2. INTRODUCCIÓN AL LENGUAJE

2.1. HISTORIA Y DESARROLLO DEL LENGUAJE

El lenguaje de programación C fue desarrollado por Dennis Ritchie en 1972 en los Laboratorios Bell mejorando el lenguaje B de Thompson, el cual, a su vez, deriva del lenguaje BCPL de Martín Richards. En un principio C sirvió para mejorar el Sistema Operativo UNIX por lo que se considera su lenguaje nativo. A partir de ahí y a raíz de su popularidad, empezaron a surgir muchas implementaciones del lenguaje C. Por ese motivo, se hizo necesario definir un estándar que está representado por el estándar ANSI.

El diseño de C incluye entonces una sintaxis simplificada y de alta portabilidad, la aritmética de direcciones de memoria permite al programador manipular bits, bytes y direcciones de memoria, por lo que se considera no sólo un lenguaje de alto nivel, sino también un lenguaje de nivel medio y el concepto de apuntador fue diseñado para generar códigos eficientes para producir una portabilidad total, es decir, que fuera posible adaptar el software escrito para un tipo de computadora o sistema operativo en otro. Logrado los objetivos anteriores C se convirtió en el lenguaje preferido por los programadores profesionales.

Una característica importante del lenguaje C, es que éste no lleva comprobaciones de errores en tiempos de ejecución, por lo tanto, el programador es el único responsable de llevar a cabo esas comprobaciones.

Se trata de un lenguaje estructurado, es decir, tiene la capacidad de seccionar y esconder del resto del programa toda la información y las instrucciones necesarias para llevar a cabo una determinada tarea. Esto permite modularizar las distintas partes del programa y reutilizar el código.

Características de C.

- El lenguaje C es un lenguaje de uso general, con una sintaxis corta y un juego de operadores potente
- C como ya se comentó es de nivel medio y explota por tanto los recursos de harware así como el manejo de las direcciones y los bits
- No posee operaciones de entrada-salida, ni métodos de archivos y tampoco maneja los objetos compuestos como las cadenas de caracteres. Estas operaciones se hacen por medio de funciones contenidas en librerías externas al lenguaje
- C es un lenguaje estructurado. La estructura básica de un programa en C se hace alrededor del concepto de función la cual puede tener sus propias variables locales, acceder a variables globales del programa y devolver valores o ninguno.
- C es un lenguaje que permite una compilación separada de los programas y enlazarlos entre si junto con las bibliotecas externas para formar el programa ejecutable completo.

- C esta definido bajo el estándar ANSI. Su creación y uso sigue las normas de este estándar y por tanto, todo lo expresado en él será utilizable con cualquier compilador que se adopte y cumpla con el estándar.

2.2. IDENTIFICADORES ESTANDAR

Los identificadores son nombres que se les da a varios elementos de un programa, tales como las variables, constantes, funciones, tipos y etiquetas. Un identificador se forma de letras y dígitos en secuencia, las letras pueden ser mayúsculas o minúsculas (una letra mayúscula es diferente a la misma letra, pero en minúsculas) y los dígitos van del 0 al 9. Además se puede utilizar el carácter especial de subrayado o guión bajo (_). Como regla, el primer carácter de un identificador debe ser una letra o el carácter de subrayado. Por ejemplo:

- `Suma`
- `suma`
- `Calculo_num_primos`
- `Abc123`
- `ab12C3`
- `_ordenar`
- `i`

Hay ciertas palabras reservadas del lenguaje denominadas palabras clave, que tienen en C un significado estándar y por tanto no pueden ser utilizadas como identificadores definidos por el programador.

2.3. PALABRAS RESERVADAS

Las palabras reservadas son entonces identificadores predefinidos que tienen un significado especial para el compilador de C. Un identificador definido por el usuario no puede tener el mismo nombre que una palabra reservada. LA siguiente es una lista de las palabras claves del ANSI-C:

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

Algunas versiones de compiladores para C pueden tener palabras reservadas adicionales. Es importante recalcar que el lenguaje C que se estudia se basa en el estándar del ANSI por lo tanto deberemos utilizar un compilador de C que esté basado en éste estándar independientemente del sistema operativo que se trabaje.

2.4. ESTRUCTURA DE UN PROGRAMA

Todo programa escrito en lenguaje C consta de una o más funciones donde una de ellas debe de llamarse *main()* y representa el programa principal de la aplicación, problema o algoritmo que se pretende resolver. El programa comienza su ejecución en la función *main()* desde la cual es posible hacer el llamado a otras funciones escritas fuera del programa principal.

Como todos los lenguajes, siempre es bueno comenzar con un ejemplo sencillo de un programa, en este caso de C, para entender su estructura. El siguiente programa es el primer programa típico, estándar que se utiliza para ejemplificar la estructura de un programa en C:

```
#include <stdio.h>    /* inclusión de un archivo o librería*/

void main(void)
   {                  /* inicia El cuerpo del prog. Principal */
     printf("Hola Mundo!\n");   /* imprime en pantalla */

    return;          /* regresa um valor al sistema */
   }
```

HolaMundo.c

Explicación:

`#include <stdio.h>`: #include es una directiva u orden para el compilador que le dice que tiene que leer un archivo de cabecera o header, en este caso con el nombre stdio.h, que es donde esta definida la función *printf()* que se utiliza en el ejemplo. Este header contiene todas las funciones de entrada y salida de C. Cuando un compilador se encuentra con esta directiva, lo sustituye por el archivo indicado.

