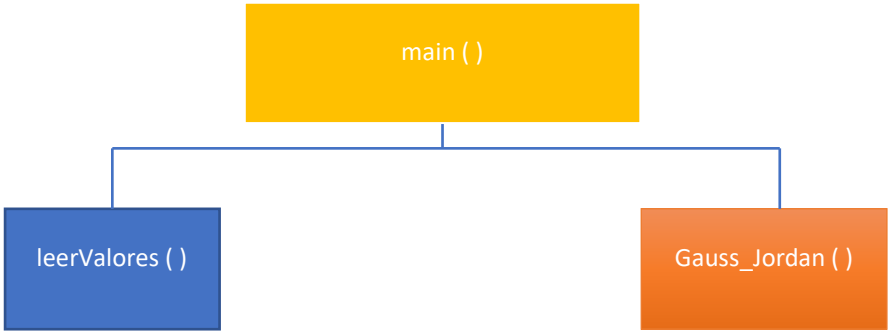
PROPUESTA DE SOLUCIÓN

Haciendo el análisis del circuito, se obtienen las siguientes ecuaciones para la malla 1 y para la malla 2:

$$(R_0 + R_1)i_1 - R_1 i_2 = V$$

$$(-R_1)i_1 + (R_1 + R_2 + R_3)i_2 = 0$$

Se propone resolver el sistema de ecuaciones usando el método de Gauss-Jordan, para lo cual se utilizará un arreglo de 2x3. El diseño a bloques del programa sería el siguiente:



- ***leerValores ( )***. Esta función solicitará el valor de la fuente de alimentación y los valores de las resistencias; estas últimas se guardarán en un vector tipo double, R[ ].
  Además, colocará los coeficientes en la matriz aumentada, Valores[ ][ ]; esta matriz está declarada de forma global para que sus valores sean conocidos en todo el programa sin necesidad de enviarlo como parámetro de función.

- ***Gauss_Jordan( )***. Aquí se resuelve la matriz usando el método de Gauss-Jordan, no necesita recibir a la matriz como parámetro debido a que es global.

- ***main( )***. En la función principal se llamarán a las funciones anteriores y se imprimirá el resultado en miliamperes.

El código propuesto queda de la siguiente forma:

```
#include<stdio.h>
double Valores[2][3];
void leerValores();
void Gauss_Jordan();

void main()
{
leerValores();
Gauss_Jordan();
printf("\n\t\t CORRIENTES DE MALLA ENCONTRADAS\n");
printf("\n\t\t CORRIENTE DE MALLA 1: %.2lf mA\n",Valores [0][2]*1000);
printf("\n\t\t CORRIENTE DE MALLA 2: %.2lf mA\n",Valores [1][2]*1000);
}

void leerValores()
{
      int i,V;
      double R[4];
      printf("\n\t VALOR DE LA FUENTE DE ALIMENTACION (Volts): ");
      scanf("%d",&V);
      for (i=0;i<4;i++)
      {
         printf ("\n\t VALOR DE R%d (Ohms): ",i+1 ); scanf("%lf",&R[i]);
      }
      Valores[0][0]=R[0]+R[1];
      Valores[0][1]=R[1]*(-1);
```

```
	Valores[0][2]=V;
	Valores[1][0]=R[1]*(-1);
	Valores[1][1]=R[1]+R[2]+R[3];
	Valores[1][2]=0;
}

void Gauss_Jordan()
{
	int i,j;
	double k;
	k=Valores[0][0];
	for(j=0;j<3;j++) Valores[0][j]=Valores[0][j]/k;
	k=Valores[1][0]*(-1);
	for (j=0;j<3;j++) Valores[1][j]=Valores[0][j]*k+Valores[1][j];
	k=Valores[1][1];
	for(j=1;j<3;j++) Valores[1][j]=Valores[1][j]/k;
	k=Valores[0][1]*(-1);
	for (j=1;j<3;j++) Valores[0][j]=Valores[1][j]*k+Valores[0][j];
}
```

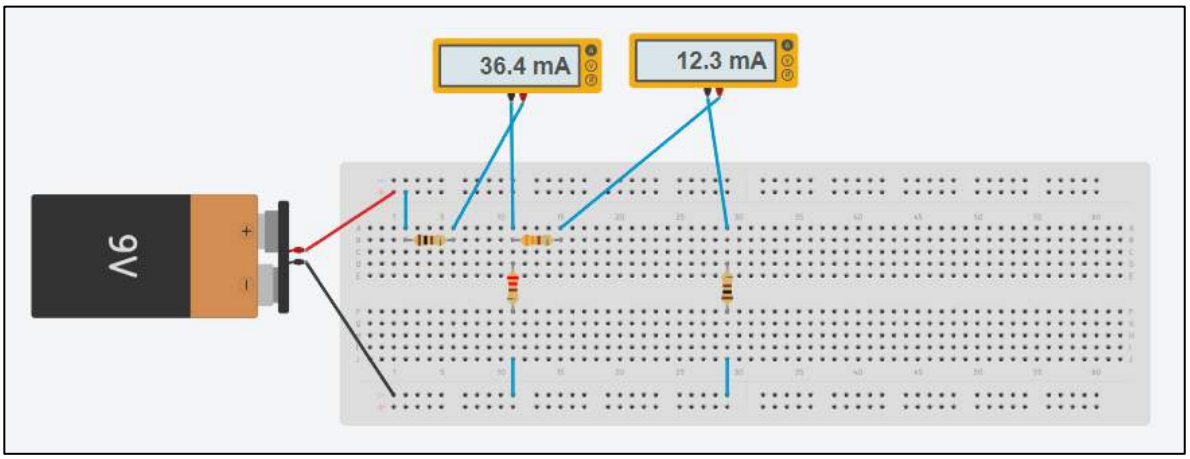
Se probó la aplicación con los siguientes valores:

R_1=100Ω, R_2=220Ω, R_3=330Ω, R_4=100Ω con V=9Volts

dando la siguiente salida:

