Notas

Programación de Computadoras

José Ortiz Bejar
Mauricio Reyes

Notas de clase

Programación de Computadoras

José Ortiz Bejar
Edificio Ω II
Facultad de Ingeniería Eléctrica

2021

Table of Contents

1 Introducción **2**
1.1 Lenguaje de programación 2
1.2 Lenguaje C 2
1.2.1 Estructura de un programa en C 3
1.2.2 Compilación 3
1.2.3 Compilador 4
1.3 Comentarios 5

2 Uso de variables y asignación **5**
2.1 Tipos de datos 6
2.1.1 Enteros 6
2.1.2 Caracteres 6
2.1.3 Punto flotante 6

3 Entrada y salida (básico) **7**

4 Operadores **9**
4.1 Aritméticos 9
4.2 Lógicos 10
4.2.1 Evaluación de corto circuito (short-circuit) 10
4.3 Operadores de comparación 10
4.4 Librería **math.h** 11
4.5 Tiro parabólico 11

5 Declaraciones condicionales **13**
5.1 El condicional *if-else* 13
5.2 Condicional *switch/case* 14

6 Instrucciones de repetición (Ciclos) **17**
6.1 Instrucción mientras *while* 17
6.2 Instrucción *do-while* 18
6.3 Instrucción *for* 18
6.4 Ciclos anidados 18
6.5 *break* y *continue* 19

7 Funciones **21**
7.1 Ejecutar una función 21
7.2 Parámetros 22
7.3 Recursión 22

8 Arreglos **24**
8.1 Inicialización de arreglos 25
8.2 Acceder a los elementos de un arreglo 26
8.3 Arreglos multidimensionales 27
8.4 Cadenas 28

9 Entrada salida de cadenas/caracteres **29**
9.1 getchar 29
9.2 gets 30

10 Apuntadores **31**
10.1 Parámetros por referencia 32
10.2 Arreglos y apuntadores 33
10.2.1 Apuntador tipo *void* 34
10.3 Ordenamiento de Burbuja (BubbleSort) 35
10.3.1 conceptos preliminares 35
10.4 Matrices y apuntadores 38

11 Memoria Dinámica **39**
11.1 Arreglos y memoria dinámica 39
11.2 La función *free* 40

12 Estrcuturas **41**
12.1 Como trabajar con Estructuras 41
12.1.1 Inicialización 43
12.1.2 Acceso a los miembros 43
12.2 Utilizar estructuras con funciones 43
12.3 `Typedef` 44

13 Archivos **47**

14 Modos se acceso **48**
14.0.1 Escritura de texto en archivos 49
14.0.2 Lectura de texto desde archivos 49
14.1 Ejemplo lectura/escritura archivos 49
14.1.1 Reubicación del apuntador 51
14.2 Lectura y escritura de datos en binario 51

List of Tables

Table 1 Especificadores de formato 7
Table 2 Operadores Aritméticos 9
Table 3 Operadores lógicos 10
Table 4 Operadores de comparación 11
Table 5 Funciones **math.h** 12
Table 6 Operaciones con punteros 34
Table 7 Funciones **C** para manejo de archivos 47
Table 8 Modos de acceso a archivos 48

List of Figures

Fig. 1 Compilación y ejecución Visual Studio Code 4
Fig. 2 Evaluación de corto circuito 11
Fig. 3 Ejemplo de una tarea que requiere el uso de un ciclo anidado 19
Fig. 4 Árbol de recursión de la función factorial 24
Fig. 5 Representación de un arreglo de tamaño n 25
Fig. 6 Arreglo de dos dimensiones 28
Fig. 7 Arreglo/apuntador en memoria 35
Fig. 8 Ejemplo ordenamiento por columnas 38

Glossary

Delete if not applicable

1. Introducción

En este curso se pretende dar una introducción a la programación en lenguaje **C**. La programación es el proceso de escribir un código de computadora, el cual le permitirá a la computadora ser capaz de resolver un problema específico. Los programas se pueden representar como pseudo-código o mediante un diagrama de flujo, y la programación es la traducción de estos a un programa de computadora.

En general, para *decirle* a una computadora que realice una tareas, se debe escribir un programa que le diga exactamente qué hacer y cómo hacerlo.

1.1 Lenguaje de programación

Un lenguaje de programación es un lenguaje artificial el cual puede ser interpretado por una computadora. El lenguaje se compone de una serie de declaraciones que en conjunto forman instrucciones o procedimientos. Estas instrucciones son las que le indican a una computadora cómo realizar la tarea a resolver.

Hay muchos lenguajes de programación diferentes, algunos más complicados y complejos que otros. Entre los lenguajes más populares se encuentran:

- JavaScript
- Python
- Java
- C#
- C/C++

Diferentes lenguajes tienen diferentes características en cuanto complejidad, eficiencia, etc. Si embargo, en este curso nos enfocaremos la programación en lenguaje **C**

1.2 Lenguaje C

C es un lenguaje de programación de alto nivel y de propósito general, el cual fue desarrollado originalmente por **Dennis Ritchie**. **C** es una de los lenguajes de programación más ampliamente utilizados, no solo debido a que el sistema operativo Unix y prácticamente todas las aplicaciones Unix están escritas en lenguaje C. Además tiene muchas otras características como que: es fácil de aprender, estructurado, eficiente (casi tan rápido como el código escrito en lenguaje ensamblador), puede realizar tareas de alto nivel y se puede compilar y ejecutar en prácticamente todas las arquitecturas de computadoras. Una amplia variedad e aplicaciones y herramientas están escritas en C, por ejemplo:

- Sistemas operativos (Windows, Linux,Android, iOS...

- Compiladores
- Ensambladores
- Editores de texto
- Bases de datos
- Frameworks de aprendizaje profundo
- ...

1.2.1 Estructura de un programa en C

Para crear un programa escrito en **C** que imprima ”Hola, mundo” en la pantalla, use un editor de texto (por ahora utilizaremos un compilador en línea https://www.onlinegdb.com/) y crear un archivo que contenga el siguiente código:

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

Listado 1. Hola Mundo

La palabra *include* en línea 1 es una palabra **reservada**, en este caso precedida por un # que indica que es una instrucción de *preprocesador*. La instrucción *include* incluye en el programa actual todas la funcionalidades en la biblioteca especificada entre los símbolos $<$ y $>$ al código de tu programa. En el listado 6 archivo es **stdio.h**, el cual contiene las funcionalidades de flujos de entrada salida (input-output streams); es decir incluye los métodos para leer y imprimir texto.

En la línea 2 se inicia la función *main()*. Esta función es la que contienen las secuencia de instrucciones que se efectúan cuando se ejecuta el programa. La línea 3 contiene una instrucción *printf*, que indica que imprime a la salida estándar.

Los símbolos { y } delimitan un bloque de código, donde { marca el inicio de un bloque de código y } indica el final. El símbolo *;* se utiliza para indicar el final de instrucciones del lenguaje **C**. La instrucción *return* devuelve el un valor e indica el final de la función.

En las instrucciones *funcion(arg1, arg2)* son llamadas a funciones son. Por ejemplo la instrucción *printf(”Hola Mundo!”)* se ejecuta la función *printf* con el argumento *”Hola Mundo!”*.

1.2.2 Compilación

Para que la computadora pueda ejecute el código escrito en **C**, primero debe ser compilado. El proceso de compilación traduce el código **C** a un programa que la computadora puede ejecutar. Un lenguaje de alto nivel como **C** permite a un programado escribir instrucciones

a la máquina en un lenguaje interpretable por las personas, el cual puede ser transformado a un lenguaje pueda entender la máquina (lenguaje máquina). En resumen, el compilador lee el código escrito por el programador y lo traduce a lenguaje máquina, que la computadora puede ejecutar directamente.

1.2.3 Compilador

El compilador es un *programa* de computadora que permite convertir un código escrito en un lenguaje de alto nivel a un código de máquina. Para nuestro curso utilizaremo GNU Compiler Collection (GCC) y MinGW para el caso de Windows. Los ejemplos en este documento asumen el uso de *Visual Studio Code*. Aunque cada compilador se ejecuta de forma especifica, para el caso de *GCC* y *MinGW* se hace como en el listado 2

Listado 2. Como compilar

```
# gcc hola.c -o hola.out
```

Donde *hola.c* es el nombre del archivo que contiene el código fuente a compilar, *hola.cout* es el nombre del archivo de salida que contendrá el programa ejecutable. Si el compilador detecta errores en el código, los mostrará en el terminal. Si no se detectan errores, el compilador producirá un archivo ejecutable.

Para ejecutar el programa compilado, se hace como en el listado 3

Listado 3. Como ejecutar

```
#./hola.out
```

La figura 1 muestra el proceso de compilación y ejeucción utilizando *Visual Studio Code*.

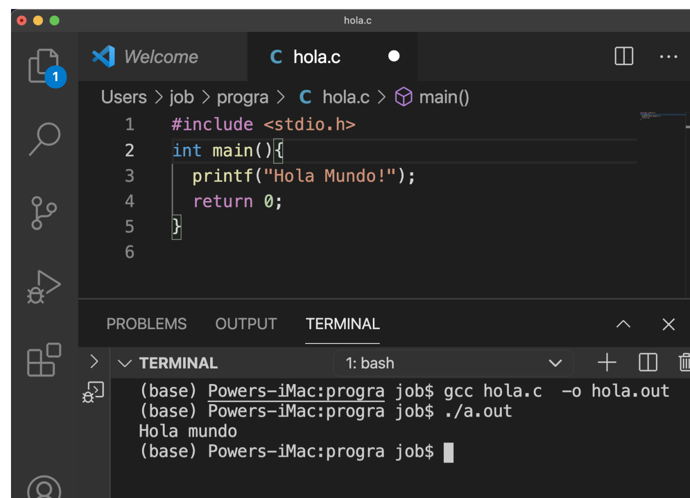

Fig. 1. Compilación y ejecución Visual Studio Code

Si se utiliza un compilador diferente, deberá revisar la documentación que para conocer la forma correcta de utilizarlo.

1.3 Comentarios

Los comentarios se utilizan para indicar algo a la persona que lea el código. El compilador trata los comentarios como un espacio en blanco y no cambian nada en el significado real del código. Hay dos sintaxis utilizadas para los comentarios en **C**, la original /* */ y la una posterior //. Algunos sistemas de documentación utilizan comentarios con formato especial para ayudar a producir automáticamente la documentación para el código.

Un comentario que comienza con /* termina cunado se encuentra la secuencia */. Todo lo que se encuentra entre estas combinaciones de caracteres es un comentario y el compilador lo trata como un espacio en blanco (lo ignora). Este tipo de comentarios se puede expandir múltiples líneas por ejemplo vea el listado 4.

```
/* Este comentario se
expande multiples
lineas */
```

Listado 4. Comentario multi-línea

En **C99** introdujo el uso de comentarios de una sola línea al estilo **C ++**. Este tipo de comentario comienza con // y se extiende hasta el final de una línea (ver 3)

```
// comentario de una solo linea
printf("hola"); //otro comentario de una solo linea
```

Listado 5. Comentario multi-línea

2. Uso de variables y asignación

Las **variables** son utilizadas para registrar el estado en cada paso de la ejecución de un programa. Las variables se almacenan en la memoria en la memoria de la computadora. Aún cuando la computadora hace referencia la ubicación de memoria de las variables utilizando notación hexadecimal, por ejemplo: **0x3F5000AC**. El programador utilizará descripciones simbólicas y el compilador se hará cargo de mantener la relación entre su ubicación en la memoria de la computadora y la descripción del programador.