El compilador tiene varias maneras de buscar un archivo de cabecera:

```
#include <stdio.h>          /*  Directorio de instalación */
#include "stdio.h"          /*  Directorio activo en el momento */
#include "c:\mi_header.h"   /*  Directorio indicado */
```

`void main(void):` Es La función o programa principal. Todos los programas de C, deben tener una función llamada *main()*, que será la primera que se ejecute al llamarse el programa. El primer *void* que tiene al principio significa que la función no va a regresar ningún valor cuando ésta termine su ejecución. La palabra reservada *void* significa "sin tipo" o "cualquier cosa". El segundo *void* significa que la función no requiere que se le proporcione ningún argumento para que pueda ejecutarse. Podría también haberse escrito como *void main()*.

`{ . . .}:` Toda función debe iniciar con una llave que abre y termina con una llave que cierra. Lo que esta dentro de las llaves se le llama el cuerpo de la función y puede incluir el llamado a funciones, declaraciones de variables, constantes, cálculos aritméticos, lógicos, etc.

`/* inclusión de un archivo o librería*/:` Esto es un comentario que no se ejecuta por el compilador. Sirve para describir el programa de forma que tenga más claridad a la hora de leer el código fuente. UN comentario puede ocupar más de una línea:

```
/* inclusión de un
archivo o librería*/
```

Un comentario entonces empieza con los caracteres /* y termina con */. Cualquier cosa que se escriba entre estos caracteres, el compilador lo interpretará como un comentario.

`printf("Hola Mundo!\n"); :`Esta es la instrucción que el compilador ejecuta para que el programa realice alguna acción. En este caso la función *printf()* muestra el mensaje "Hola Mundo" en la pantalla. Este mensaje es una cadena de caracteres que se escribe entre comillas dobles. Dentro de esta cadena al final aparece el símbolo ´\n´ que provoca que después de imprimir el mensaje, el cursor de la pantalla pase a la siguiente línea. Todas las instrucciones o sentencias de C deben terminar con un punto y coma, pues es la manera en que el compilador sabe cuando termina una instrucción y cuando empieza otra. Se pueden poner varias instrucciones en la misma línea siempre que vayan separadas or sus correspondientes puntos y comas.

`return; :`Es la instrucción que hace que las funciones regresen algún valor al sistema cuyo tipo depende del tipo declarado en la función. En el caso del ejemplo, como la función *main()* es una función sin tipo, es decir, no devuelve nada, entonces el uso del *return* sin regresar ningún valor, nos sirve para retornar al sistema.

2.5. TIPOS DE DATOS ESTANDAR (PRIMITIVOS)

Existen cinco tipos de datos atómicos en C:

char – carácter
int - entero
float – real de punto flotante
double – real de doble precisión
void – sin valor

Existen modificadores de tipos que se utilizan para alterar los tipos base (**short, long, unsigned, signed**). Estos modificadores se aplican a los tipos base **int** y **char**, no obstante **long** también se puede aplicar a **double**.

La siguiente tabla muestra todos los tipos de datos estandar en C y las combinaciones que se pueden dar con los modificadores de tipos:

tipo	bytes	v. mínimo	v. máximo		tipo	bytes	v.mínimo	v. máximo
char	1	-127	+127		long int	4	-2.147.483.648	+2.147.483.647
int	2	-32.767	+32.767		signed long	4	-2.147.483.648	+2.147.483.647
signed char	1	-127	+127		unsigned long	4	0	4.294.967.295
signed int	2	-32.768	+32.767		float	4	3.4e-38	3.4e+38
unsigned char	1	0	+255		double	8	1.7e-308	1.7e+308
unsigned int	2	0	65.535		long double	10	3.4e-4932	3.4e+4932
short int	2	-32.768	+32.767					

A reserva de trabajar las cadenas más adelante, definiremos una cadena como una secuencia de caracteres entre comillas dobles “”. Si las comillas tienen que aparecer como parte de una cadena, el carácter de las comillas debe ir precedido por el carácter \.

2.6. DECLARACIÓN DE CONSTANTES Y VARIABLES

2.6.1. VARIABLES

Una variable es un identificador o posición de memoria representada mediante un nombre que se usa para mantener un valor que puede ser modificado por el programa. Todas las variables en C deben ser declaradas antes de ser utilizadas y su sintaxis es la siguiente:

```
tipo nombre_variable [, nombre_variable, . . ., nombre_variable];
```

donde ***tipo*** determina el tipo de la variable: int, char, etc.
nombre_variable indica el nombre de la variable con las reglas para definir identificadores
Los corchetes **[ ]** indican que se puede definir en línea más de una variable del mismo tiposeparadas por **comas** y terminando con un **punto y coma**.