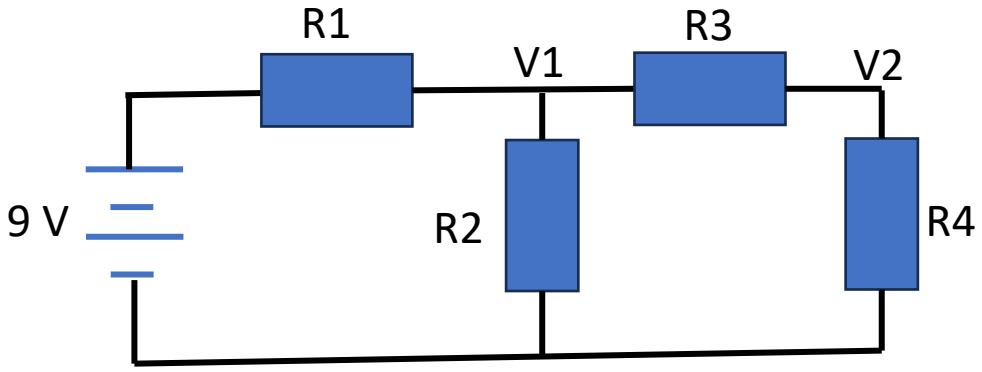
```
VALOR DE LA FUENTE DE ALIMENTACION (Volts): 9

VALOR DE R1 (Ohms): 100

VALOR DE R2 (Ohms): 220

VALOR DE R3 (Ohms): 330

VALOR DE R4 (Ohms): 100

        CORRIENTES DE MALLA ENCONTRADAS

        CORRIENTE DE MALLA 1: 36.65 mA

        CORRIENTE DE MALLA 2: 12.41 mA
```

Para validar los resultados obtenidos, se utilizó un simulador de circuitos electrónicos gratuito en línea, puedes ver el link en Referencias y Bibliografía.



Caso de estudio 3: Nodos

A partir de la siguiente configuración de circuito resistivo, hacer una aplicación que calcule los dos voltajes de nodos para diferentes valores de resistencias y fuente de alimentación. Los valores de las resistencias estarán dados en ohms y la fuente de alimentación en volts.



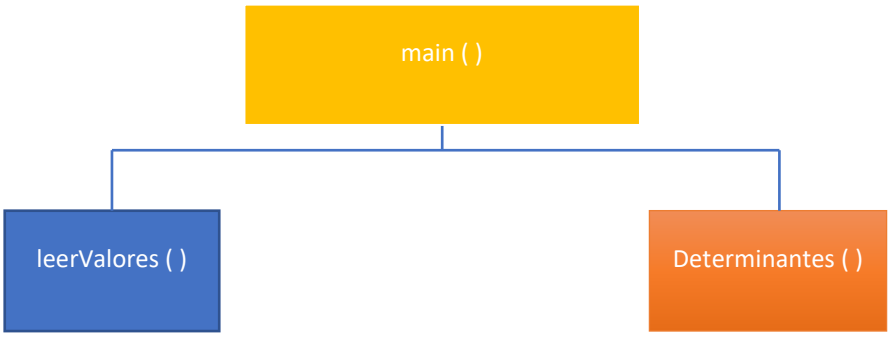
PROPUESTA DE SOLUCIÓN

Haciendo el análisis del circuito, se obtienen las siguientes ecuaciones para el nodo 1 y el nodo 2

$$\left({}^{1}/_{R_0} + {}^{1}/_{R_1} + {}^{1}/_{R_2}\right) V_1 - {}^{1}/_{R_2}\, V_2 = {}^{1}/_{R_0}\, V$$

$$\left(-{}^{1}/_{R_2}\right)V_1 + \left({}^{1}/_{R_2} + {}^{1}/_{R_3}\right)V_2 = 0$$

Se propone resolver el sistema de ecuaciones usando el método de determinantes. Para guardar los valores de los coeficientes que se utilizarán en el cálculo de los determinantes, se utilizará un arreglo de 2x4, en la última columna se guardarán los resultados. El diseño a bloques del programa sería el siguiente:



- ***leerValores ( )***. Esta función solicitará el valor de la fuente de alimentación así como los valores de las resistencias con los cuales se calcularán las conductancias que se almacenarán en el vector conductancias [ ].
  Además, colocará los coeficientes en la matriz aumentada, Valores[ ][ ]; esta matriz está declarada de forma global para que sus valores sean conocidos en todo el programa sin necesidad de enviarlo como parámetro de función.

- ***Determinantes( )***. Aquí se resuelve el sistema de ecuaciones usando la técnica de los determinantes.

- ***main( )***. En la función principal se llamarán a las funciones anteriores y se imprimirá el resultado en volts.

El código propuesto queda de la siguiente forma:

```
#include<stdio.h>
double Valores[2][4];
void leerValores();
void Determinantes();
void main()
{
leerValores();
```

```
Determinantes();
printf("\n\t\t VOLTAJES DE NODOS ENCONTRADOS\n");
printf("\n\t\t VOLTAJE DE NODO 1: %.2lf V\n",Valores [0][3]);
printf("\n\t\t VOLTAJE DE NODO 2: %.2lf V\n",Valores [1][3]);
}

void leerValores()
{
      int i,V;
      double conductancia[4];
      printf("\n\t VALOR DE LA FUENTE DE ALIMENTACION (Volts): ");
      scanf("%d",&V);
      for (i=0;i<4;i++)
      {
         printf ("\n\t VALOR DE R%d (Ohms): ",i+1 ); scanf("%lf",&conductancia[i]);
         conductancia[i]=1/conductancia[i];
      }
      Valores[0][0]=conductancia[0]+conductancia[1]+conductancia[2];
      Valores[0][1]=conductancia[2]*(-1);
      Valores[0][2]=conductancia[0]*V;
      Valores[1][0]=conductancia[2]*(-1);
      Valores[1][1]=conductancia[2]+conductancia[3];
      Valores[1][2]=0;
}

void Determinantes()
{
      double DV1,DV2,DV;
      DV=Valores[0][0]*Valores[1][1]-Valores[1][0]*Valores[0][1];
      DV1=Valores[0][2]*Valores[1][1]-Valores[1][2]*Valores[0][1];
      DV2=Valores[0][0]*Valores[1][2]-Valores[1][0]*Valores[0][2];
      Valores[0][3]=DV1/DV;
      Valores[1][3]=DV2/DV;
}
```