La descripción simbólica dada por el programador, es en si un nombre descriptivo por ejemplo para almacenar el volumen de un cubo podría asignarse *volumenCubo*. Para definir los nombres de las variable se requiere que el programador tenga en cuenta las siguietes consideraciones:

- Utilizar solo caracteres alfanuméricos y el símbolo _
- No utilizar palabras reservadas (por ejemplo *return*)
- Deben comenzar con una letra o _ (no con un número).
- No deben contener espacios en blanco.
- No pueden llevar tildes.

Algunos ejemplo válidos son por ejemplo: *i*, *contador*, *valor_absoluto*, *x3*; por otro lado *valor absoluto* o *9x* son nombres no válidos. Para asignar un valor a una variable lo hacemos mediante el operador =. El listado 4 muestra un fragmento de código con algunos ejemplos de declaración y asignación de variables.

```
int x;    // declaracion de una variable
x=3;      // asignacion del valor 3 a la variable x
int y=0;  // declaracion y asignacion
```

Listado 6. Declaración y asignación

2.1 Tipos de datos

2.1.1 Enteros

Un entero es un número sin decimales, por ejemplo $1, 2, 3$ y 4 son valores enteros y 3.1416 y 9.81 no son enteros. Si se trata de guardar un valor 1.5 en tipo entero, este se trunca y se almacena como un 1. En **C** hay dos tipos principales de enteros: *int* que utiliza 32 bits para representar enteros en el rango de -2147483648 a 2147483647 y *long long int* que utiliza 64 bit para representar enteros entre -9223372036854775808 y 9223372036854775807. El listado 3 muestra ejemplo de declaraciones de variables del tipo entero.

```
int x = 3;             // entero de 32 bits
int y = 12147483647;  // entero de 64 bits
```

Listado 7. Enteros

2.1.2 Caracteres

Un carácter es un tipo entero que utiliza 8 bits de memoria, se utiliza la palabra reservada *char* para definir una variable del tipo carácter. Soporta valores enteros entre 0 y 255 y es comúnmente utilizado para almacenar texto en formato ASCII. Un char se puede inicializar a partir de un carácter o un entero, pero en ambos casos se almacenará el valor ASCII. El listado 3 muestra unos ejemplos en el que tanto *c1* como *c2* contienen el valor ASCII de la letra Z.

```
char c1 = 'Z';
char c2 = 90;
```

Listado 8. Carácter

2.1.3 Punto flotante

Para el caso de números con punto flotante (decimales) se tienen los tipos de datos *float*, *double* y *long double*. Los *float* utlizan 32 bits y tienen un tamaño en memoria. El tipo *double* utiliza 64 bits de memoria por lo que tendrá una mayor precisión que un tipo *float*.Ambos tipos solo almacenan aproximaciones de los números reales. El listado 3 muestra un ejemplo de como definir variables de punto flotante.

```c
float   f = 3.1416;
double d = 3.141592653589793;
```

Listado 9. Punto flotante

3. Entrada y salida (básico)

Los mecanismos básicos para proporcionar una entrada y mostrar una salida hacia y desde un programa son los métodos *printf* y *scanf*. Las funciones *printf* y *sacnf* son parte d biblioteca **stdio.h**, que incluimos en la cabecera del programa.

La función *printf* se utiliza para imprimir en la salida diferentes tipos de datos como: carácteres, cadenas, flotantes, enteros, octal y valores hexadecimales. Para indicar el tipo de datos a imprimir se utiliza el símbolo % seguido de un especificador del formato, por ejemplo para una variable entera se utiliza *%d*. De forma más general se utiliza *printf(format,arg1,...,argn)*. En la cadena de **formato** se especifica la forma en que se formatearan cada uno de los datos en las variables. También puede se solo una cadena que no contenga especificadores del formato, o una combinación de especificadores e incluir secuencias de escape. Cuando hay a argumentos se deben incluir un especificador por cada uno de ellos. El listado 8 muestra un ejemplo del uso de *printf*.

```c
#include <stdio.h>
int main(){
    int a=2,b=3;
    float c=3.14163;
    printf("El resultado de %d+%d=%d\n",a,b,a+b);
    printf("Aqui un valor flotante %f\n", c);
  }
```

Listado 10. Uso de *printf*

La tabla 1 resume los especificadores disponibles para diferentes tipos de datos.

%c	caracter
%d	entero con signo, en base decimal
%lld	entero con signo, en base decimal (long long)
%u	entero sin signo, en base decimal
%o	entero base octal
%x	entero base hexadecimal
%e	real de punto flotante, con exponente
%f	real de coma flotante, sin exponente
%Lf	real de coma flotante, sin exponente (long double)
%s	cadena de caracteres
%p	puntero o dirección de memoria

Table 1. Especificadores de formato

La función *scanf()* se utiliza para leer datos desde un medio de entrada (comúnmente teclado, archivo o alguna interfaz), los datos son cargados en la memoria de la computadora. La función se utiliza como *scanf(formato,arg1,...argn)*. Al igual que para *printf* en la cadena de *formato* se especifica el tipo de datos de cada uno de los argumentos. En *scanf* requiere como entrada la posición de la memoria donde se almaceno la variable, esto para *guardar* ahí el dato leído. Para hacer referencia a la posición de memoria, se coloca el carácter *ampersand* (&) delante del nombre de cada variable (la excepción son los arreglos). Terminamos esta sección con el listado 12, el cual ilustra el uso de *scanf*.

```
1  #include <stdio.h>
2  int main(){
3      char nombre[10]; //un arreglo lo veremos mas adelante
4      int edad;
5      printf("Introduce tu nombre: ");
6      scanf("%s",nombre);
7      printf("Introduce tu edad: ");
8      scanf("%d",&edad);
9      printf("Hola %s, como te va en los %d?\n", nombre, edad);
10     return 0;
11  }
```

Listado 11. Uso de *scanf*

4. Operadores

Son el conjunto de operadores (símbolos) que nos permiten realizar operaciones aritméticas, lógicas y de comparación. En lenguaje **C** la mayoría de esto símbolos se corresponden con los que ya aprendimos en cursos de matemáticas o lógica.

4.1 Aritméticos

Al igual que en matemáticas, en **C** la operaciones tiene una jerarquía, es decir siguen un orden evaluación. Por ejemplo, la multiplicación (*) y división (/) tienen una mayor precedencia que las operaciones de suma (+) y resta (-), lo que implica que * y / deben evaluarse primero. Por ejemplo, en la expresión $6/2+1$; primero se evaluará $6/2$ y al resultado se le sumará 1. Para especificar el orden de evaluación se utilizan paréntesis *()*. Por ejemplo en la expresión $6/(2+1)$ los paréntesis indican que primero se realiza $2+1$ y el resultado divide a 6.

La tabla 2 muestra los operadores aritméticos disponibles n lenguaje **C** .

Operador	**ejemplo**	**función**
$=$	int $a=3$;	asigna el valor de 3 en la variable a
$+$	$3+4$	Suma
$-$	$3-4$	Resta
$*$	$3+4$	Multiplicación
$/$	$4/2$	División
$+=$	$a+=1$	Suma y asigna (equivalente a $a=a+1$)
$-=$	$a-=2$	Resta y asigna (equivalente a $a=a-2$)
$*=$	$a*=2$	Multiplica y asigna (equivalente a $a=a*2$)
$/=$	$a/=5$	Divide y asigna (equivalente a $a=a/52$)
$++$	$a++$	Suma 1 y asigna (equivalente a $a=a+1$ y $a+=1$)
%	5%3	Módulo/resto de la división entera. En el ejemplo regresa 2 ya que 3 cabe 1 vez en 5 y sobran 2 (el resto).
$\%=$	$a\%=5$	Módulo y asignación (equivalente a $a=a\%5$)

Table 2. Operadores Aritméticos

El listado 21 muestra el uso de diferentes operadores aritméticos en enteros. Todos los operadores a excepción de los de módulo operan números enteros y de punto flotante.

```
1 #include <stdio.h>
2
3 int main() {
4     int n = 100; // inicializamos un entero n igual a 100
5
6     n = n/10; // se actualiza el valor a 10
7     n = n*5; //  ahora toma el valor de 50
8     n = n+50; // nuvamente vale 100.
```

```
9
10      n = n + 50*2; //200 el operado * tiene precedencia sobre la suma.
11      n = (n + 50) * 2; //500 porque los parentesis modifican el orden.
12
13      n /= 10; // 50 dividimos y asignamos
14      n = n%15; // ya que 15 en 50 cabe 3 veces y sobran 5 (valor de n)
15      n++; // sumamos 1 a n y termina con un valor de 6.
16
17      printf("Despues de todas esa operaciones n=%d", n);
18
19      return 0;
20  }
```

Listado 12. Ejemplo del uso de opreadores aritméticos

4.2 Lógicos

Un operador lógico funciona de forma similar a los operadores aritméticos, sólo que utiliza valores booleanos, es decir solo puede tomar valores de **falso** o verdadero. En muchos lenguajes de programación se utilizan como valores booleanos *true* y *false*, pero en el caso de **C** se utiliza el valor 0 para verdadero y 1 para el valor de falso. Los operadores lógicos disponibles se muestran en la tabla 3, como podrás observar son los que que ya conoces de tu clase de lógica.

a	b	a && b (AND)	a || b (OR)	! a (NOT)
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Table 3. Operadores lógicos

4.2.1 Evaluación de corto circuito (short-circuit)

Los operadores || (and) y && (or) son operadores de corto circuito (short-circuit). Es decir, se evalúan las expresiones de izquierda a derecha y se detiene la evaluación tan pronto como conoce la respuesta. El valor de verdad es el último valor en ser evaluado. La figura 2

4.3 Operadores de comparación

Este conjunto de operadores los utilizamos principalmente para comparar valores numéricos y es posible encadenarlos mediante operadores lógicos. La tabla 4 muestra algunos ejemplos del uso de operadores de comparación.

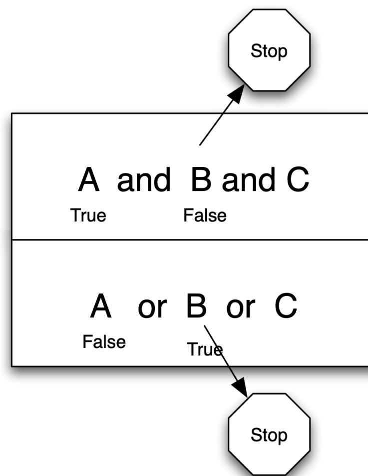

Fig. 2. Evaluación de corto circuito

Operador	**ejemplo**	**función**
$<$	$3 < 5$	menor que, regresa 1
$>$	$3 > 5$	mayor que, regresa 0
$! =$	$3! = 5$	diferente de, regresa 1
$==$	$3 == 5$	Igualdad, regresa 0
$>= 2$	$3 >= 5$	mayor o igual, regresa 0
$<= 2$	$3 <= 5$	menor o igual, regresa 1
encadenamiento	$a < 1$ && $a >= 2$	regresa 1 si el valor de a está entre 2 y 10

Table 4. Operadores de comparación

4.4 Librería math.h

La librería **math.h** incluye muchas otras operaciones matemáticas como raíz cuadrada, funciones trigonométricas, logaritmos, exponencial, etc. Las funciones disponibles se muestran en la tabla **??**.

4.5 Tiro parabólico

Un portero despeja balón, con una velocidad inicial v_i en m/s. Si la pelota sale del suelo con un ángulo de θ (expresado en radianes) y cae sobre el campo sin que lo toque ningún jugador. Escribir un programa que lea los valores v_i y θ y calcule la altura máxima h_{max} que alcanzará el balón.