Por ejemplo:
int i,j,k;
float largo, ancho;
char c;

El inicio de un programa en C se ve de la siguiente manera:

```
main()
{
   declaración de variables;

   proposiciones;

  return;
}
```

Las llaves {…} encierran un bloque de proposiciones y se usan para enmarcar declaraciones y proposiciones. Toda declaración debe ir antes de las proposiciones. Las declaraciones tienen dos objetivos:

1. Piden al compilador que separe la cantidad de memoria necesaria para almacenar los valores asociados con las variables.
2. Debido a que se especifican los tipos de datos asociados con las variables, éstas permiten al compilador instruir a la máquina para que desempeñe correctamente ciertas operaciones.

Ámbito de las variables

Según el lugar donde se declaren las variables, tendrán un ámbito. Dentro de las funciones, es decir, *variables locales*, en la definición de los parámetros de las funciones o *parámetros formales* y fuera de todas las funciones llamadas *variables globales*.

Variables Locales: Son aquellas que se declaran dentro de una función. Las variables locales sólo pueden ser referenciadas (utilizadas) por sentencias o instrucciones que estén dentro de la función que han sido declaradas. No son conocidas fuera de la función. Pierden su valor cuando se sale y se entra en la función. Una variable local también es aquella que está siendo referenciada sólo por sentencias que estén dentro del bloque donde están declaradas. Un bloque es aquel que está encerrado entre dos llaves.

```
void func1(void)     //la variable x declarada en func1(…)
{                    //no es la misma que la declarada en func2(…)
 int x=10;
}

void func2(void)    //La variable "y" declarada en func2(…)
{                   //se crea solo al entrar en el fragmento
int x=-20;          //del programa y desaparece al salir de él
   {
    int y;
    y=2;
   }
}
```

Parámetros Formales: Son los argumentos de una función. La declaración de éstas variables se hace entre los paréntesis de una función. Su comportamiento es igual que las variables locales de cualquier función, es decir, desaparecen al terminar su ejecución.

```
Float mul(float x, float y)  // Los parámetros formales de
{                            // la function mul() son x e y
  Return (x*y);
}
```

Variables Globales: Son conocidas a lo largo de todo el programa, por lo tanto se pueden usar en cualquier parte del código. Mantienen sus valores durante toda la ejecución. Las variables globales se declaran por lo general al principio del programa, fuera de todas las funciones, incluida la función *main()*. Inicialmente toman el valor cero o nulo según el tipo.

```
int x;                 //la variable x es global, pues está
                       //declarada fuera de todas
void fun1(…)           // las funciones
{
```

```
y=x;                    // en la function fun1(…) se accede a la
x=10;                   //variable global x
}

void fun2(…)            // en la function fun2(…), x se refiere a
{                       // la variable local
 int x;
 x=3;
}
```

Especificadores de Almacenamiento:
Existen cuatro tipos de especificadores de almacenamiento: *extern, static, register*. Estos especificadores indican al compilador cómo debe almacenar las variables. Su sintaxis es:

```
especificador tipo nombre_variable [, nombre_variable, . . ., nombre_variable];
```

extern: Dado que en C es normal la compilación por separado de módulos que componen un programa completo, guardados cada uno en su correspondiente archivo, puede darse el caso de que se deba utilizar una variable global que se conoczca en los módulos que nos interese sin perder su valor. La palabra extern indica al compilador que las variables que siguen ya han sido declaradas en alguna otra parte y por tanto, no se crea una nueva variable.

```
int x,y;
char c;
. . .
fun1()
{
 x=10;
}
```

Archivo 1

```
extern int x,y;
extern char c;    // Las variables x,y,c declaradas en este
. . .             // archivo2 son exactamente las declaradas en
fun11()           // el archivo1
{
 x=y;
}
```

Archivo 2

static: Si se quiere mantener el valor de una variable local entre una llamada y otra sin perder por ello su ámbito, ésta tendrá que ser declarada como *static*. Cuando se aplica *static* a una variable local, su valor se retiene entre llamadas de funciones y se crea un almacenamiento permanente igual que las variables globales (aunque sigue desconocida

fuera de su ámbito). Cuando se aplica *static* a una variable global, se indica al compilador que cree una variable conocida únicamente en el archivo. Ninguna función de otros archivos puede tener acceso a ella.

```
serie(void)      // Cada llamada a la función serie1(…) produce
{                // un Nuevo valor n, basándose en el número
 static int n;   // anterior
 n=n+23;
}

serie2(void)     // en la segunda función serie2(…), el valor de
{                // n se inicializa a 10, pero no cada vez que se
static int=10;   // llame a la function serie2(…), como es el caso
n=n+23;          // de las variables locales
}
```

register: En vez de que una variable sea mantenida dentro de una posición de memoria de la computadora, se guarda en registros internos del microprocesador. De esta manera el acceso a ellas es más rápido y directo. Se aplica únicamente para las variables locales y nunca para las variables globales. Se suele usar para almacenar los contadores de los ciclos (for, while, repeat).