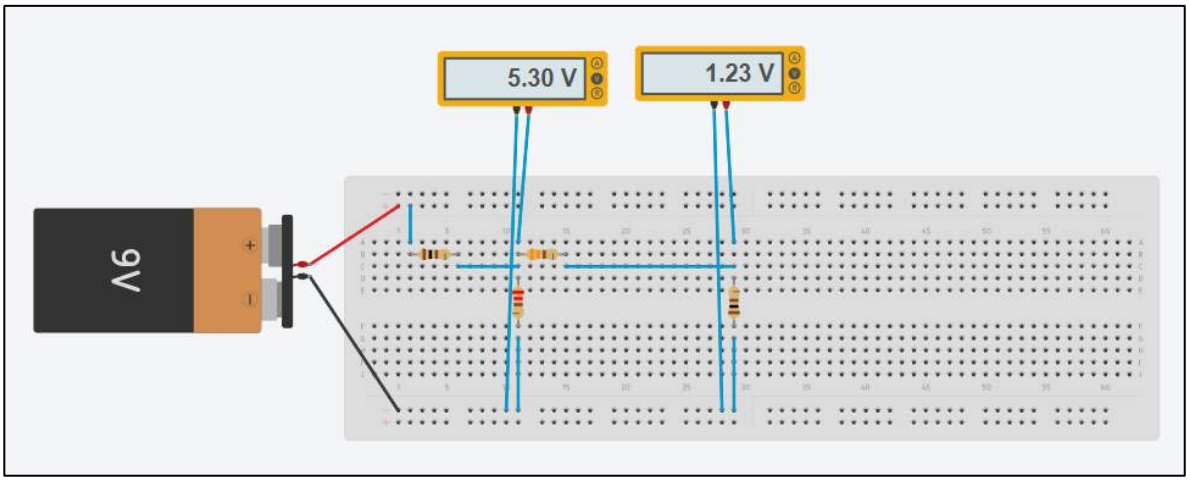
Se probó la aplicación con los siguientes valores:

R_1=100Ω, R_2=220Ω, R_3=330Ω, R_4=100Ω con V=9Volts

dando la siguiente salida:

```
VALOR DE LA FUENTE DE ALIMENTACION (Volts): 9

VALOR DE R1 (Ohms): 100

VALOR DE R2 (Ohms): 220

VALOR DE R3 (Ohms): 330

VALOR DE R4 (Ohms): 100

        VOLTAJES DE NODOS ENCONTRADOS

        VOLTAJE DE NODO 1: 5.33 V

        VOLTAJE DE NODO 2: 1.24 V
```

Se muestra la simulación del circuito con los mismos valores de resistencias y voltaje con el fin de validar los resultados obtenidos



Referencias y bibliografía:

Kernighan B., Ritchie D. (1978). El lenguaje de programación C. Prentice Hall.

García-Bermejo Giner, J. R. (2008). Programación estructurada en C. Pearson Educación.

Jiménez Castells, M. y Otero Calviño, B. (2015). Fundamentos de ordenadores: programación en C. Universitat Politècnica de Catalunya.

Menchaca García, F. R. (2010). Fundamentos de programación en Lenguaje C. Instituto Politécnico Nacional.

Joyanes Aguilar, L. (2005). C algoritmos, programación y estructuras de datos. McGraw-Hill España.

Gaxiola Pacheco, C. G. y Flores Gutiérrez, D. L. (2008). Metodología de la programación con pseudocódigo enfocado al lenguaje C. Plaza y Valdés, S.A. de C.V.

Moreno Pérez, J. C. (2015). Programación. RA-MA Editorial.

Shildt H (2000) C The complete reference. McGrawHill

Chapra & R. Canale (2010), Métodos Numéricos para Ingenieros. Sexta Edición. Editorial Mc-Graw Hill.

Dorf, R., & Svoboda, J. (2011). Circuitos eléctricos. Alfaomega

https://www.calcuvio.com/codigo-resistencias

https://www.tinkercad.com/dashboard

Capítulo 5

ARCHIVOS

OBJETIVO GENERAL

Al finalizar el capítulo, el estudiante será capaz de hacer programas que involucren uso de archivos.

.

OBJETIVOS PARTICULARES:

- Conocer y saber utilizar funciones para lectura de archivos.
- Conocer y saber utilizar funciones para escritura de archivos.

5.1 Introducción

Hasta el momento todos los programas que has escrito tienen algo en común: una vez que los cierras, los datos desaparecen. En este capítulo empezarás a escribir programas que te permitan almacenar datos en un archivo usando funciones propias de la librería stdio.h. En el contexto general, un *archivo* es una sección de almacenamiento, regularmente un disco, identificada bajo un nombre. Sin embargo, en el lenguaje C, un archivo es considerado como una secuencia de bytes continuos que pueden ser leídos uno por uno.

En C estándar hay dos tipos de archivo: binario y texto. El compilador de C ve al tipo binario como una secuencia de bytes mientras que al tipo de texto como una secuencia de caracteres que se estableció para tener compatibilidad con el sistema operativo de Microsoft.

El paquete de entrada/salida estándar de C funciona con *buffers*. Un buffer es un área de memoria intermedia entre el programa y el archivo, como se muestra en la figura 36. Veamos cómo funciona el concepto del buffer. Supongamos que un programa lee un archivo, entonces se copia un bloque del archivo al buffer y desde allí, el programa examina el contenido byte por byte. Esto incrementa la velocidad de transferencia de datos porque el acceso a un buffer es más rápido que el acceso a un dispositivo físico, tal como lo es un disco.

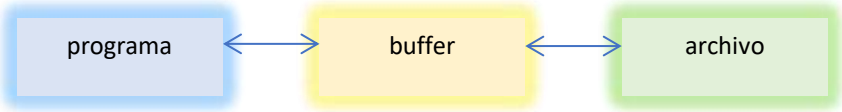


Figura 36. Esquema de relación de un buffer con el archivo y el programa.