Recuerde que la altura máxima está dada por la siguiente ecuación:

$$h_{max} = \frac{v_i^2 \sin^2(\theta)}{2g}$$

donde $g = 9.81$ es la constante de gravedad. El listado para realizar el calculao se

Función	Descripción
acos	arcocoseno
asin	arcoseno
atan	arcotangente
atan2	arcotangente de dos parámetros
floor	función suelo
cos	coseno
cosh	coseno hiperbólico
exp(double x)	función exponencial, computa e^x
fabs	valor entero
ceil	menor entero no menor que el parámetro
fmod	residuo de la división de flotantes
frexp	fracciona y eleva al cuadrado.
ldexp	tamaño del exponente de un valor en punto flotante
log	logaritmo natural
log10	logaritmo en base 10
modf	obtiene un valor en punto flotante íntegro y en partes
pow	eleva un valor dado a un exponente, x^y
round	redondea al valor entero más próximo
sin	seno
sinh	seno hiperbólico
sqrt	raíz cuadrada
tan	tangente
tanh	tangente hiperbólica
trunc	trunca el valor, remueve los numeros después del ”.”

Table 5. Funciones **math.h**

despliega en el listado 12

```
#include <stdio.h>
#include <math.h> // Incluimos los metodos en math.h

int main() {
    float vi, theta, hmax, g = 9.81;
    scanf("%f %f", &vi, &theta);
    hmax = (pow(vi, 2) * pow(sin(theta), 2)) / (2.0 * g);
    printf("%.4f\n", hmax);

    return 0;
}
```

Listado 13. Ejemplo del uso de la bilbioteca **math.h**

5. Declaraciones condicionales

Las **declaraciones condicionales** se utilizan para indica si se debe ejecutar o no un bloque de código. Los dos tipos generales son *if-else* y *switch-case*. Las condiciones en declaraciones condicionales se especifican utilizando operadores de comparación (ver tabla 4), recuerde que en **C** estos operadores regresan un valor de verdad 1 y 0.

5.1 El condicional *if-else*

De forma general este, condicional se puede se utiliza como se muestra en el listado 15. El condicional *if* puede contener (aunque no es necesario) condicionales alternativas indicadas con *else if*, en ambos casos se especifica una expresión condicional *cond*, esta última es construida con las reglas de la tabla 4. De igual forma, se puede indicar una opción que contenga un bloque de instrucciones a ejecutar en caso de no cumplirse ninguna de las cláusulas *if/else if*, esto se hace mediante el uso de la instrucción *else*.

```
1  #include <stdio.h>
2
3  int main(){
4      if (cond1){
5          // todo lo que este dentro solo se ejecuta si cond1 es verdadero
6      }
7      else if (cond2){ //tantas instrucciones else if como casos a
8          // todo lo que este dentro solo se ejecuta si condn es verdadero
9      }
10     else{// condicion por defecto
11       // solo se ejecuta si no se cumplio ninguna de las condicionales
       anteriores
12     }
13     return 0;
14 }
```

Listado 14. Ejemplo uso del condicional *if*

Las instrucciones *else/else if* simpre deben ser precedidas de un *if*. Ahora veamos un ejemplo. Suponga que queremos probar si un número *n* es par y en caso afirmativo imprimir *n* **es par**. Esto lo podemos resolver mediante una declaración que condicione el bloque de la instrucción *printf* a que *n* sea par. listado 12 muestra una posible solución.

```
1  #include <stdio.h>
2
3  int main(){
4      int n;
5      scanf("%d", &n); // leemos el entero
6      if (n%2==0){   // inicia el bloque condicional delimitado por {}
7          // todo lo que este dentro solo se ejecuta si n%2==0
8          printf("%d es par", n);
9      }
10     return 0;
```

```
11 }
```

Listado 15. Ejemplo uso de condicional *if*

En el ejemplo del listado 12 solo tenemos una instrucción *if*. Podemos ampliar el código en el listado 12, para que imprima la frase **”*n* es impar”** cuando *n* sea un entero impar como se muestra en el listado 14. Note que en este caso, dado que *n* solo puede ser par o impar podemos utilizar la instrucción *else* pero si hubiese más de dos opciones tendríamos que haber utilizado al menos una instrucción *else if*.

```
1  #include <stdio.h>
2
3  int main(){
4      int n;
5      scanf("%d", &n);
6      if (n%2==0){
7          printf("%d es par",n);
8      }
9      else{
10         printf("%d es impar",n);
11     }
12     return 0;
13 }
```

Listado 16. Ejemplo uso de condicional *if/else*

5.2 Condicional *switch/case*

Esta forma condicional prueba la igualdad de una variable con un número de posibles valores distintos y ejecuta el código correspondiente al valor de la variable. La sintaxis se muestra en el listado 16. La sintaxis difiere de como se especifican los demás bloques de código, ya que el bloque del cada caso se especifíca entre *case valor:* y *break*.

```
1  #include <stdio.h>
2  int main(){
3      switch (val) {
4          case 1:
5              // Este bloque de codigo se ejecuta si valor == 1.
6              break; // ya no compara los siguientes valores
7          case 2:
8              // Este bloque de codigo se ejecuta si valor == 2.
9              break;
10         default:
11             // Este bloque de codigo se ejecuta si diaDelMes es distinto
       a todos los valores listados en los casos anteriores.
12             break;
13     }
14     return 0;
15 }
```

Listado 17. Sinstaxis condicional switch/case

Como ejemplo considere el listado 17, resuelve el mismo problema que el listado 14, determina si un número es par o impar.

```
#include <stdio.h>

int main(){
    int n;
    scanf("%d", &n);
    int valor=n%2;
    switch (valor) {
        case 0:
            printf("%d es par",n);
            break;
        default:
            printf("%d es impar",n);
            break;
    }
    return 0;
}
```

Listado 18. Ejemplo uso de condicional *switch*

Para complementar el uso de *switch-case*, considere el siguiente ejemplo de determinar el horóscopo chino de una persona utilizando su año de nacimiento. El horóscopo chino consta de 12 signos (1. *Rata*, 2. *Toro* 3. *Tigre*, 4. *Conejo*, 5. *Dragón*, 6. *Serpiente*, 7. *Caballo*, 8. *Cabra*, 9. *Mono*, 10. *Gallo*, 11. *Perro* y 12. *Cerdo*). El signo puede determinarse de forma cíclica comenzando en el año de 1924 con *Rata*, 1925 *Toro* y así hasta 1935 *Cerdo* y a partir de 1936 se repite el ciclo comenzando nuevamente con rata. El código del listado 21 muestra un fragmento código necesario, lo primero que hay que notar es que debemos tener 12 casos, cada una se corresponden con los años que han transcurrido desde 1924 (por eso la resta). Se utiliza módulo 12 para ciclar el resultado en ese número. Consideramos para el caso por defecto años previos a 1924.

```
#include <stdio.h>

int main(){
    int year;
    scanf("%d", &year);
    int valor=(year-1924)%12;
    switch (valor) {
        case 0:
            printf("el signo es Rata\n");
            break;
        // codigo omitido ... para casos 1 a 10
        case 11:
            printf("el signo es Cerdo\n");
            break;
        default:
           printf("No es posble determinar el signo para %d\n", year);
           break;
    }
```

```
19      return 0;
20  }
```

Listado 19. Ejemplo del horóscopo chino

6. Instrucciones de repetición (Ciclos)

Los instrucción de repetición o ciclos permiten al programador ejecutar el mismo bloque de código múltiples veces. La repetición depende de una instrucción condicional.

6.1 Instrucción mientras *while*

La sintáxis básica de un ciclo *while* se muestra en el listado 7.

```
#include <stdio.h>
int main(){
    while(cond){
        // instrucciones a ejecutar en el ciclo
    }
}
```

Listado 20. Uso de *while*

El bloque de código acotado por las llaves {...} será ejecutado repetidamente hasta que la expresión condiciones *cond* deje de ser verdadera. El uso de las llaves {...} en ciclos *while* es mandatorio si se desea incluir más de una sentencia dentro del ciclo. por ejemplo los códigos en los listados 9 y 10 son equivalentes

```
#include <stdio.h>
int main(){
        ...
    while(cond)
        x+=x;
        y+=1;
    ...
}
```

Listado 21. Uso de *while2*

```
#include <stdio.h>
int main(){
        ...
    while(cond){
        x+=x;
    }
    y+=1;
    ...
}
```

Listado 22. Uso de *while2*

Note que aun cuando la línea 6 del listado 9 ($y+=1;$) está indentada, solo la primera línea 5 ($x+=x;$) es parte de las instrucciones que se repetirán como parte del ciclo *while*. Lo cual dependiente del objetivo podría llevar a resultados erroneos , como convertirse en un ciclo infinito. Por lo que es recomendable el uso de las las llaves {...} para delimitar claramente el bloque de código a repetir.

6.2 Instrucción *do-while*

La instrucción *do-while* es muy similar a un ciclo *while*, la diferencia es que mientras *while* evalua la declaración condicional previo al inicio de cada iteración, *do-while* comprueba la declaración condicional al final de cada iteración, lo cual implica que el bloque de código se ejecutará al menos una vez.

La sintaxis se describe en el listado 7.

```
#include <stdio.h>
int main(){
    do{
        // instrucciones a repetir
    }while(cond);
  }
```

Listado 23. Uso de *while2*

6.3 Instrucción *for*

Una instrucción *for* específica un tipo de ciclo que permite al programador controlar el número de iteraciones/repeticiones del ciclo.

La sintaxis básica de un ciclo *for* se muestra en el listado 4

```
    for(inicializacion;condicional;actualizacion){
        // instrucciones a repetir
    }
```

Listado 24. Uso de *for*

Como se puede observar, el ciclo *for* se compone de tres expresiones (parámetros): una expresión de *inicializacion*, la cual es ejecutada una sola vez, antes de empezar las repeticiones. Una expresión *condicional*, la cual es evaluada una vez por cada repetición justo antes de comenzar cada iteración. Esta última se utiliza para finalizar el ciclo cuando es evaluada como falsa (0). La expresión *actualizacion*. Es evaluada una vez por repetición justo después de terminar la repetición. El listado 4 muestra un ejemplo de un ciclo for que imprime los números entre 1 y 10.

```
    for(int i=0;i<10;i++){
        printf("%d\n", i);
    }
```

Listado 25. Ejemplo del uso de *for*

6.4 Ciclos anidados

Es común que se requiera anidar ciclos, es decir utilizar un ciclo dentro de otro ciclo. Por ejemplo, vamos a escribir un programa que imprima un rectángulo como el que se muestra en la figura 3.

o o o o o o

o o o o o o

o o o o o o

Fig. 3. Ejemplo de una tarea que requiere el uso de un ciclo anidado

Para imprimir el cuadrado se requiere un ciclo que itere el número de columnas (j) y otro más para el número de renglón (i). El listado 12 muestra una posible implementación utilizando la instrucción *for*.

```
#include <stdio.h>
int main()
{
    for(int i=0;i<3;i++){   // iterar de 0 a 2
        for(int j=0;j<6;j++){// iterar de 0 a 5
         printf("o ");   // imprime una o por cada j
        }
        printf("\n"); // imprime un \n por cada fila
    }
    return 0;
}
```

Listado 26. Impresión de un rectangulo de 3×6 *for*

6.5 *break* y *continue*

Existen situaciones donde es de interés interrumpir o reiniciar un ciclo. En **C** existen dos instrucciones que se pueden usar en de las distintas estructuras de control y principalmente en los ciclos , que permiten controlar las dos situaciones mencionada. Las instrucciones son *break* y *continue*:

- *break*: Indica detener la ejecución de un ciclo y salirse de él.
- *continue*: Interrumpe la iteración actual y regresa al principio del ciclo para realizar otra iteración, si aún se no se cumple la condición de paro.