2.6.2. CONSTANTES

Una constante es un valor fijo que no puede ser modificado por el programa. Puede ser de cualquier tipo de datos básicos. Las constantes de carácter van encerradas entre comillas simples, Las constantes enteras se especifican con números sin parte decimal y las de punto flotante con su parte entera separada por un punto de su parte decimal. Las cadenas de caracteres irán encerradas entre comillas dobles. Por ejemplo:

Flotantes	Enteros	Cadenas	Caracter
3.10	10	“Hola Mundo”	‘a’
0.987	-1234	“”	‘#’

Una constante en C se define de la siguiente manera:

```
#define   identificador   valor
```

Donde ***#define*** es la directiva de C que se utiliza para definir constantes.
identificador es el nombre de la constante y
valor es el valor asociado a la constante.

Ejemplos:

```
#define     entero      10
#define     real        1.09982
#define     cad         "Esto es una cadena"
#define     car         'a'
```

2.7. EXPRESIONES, PROPOSICIONES Y ASIGNACIONES

Las expresiones son combinaciones de contantes, variables, operadores y llamadas a funciones. Algunos ejemplos son:

```
tan(1.8)
a+b*3.0*9.3242
3.77+sen(3.14*98.7)
```

Un **operador** es un símbolo que indica al compilador que se lleven a cabo específicas manipulaciones matemáticas o lógicas. En C se tienen tres clases de operadores: aritméticos, relaciones y lógicos incluyendo a los de bits.

El signo de = es el operador básico de asignación en C. Cuando una expresión va seguida de un punto y coma entonces ésta se convierte en una proposición. Las proposiciones de asignación tienen la siguiente sintaxis:

```
variable = expresión;
```

Por ejemplo:

```
x=Tan(1.8);
r=a+b*3.0*9.3242;
3.77+sen(3.14*98.7);
```

Operadores Aritméticos Binarios:

Operador	Operación
+	Suma. Los operandos pueden ser enteros o reales
-	Resta. Los operandos pueden ser enteros o reales
*	Multiplicación. Los operandos pueden ser enteros o reales
/	División. Los operandos pueden ser enteros o reales. Si ambos operandos son enteros el resultado es entero. En el resto de los casos el resultado es real.
%	Módulo o resto de la división entera. Los operandos tienen que ser enteros.
-(unario)	Menos unario. Los operadores pueden ser enteros o reales.

Prioridad de los Operadores Aritméticos:

Operadores	Asociatividad
-(unuario)	derecha a izquierda
*, /, %	izquierda a derecha
+, -	izquierda a derecha
=	derecha a izquierda

Operadores Relacionales:

Operador	Operación	Ejemplos
<	Primer operando menor que el segundo	a<3
>	Primer operando mayor que el segundo	b>w
<=	Primer operando menor o igual que el segundo	-7.7<=-99.335
>=	Primer operando mayor o igual que el segundo	-1.3>=(2.0*x+3.3)
==	Primer operando igual que el segundo	c= ='w'
!=	Primer operando distinto del segundo	x!=-2.77

Operadores Lógicos:

Operador	Operación	Ejemplo
&&	AND. Da como resultado el valor lógico 1 si ambos operandos son distintos de 0. Si uno de ellos es cero el resultado es el valor lógico 0. Si el primer operando es igual a cero, el segundo operando no es evaluado.	(z<x) && (y>w)
||	OR. El resultado es cero si ambos operandos son 0. Si uno de los operandos tiene un valor distinto de 0, el resultado es 1. Si el primer operando es distinto de 0, el segundo operando no es evaluado.	(x= =y) || (z!=p)
!	NOT. El resultado es 0 si el operando tiene un valor distinto de cero, y 1 en caso contrario. El resultado es de tipo int. El operando puede ser entero, real o un apuntador.	!a

Operadores de manejo de Bits

Los operadores para este tipo de operaciones tienen que ser de tipo entero de uno o dos bytes o char, no pueden ser reales.

Operador	Operación
~	Complemento a 1. El operando tiene que ser entero
&	AND a nivel de bits
|	OR a nivel de bits
^	XOR a nivel de bits
<<	Corrimiento (desplazamiento) a la izquierda
>>	Corrimiento (desplazamiento) a la derecha

Operadores de Asignación:

Operador	Operación
++	Incremento
--	Decremento
=	Asignación simple
+=	Suma más asignación
-=	Resta más asignación
|=	Operación OR sobre bits más asignación
&=	Operación AND sobre bits más asignación
>>=	Corrimientos a la derecha más asignación
<<=	Corrimientos a la izquierda más asignación
*=	Multiplicación más asignación
/=	División más asignación
%=	Módulo más asignación

Tabla de prioridades de los operadores:

Operadores	Asociatividad
( )	Izquierda a derecha
-(unario), ++, --, ~, !, *, &(los últimos dos como operadores apuntadores)	Derecha a izquierda
*,/,%	Izquierda a derecha
+,-	Izquierda a derecha
<,<=,>,>=	Izquierda a derecha
==, !=	Izquierda a derecha
&	Izquierda a derecha
^	Izquierda a derecha
|	Izquierda a derecha
&&	Izquierda a derecha
||	Izquierda a derecha
=, +=, -=, *=, <<=, >>=, %=	Derecha a izquierda
, (operador coma)	Izquierda a derecha

Otros operadores:

- El operador de **indirección (*)** accede a un valor indirectamente a través de un apuntador. El resultado es el valor direccionado por el operando.

- El operador de **dirección (&)** indica la dirección de su operando. Este operador no se puede aplicar a un campo de bits perteneciente a una estructura o a un identificador declarado con el especificador **register**.