A las acciones que se realizan en un archivo se les conoce como *operaciones de archivos*. En este capítulo verás los temas necesarios para hacer operaciones básicas de archivos: abrir, leer, escribir y cerrar. Si deseas profundizar en el tema, te sugiero consultar los libros de la sección Referencias y Bibliografía, especialmente el libro de Francisco Ceballos. Al final del capítulo te presento dos casos de estudio en donde podrás ver cómo usar las funciones de archivo.

5.2 La función fopen( ) y función fclose( )

Para abrir un archivo se utiliza la función fopen( ), definida en stdio.h, que recibe dos parámetros, el nombre del archivo y el *modo* en que se abre, y regresa un *puntero*. Un puntero es una variable que guarda una dirección de memoria. Si el archivo se abrió con éxito, fopen( ) regresa un puntero de tipo FILE que contiene, entre otros datos, el buffer asociado al archivo; en caso contrario, regresa un puntero NULL, que significa que hubo un error tal como disco lleno o nombre de archivo inválido. Recuerda que el buffer es una memoria intermedia.

Los *modos* en que se puede abrir un archivo se muestran en la primera columna de la Tabla 18. Verás, en la misma tabla, que se hace referencia a un *archivo texto* y a un *archivo binario*. Cuando C ve un archivo texto tiene que hacer una conversión de caracteres de final de línea, si se corre desde un sistema operativo de Microsoft.

Tabla 18. Modos para abrir un archivo.

Modo	Descripción
r	Abre un archivo texto para leer; si no existe, produce error.
w	Abre un archivo texto para escribir; si no existe, lo crea; si existe, elimina el contenido.
a	Abre un archivo texto para escribir; si no existe, lo crea; si existe, lo añade al final de su contenido.
r+	Abre un archivo texto para leer y escribir; si no existe, produce error.
w+	Abre un archivo texto para leer y escribir; si no existe, lo crea; si existe, elimina el contenido.
a+	Abre un archivo texto para leer y escribir; si no existe, lo crea; si existe, lo añade al final de su contenido.
rb	Abre un archivo binario para leer; si no existe, produce error.
wb	Abre un archivo binario para escribir; si no existe, lo crea; si existe, elimina el contenido.
ab	Abre un archivo binario para escribir; si no existe, lo crea; si existe, lo añade al final de su contenido.
rb+	Abre un archivo binario para leer y escribir; si no existe, produce error.
wb+	Abre un archivo binario para leer y escribir; si no existe, lo crea; si existe, elimina el contenido.
ab+	Abre un archivo binario para leer y escribir; si no existe, lo crea; si existe, lo añade al final de su contenido.

Sintaxis para abrir un archivo con fopen ( ):

```
fp = fopen ( “nombre_archivo” , “modo” ) ;
```

donde:

fp es un puntero de tipo FILE,
fopen( ) es la función para abrir un archivo,
nombre_archivo es el nombre del archivo que se desea abrir,
modo indica el modo en que se abrirá el archivo (Tabla 18),
; indica al compilador la terminación de la instrucción.

Ejemplo de cómo abrir un archivo

```
#include<stdio.h>
main()
{
        FILE *fp;
        fp=fopen ("C:\\Documents\\prueba.txt","r");
        if (fp==NULL) printf("Error al abrir el archivo");
}
```

Es necesario declarar primero un puntero de tipo FILE para almacenar el valor devuelto por fopen( ). Observa que en el parámetro del nombre del archivo están, entre comillas dobles, la ruta y la extensión del archivo que se desea abrir. Antes de hacer alguna operación de escritura o lectura hay que verificar que el archivo se haya abierto con éxito, para eso se pregunta si el valor del puntero retornado es NULL.

Al finalizar el programa, el archivo se cierra automáticamente. Sin embargo, es buena práctica forzar el cierre mediante la función *fclose( )*, también definida en stdio.h. Esta función recibe como parámetro el nombre del puntero que identifica al archivo y regresa un 0 si el cierre fue exitoso o un valor de EOF si hubo problemas tal como disco lleno o que el medio de almacenamiento haya sido removido. EOF (End of File), fin de archivo en español, es un valor especial que indica que se ha alcanzado el fin del archivo. Es buena práctica de programación preguntar por el valor revuelto por fclose ( ) para saber si hubo algún error durante su ejecución.

Sintaxis para cerrar un archivo con fclose ( ):

```
fclose ( fp  )
```

donde:

fclose( ) es la función para cerrar un archivo,
fp es un puntero de tipo FILE que señala el archivo para cerrar.

Ejemplo de cómo cerrar un archivo

```
if (fclose(fp) != 0 ) printf("Error al cerrar el archivo");
```

5.3 Funciones de lectura de archivos

En una operación de lectura se utiliza la función feof( ) se utiliza para saber si ya se ha alcanzado el fin de archivo. Esta función devuelve un valor 0 si todavía no se ha terminado el archivo y un valor distinto de 0 cuando suceda. El parámetro que recibe es el puntero asociado al buffer del archivo.

Sintaxis de feof( ):

```
feof ( fp  )
```

donde:

feof( ) es la función para saber si ya se ha alcanzado el fin de archivo,
fp es un puntero asociado al buffer del archivo.

Ejemplo de cómo usar feof( )

```
while(! feof (fp) { /*sentencias de lectura*/}
```

En este ejemplo, se utiliza el valor retornado por feof( ) para seguir en el ciclo while y continuar leyendo, cuando se llegue al fin de archivo y la función retorne un valor diferente de cero, el ciclo terminará y ya no seguirá leyendo. Hay tres funciones en el C estándar con las cuales puedes leer un archivo:

- fgetc( ). Lee caracter por caracter.
- getw( ). Lee palabra por palabra.
- fgets( ). Lee una cadena de caracteres.
- fscanf( ). Lee una cadena con formato.
- fread( ). Lee registro por registro.