El listado 13 muestra el uso de la instrucción *break* para interrumpir un ciclo cuando el entero leído sea 0, mientras que si $n! = 0$ imprime el resultado de $n * n$.

```
#include <stdio.h>
int main()
{
    int n;
    while(1){
        scanf("%d",&n);
```

```
7         if(n==0)
8             break; // si n es 0 se temina el ciclo
9          printf("%d\n", n*n); // esto solo se ejecuta si n!=0
10    }
11    return 0;
12 }
```

Listado 27. Uso de *break*

El listado 12 ilustra el uso de la instrucción *continue* para imprimir solo los números pares. En la líneas se prueba si n es impar, si así ocurre se ingnora el resto del ciclo y se inicia la siguiente iteración.

```
1 #include <stdio.h>
2 int main()
3 {
4     for(int i=0;i<10;i++){
5          if(i%2)
6              continue;
7          printf("%d ",i); // esto solo se ejecuta si n%2==0
8     }
9     printf("\n");
10    return 0;
11 }
```

Listado 28. Uso de *continue*

7. Funciones

Una función es un bloque de instrucciones que están aislado/separadas del código principal. Una función se ejecuta como una instrucción desde el otra función, esta puede ser desde la función **main** o la misma función. Cuando se se completan las instrucciones en terminado d el código de la función, se regresa el flujo del programa al punto donde se hizo la llamada a la función. La estructura general de un función es como se muestra en el listad 5:

```
1 tipo_de_dato nombre_de_la_funcion(tipo1 arg1, tipo2 arg2,..., tipon argn)
    {
2     //instrucciones que componen la funcion
3     return var; \\ donde var es del tipo tipo_de_dato
4 }
```

Listado 29. Ejemplo del uso de opreadores aritméticos

donde *tipo_de_dato* específica el tipo de dato (*int*, *char*, etc) valor de regreso de la función, *nombre_de_la_funcion* sigue las mismas reglas que las utilizadas para nombrar variables. Entre *()* se especifican separados por comas la lista de argumentos y los tipos de los mismos. A continuación revisemos el ejemplo del listado 5.

```
1 int suma(int a, int b){
2     int s=a+b;
3     return s;
4 }
```

Listado 30. Ejemplo una función que suma dos variables

En la función *suma*. El primer *int* especifica el tipo de del valor que regresará la función; *suma* es el nombre de la función (este lo utilizaremos para ejecutar la función); *int a* e *int b* son los parámetros o argumentos de la función. Estos último se defines como *tipo_de_dato* identificador. El código de la función está delimitado por llaves {...}. Toda función debe de tener un comando *return* (excepto si el tipo de retorno se específica como *void*).

7.1 Ejecutar una función

Para ejecutar una función, ésta primero debe haber sido declarada en cualquier línea anterior a la línea que la llama. En el listado 16

```
1 #include <stdio.h>
2 void hola(){
3     printf("Hola Mundo!\n");
4 }
5 int suma(int a, int b){
6     int s=a+b;
7     return s;
8 }
9 int main() {
10    int sum;
```

```
11    hola();
12    sum = suma(7,3);
13    printf("%d\n", sum);
14    return 0;
15 }
```

Listado 31. Ejemplo del uso de funciones

Note que el hecho de que las funciones *suma* y *hola* preceden la declaración de *main* es lo que permite que puedan ser utilizadas desde este último. Para llamar la función *hola* todo lo que tuvimos que escribir fue *hola()*; esto es debido a que no requiere de ningún parámetro y su tipo de retorno es void. Por otro lado, la función *suma* requiere de dos parámetros y su tipo de retorno es *int*. Para pasarle datos al a función, simplemente listamos los datos en el orden dado cuando definimos la función. El valor retornado por la función lo asignamos a la variable *sum*.

7.2 Parámetros

Los parámetros son usados para pasar los datos sobre los que operan la funciones. Como ya vimos en el ejmeplo anterior, los datos que se desea pasar a una función se pasan entre *()*, en el orden determinado por la definición de la función. El primer parámetro se asignará a la primera variable listada en la definición de la función, el segundo parámetro se asignará a la segunda variable y así sucesivamente. Además, el número de valores debe tener el número correcto de parámetros y el tipo de datos especificado en la definición de función, de lo contrario ocurrirá un error de compilación.

Por ejemplo, *suma(1,3)* es una forma correcta de utilizar la función suma; mientras que *suma(1,3,2)* y *suma(1.0f,3)* son formas incorrectas, en el primer caso el número de parámetros es incorrecto y en segundo el tipo de datos del primer dato es un flotante y debería ser un tipo *int*.

7.3 Recursión

Una función también puede ejecutar otras funciones, o incluso ejecutarse a sí misma. Cuando una función se llama a sí misma, decimos que es una función recursiva. Consideremos como un ejemplo la función recursiva para calcular el factorial de un número. El factorial de un número n, escrito $n!$, de define como el producto de cada número del 1 al n y puede definirse en términos de si mismo como en la ecuación 1.

$$\begin{matrix} 1 & n = 1 \\ n \times (n-1)! & n > 1 \end{matrix} \tag{1}$$

Una función recursiva para calcular el factorial de un número n lo podemos definir como en el listado 16.

```
1 #include <stdio.h>
2 int factorial(int n){
3     if(n==1){
```

```
4           return 1;
5       }
6       else{
7           return n*factorial(n-1);
8       }
9   }
10  int main() {
11      int fac;
12      fac = factorial(5);
13      printf("%d\n", fac);
14      return 0;
15  }
```

Listado 32. Ejemplo de factorial recursivo funciones

Para ver como opera la función calcules el 4! el cual será $4\times3\times2\times1=24$. Iniciamos la ejecución con la instrucción *factorial(4)*, entonces ocurre lo siguiente:

- como $n! = 1$ se ejecuta el código dentro del *else* en la línea 7. Esto regresará $4\times$ *factorial(3)*.
  - como $n=3$, se vuelve a ejecutar la línea 7, pero ahora como $3\times$ *factorial(2)*.
    - como $n=3$, se vuelve a ejecutar la línea 7, pero ahora como $3\times$ *factorial(2)*.
      - para $n=2$, nuevamente se ejecuta la línea 7, pero como $2\times$ *factorial(1)*.
      - para $n=1$, se ejecutará la línea 3 y regresará el valor de 1.
      - entonces, *factorial(2)* regresa $2\times1=2$
    - una vez resuelto *factorial(2)*, se regresa $3\times2=6$.
  - Finalmente, se resuelve $4\times$ *factorial(3)* como $4\times6=24$

El proceso descrito puede verse graficamente en la figura 4.

Las soluciones recursivas usualmente resultan en un programa simple. La desventaja es que por lo general tienen implica un costo extra de tiempo de ejecución y memoria utilizada. Esto debido a a que cada vez que se llama a una función, está es colocada en memoria. Como no se sale la función factorial hasta que n es igual a 1, esto puede implicara que se tienen n funciones en memoria. Esto no es un problema para casos pequeños, pero para casos más grandes puede derivar en problemas de memoria.

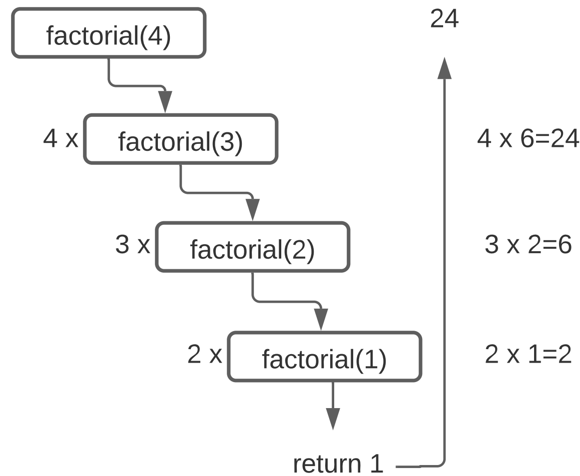

Fig. 4. Árbol de recursión de la función factorial

8. Arreglos

Hasta este punto solo hemos utilizado variables que almacena un valor único, valores enteros, flotantes y caracteres. Ahora revisaremos una alternativa que nos permite almacenar múltiples valores en una sola *unidad*. Este estructura es denominada como arreglo (array en inglés).

Un arreglo es una colección de tamaño fijo, la cual se utiliza para almacenar un conjunto de elementos del mismo tipo; estos son almacenados en en un bloque continuo de de memoria. Por ejemplo, un arreglo de n enteros contendrá n valores de tipo *int*, mientras que uno arreglo de carácteres almacenará un valores de tipo *char*. El número de elementos n es al *tamaño* o *dimensión* del arreglo. Para declarar **C** una variable que hará referencia a un arreglo se utiliza la siguiente sintaxis:

```
tipo_de_dato nombre_del_arreglo [tamaño];
```

por ejemplo para definir un arreglo de 5 enteros podemos utilizar la siguiente sentencia:

```
int arreglo[5];
```

Los elementos en un arreglo pueden haciendo referencia a la posición dentro del mismo, a esa posición se le conoce como índice. En **C** los índices comienzan desde el valor 0, por lo que el primer elemento del arreglo será índice 0, mientras que último será el de la posición $n-1$ (para un arreglo de n elementos). Por ejemplo para acceder al valor almacenado en el posición 3, ser hará con la expresión **arreglo***[*3*]*. El valor retornado mediante el uso de

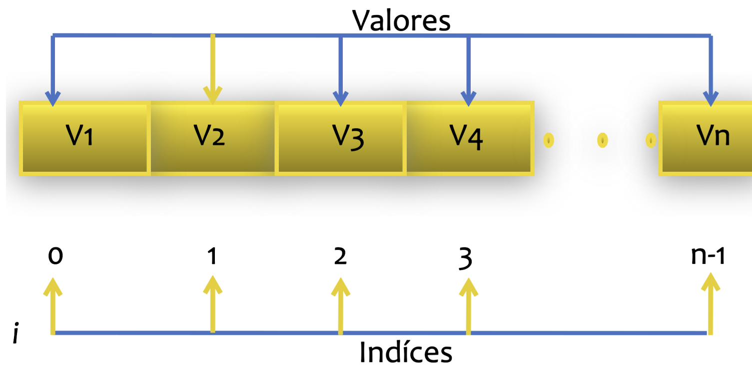

Fig. 5. Representación de un arreglo de tamaño *n*

esta expresión es utilizado de la misma forma que cualquier otro valor entero. Podemos representar gráficamente un arreglo como en la figura 5, donde cada valor *v* está contenido en una celda(de memoria) el valor del índice se indica mediante el uso de un subíndice.