2.8. ENTRADA Y SALIDA BÁSICA

Las operaciones de entrada y de salida no forman parte del conjunto de sentencias de C, sino que pertenecen al conjunto de funciones de la librería estándar de entrada y de salida de C, es decir, están definidas en el archivo ***stdio.h***. Cuando se emplea alguna función de entrada y salida, se debe incluir éste archivo al comienzo del programa mediante la directiva: **#include <stdio.h>** y el compilador inserta el código fuente al comienzo del archivo de nuestro programa.
Las siguientes funciones son algunas de las más utilizadas para la entrada y salida de datos en C: **printf, scanf, getch, getchar, puts, gets**. Todas y cada una de ellas tiene una sintaxis que las identifica.

2.8.1. Salida de datos con formato (uso de printf)

Para visualizar los datos por la pantalla se dispone de la función *printf(…)* que permite formatear la salida. Esta función puede visualizar una combinación de valores numéricos, caracteres y cadenas de caracteres. Su sintaxis es la siguiente:

```
printf(cadena_de_control, arg1, arg2, . . ., argn);
```

La ***cadena de control*** especifica cómo va a ser la salida. Es una cadena delimitada por comillas, por caracteres ordinarios, secuencias de escape y especificaciones de formato bajo el cual se requiere la salida de la información hacia pantalla.

La lista de argumentos **arg1, arg2, . . . argn** representa el valor o valores a escribir en la pantalla. Una especificación de formato está compuesta por:

```
%[flags][width][.prec][F|N|h|l|L] tipo_de_dato
```

Cada uno de los datos que se desee mandar a imprimir debe ir antecedido por el carácter % y después debe de venir (en ese orden) lo siguiente (no es necesario poner todo, lo que se encuentra entre corchetes es opcional):

COMPONENTE	**ESPECIFICACIÓN**
flags	Justificación, etc.
width	Número de dígitos significativos parte entera
.prec	Número de dígitos significativos parte real
F|N|h|l|L	Modificadores de Salida N= Near apuntador F= far Apuntador h= entero corto l= entero largo L= real largo
Tipo_de_dato	c= imprime un carácter d= imprime un entero e= notación científica s= imprime una cadena f= decimal en punto flotante

Ejemplo:

```
printf(“hola Puebla son las %d\n”, tiempo);
```

Otro ejemplo:

```
n1=5;
n2=6.7;
printf(“El dato 1 es: %d y el dato 2 es %f\n”, n1,n2);
```

Secuencia	Nombre
\ n	Nueva línea
\ t	Tab horizontal
\ v	Tab vertical (solo para impresora)
\ b	Backspace (retroceso)
\ r	Retorno de carro
\ f	Alineación de página (solo para impresora)
\ a	Bell (sonido)
\ ´	Comilla simple
\ ”	Comilla doble
\ 0	Nulo
\ \	Backslash (barra hacia atrás)
\ ddd	Carácter ASCII. Representación Octal
\ xdd	Carácter ASCII. Representación hexadecimal

Secuencias de escape en lenguaje C

```
char ch = 'A';
int m = 4;
float x = 14.250;

printf("ch=%c, m=%d", ch, m);
printf("ch=%d, x=%f", ch, x);
printf("ch=%x,\n m=%d ", ch, m);

printf("x=%4.1f, m=%05d", x, m);
printf("La suma x+m=%f",(x+m));
printf("La tecla es %c\n",getch());
```

```
El programa produce
las siguientes salidas.


ch = A , m= 4
ch = 65, real = 14.250
ch = 41,
m = 4   (N.B salto de linea).
x = 14.2, m = 00004
Argumento como expresión(x+m)
El argumento es una función.
```

2.8.2. Entrada de datos con formato (uso de scanf)

La función *scanf(…)* lee los datos desde el teclado con formato. Su sintaxis es:

```
scanf(cadena_de_control, arg1, arg2, . . ., argn);
```

La **cadena de control** contiene información sobre el formato de las entradas y son idénticos a los formatos vistos en la función *printf(…)*. Es decir, se forma por códigos de formato de entrada, que están precedidos por un signo de % y encerrados entre comillas.

Código	Significado
c	Lee un único carácter
d	Lee un entero decimal base 10
i	Lee un entero en base 10, 16 u 8
f	Lee un número en punto flotante
e	Lee un número en punto flotante
h	Lee un número entero corto
s	Lee una cadena de caracteres
o	Lee un entero en octal
x	Lee un número hexadecimal

Códigos de formato de scanf(…)

Los argumentos ***arg1, arg2, arg3,. . ., argn*** son direcciones de las variables y para ellos se debe utilizar el operador de direcciones (&). La lista de valores por tanto representa el valor o valores a escribir en pantalla. Una especificación de formato para scanf() esta compuesta por:

Símbolo	Significado
*	Un asterisco a continuación de % suprime la asignación del siguiente dato en la entrada.
Ancho	Máximo numero de caracteres a leer de la entrada. Los caracteres en exceso no son tenidos en cuenta.
f	Indica que se quiere leer un valor apuntado por una dirección far(dirección segmentada).
N	Indica que se quiere leer un valor apuntado por una dirección near (dirección dada por el valor offset). F y N no pertenecen al C estándar.
H	Se utiliza como prefijo con los tipos d, i, n, o y x, para especificar en el argumento es short int, o con u para especificar un short unsigned int.
L	Se utiliza como prefijo con los tipos d,i,n,o, y x, para especificar que el argumento es long int. También se utiliza con los tipos e,f y g para especificar un double.