A continuación, veremos cada una de ellas.

fgetc ( ) lee el caracter de la posición indicada por el puntero de lectura/escritura del archivo y avanza el puntero a la siguiente posición. Devuelve el caracter leído si la operación fue un éxito y un EOF si ocurre un error o se ha llegado al final del archivo.

Sintaxis de fgetc( ):

```
fgetc ( caracter, fp  ) ;
```

donde:

fgetc( ) es la función para leer un caracter desde un archivo,
fp es un puntero asociado al buffer del archivo.

Ejemplo de cómo utilizar fgetc( )

```
char cadena[40];
while ( ! ferror ( fp ) && ! feof (fp))
      cadena[i++]=fgetc(fp);
cadena[--i]='\0';
```

En este ejemplo, fgetc( ) lee el archivo caracter por caracter y lo va copiando en cadena[ ], observa que la última línea sirve para escribir el caracter nulo al final de la cadena, ya que esto no se hace de forma automática. Para ver el contenido de cadena[ ] en la pantalla, sólo deberás escribir *printf("%s",cadena);*.

getw ( ) lee un dato binario de tipo int, conocido como *palabra*, en la posición indicada por el puntero de lectura/escritura del archivo y avanza el puntero a la siguiente posición. Devuelve el número leído si la operación fue un éxito y un EOF si ocurre un error.

Sintaxis de getw( ):

```
getw ( fp ) ;
```

donde:

getw( ) es la función para leer un dato tipo int desde un archivo,
fp es un puntero asociado al buffer del archivo.

Ejemplo de cómo utilizar getw( )

```
while(ferror(pf) || feof(pf))
    {
      n=getw(pf);
      printf("%d ", n);
     }
```

Si es necesario volver a colocar el puntero de lectura/escritura al inicio del archivo, sólo tienes que usar la función *rewind( )* con el nombre del puntero asociado al archivo como argumento de la función.

fgets ( ) lee una cadena de caracteres desde un archivo. Se considera una cadena a la secuencia de caracteres que se encuentra desde la posición actual del puntero hasta el caracter '\n', incluyéndolo, o hasta llegar al fin del archivo, o hasta n-1 caracteres; n se especifica como un parámetro de la función. Devuelve un puntero al inicio de la cadena a leer si la operación fue un éxito y NULL si ocurre un error.

Sintaxis de fgets( ):

```
fgets ( cadena,n, fp  ) ;
```

donde:

fgets( ) es la función para leer una cadena de un archivo,
cadena, aquí se guardará la cadena leída del archivo,
n es el número de caracteres de la cadena;
fp es un puntero asociado al buffer del archivo.

Ejemplo de cómo utilizar fgets( )

```
cadena[50];
while(fgets(cadena,50,fp) !=NULL)
printf("%s",cadena);
```

fscanf ( ) funciona como scanf( ) pero en lugar de leer desde teclado, lo hace desde un archivo apuntado por un puntero de tipo FILE. Devuelve el número de argumentos leídos y asignados si la operación fue un éxito, un 0 si ocurre un error en la asignación de valores. y un EOF si se ha llegado al fin del archivo.

Sintaxis de fscanf( ):

```
fscanf ( fp , formato) ;
```

donde:

fscanf( ) es la función para leer desde un archivo,
fp es un puntero asociado al buffer del archivo.
formato es la especificación del formato usado.

Ejemplo de cómo utilizar fscanf( )

```
fscanf(fp, "Color %s fue selecionado por %d personas\n", color, &num);
```

Recuerda poner & antes del nombre de la variable en donde se almacenará el dato, excepto para una cadena.

fread ( ) permite leer datos en un archivo en forma de registros o bloques. Lee *n* elementos de longitud *tamaño* bytes desde el archivo apuntado por *fp* y los almacena en *variable*. Devuelve el número de elementos leídos si la operación fue un éxito, si el número devuelto es menor que el número especificado en su llamada entonces ocurrió un error durante la lectura.

Un registro puede ser una variable de cualquiera de los tipos mencionados en la Tabla 1, incluso una cadena de caracteres. Sin embargo, lo más usual es que los registros utilicen tipos struct, como lo verás en el segundo caso de estudio de este capítulo.

Sintaxis de fread( ):

```
fread ( &dato, tamaño, n, fp) ;
```

donde:

fread( ) es la función para leer información en forma de registros,
& se usa igual que en scanf( ), se omite si se trata del nombre de una cadena,
dato es el nombre de la variable en donde se guarda el dato leído del archivo,
tamaño cantidad de bytes utilizadas por el dato,
n es el número de datos que se leerán desde el archivo,
fp es un puntero asociado al buffer del archivo.

Ejemplo de cómo utilizar fread( )

```
char cadena[50];
fread(cadena, sizeof(cadena),1,fp);
```

sizeof( ) es una función que retorna el número de bytes que está ocupando la variable que se le envía como parámetro, usando esta función nos evitamos hacer el cálculo de bytes.

5.4 Funciones de escritura de archivos

En ocasiones, cuando se busca realizar una operación con los archivos se puede producir un error que puede ser detectado por la función ferror( ) que devuelve un valor distinto de cero cuando esto sucede, y un valor de 0 si hubo éxito en la operación. Una vez que ferror( ) ha detectado un error se queda activado hasta que sea desactivado por la función clearerr( ). Ambas funciones reciben un puntero al buffer del archivo como argumento.

Ejemplo de cómo utilizar ferror( ) y clearerr( )

```
if ( ferror ( fp ) )
{
   printf("Error al escribir en el archivo");
   clearerr(fp);
}
```

Para escribir en un archivo, C estándar tiene tres funciones:

- fputc( ). Escribe caracter por caracter.
- putw( ). Escribe palabra por palabra.
- fputs( ). Escribe una cadena.
- fprintf( ). Escribe una cadena con formato.
- fwrite( ). Escribe registro por registro.

A continuación, las veremos con detalle.

fputc( ) escribe un caracter en la posición indicada por el puntero de lectura/escritura del archivo. Devuelve el caracter escrito si la operación fue un éxito y un EOF si ocurre un error.