8.1 Inicialización de arreglos

Al igual que el resto las variables, los elementos de un arreglo deben de ser inicializados antes de que puedan ser usados; de lo contrario, es probable que se presenten resultados inesperados. Existen múltiples formas de inicializar un arreglo. Una posibilidad es primero declarar el arreglo y luego inicializar algunos o todos los elementos de mismo, esta opción se muestra en el listado 9:

```
#include <stdio.h>

int main(){
    int arreglo[3];
    arreglo[0]=1;
    arreglo[1]=2;
    arreglo[2]=3;
}
```

Listado 33. Inicialización de un arreglo

Otras dos opciones se muestran en el listado 7. Tanto en la línea 4 como en la 5 se especifican todos separados por , los valores del arreglo entre las llaves {...}, la diferencia es que en 4 especificamos la dimensión del arreglo, mientras que en 5 el compilador determina el tamaño, basado en cuántos elementos se proporcionan.

```
#include <stdio.h>

int main(){
    int arr1[3]={1,2,3};
    int arr2[]={1,2,3,4,5};
}
```

Listado 34. Inicialización de un arreglo

Los valores del arreglo también pueden inicializarse con valores que no se conocen de antemano. Por ejemplo, podemos leer los valores desde el teclado como en el listado 16

```
#include <stdio.h>

int main(){
    int arr[2];
    printf("Elemento arr[0]");
    scanf("%d", arr[0] ); //note que en este caso no se pone &
    printf("Elemento arr[1]");
    scanf("%d", arr[1] );
}
```

Listado 35. Inicialización de un arreglo leyendo los valores desde el tecaldo

8.2 Acceder a los elementos de un arreglo

Como ya mencionamos para acceder a los elementos de un arreglo se debe hacer referencia al posición, esto utilizando el índice. Este último debe de ser un entero en el rango de 0 a $n-1$, donde n es la dimensión del arreglo. El índice puede especificarse directamente como un valor entero, mediante una variable ser el resultado de un expresión aritmética. El listado **??** muestra algunos ejemplos para especificar el índice.

```
...
arr[7];
arr[i];
arr[i%5 + 1];
...
}
```

Listado 36. Acceso a los elmentos de un arreglo

Los arreglos también pueden ser utilizados como parámetros en funciones. Cuando se declara la función, se especifica el arreglo como parámetro, pero no se incluye la dimensión entre los corchetes (*[]*). El arreglo se utiliza normalmente dentro de la función. El listado **??** muestra una función que regresa la suma de todos lo elementos en un arreglo. En el listado función *suma* toma u arreglo de enteros enteros arreglo y un entero que especifica el tamaño. Para el ejemplo del listado el programa imprime el valor 10.

```
#include <stdio.h>

int suma(int arr[], int n){
    int sum=0;
    for (int i=0;i<n;i++){
        sum+=arr[i];
    }
    return sum;
}

int main(){
```

```
12      int arr[]={1,2,3,4};
13      printf("%d", suma(arr,4));
14
15  }
```

Listado 37. Acceso a los elmentos de un arreglo

8.3 Arreglos multidimensionales

En **C** podemos definir arreglos multidimensionales. Para crear un arreglo bidimensional (matriz) se escribe:

```
tipo_de_dato nombre_del_arreglo [n][m];
```

El arreglo tendrá una dimensión $n \times m$ de elementos del mismo tipo. Podemos pensar en un arreglo multidimensional como un ”arreglo de arreglos”. El primer índice indicará cuál de los n sub-arreglos acceder, y el segundo índice designa uno de los m elementos dentro de ese sub-arreglo. La inicialización y el acceso funcionan de forma similar que para los arreglos unidimensionales. El listado 20 muestra las distintas forma de inicialización de un arreglo de dos dimensiones.

```
1  #include<stdio.h>
2  int main() {
3      int matriz[2][3];
4      matriz[0][0] = 6;
5      matriz[0][1] = 0;
6      matriz[0][2] = 9;
7      matriz[1][0] = 2;
8      matriz[1][1] = 0;
9      matriz[1][2] = 1;
10     //una inicializacion alternativa seria:
11     //int matriz[2][3]={{6,0,9},{2,0,1}} ;
12     for (int i = 0; i < 2; i ++) {
13         for (int j = 0; j < 3; j ++) {
14             printf("%d ", matriz[i][j]);
15         }
16         printf("\n");
17     }
18     printf("\n");
19 }
```

Listado 38. Arreglos multidimensionales

La representación gráfica de una arreglo la podemos entender como una tabla en el que los índices i y j especifican la posición del reglón y la columna del dato dentro de la tabla (ver figura 6)

Es posible generalizar la declaración de arreglos de k dimensiones como se muestra en el listado 39:

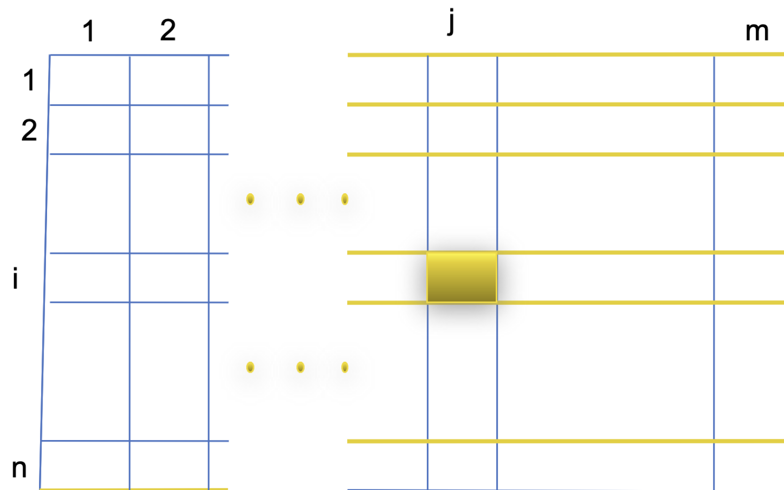

Fig. 6. Arreglo de dos dimensiones

Listado 39. Arreglos *k*-dimensionales

```
tipo_de_dato nombre_del_arreglo [n1];
tipo_de_dato nombre_del_arreglo [n1][n2];
.
.
.
tipo_de_dato nombre_del_arreglo [n1][n2]...[nk];
```

8.4 Cadenas

Las cadenas de texto en **C** sonr representadas como un arreglo de carácteres. En otras palabras, una cadena es simplemente un arreglo de *char*, por lo que puede ser manipulada de la misma forma que los arreglos que ya hemos revisado. El listado 13 muestra como definir e iterar los elementos en una cadena de carácteres.

```
#include <stdio.h>
int main()
{
    char S[]={'H','o','l','a',' ', 'm','u','n','d','o','\0'};
    int i=0;
    while(S[i]!='\0'){
        printf("%c", S[i]);
        i++;
    }
    printf("\n");
    return 0;
```

```
12 }
```

Listado 40. Cadenas de carácteres

El programa de listado 13 imprime la *Hola mundo* letra a letra. Note que el último elemento del arreglo *S* termina con el carácter especial *'\0'* conocido como carácter nulo. Este último es utilizado para indicar el final de la cadena.

Las cadenas también pueden ser inicializadas utilizando cadenas constantes. En este caso, no se necesita un carácter nulo al final, el compilador automáticamente insertará. El listado 13 muestra como definir una cadena mediante el uso del valor literal en el carácter *"*.

```
1  #include <stdio.h>
2  int main()
3  {
4      char S[]="Hola Mundo";
5      int i=0;
6      while(S[i]!='\0'){
7          printf("%c", S[i]);
8          i++;
9      }
10     printf("\n");
11     return 0;
12 }
```

Listado 41. Cadenas de carácteres

9. Entrada salida de cadenas/caracteres

Hasta ahora hemos leído y escrito utilizando los métodos *scanf* y *printf* los cuales nos permiten especificar el formato de la entrada y salida. En esta sección revisaremos un conjunto de métodos que nos permiten leer y escribir cadenas y caracteres. Los métodos *getchar*, *putchar*, *gets* y *puts*.

9.1 getchar

El método *getchar* toma como entrada una varible de tipo int donde almacena lod. Debido a que la función *getchar* devuelve los caracteres ingresados, cuando encuentra el carácter de fin de archivo **EOF**, el cual es generalmente la constante -1. Por lo tanto, en este caso, la función *getchar* devuelve un valor negativo, y es incorrecto asignar un valor negativo a una variable de tipo *char*. Cuando se ingresan los datos desde el teclado las secuencias *CTRL + D* en Linux, *CTRL + Z* Windows ingresan **EOF**. Podemos utilizar la función *getchar* para leer una cadena de varios caracteres, leyendo en un bucle la cadena carácter a carácter (ver listado 22).

```
1  #include <stdio.h>
2
3  int main(){
```

```
4     int i=0;
5     char c=getchar();
6     char C[100];
7     while(c!='\n'){   //leemos hasta el final de linea
8       if(c!=' '){   //ignoramos los espacios en balnco
9         C[i]=c;
10        i++;
11      }
12      c=getchar();
13    }
14    i=0;
15    while(C[i]!='\0'){
16      printf("%c",C[i]);
17      i++;
18    }
19    printf("\n");
20    return 0;
21  }
```

Listado 42. Uso de *getchar* para leer una cadena carácter a carácter

9.2 gets

El método *gets* se usa para leer cadena desde el dispositivo de entrada estándar (teclado) hasta un retorno de carro *n* o **EOF**, pero el retorno de carro no se toma como parte de la cadena.

La función *gets(s)* se comporta de foma similar a *scanf ("%s", &s)*. La diferencia es que cuando se utilza *scanf* al ingresar un espacio, se considerará el final de la cadena de entrada y los caracteres posteriores al espacio se tratarán como la siguiente entrada, mientras que el método *gets* tomará la cadena como toda la entrada hasta que se ingresa un retorno de carro. (partir de VS2015 (Windows) *gets_s* reemplaza a *gets*).

10. Apuntadores

Antes de comenzar, recordemos que una variable es un referencia a un bloque especifico de memoria, dentro de la cual ser almacena el valor de un dato. El nombre de la variable nos permite recuperar o re-asignar el valor en esa localidad. El tamaño del bloque de memoria depende del tipo de datos que asignamos a la variable. Por ejemplo, en una máquina de 32 bits, un variable del tipo *int* usará 4 bytes, mientras que en máquinas antigua de 16 bits será de 2 bytes, y en máquinas de 64 bits podrían utilizarse 8 bytes. En **C** número de bits utilizados para no será necesariamente el mismo en todas las computadoras. Por ello existen los diferentes tipos de variables enteras (recuerde enteros largos *long int* o cortos como *short int*). Podemos identificar el tamaño de los diferentes tipos con la función *sizeof*. El listado 9 puede utilizarse para identificar los tamaños de algunos tipos numéricos.

```
#include <stdio.h>
int main(){
    printf("Un entero utiliza %ld bytes\n", sizeof(char));
    printf("Un entero utiliza %ld bytes\n", sizeof(int));
    printf("Un entero largo utiliza %ld bytes\n", sizeof(long));
    printf("Un float utiliza %ld bytes\n", sizeof(float));
    printf("Un double utiliza %ld bytes\n", sizeof(double));
}
```

Listado 43. Ejemplo del uso de la función *sizeof*

Cuando definimos una variable sin inicializarla estamos solamente asignado un especie de *alias* a un bloque de memoria que tendrá espacio suficiente para almacenar ese tipo de datos. Mientras que cuando le asignamos un valor lo que hacemos es escribir el valor en esa localidad específica. Por tanto, podemos decir que hay dos valores asociados con la instancia de cada variable; uno es el valor almacenado y el otro la dirección de la localidad de la memoria.

Un **apuntador** es una variable que almacena la dirección de memoria de una variable. El tamaño de una variable de este tipo depende del sistema. En computadoras antiguas con poca memoria, la dirección puede ser contenida en 2 bytes. Sin embargo, en computadoras más recientes se pueden requerir más bytes para almacenar una dirección. En **C** una variable apuntador se define utilizando el carácter * al inicio del nombre de la variable, y asignado un tipo, el cual de coincidir con el tipo de dato que se almacenará en la dirección. Por ejemplo para declarar un apuntador a una variable tipo entero utilizaríamos *int *ptr* donde el * indica que es una variable apuntador, es decir, que se reserven los bytes necesarios para alojar una dirección en la memoria.