Tipo	El tipo determina si el dato de entrada es interpretado como un carácter, como una cadena de caracteres o como un número.

Ejemplo:

```
printf(“Da un numero: “);
scanf(“%d”,&n);
```

Cuando se especifica más de un argumento, los valores correspondientes en la entrada hay que separarlos por uno o más espacios en blanco, tabuladores y cambios de línea. La función *scanf()* devuelve un entero correspondiente al número de datos leídos de la entrada:

Otro ejemplo:

```
main()
{
int a,r;
float b;
char c;
printf(“Introducir un valor entero, un real y un caracter\n =>”);
r=scanf(“%d%f%c\n”,&a,&b,&c);
printf(“Numero de datos leidos: %d\n”,r);
Printf(“datos leidos: %d%f%c\n”,a,b,c);
}
```

`Int n;`
`Float x;`
`Char cadena[20];`
`........`
`scanf("%f %d",&x, &n);`
	Dos grupos; El primero es float (%f) y el segundo es un entero (%d). Las variables se almacenan en las direcciones(&x y &n).

`scanf("%s ", cadena);`	La función lee una cadena de caracteres. El argumento no lleva el operador &.

```
   scanf("%s %d %f", cadena, &n, &x);

Los datos se pueden introducir de las siguientes maneras.
```

Mensaje 120 0.25	Mensaje 120 0.25	Mensaje 120 0.25

Para leer datos de tipo cadena no lleva el operador de dirección &. Y tenemos que cambiar el formato %s por %[^\n]. Por ejemplo:

```
char nombre[40];
```

```
scanf(“%[^\n]”, nombre);
printf(“%s”, nombre);
```

Entrada: Francisco Javier
Salida: Francisco Javier

Si en lugar de especificar el formato %[^\n] se hubiera especificado el formato %s, el resultado hubiera sido: Francisco.

`Char *cad;`	
`Scanf("%s ", cad);`	No se puede leer un espacio en blanco.
`Scanf("%[ ABCD]",cad);`	Lee solo los espacios en blanco y las letras especificadas(,A,B,C,D).
`Scanf("%[^\nABCD]",cad);`	Lee todos los caracteres salvo (\n) y los caracteres A,B,C,D)
`Scanf("[^\n]",cad);`	Lee todos los caracteres salvo (\n).

2.8.3. Entrada y Salida sin formato

Para leer un carácter desde el teclado o imprimirlo en pantalla, se emplean las siguientes funciones:

Funciones	DESCRIPCIÓN
var_char=getchar();	Lee un carácter de teclado, espera un salto de carro.
var_char=getche();	Lee un carácter con eco, no espera salto de carro.
var_char=getch();	Lee un carácter sin eco, no espera salto de carro.
gets(var_cadena);	Lee una cadena del teclado.
putchar(var_char);	Muestra un carácter en pantalla.
puts(variables);	Muestra una cadena en pantalla.

En la tabla anterior se encuentran también las instrucciones para leer o escribir una cadena de caracteres. Estas son: *gets(…) y puts(…)* respectivamente. A continuación unos ejemplos.

```
char ch;
ch = getchar();     Lee un carácter y lo asigna a ch.
putchar(ch);        Escribe en pantalla el contenido de ch.
puchar('B');        Escribe el carácter B en pantalla.
puchar( 98);        Igual que antes,98 es el código ASCII de B
```

2.9. ESTRUCTURAS DE CONTROL

Cuando se resuelven problemas generando programas de computadora, en este caso en lenguaje C, lo más probable es que necesitemos de algún tipo de elementos de control lógico o comprobaciones de condiciones que sean ciertas o falsas (selección o decisión). Además los programas pueden requerir que un grupo de instrucciones se ejecute repetidamente un determinado número de veces, o hasta que se satisfaga una condición lógica (ciclo de repetición).
En el lenguaje C se incluyen entonces la instrucciones de decisión *if y switch* y las sentencias de repetición, for, while, do/while.

2.9.1. La Proposición if

La sintaxis de la proposición if es la siguiente:

```
if (expresión)
   {
    proposiciones;  /* bloque */
   }
proposición siguiente;
```

Donde *expresión* debe ser una expresión numérica, relacional o lógica.
Si se tiene una sola proposición, no es necesario introducir las llaves. Si tenemos varias proposiciones a realizar, estas irán separadas con sus respectivos puntos y comas.
Si el resultado de la expresión es verdadero (cualquier valor diferente de cero), se ejecutará la proposición o proposiciones. Si el resultado es falso entonces no se realiza ninguna acción y continua el flujo del programa a la siguiente línea de código después del if (proposición_siguiente).

```
if(a>b)    if((a>b)!=0)   1.   Las dos expresiones son idénticas.

if(a)      if(a!=0)       2.   Las dos expresiones son idénticas.

if(!a)     if(a==0)       3.   Obsérvese que (!a) dará un valor
                          verdadero sólo cuando a sea falso.
if(a==b)   if(a=b)        4.   En la primera se hace una comparación
                          entre a y b; en la segunda se b a a.
                          Ambas son correctas pero distintas.
if(a)      if a           5. Muy simplificada, poca legibilidad.
```