Sintaxis de fputc( ):

```
fputc ( caracter, fp ) ;
```

donde:

fputc( ) es la función para escribir un caracter a un archivo,

caracter es el caracter enviado a archivo,
fp es un puntero asociado al buffer del archivo.

Ejemplo de cómo utilizar fputc( )

```
char mensaje[40]=" Mensaje copiado a archivo";
while ( ! ferror ( fp ) && mensaje[i] )
     fputc (mensaje[i++], fp);
```

putw ( ) escribe una *palabra*, esto es, un dato binario de tipo int en la posición indicada por el puntero de lectura/escritura del archivo. Devuelve el número escrito si la operación fue un éxito y un EOF si ocurre un error.

Sintaxis de putw( ):

```
putw ( número, fp  ) ;
```

donde:

putw( ) es la función para escribir un dato tipo int a un archivo,
número es el número enviado a archivo,
fp es un puntero asociado al buffer del archivo.

Ejemplo de cómo utilizar putw( )

```
for(i=0;i<10;i++)
    {
      printf("Valor para el archivo: ");
      scanf("%d",&num);
     putw(num,fp);
     if (ferror(pf) )
      {
         printf ("Error al escribir en archivo");
         exit(1);
       }
     }
```

fputs ( ) copia una cadena de caracteres a un archivo, excepto el caracter nulo. Devuelve un 0 si la operación fue un éxito y un valor distinto de 0 si ocurre un error.

Sintaxis de fputs( ):

```
fputs ( cadena, fp  ) ;
```

donde:

fputs( ) es la función para escribir una cadena a un archivo,
cadena es la cadena enviada a archivo,
fp es un puntero asociado al buffer del archivo.

Ejemplo de cómo utilizar fputs( )

```
fputs(cadena,fp);
fputc ('\n', fp);
```

Ya que fputs( ) no copia el capacter nulo, se agrega al final usando fputc( ) para facilitar la posterior operación de lectura.

fprintf ( ) funciona como printf( ) pero en lugar de imprimir en pantalla, imprime en un archivo apuntado por un puntero de tipo FILE. Devuelve el número de caracteres escritos si la operación fue un éxito y un número negativo si ocurre un error.

Sintaxis de fprintf( ):

```
fprint ( fp , formato) ;
```

donde:

fprintf( ) es la función para imprimir en un archivo,
fp es un puntero asociado al buffer del archivo.

formato es la especificación del formato usado.

Ejemplo de cómo utilizar fprintf( )

```
fprintf(fp, “Valor %d = %.2f\n”, n, voltaje);
```

Usar fprintf( ) para guardar números no es muy recomendable si se desea ahorrar espacio en disco, ya que esta función utiliza un byte por cada dígito. Así, por ejemplo, guardar -34534.238 con fprintf( ) ocuparía 10 bytes en lugar de los 4 bytes que ocupa un número de tipo flotante.

fwrite ( ) permite escribir datos en un archivo en forma de registros o bloques. Se utiliza regularmente para guardar arreglos o datos tipo struct (verás este tipo de dato en el segundo caso de estudio). Sin embargo, también es posible utilizarlo para guardar los otros tipos de datos, tales como int, float, char, incluyendo cadenas de caracteres. Devuelve el número de elementos escritos si la operación fue un éxito, si el número devuelto es menor que el número especificado en su llamada entonces ocurrió un error durante la escritura.

Esta función, a diferencia del fprintf( ), almacena los datos numéricos en formato binario; esto es, que un int ocupa 2 o 4 bytes, dependiendo del compilador, un float 4 bytes, etc.

Sintaxis de fwrite( ):

```
fwrite ( &dato, tamaño, n, fp) ;
```

donde:

fwrite( ) es la función para guardar información en forma de registros,
& se usa igual que en scanf( ), se omite si se trata del nombre de una cadena,
dato es el nombre de la variable que contiene el dato que se guarda en el archivo,
tamaño cantidad de bytes necesarias para almacenar el dato,
n es el número de datos que se escribirán en el archivo,
fp es un puntero asociado al buffer del archivo.

Ejemplo de cómo utilizar fwrite( )

```
char cadena[50];
fwrite(cadena, sizeof(cadena),1,fp);
```

Observa que en la llamada a fwrite( ) se omitió el símbolo & porque el dato a escribir en el archivo es una cadena. sizeof( ) es una función que retorna el número de bytes que está ocupando la variable que se le envía como parámetro.

5.5 Ejercicios y Problemas

Ahora verás cómo utilizar las funciones estudiadas. En el primer caso de estudio se usarán las funciones para escribir con formato usando fprintf( ). En el segundo caso, se leerá y escribirá información en bloques usando tipos struct con las funciones fread( ) y fwrite( ).

Caso de estudio 1: Ley de Ohm con archivo

Hacer un programa que calcule el voltaje en las terminales de una resistencia usando la ley de Ohm. El programa debe permitir guardar el resultado en un archivo llamado Voltaje.txt. El voltaje estará dado en volts, la corriente en amperes y la resistencia en ohms.

PROPUESTA DE SOLUCIÓN

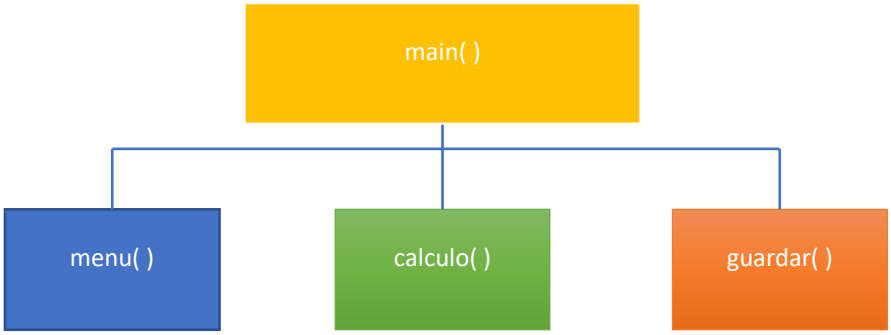
Se propone el uso de fprintf( ) para guardar los datos con un formato especificado en un archivo texto llamado voltaje.txt que se generará en la carpeta en la que se esté ejecutando el programa. El diseño modular del programa es el siguiente:

- ***menu ( ).*** Esta función preguntará al usuario si desea guardar la información a un archivo. Además, permitirá seguir haciendo cálculos o salir de la aplicación.