Cuando no inicializamos una variable y declaración ocurre fuera de una función, los compiladores ANSI la inicializarán automáticamente a cero (valor por defecto). De igual forma un apuntador cuando no tiene asignado un valor; si la declaración es fuera de cualquier función, es inicializado a un valor que garantice que no apunte a una instancia **C** (objeto o función). El apuntador se inicializa *NULL* y se denomina como un apuntador nulo (null pointer).

Un apuntado nulo, dependiendo del compilador puede o no evaluarse a cero. Por ello, para hacer el código fuente compatible entre distintos compiladores, existe el macro *NULL*, esto no permite determinar si una apuntador es nulo como *if(*ptr *== NULL)*.

Para acceder a la dirección de memoria almacenada en un puntero se utiliza el operador unitario *&*. Por ejemplo si tuvieramos una variable entera **k** y el apuntador **ptr** la sentencia **ptr**==*&***k**, indicaría que la dirección de la variable **k** será copiada en en **ptr**, es decir se tendrá un apuntador a **k**.

El operador unitario * conocido como *indirección* o de *desreferencia* permite escribir/recuperar el valor en la dirección de memoria reverenciada por el apuntador. Por ejemplo, las sentencia ***ptr**=4 escribirá un 4 en la locadidad de memoria apuntada por **ptr**. Así que como **ptr** apunta al mismo lugar que **k**, el valor de este último será actualizado. Al utilizar el operador * se hace referencia al valor al que **ptr** apunta y no al valor de el apuntador en si.

El listado 16 ilustra el uso de los operadores * y *&*, el especificador de formato *%p* es utilizado para mostrar la dirección de las variables del tipo puntero.

```
1  #include <stdio.h>
2  int k; int *ptr;
3  int main (void){
4     k = 2021;
5     if(ptr==NULL)
6      printf("El apuntador ptr es nulo\n");
7     ptr = &k;
8     if(ptr!=NULL)
9       printf("El apuntador ptr apunta a %p\n", ptr);
10    printf("k tiene el valor %d y esta almacenado en %p\n", k, &k);
11    *ptr=2022;
12    printf("ptr tiene el valor %d y apunta %p\n", *ptr, ptr);
13    printf("k tiene el valor %d y esta almacenado en %p\n", k,\ &k);
14   return 0;
15 }
```

Listado 44. Ejemplo del uso de los operadores para aputadores

10.1 Parámetros por referencia

En **C** los parámetros de tipos primitivo siempre se pasan por valor. Los punteros son útiles cuando se requiere pasar parámetro por referencia, ya que no permite utilizar la dirección de la variable, es decir, que lo que se envía es un puntero a su valor. En este caso, el puntero funciona como un parámetro sólo de entrada que permite modificar el valor de la variable a la que apunta. Para indicar en una función que se realizara un paso por referencia se debe especifícar la variable como tipo puntero utilizando la sintaxis *tipo* ***varname**, y al momento de utilizar la función se debe pasar especificar como parámetro la dirección utilizando el operador *&*. La instrucción *scanf(*"%d" *&***varname***)* es un ejemplo de una función que hace paso por referencia, en esta caso el valor leído del teclado es escrito en la dirección de memoria de la variable **varname**.

Como ejemplo de una función que hace paso por referencia, consideré el listado 15. La función **swap** toma los valores de las direcciones (punteros) de las variables **a** y **b** e intercambia los valores almacenados en cada una de las direcciones.

```
#include <stdio.h>
void swap(int *a, int *b){
    int tmp=*a;  // copiamos el valor en b
    *a=*b; // escribimos el valor almacenado en la direccion de b en a
    *b=tmp; // escribimo el valor tmp en la direccion referenciada por b
}
int main()
{
    int a=1,b=2;
    printf("Antes del swap a=%d , b=%d\n", a,b);
    swap(&a,&b); // pasamos como parametros las direcciones de a y b
    printf("Despues del swap a=%d , b=%d\n", a,b);
    return 0;
}
```

Listado 45. Ejemplo de una función con paso por referencia

10.2 Arreglos y apuntadores

Los arreglos pueden ser considerados como puntero del tipo *constante*, es decir que apunta a un lugar fijo. Como ya vimos los arreglos se acceden mediante el uso de índices, los cuales con especificados ente los caracteres *[]*. Los arreglos pueden ser accedidos/operados utilizando punteros y la denominada aritmética de punteros. En el código del listado 22 las funciones *imprime1* e *imprime2* son equivalentes, las expresiones "*int* **A** *[]*" e *int* ***ptr**" hacen ambas referencia a un apuntador del tipo entero y pueden ser "intercambiadas", así en la linea 18 pasamos como argumento a *imprime1* el puntero **ptr** y en la línea 19 el arreglo **A** a la función *impirme2*, es decir intercambiamos un arreglo con un apuntador.

```
#include <stdio.h>
void imprime1(int A[], int n){
    printf("Ejecutando funcion con arreglo: ");
    for(int i=0;i<n;i++)
        printf("%d ", A[i]);
    printf("\n");
}
void imprime2(int *ptr, int n){
    printf("Ejecutando funcion con puntero: ");
    for(int i=0;i<n;i++)
        printf("%d ", *(ptr+i));
    printf("\n");
}
int main()
{
    int A[]={1,1,2,2,3};
    int *ptr=&A[0]; //Apuntamos ptr a la direccion que almacena A[0]
    imprime1(ptr,5);
```

```
19      imprime2(A,5); // A es un puntero!
20      return 0;
21  }
```

Listado 46. Ejemplo equivalencia puntero/arreglo

En la línea 12 de listado 22 le sumamos el valor de la variable *i* al apuntador **ptr**, esto es un ejemplo de aritmética de punteros. Los punteros pueden utilizarse en operaciones aritméticas, de asignación y de comparación. La tabla **??** muestra un resumen de las operaciones que puede ser realizadas con apuntadores.

Operación	**Tipo de regreso**	**Explicación**
pt1++	Apuntador	Desplazamiento de una posición hacia adelante
pt1–	Apuntador	Desplazamiento de una posición hacia atrás
pt1+n	Apuntador	Desplazamiento de *n* posiciones hacia delante
pt1-n	Apuntador	Desplazamiento de *n* posiciones hacia atrás
pt1-**pt2**	Entero	Distancia entre las posiciones
pt1==*NULL*	0 o 1	Comprueba si el puntero es nulo
pt1!=*NULL*	0 o 1	Comprueba si el puntero es diferente de nulo
pt1==**pt2**	0 o 1	Comprueba si **pt1** apunta un sitio diferente **pt2**
pt1>**pt2**	0 o 1	Comprueba si **pt1** es menor que **pt2**
pt1>=**pt2**	0 o 1	Comprueba si **pt1** es mayor o igual **pt2**
pt1<**pt2**	0 o 1	Comprueba si **pt1** es menor **pt2**
pt1<=**pt2**	0 o 1	Comprueba si **pt1** es menor o igual **pt2**
pt1! =**p21**	0 o 1	Comprueba si **pt1** es diferente de **pt2**
pt1=**pt2**	Apuntador	Asignación

Table 6. Operaciones con punteros

De la tabla 6, podemos ver que los apuntadores pueden ser comparados con operaciones relacionales y aritméticas. Este tipo de operaciones solo tiene sentido entre en apuntadores dirigidos al mismo arreglo. En las operaciones con punteros estamos realizado operaciones sobre direcciones de memoria, esto se aprecia en la figura 7. En la figura 7 el apuntador **ptr** es asignado al inicio de un arreglo de 5 enteros *A*, es decir referencia la dirección *0xA610*. Al hacer la operación (**ptr**+1) se incrementa una posición con respecto de **ptr**, pero como cada elemento requiere 4 bytes se avanzará a la posición *0xA614*. La posición (**ptr** + 4) e estará 16 bytes delante, en la localidad *0xA620*, lo que es equivalente a avanzar 16 bytes (el tamaño de 4 enteros, en hexadecimal 0xA610 + 0x10).

10.2.1 Apuntador tipo *void*

Un apuntador puede ser asignado a otro si son del mismo tipo, cuando son de tipos distintos hay que usar un operador de conversión (cast). Un apuntador del tipo *void* asignarse a cualquier tipo de puntero o bien ser asignados a un apuntador de cualquier tipo. Sin

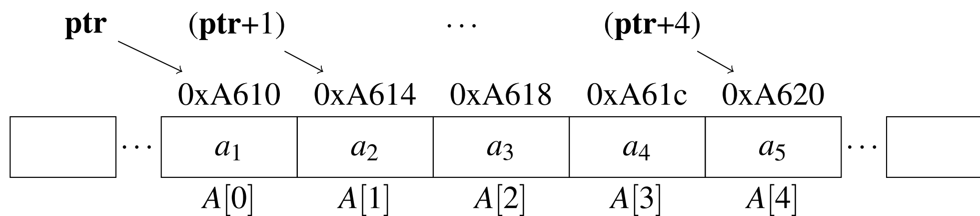

Fig. 7. Arreglo/apuntador en memoria

embargo un puntero *void* no puede ser desreferenciado. Un apuntador de tipo *void* es un puntero genérico, que puede recibir el valor de cualquier otro puntero incluso *NULL*. El listado 19 ilustra el uso del apuntador *void*.

```
#include <stdio.h>
int main()
{
    int a = 1;
    float b=2.0;
    printf("a=%d b=%f\n", a,b);
    void *ptr=&a; // apunta a un entero

    *( int *) ptr=2;
    printf("a=%d\n", a);
    ptr = &b; // apunta a un float
    *( float *) ptr=3.5;
    printf("b=%f\n", b);
    // printf("%f\n", *ptr); no se permite desreferenciar apuntadores tipo void

    return 0;
}
```

Listado 47. Ejemplo del uso de un apuntador tipo *void*

10.3 Ordenamiento de Burbuja (BubbleSort)

El ordenamiento es una tarea común que realizamos continuamente. En términos simples, es colocar la información de una manera que de una noción de relación $<$ o $>$, está es usulamente léxico-gráfica o numérica. En la computación el ordenamiento de datos tiene una función muy importante, ya sea como un fin en sí o como parte de otros procedimientos más complejos. En este curso solo revisaremos uno de los ordenamientos más simples. Sin embargo, existen una gran variedad de estos.

10.3.1 conceptos preliminares

Clave : La parte de un registro por la cual se ordena la lista. Por ejemplo, una lista de registros con campos nombre, dirección y teléfono se puede ordenar alfabéticamente

de acuerdo a la clave nombre. En este caso los campos dirección y teléfono no se toman en cuenta en el ordenamiento.

Criterio de ordenamiento (o de comparación): El criterio que se utiliza para asignar valores a los registros con base en una o más claves. De esta manera decidimos si un registro es mayor o menor que otro.

Registro : Un grupo de datos que forman la lista. Pueden ser datos atómicos (enteros, caracteres, reales, etc.) o grupos de ellos, en **C** será estructuras (las cuales veremos más adelante en el curso, por ahora nos concentraremos en ordenar un arreglo de números enteros).

El método de la burbuja, es probablemente el algoritmo de ordenamiento más sencillo. Consiste en iterar repetidamente un arreglo, comparando los elementos adyacentes de dos en dos. Si un elemento es mayor que el que está en la siguiente posición se intercambian.

La estrategia general es la siguiente: recorrer el arreglo comparando los elementos adyacentes e intercambiándolos sino están en orden relativo. Esto tiene el efecto de que el elemento de mayor valor se vaya desplazando como una *Burbuja* hasta la última posición del arreglo. A continuación se repite la operación, haciendo que se desplace el segundo valor más grande hasta la penúltima posición. Este proceso continua hasta que se hayan desplazado todos los elementos a sus posiciones correctas.