Por ejemplo:

```
if (grado>=90)
    printf("\n FELICIDADES");
printf("\n su grado es %d", grado);
```

La Proposición else

El *else* es optativo y su aplicación resulta en la ejecución de una o una serie de sentencias (bloque) en el caso de que el resultado de la expresión del *if* sea *falso*. La sintaxis de la proposición if-else es la siguiente:

```
if (expresión)
   {
    proposición_1; /* bloque 1 */
   }
else {
      proposición_2; /*bloque 2*/
      }
proposición_siguiente;
```

Por ejemplo:

```
if (x<=y)
      min=x;
else
       min=y;
```

If(expresión)
{
Sentencia 1 ;
Sentencia 2 ;
}
sentencia 3 ;
sentencia 4 ;
sentencia 5 ;
	if(expresión)
	{
	sentencia 1 ;
	sentencia 2 ;
	}
	else
	{
	sentencia 3 ;
	sentencia 4 ;
	}
	sentencia 5 ;
		1. En primer ejemplo, no se usa el else y por lo tanto las sentencias 3 , 4 y 5 se ejecutarán siempre.
		2. En el segundo, las sentencias 1 y 2 se ejecutan solo si la expresión es cierta y nc se ejecutarán la 3 y la 4 para saltar directamente a la 5.
		En caso de que la expresión resulte falsa se realizarán la 3 y la 4 en lugar de la 1 y la 2 y luego la 5.

Las sentencias if-else pueden estar anidadas, es decir, como proposición_1 o proposición_2 puede escribirse otra sentencia if.

```
if(exp1)
  sentencia 1;
else if(exp2)
  sentencia 2;
else if(exp3)
  sentencia 3;
else
  sentencia 5;
```

```
if(exp1)
  sentencia 1;
else if(exp2)
      sentencia 2;
     else if(exp3)
          sentencia 3;
          else
          sentencia 5;
```

```
Se suele escribir según la
modalidad de la izquierda.

A la derecha se han expresado
las  asociaciones entre los
distintos else-if para mayor
legibilidad.
```

Por ejemplo:

```
if (a>b)
    printf("%d es mayor que %d",a,b);
else if (a<b)
             printf("%d es menor que %d",a,b);
        else printf("%d es igual que %d",a,b);
```

En las sentencias de selección en las que hay sólo un *else*, se pueden reescribir mediante el operador condicional (*? :*) cuya sintaxis es la siguiente:

```
(condición) ? (expresión1) : (expresión2) ;
```

Las expresiones deben ser simples y no deben formar grupos de instrucciones. Primero se evalúa la *condición* y si ésta es verdadera se ejecuta la *expresión1*, sino se ejecuta la *expresión2*.

```
if(x<1)
 puts("No toques nada...");
else
 puts("Continúe,joven...");
```

```
El código se puede reescribir como:

(x<1)? puts("No toques nada..."):
       puts("Continúe,joven...");
```

2.9.2. La Proposición switch

Ésta proposición permite ejecutar una de varias acciones, en función del valor de una expresión. La sintaxis de esta proposición es:

```
switch (expresión-test)
   {
    case constante1: sentencia;
    case constante2: sentencia;
    case constante3: sentencia;
     . . .
    case constanten: sentencia;
    default: sentencia;
   }
```

Expresión-test es una constante entera, una constante de caracteres, o una expresión constante. El valor es convertido a tipo *int*. *Sentencia* es una sentencia simple o compuesta.
El switch empieza con la evaluación de la expresión, y luego se compara sucesivamente con todas las etiquetas. Cuando se iguala con alguna de ellas se ejecutan las sentencias correspondientes. Se suele utilizar la palabra reservada *break* dentro de las etiquetas (*case:*) la cual provoca que termine la ejecución del switch después de ejecutar las sentencias de alguna etiqueta. Al final aparece una etiqueta optativa llamada *default:* que quiere decir que, si no se ha cumplido ningún *case* se ejecute lo que sigue.

```
switch(ch = getchar())
{
 case 'a': printf("Azul");
           break;
 case 'b': printf("Blanco");
           break;
 case 'r': printf("Rojo");
           break;
 default : printf("Error");
}
```

```
      Selección de colores.

Cada case termina con un break, con lo
cual se selecciona el color y el
control sale fuera del switch.
```

```
.............
case 'a':
case 'A': printf("Azul");
           break;
case 'b':
case 'B': printf("Blanco");
           break;
case 'r':
case 'R': printf("Rojo");
           break;
.........
Los case 'a' y 'A', tiene la
misma sentencia.
```

```
...........
case 'a': printf("Azul");
case 'b': printf("Blanco");
case 'r': printf("Rojo");
...........


Si en la comparación se elige el case
'a', se ejecutarán todas las
sentencias que hay a continuación
hasta el siguiente break o la opción
default.
```

2.9.3. La Proposición de iteración for

Cuando se desea ejecutar una acción simple o un bloque de instrucciones, repetidamente un número de veces conocido, es recomendable utilizar la proposición *for*, cuya sintaxis es la siguiente:

```
for(inicialización; condición; incremento)
   {
     sentencia1;
     sentencia2;
     . . . . .
     sentencia_n;
   }
```

donde:

inicialización es una proposición de asignación para establecer el valor inicial de la variable de control. Se pueden establecer varias variables de control en un ciclo for.