- ***calculo( )***. Aquí se hará el cálculo del voltaje, según la ley de Ohm.

- ***guardar( )***. Esta función abrirá el archivo usando el modo “a” para que no borre el contenido anterior cuando el usuario desee agregar más registros. Guardará los datos usados usando fprintf( ).

- ***main( )***. Desde esta función se llamará primero a calculo ( ) y después a menu() para saber la opción del usuario.



El código propuesto queda de la siguiente forma:

```
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
void calculo(void);
char menu(void);
void guardar(void);
int V,I,R;
void main()
{
char opc;
calculo();
do{
 opc=toupper(menu());
 switch(opc)
 {
      case 'G':
                  guardar();
```

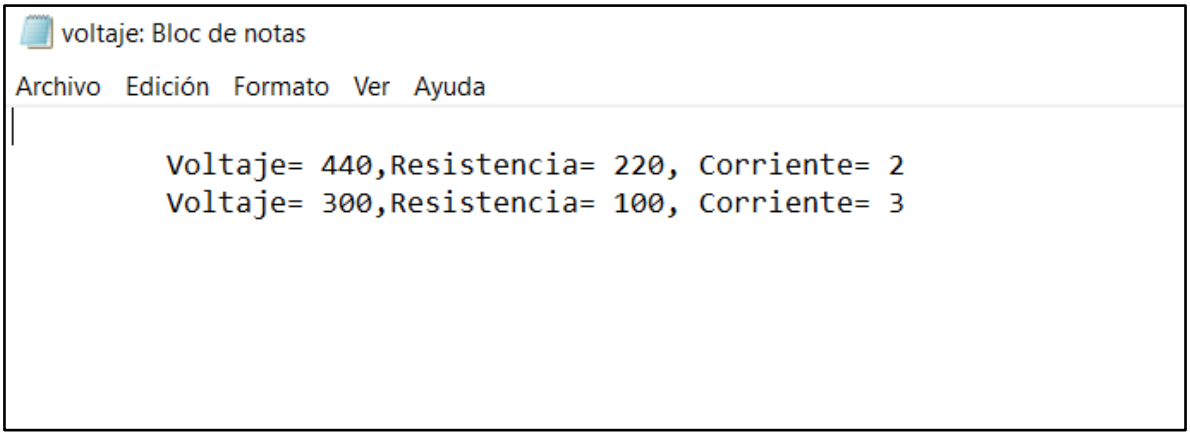
```
            break;
      case 'C':
                  calculo();
                  break;
      case 'S':
                  break;
      default:
           printf("\n\t\t Opcion no valida. Intente de nuevo:");
  }
}while (opc!='S');
}

void calculo(void)
{
printf ("\n\t =========APLICACION PARA CALCULAR VOLTAJE SEGUN LA LEY DE
      OHM==========");
printf("\n\t\t Valor de la Resistencia (en ohms): ");
scanf("%d",&R);
printf("\n\t\t Valor de la Corriente (en amperes): ");
scanf("%d",&I);
V=I*R;
printf("\n\t\t----------------------------------------");
printf("\n\t\t Valor del Voltaje (en volts): %d\n", V);
}

char menu(void)
{
fflush(stdin);
printf("\n\t\t (G) guardar los datos en archivo");
printf("\n\t\t (C) continuar calculando voltajes");
printf("\n\t\t (S) salir de la aplicacion");
printf("\n\t\t Teclee una opcion: ");
return(getchar());
}
void guardar(void)
{
 FILE* puntero;
 if ((puntero=fopen("voltaje.txt","a"))==NULL)
 {printf("\nError al abrir el archivo\n");
 exit(1);
 }
 fprintf(puntero,"\n\t Voltaje= %d,Resistencia= %d, Corriente= %d",V,R,I );
 fclose(puntero);
}
```

Observa el return de menu( ); se llama a la función getchar( ), el valor devuelto por ella se convierte en el parámetro de return( ).

La primera vez que se ejecuta el programa comienza haciendo el cálculo del voltaje y preguntando si se desean guardar los datos. Cuando le usuario opte por almacenarlos, se creará el archivo voltaje.txt y se guardará la cadena con formato usada en fprintf( ). El modo en que se abre el archivo es “a” para ir agregando los valores. Aquí te muestro una imagen de la pantalla de salida del programa.

```
=========APLICACION PARA CALCULAR VOLTAJE SEGUN LA LEY DE OHM===========
        Valor de la Resistencia (en ohms): 220

        Valor de la Corriente (en amperes): 2

       ---------------------------------------
        Valor del Voltaje (en volts): 440

        (G) guardar los datos en archivo
        (C) continuar calculando voltajes
        (S) salir de la aplicacion
        Teclee una opcion: g

        (G) guardar los datos en archivo
        (C) continuar calculando voltajes
        (S) salir de la aplicacion
        Teclee una opcion: c

=========APLICACION PARA CALCULAR VOLTAJE SEGUN LA LEY DE OHM===========
        Valor de la Resistencia (en ohms): 100

        Valor de la Corriente (en amperes): 3

       ---------------------------------------
        Valor del Voltaje (en volts): 300

        (G) guardar los datos en archivo
        (C) continuar calculando voltajes
        (S) salir de la aplicacion
        Teclee una opcion: g
```

Observa, con el explorador de archivos, que después de escoger la opción ‘g’ se crea el archivo voltaje.txt. Si abres el archivo con un editor de texto, comprobarás que el contenido del archivo son los datos de los cálculos.



Caso de estudio 2 : Control de inventario de un laboratorio de electrónica

Se desea tener un archivo llamado *inventario.txt* en donde se guarde el inventario de un laboratorio de electrónica. Hacer un programa que muestre al usuario un menú en donde pueda escoger: leer el archivo, escribir en él o terminar el programa.