Revisemos el procedimiento con el siguiente ejemplo Por ejemplo, si comenzáramos con el arreglo:

9	6	8	12	3	1

primero se compararían el 9 y el 6 y, como no están en el orden correcto, se intercambian con lo que quedaría:

6	9	8	12	3	1

Después se compara el 9 con el 8 y, de nuevo al ver que no están en orden, se intercambian obteniéndose:

6	8	9	12	3	1

Luego se compara el 9 con el 12. Puesto que ya están en orden correcto, no se intercambian. En lugar de ello, se toma la siguiente pareja de valores para compararlos, es decir, se compara el 12 con el 3.

6	8	9	12	3	1

Puesto que no están en el orden correcto, se intercambian obteniendo:

6	8	9	3	12	1

Finalmente se compara el 12 con el 1 y se intercambian, quedando:

6	8	9	3	1	12

Esto completa la primera iteración, dejando el número de mayor valor en la última posición del arreglo. Sin embargo, pero no es posible asegurar lo mismo de los números restantes. La iteración subsiguiente deberá colocar el segundo mayor elemento en la penúltima posición (la correcta). Para ordenar completamente el arreglo, se deben realizar $n-1$ iteraciones.

El seudo-código para el método de la burbuja se muestra en el algoritmo 1. Como actividad implemente una función que realice el ordenamiento de la burbuja, utilice una función que use paso por referencia. La función tendrá como tipo de regreso *void* está modificará el arreglo para que queden los elementos en orden.

Algoritmo 1 Ordenamiento de burbuja

Entrada: A, un arreglo desordenado y n número de elementos en A

Salida: ordena A

1: $bubbleSort(A,n)$
2: **para** $i=0$ hasta n **hacer**
3:     **para** $j=0$ hasta $n-1-i$ **hacer**
4:         **si** $A[j]$¿$A[j+1]$ **entonces**
5:             $aux = A[j]$
6:             $A[j]$=$A[j+1]$
7:             $A[j+1]$= aux
8:         **fin si**
9:     **fin para**
10: **fin para**

10.4 Matrices y apuntadores

Como vimos en secciones previas las matrices se declaran utilizando *[][]*, donde los elementos se almacenan en forma de filas y columnas. Podemos pasar una matriz como parámetro a una función de la siguentes forma:

```
funcion(int M[][15], int filas)
funcion(int *M, int filas, int columnas)
```

Como ejercicio implemente un método de ordenamiento que reciba como parámetros la dirección de primer elemento de una matriz, la cantidad de filas y columnas. El método deberá regresar la matriz con sus columnas ordenadas.

Entrada

7	5	3	2	1
4	6	0	8	8
3	2	3	1	4

Salida

3	2	0	1	1
4	5	3	2	4
7	6	3	8	8

Fig. 8. Ejemplo ordenamiento por columnas

11. Memoria Dinámica

En **C** el manejo de memoria dinámica se gestiona de forma *manual*, es decir el programador debe reservar y liberar la memoria de forma explicita. La gestión de la memoria se realiza mediante el uso de dos funciones:

- *malloc*: Se utiliza para reservar(alocar) memoria.
- *free*: se utiliza para liberar memoria.

La función *malloc* es parte de la biblioteca en **stdlib.h**, esta nos permite reservar un bloque de memoria de un tamaño definido. El listado 11 ilustra el uso de la función *malloc* para reservar memoria para un puntero del tipo *int*, el tamaño se especifica en bytes.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      int *ptr = malloc(4);
5      printf("Direccion del Puntero: %p\n", ptr);
6      printf("Valor almacenado: %d\n", *ptr);
7      *ptr = 42; //Modificamos el valor almacenado
8      printf("Valor almacenado: %d\n", *ptr);
9
10 }
```

Listado 48. Uso básico de *malloc*

11.1 Arreglos y memoria dinámica

Cuando se requiere alocar memoria a na cantidad variable de números, que leeremos desde el teclado. Primero pedimos el número n de enteros a ingresar, y luego se leerán los n enteros. Este ejemplo se muestra en el listado 19, también se incluye la rutina **imprime** para verificar que los datos ingresados.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void imprime(int A[], int n){
5      for(int i=0;i<n;i++){
6          printf("%d ", A[i]);
7      }
8      printf("\n");
9  }
10 int main(){
11     int n;
12     scanf("%d", &n);
13     int *A=malloc(sizeof(int)*n);
14     for(int i=0;i<n;i++)
15         scanf("%d", A+i); // equivalente a scanf("%d", &A[i]);
16     imprime(A,n);
```

```
17
18 }
```

Listado 49. Uso de *malloc* para reservar memoria para un arreglo

Si bien *malloc* recibe la cantidad de bytes a alocar, en general esa memoria se requiere para almacenar algún tipo de datos especifico (*int*, *char*, etc) . Por ello, el número de bytes requeridos se calcula con el método *sizeof* multiplicado por el número de elementos. Note que es posible utilizar el apuntador ***A** de forma similar a como utilizamos un arreglo (como lo hacemos en la linea 6) arreglo/vector (array). Si bien apuntadores y arreglos son similares, no son exactamente iguales. Realice el ejercicio de que sucede si ejecuta *sizeof* con arreglo de 10 elementos, y compare que con lo que ocurre al hacerlo con un puntero de memoria para esa misma cantidad de elementos.

11.2 La función *free*

En **C** es necesario liberar de forma explícita la memoria que hemos alocado. Este proceso se realiza mediante la función *free*, la cual recibe como parámetro apuntador. Los lineamientos para la gestión de memoria serían: alocar el apuntador lo más tarde posible, es decir cuando se requiera su uso; y liberar la memoria lo más rápido posible, tan pronto como no sean requeridos los datos almacenados. Siempre que sea posible, se debería alocar y liberar en la misma (principio de *cercanía*).

12. Estrcuturas

Las ***estructuras*** son colecciones de variables de distinto tipo relacionadas bajo un *nombre* haciendo que su control o manejo sea mucho más sencillo. Podemos decir también, que las estructuras son como *arreglos*, pero que su diferencia es que las estructuras si permiten el almacenamiento de diferentes tipos de datos.

La utilización más común o más importante de las *estructuras* es cuando queremos programar y utilizar ***bases de datos*** (DB's), ya que nos permiten manejar de manera más sencilla los registros o fichas.

Entonces, tenemos que las ***estructuras son tipos de datos derivados***, su sintaxis declarativa es la siguiente:

```
struct tipo_estructura{
    tipo_variable nombre_variable1:
    tipo_variable nombre_variable2:
    tipo_variable nombre_variable3:
    ...
    tipo_variable nombre_variableN:
}
```

Listado 50. Sintaxis declarativa de *struct*

Donde:

- *tipo_estructura* es el nombre del nuevo tipo de dato que se está creando.
- `tipo_variable` y `nombre_variable` son las variables que forma parte de la estructura.

Estas variables se les conoce como ***miembros***. Los **miembros** deben de tener nombres únicos, mientras que dos o más estructuras pueden contener miembros con el mismo nombre. También debemos de tener claro, que los miembros de las estructuras pueden ser variables de los *tipos básicos* o *arreglos*, incluso otras *estructuras*, pero no pueden contener instancias de sí mismas.

12.1 Como trabajar con Estructuras

Para poder declarar una variable de tipo estructura, la estructura tiene que estar declarada previamente. Se debe declarar antes de la función *main*. Para definir y declarar las variables tipo estructura existen diferentes maneras, pero a continuación se presentan las más comunes y más utilizadas.

```
struct ejemplo ca, arr[10];
```

```
ejemplo ca, arr[10];
```

Las formas anteriores son declaran la variable *ca* de tipo *ejemplo* y *arr* de tipo arreglo de *ejemplo* con una dimensión de 10.

Las siguientes formas son las más recomendables, ya que nos permiten declarar las variables de tipo estructura cuando sea necesario utilizarlas o sea el caso al mismo tiempo que se crea la estructura.

```
struct ejemplo{
    char c;
    int i;
}
...
struct ejemplo ca, arr[10];
```

la otra forma es:

```
struct ejemplo{
    char c;
    int i;
}ca, arr[10];
```

De las dos últimas formas tenemos que en el primer caso declaramos la estructura, y en el momento en que necesitamos las variables, las declaramos. En el segundo las declaramos al mismo tiempo que la estructura. El problema del segundo método es que no podremos declarar más variables de este tipo a lo largo del programa, a menos que volvamos a la estructura y las declaremos.

Recordemos que una de las operaciones válidas entre estructuras es que pueden asignarse variables de estructura a variables de estructura del mismo tipo, dentro de las operaciones ***no válidas*** es que la comparación entre estructuras.

```
struct fecha{
    int dia, mes, anio;
    int diames;
    char nombre_mesanio[10];
}

struct persona{
    char nombre[35];
    char direccion[50];
    long int codigopostal;
    long int segurosocial;
    double salario;
    fecha nacimiento;
    fecha contrato;
}
```

Listado 51. Ejemplo del uso de *struct*

12.1.1 Inicialización

Las estructuras pueden inicializarse mediante el uso de listas como si fueran arreglos. Comenzamos declarando la variable a continuación del nombre un signo de igual con los inicializadores entre llaves y separados por coma, ejemplo:

```
ejemplo ca = {'a', 10};
```

Si fuera el caso de que en nuestra lista de inicialización aparecieran menos datos de los que tiene la estructura, los demás miembros son inicializados automáticamente en *0* ***(cero)***. Las variables de estructura también pueden ser inicializadas en enunciados de asignación asignándoles valores a los miembros individuales de la estructura.

```
ejemplo ca.c = 'a';
ejemplo ca.i = 10;
```

otro ejemplo pero ahora con el tipo *fecha*:

```
struct fecha f = {18,11,1983,254."Noviembre"};
```

12.1.2 Acceso a los miembros

Como se podrá haber visto en los códigos anteriores el acceso a los miembros de las variables de estructura se hace a traves del operador unario ¨.; el cual, se coloca entre el *nombre de la variable* y *nombre del miembro*.

Por ejemplo si quisiéramos imprimir en pantalla el miembro tipo *char* de la variable de estructura ***ca***, hacemos uso del siguiente enunciado:

```
printf("%c", ca.c);
```

otro ejemplo con la variable de estructura fecha llamada ***f***

```
printf("Nombre del mes: %s", f.nombre_mesanio);
```

12.2 Utilizar estructuras con funciones

Las estructuras pueden ser pasadas a funciones pasando miembros de estructura individuales o pasando toda la estructura. Cuando se pasan estructuras o miembros individuales de estructura a una función se pasan por llamada por valor. Para pasar una estructura en llamada por referencia tenemos que colocar él ***** o ***&***.

Los arreglos de estructura como todos los demás arreglos son automáticamente pasados en llamadas por referencia. Si quisiéramos pasar un arreglo en llamada por valor, podemos definir una estructura con único miembro el arreglo. Dicho lo anterior, entonces una función puede devolver una estructura como valor.

```
struct punto {
    int x;
    int y;};
```

```
/* creo_punto: crea un punto a partir de sus coordenadas */
punto creo_punto(int a, int b){
    punto temp;
    temp.x=a;
    temp.y=b;
    return temp;
}

/* sumo_puntos: suma dos puntos */
punto sumo_puntos(punto p1,punto p2){
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
/* imprimir_punto: imprime las coordenadas de un punto */
void imprimir_punto(punto p){
    printf("Coordenadas del punto: %d y %d \n", p.x, p.y);
}
```

Listado 52. Ejemplo *struct* en funciones.