Condición es una expresión relacional o lógica que determina cuando terminará el ciclo de repetición.

Incremento define cómo cambiará la variable de control o las variables de control, cada vez que cambia este.

`for(digito=0; digito<=9; ++digito)` `  printf("%d \n", digito);`	Programa que visualiza los dígitos del 0 al 9. Ver ejemplo (while).

`for( ; ; )  printf("Esto no termina");`	1. bucle infinito.

`for( ;cañas <=20; )`
`  { cañas += 1;`
`  borachera *=2;`
`  }`
	2. Sin índice ni su actualización, pero se actualiza dentro del cuerpo del bucle.

`for(cont=0, total=0; cont<10; ++cont)`
`{total += cont;`
`printf("cont=%d,total=%d\n",cont,total);`
`}`
	3. Dos contadores en el índice.

2.9.4. La Proposición de Iteración while

Esta proposición ejecuta una sentencia simple o compuesta, cero o más veces dependiendo del valor de una expresión. Su sintaxis es la siguiente:

```
while (expresión)
   {
    sentencia1;
    sentencia2;
    . . . . .
    sentencia_n;
   }
```

Una proposición while se ejecuta mientras la expresión sea verdadera (cualquier valor distinto de cero). En el momento en que la expresión sea falsa, entonces se ejecuta la siguiente línea después del fin del ciclo. Por ejemplo:

```
char ca;
ca=0;
while(ca!='A')
   {
     ca=getchar();
     putchar(ca);
   }
```

```
int digito = 0;
while(digito <= 9){
   printf("%d \n", digito);
   ++digito;
}
Programa que visualiza los
Dígitos del 0 al 9.
```

```
int digito = 0;
while(digito <= 9)
   printf("%d \n", digito++);


Una variante más concisa que la primera.
```

2.9.5. La Proposición do-while

Esta proposición ejecuta una sentencia o varias, una o más veces dependiendo de l valor de una expresión. Su sintaxis es:

```
do {
        sentencia1;
        sentencia2;
        . . . . .
        sentencia_n;
       } while (expresión);
```

En la sentencia *do-while* se ejecuta primero la sentencia o bloque de sentencias que estan dentro del *do*. Después se evalúa la *expresión* y si ésta es falsa termina la proposición *do-while*, sino se repote la sentencia o sentencias que están dentro del *do*. Por ejemplo:

```
int n;
do {
        printf("\n da un número: ");
        scanf("%d", &n);
      } while (n>100);
```

2.9.6. Las sentencias break, continue y la función exit()

La sentencia break: Esta sentencia ya descrita en el uso del switch, también sirve para terminar ciclos producidos por el for, while y do-while, antes de que se cumpla la condición normal de terminación. Por ejemplo:

```
for (t=0; t<100;t++)
   {
    contador 1;
    do{
          Printf( “%d”, contador);
    contador++;
    if contador(==10)
          break;
         }while (1);
   }
```

```
char c ;
printf("Este es bucle indefinido");
 while(1){
   printf( " Apriete una tecla:");
    if( (c = getch()) == 'S')
     break ;
   printf("\n No fue la correcta");
 }
 printf("\n Tecla correcta.");
```

Obsérvese que la expresión while(1) siempre es cierta con lo que el se ejecutará hasta que el operador oprima la tecla S, permitiendo la ejecución de la instrucción break dando por finalizado el bucle while.

La sentencia continue: Com esta sentencia, en vez de terminar um ciclo, termina con La realización de La iteración en curso saltando El resto de la pasada, ES decir, todas las sentencias que se encuentran después de *continue*, para ir a la siguiente iteración. Igual que *break*, se puede utilizar en los ciclos for, while y do-while.

```
char c ;
for(;;){
   c = getch(); /*lee sin eco */
   if(c=='a' || c=='e' || c=='i'||
      c=='o' || c=='u')
     continue;
   putchar(c);
   putchar(c);
 }
```

En este programa, se leen los caracteres que se pulsen. Si se pulsa una vocal minúscula, se realiza un salto al comienzo del bucle, en caso contrario el carácter se escribe dos veces en pantalla.

Por ejemplo:

```
do{
       printf(“Da un numero: “);
       scanf(“%d”,&x);
       if (x<0) continue;
       printf(“El numero es %d”,x);
     }while (x!=100);
```

La función exit(): Otra forma de terminar un ciclo de repetición desde dentro es utilizar la función exit(). A diferencia con el *break,* la función *exit()* terminará con la ejecución del programa y regresará el control al sistema operativo.

Por ejemplo:

```
for(i=0; i<=1000; i++)
   {
    printf("El valor de i es %d", i);
    if (i>10) exit();
   }
```