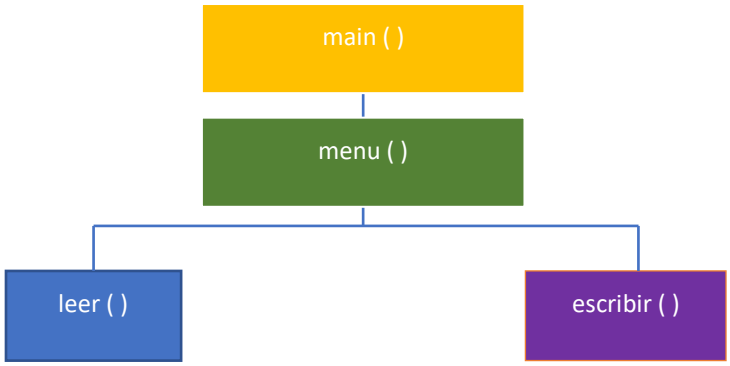
PROPUESTA DE SOLUCIÓN

Para llevar el inventario del laboratorio, se propone llevar los siguientes datos:
a) La descripción del equipo, y
b) La cantidad de ese equipo

Cuando se desea guardar este tipo de información, donde cada elemento tiene dos o más componentes, se utilizan los tipos struct, que son un *tipo de dato definido por el usuario*. Se dice definido por el usuario porque el programador establece lo que guardará cada variable de este tipo. Para este problema se propone el siguiente tipo de dato:

```
struct registro
{
        char nombre[20]; // aquí se guardará la descripción
        int cantidad;  //aquí se guardará la cantidad de este elemento
};
```

Se propone el siguiente diseño de programación modular:



- ***menu ( )***. Esta función mostrará el siguiente menú al usuario:
  1. Ver el archivo del inventario
  2. Agregar datos al archivo del inventario
  3. Salir de la aplicación

  Si el usuario escoge la primera opción, menú() llamará a la función leer( ) y si escoge la segunda, a escribir( ).

- ***leer( )***. Esta función abrirá el archivo inventario.txt. Si el archivo no existe, mandará un mensaje de error y terminará el programa.

- ***escribir( )***. Esta función abrirá el archivo usando el modo "a" para que no borre el contenido anterior cuando el usuario desee agregar más registros. Dejará de pedir datos cuando el usuario teclee un valor NULL, esto es, ctrl+z.

- ***main( )***. Desde esta función se llamará a menu().

El código propuesto queda de la siguiente forma:

```
#include <stdio.h>
#include <stdlib.h>
void escribir(void);
void leer(void);
void menu(void);
struct registro
{
      char nombre[20];
```

```
      int cantidad;
};
typedef struct registro equipo;

void main()
{
menu();
}

void menu(void)
{
      int opcion;
do{

      system("cls");
      printf("\n ====APLICACION PARA LLEVAR EL INVENTARIO DEL LABORATORIO DE
      ELECTRONICA===\n");
      printf("\n\t\t 1. Ver el archivo del inventario");
      printf("\n\t\t 2. Agregar datos al archivo del inventario");
      printf("\n\t\t 3. Salir de la aplicacion");
      printf("\n\t\t Escoge una opcion: ");
      scanf("%d",&opcion);
      switch(opcion)
      {
         case 1:
                           leer();
                           break;
         case 2:
                           fflush(stdin);
                           escribir();
                           break;
         case  3:
                           break;
         default:
                     printf("\n\t OPCION NO VALIDA\n");
      }
  }while (opcion!= 3);
}
void escribir(void)
{
equipo elemento;
FILE *fp;
char cantidad_temporal[4];
if((fp=fopen("inventario.txt","a"))==NULL)
```

```
 {
      printf("\n\t Problemas al abrir el archivo\a\n");
      exit(1);
 }
 printf("\n\t\t Nombre del esquipo: ");
 while(gets(elemento.nombre)!=NULL)
 {
      printf("\n\t\t Cantidad: ");
      gets(cantidad_temporal);
      elemento.cantidad=atoi(cantidad_temporal);
      fwrite(&elemento,sizeof(elemento),1,fp);
      printf("\n\t\t Nombre del esquipo: ");
 }
 fclose(fp);
}

void leer(void)
{
equipo elemento;
FILE *fp;
if((fp=fopen("inventario.txt","r"))==NULL)
 {
      printf("\n\t Problemas al abrir el archivo\a\n");
      exit(1);
 }
   printf("\n\t\t=====================================\n");
 printf("\n\t\t   EQUIPO   \t\t\tCANTIDAD\n");
 fread(&elemento,sizeof(elemento),1,fp);
 while(!feof(fp))
 {
      printf("\n\t\t %s", elemento.nombre);
    printf("\t\t\t %d\n", elemento.cantidad);
      fread(&elemento,sizeof(elemento),1,fp);
 }
  printf("\n\t\t=====================================\n");
  printf("\n\t\t FIN DE ARCHIVO\n");
  fclose(fp);
  system("pause");
}
```

Observa el funcionamiento de fflush(stdin)para limpiar el buffer de entrada antes de escribir en el archivo.

Referencias y bibliografía:

Ceballos, Francisco Javier (2015). Enciclopedia del Lenguaje C. Alfaomega.

Kernighan B., Ritchie D. (1978). El lenguaje de programación C. Prentice Hall.

García-Bermejo Giner, J. R. (2008). Programación estructurada en C. Pearson Educación.

Jiménez Castells, M. y Otero Calviño, B. (2015). Fundamentos de ordenadores: programación en C. Universitat Politècnica de Catalunya.

Menchaca García, F. R. (2010). Fundamentos de programación en Lenguaje C. Instituto Politécnico Nacional.

Joyanes Aguilar, L. (2005). C algoritmos, programación y estructuras de datos. McGraw-Hill España.

Gaxiola Pacheco, C. G. y Flores Gutiérrez, D. L. (2008). Metodología de la programación con pseudocódigo enfocado al lenguaje C. Plaza y Valdés, S.A. de C.V.

Moreno Pérez, J. C. (2015). Programación. RA-MA Editorial.

Shildt H (2000) C The complete reference. McGrawHill