12.3 Typedef

Es posible agrupar un conjunto de elementos de tipo estructura en un arreglo. Esto se conoce como arreglo de estructuras: El lenguaje *C* dispone de una declaración llamada `typedef` que permite la creación de nuevos tipos de datos. Esta palabra reservada se utiliza comúnmente para crear pseudónimos para los tipos básicos de datos.

A continuación se muestra un programa completo que nos permite barajar y distribuir un mazo de cartas.

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

struct carta {
    int numero;
    char palo[7];
};

typedef carta Carta;

typedef char Palo[7];

void inicializar_mazo(Carta m[],Palo p[]);

void barajar(Carta m[]);

```

```
18 void imprimo(Carta m[]);
19
20 int main (){
21     Carta mazo[48];
22     Palo p[4] = {"copa","oro","espada","basto"};
23
24     srand(time(NULL));
25
26     inicializar_mazo(mazo,p);
27     barajar(mazo);
28     imprimir(mazo);
29     system(    PAUSE    );
30 }
31
32 void copiar(char a, char b, int largo){
33     int i;
34     for (i=0;i < largo;i++)
35         a[i]=b[i];
36 }
37
38 void inicializar_mazo(Carta m[],Palo p[]){
39     int i;
40     for (i=0; i < 48;i++){
41         m[i].numero=(i % 12)+1;
42     copiar(m[i].palo,p[i/12],7);
43     }
44 }
45
46 void barajar(Carta m[]){
47     int i,j;
48     Carta temp;
49     for (i=0; i < 48; i++){
50         j = rand() % 48;
51         t  emp = m[i];
52         m[i] = m[j];
53         m[j] = temp;
54     }
55 }
56
57 void imprimir(Carta m[]){
58     int i,j;
59     char c;
60
61     for (i=0; i < 48; i++){
62         printf (    %i de     , m[i].numero);
63         printf (    %s     , m[i].palo);
64         printf (   \ n   );
65     }
66 }
```

Listado 53. Ejemplo para barajar y distribuir un mazo de cartas.

El programa representa el mazo de cartas como un arreglo de estructuras, donde cada estructura contiene el número de la carta y el palo.

Las funciones son: `inicializar_mazo` que inicializa un arreglo de cartas con los valores de las cartas ordenado del 1 al 12 de cada uno de los palos, `barajar` recibe un arreglo de 48 cartas y para cada una de ellas se toma un número al azar entre el 0 y el 47. A continuación se intercambia la carta original con la seleccionada al azar, y finalmente la función `imprimir` imprime las cartas luego de barajar.

13. Archivos

En programación, es posible que necesitemos que algunos datos de entrada específicos se generen varias veces. A veces, no es suficiente mostrar solo los datos en la consola. Si la cantidad de dato es grande, solo es posible mostrar una cantidad limitada de estos en la consola, y dado que la memoria es volátil, es imposible recuperar los datos generados mediante programación una y otra vez. Sin embargo, si necesitamos hacerlo, podemos almacenarlo en el sistema de archivos local(memoria persistente) el cuál podemos en todo momento. Aquí surge la necesidad de manejar archivos en **C** .

El manejo de archivos en **C** nos permite crear, actualizar, leer y eliminar los archivos almacenados en el sistema de archivos local a través desde un programa en **C** . Las siguientes operaciones se pueden realizar en un archivo.

- Crear un archivo
- Abrir un archivo
- Leer desde un archivo
- Escribir a un archivo
- Eliminar un archivo

Instruccion	**Descripción**
fopen	Abre un archivo nuevo (puede o no existir)
fprintf	Escribe datos en un archivo
fscanf	Lee datos de un archivo
fputc	Escribe un carácter en un archivo
fputs	Escribe un secuencia de carácteres en un archivo
fgetc	Lee un carácter en un archivo
fgets	Lee una cadena del *stream* de entrada y la almacena
fclose	Cierra un archivo abierto
fseek	Asigna una posición a un puntero a archivo
fputw	Escribe un entero en un archivo
fgetw	Lee un entero desde un archivo
ftell	regresa la posición actual
rewind	pone el apuntador al inicio del archivo

Table 7. Funciones **C** para manejo de archivos

Modo	Descripción
r	Modo lectura
w	Ecitura
a	Actualización (agregar al final)
r+,w+	lectura y escritura
a+	Actualización, lectura y escritura
rb	Abre un archivo binario en modo de lectura
wb	Abre un archivo binario en modo de escritura
ab	Abre un archivo binario en modo de actulaización
ab+	Binario Actualización, lectura y escritura

Table 8. Modos de acceso a archivos

14. Modos se acceso

Al abrir un archivo, debemos especifica que operaciones queremos realizar, es decir si leer, escribir o actualizar. Esto, se conoce como modo de acceso y la función *fopen* nos permite especificarlo. La función *fopen* recibe como parámetro dos cadenas una con la ruta del archivo y otra más con la descripción del modo de acceso.

El formato para utilizar *fopen* se muestra en el listado 2

```
1    FILE *stream fopen(char *path , char *mode);
```

Listado 54. Sintaxis *fopen void*

donde **path** es la ruta del archivo, **mode** el modo de acceso que puede ser cualquiera de los listados en la tabla 8 y **stream** es un apuntador a un tipo *FILE* (valor de regreso).

El primer parámetro es una cadena que indica la ruta del archivo (puede ser absoluta o relativa), el segundo es el modo de acceso al archivo y puede ser cualquiera de los indicados en la tabla 8.

```
1    // Escritura
2    int fprintf(FILE *stream, const char *format [,argumento, ...])
3    int fputc(int c, FILE *stream)
4    int fputs(const char *s, FILE *stream)
5    int fwrite(const void *stream, int size, int nmemb, FILE *stream)
6
7    // Lectura
8    int fscanf(FILE *stream, const char *format [, argument, ...])
9    int fgetc(FILE *stream)
10   char* fgets(char *s, int numChars, FILE *stream)
11   int fread(void *stream, int size, int nmemb, FILE *stream)
12
13   //Reubicar el apuntador
14   int fseek(FILE *stream, long int offset, int origin)
```

Listado 55. Sintaxis funciones sobre archivos *void*

14.0.1 Escritura de texto en archivos

Las funciones *fprintf*,*fputc*, *fputs* escriben en el *stream* de salida. Todas reciben el apuntador al *stream* de salida el cual puede ser una archivo o **stdout** y los datos escribir.

La función *fputc* solo escribe un carácter a la vez, mientras que *fprintf* y *fputs* soportam *"strings"*. Para *fprintf* también se puede especificar una cadena formato. En las siguientes secciones veremos como escribir y recuperar estructuras y arreglos desde archivos binarios.

14.0.2 Lectura de texto desde archivos

La función *fgetc* lee y devuelve un carácter (representado como un *int*), cuando llega la final del archivo regresa **EOF** (End Of File) para indicar un error o que se alcanzo el final del archivo. La función, utiliza las funciones *feof* o *ferror* para distinguir entre un error y una condición de fin de archivo. Esta función solo recibe el **stream** de datos desde donde se leerá el carácter (linea 9 en el listado 15).

La función *fgets* e una cadena del **stream** de entrada y la almacena en un apuntador a cadena. La función lee caracteres desde la posición actual en el **stream** hasta el primer carácter de nueva línea (incluyendo al carácter *'\n'*), hasta el final de la secuencia o hasta que el número de caracteres leídos sea igual a numChars – 1 (el segundo parámetro ver línea 10 del listado 15), lo que ocurra primero. El resultado almacenado se le agrega al final el carácter nulo (*'\0'*).

El método *fscanf* lee datos formateados de una secuencia (similar a *scanf*), la diferencia es que en *fscanf* se especifica el **stream** de entrada.

Hay disponibles versiones más seguras de estas funciones

14.1 Ejemplo lectura/escritura archivos

Para los ejemplos asumiremos que se tiene el archivo de texto plano **hunters.txt** con el siguiente contenido:

```
Kyllua Zoldyck 5.2
Leorio Paradinight 7.5
Gon Freecss 8.2
Hisoka Morow 10.0
Kurapika Kurta 9.5
Illumi Zoldyck 9.8
```

Cuando de ejecuta la función *fopen* en modo lectura. Está primero En primero localiza el el archivo a abrir, una vez identificado es cargado desde el disco y colocado en el búfer. El búfer se utiliza para proporcionar eficiencia a las operaciones de lectura. La función regresa un apuntador un al primer carácter del archivo. Considere el ejemplo del listado 12 que abre un archivo en modo de lectura. El archivo es leído e impreso carácter a carácter, note que también se leen e imprimen los saltos de línea.

```c
#include <stdio.h>
int main( ){
     FILE *f=fopen("hunters.txt","r"); //Se abre el archivo en modo lectura
      char c=fgetc(f); //se lee el primer caracter
      while (c !=EOF ){// verifica si se llego al final del archivo
         c = fgetc (f);//se lee siguiente caracter
         fprintf(stdout, "%c", c); // se despliega el caracter
     }
     fclose(f); //se cierra el archivo
     return 0;
}
```

Listado 56. Ejemplo de lectura

Ahora revisemos un ejemplo en el que utilizamos un formato tanto para leer como para escribir. El listado 19 ilustra el uso de las funciones *fscanf* y *fprintf*.

```c
#include <stdio.h>
int main(){
  FILE *f, *fc;
  f=fopen("hunters.txt", "r"); // Abrir archivo en modo lectura
  fc=fopen("truncadas.txt", "w"); // Abrir archivo en modo de escritura
  char nombre[12]; //para almacenar un nombre de maximo 12 caracteres
  char apellido[12];
  float calificacion;
  // leer dos cadenas y un flotante
  fscanf(f,"%s %s %f", nombre, apellido, &calificacion);
  while(feof(f)==0){// verificar si se alcanzo el final del archivo
    //escribir en truncadas.txt
    fprintf(fc, "%s %s %d\n", apellido, nombre, (int)calificacion);
    fscanf(f,"%s %s %f", nombre, apellido, &calificacion);
  }
  fclose(fp);
  fclose(f);
}
```

Listado 57. Ejemplo de lectura utilizando *fscanf/fprintf*

En el listato 19 se lee el archivo **hunters.txt** y se copia la información **truncadas.txt**. En la función para leer (*fscanf*) especificamos el formato para cada una de las secuencias a leer. Al escribir los datos en el segundo archivo se intercambia la posición del **nombre** y **apellido**, además truncamos el valor flotante a un *int*. Si leemos el archivo **truncadas.txt** deberá contener lo siguiente:

```
Zoldyck Kyllua 5
Paradinight Leorio 7
Freecss Gon 8
Morow Hisoka 10
Kurta Kurapika 9
Zoldyck Illumi 9
```

14.1.1 Reubicación del apuntador

La función *fseek* mueve el apuntador de archivo (si lo hay) asociado con la secuencia a una nueva ubicación **offset** bytes desde el **origin**. La siguiente operación en la secuencia se lleva a cabo en la nueva ubicación. El argumento **origin** debe ser una de las siguientes constantes, definidas en la **stdio.h**: *SEEK_CUR* posición actual del apuntador, *SEEK_END* Fin del archivo o *SEEK_SET* inicio del archivo.

La función de *rewind* reposiciona el apuntador asociado a un archivo al principio del archivo. Una llamada a *rewind* es similar a *fseek*(strema, 0,SEEK_SET). Sin embargo, a diferencia de *fseek*, keywrewind borra los indicadores de error de la secuencia, así como el indicador de fin de archivo. Además, a diferencia de *fseek*, no devuelve un valor para indicar si el puntero se movió correctamente.

Para borrar el búfer del teclado, se puede utilizar *rewind* sobre **stdin**, la cual por defecto está asociada con el teclado.

14.2 Lectura y escritura de datos en binario